UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Fachgebiet Spezifikation und Modellierung von Softwaresystemen
Warburger Straße 100
33098 Paderborn

On-The-Fly Safety Checking – Customizing Program Certification and Program Restructuring

Dissertation

zur Erlangung des akademischen Grades
Doktorin der Naturwissenschaften (Dr. rer. nat.)
der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von
Marie-Christine Jakobs, Master of Science

Paderborn, 2017

Abstract

Many artifacts that we use, for example, washing machines, TVs, mobile phones, and cars, run software. Software has a high impact on our daily life. Thus, it should be error free. The first version of a software is almost never free of errors. To eliminate errors, companies spent an enormous time on the validation of their software. For example, formal verification is used to find errors and to finally show that safety critical software does not contain errors with respect to the specification. Software originating from a company which runs an extensive validation phase during software development can be installed on our systems without any further checks.

In the past, users often installed software from large companies. Typically, they had installed different software from that company before and experienced its quality. Due to the long-term business relationship, users could estimate the quality of software produced by these companies. However, the software usage behavior changed. For instance, users regularly download applications on their mobile phones from software markets like Google Play. Applications are no longer only written by large companies but often by unknown software producers, which may be a private person or new in the market and from whom one never purchased a software product before. There is no reason why one should trust that these producers build error free software products. We need other approaches to determine whether a software product received from such producers is free of errors.

In this thesis, we develop techniques that eliminate the trust in the software producer and still convince a consumer that a bought software product is free of errors. Our techniques are based on formal verification, are fully automatic, and are configurable to the property of interest as well the verification technique. The underlying idea of our techniques is that the producer performs the expensive verification and uses the proof information obtained by the verification to simplify and speed up the consumer's reinspection. We present techniques from two research directions. First, we present configurable program certification approaches which follow the classical idea of certification. The producer attaches some witness information, for example, some parts of the proof, to the program. The consumer solely needs to validate the certificate. This means, he needs to check that it is a proper witness. Second, we suggest the Programs from Proofs approach. This approach uses the proof information to restructure the program such that the verification of the restructured program is simpler. After restructuring, for example, removal of infeasible paths or splitting of syntactical paths, the consumer can run a simple and efficient dataflow analysis. Finally, we discuss the integration of Programs from Proofs with configurable program certification. For all our approaches, we show that they are adequate. A successful reinspection implies that the program adheres to the checked property and the reinspection succeeds when the producer does not try to cheat. Furthermore, we implemented prototypes for our approaches and used them in an extensive, practical evaluation.

Zusammenfassung

Viele Gegenstände des täglichen Gebrauchs führen Software aus. Software hat einen starken Einfluss auf unser tägliches Leben und sollte daher fehlerfrei sein. Anfänglich ist Software aber selten fehlerfrei. Firmen betreiben einen hohen Validierungsaufwand, um Fehler in der Software zu eliminieren. Zum Beispiel wird formale Verifikation eingesetzt, um Fehler zu finden und schlussendlich zu zeigen, dass sicherheitskritische Software den Anforderungen genügt. Software, die aus dem Hause einer Firma stammt, in der eine aufwändige Validierungsphase Teil des Entwicklungsprozesses ist, kann bedenkenlos auf unseren Systemen installiert werden.

Früher haben Verbraucher vorwiegend Software aus großen Softwarehäusern installiert, von denen sie bereits Software benutzt und die Qualität dieser Software erfahren haben. Aufgrund der langzeitigen Geschäftsbeziehung konnten Verbraucher die Softwarequalität dieser Hersteller abschätzen, aber die Softwarenutzung hat sich verändert. Heute installieren Verbraucher zum Beispiel regelmäßig Anwendungen auf ihren Handys, die sie auf Plattformen wie Google Play finden. Zusätzlich werden solche Anwendungen nicht mehr ausschließlich von großen Firmen geschrieben, sondern immer häufiger auch von unbekannten Entwicklern, Privatpersonen oder Marktneulingen, sodass nicht unbedingt zuvor Software von diesen Herstellern bezogen wurde. Daher kann nicht erwartet werden, dass der Verbraucher auf die Fehlerfreiheit der Software vertraut, und es werden andere Verfahren benötigt, um den Verbraucher davon zu überzeugen, dass gekaufte Software fehlerfrei ist.

In dieser Arbeit entwickeln wir Techniken, die ohne Vertrauen in den Softwarehersteller auskommen und dennoch den Verbraucher davon überzeugen, dass sein gekauftes Softwareprodukt fehlerfrei ist. Unsere Techniken basieren auf formaler Verifikation, sind automatisch und können an die einzuhaltende Spezifikation und die zu verwendende Verifikationstechnik angepasst werden. Die Idee unserer Techniken ist wie folgt. Der Produzent beginnt mit einer aufwändigen Verifikation und nutzt die Informationen aus der Verifikation, um die Nachprüfung des Konsumenten zu vereinfachen und zu beschleunigen. Wir verfolgen zwei unterschiedliche Forschungsansätze. Das Konzept der konfigurierbaren Programmzertifizierung nutzt die Idee klassischer Zertifizierung. Der Produzent liefert das Programm mit einem Zertifikat aus. Das Zertifikat enthält Informationen, zum Beispiel Teile seines Verifikationsbeweises, die bezeugen, dass der Produzent die Verifikation erfolgreich durchgeführt hat. Somit muss der Verbraucher nur prüfen, ob das Zertifikat gültig ist, also tatsächlich bezeugt, dass die gewünschte Verifikation erfolgreich absolviert wurde. Das Konzept Programs from Proofs nutzt dagegen die Informationen aus der Verifikation, um ein einfacher zu beweisendes Programm zu erzeugen. Zum Beispiel werden nicht ausführbare Pfade entfernt oder Pfade voneinander separiert. Somit muss der Verbraucher in seiner Nachprüfung lediglich eine einfache Datenflussanalyse durchführen. Des Weiteren betrachten wir auch eine Integration der beiden Ansätze. Außerdem beweisen wir für alle Techniken, dass sie nutzbar sind. Das heißt, eine erfolgreiche Nachprüfung des Verbrauchers bedingt, dass das Softwareprodukt die geprüfte Eigenschaft erfüllt. Weiterhin ist garantiert, dass die Nachprüfung erfolgreich ist, wenn sich der Produzent an die Regeln hält. Neben diesen theoretischen Eigenschaften haben wir die Techniken mittels eines Prototypens intensiv evaluiert.

Ackknowledgements

First, I would like to thank my advisor Heike Wehrheim for her guidance, her support and for offering me the opportunity to work on this topic. Furthermore, I owe my colleagues and former colleagues Steffen Beringer, Galina Besova, Tobias Isenberg, Jürgen König, Julia Krämer, Alexander Schremmer, Dominik Steenken, Nils Timm, Manuel Töws, Oleg Travkin, Sven Walther, and Steffen Ziegert a big gratitude. They let me work in a friendly and professional atmosphere. I always loved going to work. Special thanks to Oleg Travkin and Sven Walther for the enjoyable, often off-topic discussion as well as to Tobias Isenberg for the productive atmosphere in our office. I also want to thank our secretary Elisabeth Schlatt for her kind administrative support as well as my research colleagues in the CRC 901 who helped me see the big picture of my research.

Moreover, I like to say thank you to my PhD committee members Prof. Dr. Heike Wehrheim, Prof. Dr. Dirk Beyer, Prof. Dr. Eric Bodden, Prof. Dr. Marco Platzner, and Dr. Marie Christin Platenius.

I thankfully acknowledge the contributions of Henrik Bröcher and our student assistant Mike Czech to the implementation. Mike Czech realized the sign analysis and Henrik Bröcher extended the available set of partitioning strategies.

I am also grateful to Dirk Beyer and his group who let me run my experiments in their VerifierCloud and offered me the opportunity to realize my approaches in their software analysis tool CPAchecker. Especially, Philipp Wendler and lately also Karlheinz Friedberger always helped me whenever I had some administrative questions or questions regarding the implementation.

Last but not least, I want to say thank you to my family and my boyfriend for their support, their patience, and understanding, especially that I regularly spent my free time writing this thesis.

Contents

List of Figures

List of Tables

1 Introduction

Software plays an important role in our daily life. Still, software runs in classical environments, e.g., on computers. More importantly, the number of non-classical devices executing software grows enormously. Today, many of our household appliances, e.g., our washing machines, are controlled by software. Our TVs are connected with the internet. Mobile phones can be extended with software applications. Furthermore, the number of assistant systems in cars, typically written in software, rises. Considering current trends like e.g. smart homes or smart meters, our dependence on software will even further increase in future. Due to the broad application area and the high impact on our daily lives, more than ever software must work as intended. It should be error free.

The first version of a software program is almost never error free. Since the beginning of software development, people proposed and used software validation techniques, which try to find software errors. Nowadays, developers can select from a broad set of validation techniques like e.g. software model checking [JM09b], program analysis [NNH05], symbolic execution [Kin76], or testing [SL12]. Nevertheless, software validation still takes a significant proportion of the software development time. In principle, the time spent on validation depends on the characteristics of the software projects. For larger, safety critical projects, validation can take up to 80% of the development time [SL12, pp. 16 f]. Thus, software validation is a major cost factor in the software development process.

While the software quality problem remains – software programs still contain errors –, software development changed over the years. In the early stages, monolithic software is developed in companies. Ideas like e.g. component-based design [HC01] and service oriented architectures [Erl05] caused software and its development to become modular. Existing components can be reused in different software projects and some functionality of a software can even be realized by a call to an external service, e.g., a web service developed in a different company. Simultaneously, modularity also eases the development of open source software, which is often built by a community. In some open source projects, users may become part of the community and actively contribute to the open source software. Recently, customization of software becomes more and more important. Using the concept of software product lines [PBvdL05], companies do not longer develop a single software solution, but develop variations for different functionality which can be configured to a customized software solution. Furthermore, accompanied by the increasing popularity of

smartphones and tablets, more and more people extend the vendor defined functionality of their devices. To fit their needs, they search for suitable software (applications) on market platforms like e.g. Google Play and install software, which possibly cooperates with already installed software. Since such market platforms are mostly open to everyone, users run software from different vendors, which are not necessarily companies but possibly also private persons.

As throughout the years, software users still do not and also in future will not like to experience (too many) software errors. To avoid the experience of errors, users aim at selecting software with high quality. Often, software that is thoroughly validated against certain correctness properties comes with such a high quality. Hence, users try to estimate whether a supplied software is thoroughly validated. Since the number of software applications run by today's users increases, in total more time must be spent for the estimation. Additionally, we claim that judging thoroughly validation in today's global software markets, which are accessible to suppliers all over the world, is more difficult. We explain this claim in the next section in more detail.

1.1 Software Markets Easily Accessible to Everybody

In this section, we discuss two examples for global software markets, which are easily accessible to suppliers, private persons as well as companies, spread all over the world. We start with an already existing class of markets, the smartphone application markets. Thereafter, we present our future vision of these global software markets. Based on the insights obtained from these markets, especially our vision of the future market, we then explain why estimating software quality in global software markets is difficult.

1.1.1 Smartphone Application Markets

Google Play[1] and Apple's AppStore[2] are two popular market platforms for smartphone applications. Customers can search, buy, and download applications for their smartphones. Application developers can offer their applications on these platforms. In contrast to Google Play, applications are only published in the AppStore after they passed a review process. Thus, both platforms already realize software markets that are open to nearly anyone. Nevertheless, they are limited. In principle, applications can cooperate. For example, Android apps offered in Google Play use (implicit) intents to call the functionality of other apps. To obtain a larger functionality, applications can be composed. However, the customer must build the composition on his own, e.g., search for components or check whether components can cooperate with each other. The customers can only search for existing applications. On-the-fly computing tries to overcome these drawbacks.

1.1.2 The Vision of On-The-Fly Computing

App stores are real world examples for markets in which software solutions from all over the world can be offered by nearly anyone. To participate as a producer, one must only be able to create an app. However, these markets lack an important trend, namely *customization*. Still, customers can only select existing solutions.

[1] `https://play.google.com/`
[2] `https://www.apple.com/appstore`

During our research within the CRC 901 "On-The-Fly Computing"[3], we already considered future markets, so called on-the-fly markets (OTF markets), which offer solutions for the individual customer needs. We assume that these future markets are highly dynamic. Participants may enter or leave the market at any time, change, e.g., extend, their offerings or prices. Furthermore, participants, except for maybe customers, act entrepreneurial and strategically. Our CRC's vision is that based on a customer request an individual, high quality solution, software plus execution (environment), for the customer is automatically built on demand ("on-the-fly"). The software solution is composed of existing software services which are freely traded on global software markets. While the composition process is under control by the one who builds the software solution, he relies on the quality of the composed, single services received from the market. Probably, a high quality of the composed software solution can only be achieved when each single software service in the composition is of high quality. Next, we discuss why reliably estimating the quality of these single services is especially difficult or even impossible in on-the-fly markets.

1.1.3 The Quality Problem in On-The-Fly Software Markets

Two characteristics of on-the-fly markets intensify the quality problem on software services in these markets. Due to its *dynamics*, participants, e.g., producers of software services, may have entered the market recently and may also stay only for a short period of time. Furthermore, its *liberal accessibility* results in a large and heterogeneous set of producers. Software services constructed by different producers vary in their quality. One producer may be less experienced than another producer. Furthermore, producers also plan their software service quality strategically. For example, if a producer wants to earn a lot of money in a short period of time and afterward leaves the market, he will strategically decide not to invest in validation and to offer services with low quality. In contrast, producers which aim at long-term business relationships may validate more thoroughly to produce high quality services. At worst, a producer is malicious. His goal is to harm consumers. Thus, he provides a faulty software intentionally.

For a consumer, the one who builds the composition, it is difficult to estimate the quality of a software service produced by a particular producer. Since in on-the-fly markets many producers offer software services, often a consumer purchases a software for the first time from that particular producer and he only purchases once from that producer. Consumers cannot use their past experience to estimate the quality and to establish a trust relation. Additionally, when a producer entered the market recently, the consumer likely cannot profit from experiences of other consumers. In such a case, no consumer or only a few consumers rated the producer's quality so far. No reliable reputation for the producer and his quality exists.

To convince the consumer of the quality of a software service, trust and reputation do not work. We require additional mechanisms and techniques. In the following, we discuss existing mechanisms and techniques that try to convince the consumer of the software quality, i.e., that try to convince him that a software service adheres to certain correctness properties. Furthermore, we propose our general approach to convince a consumer that a software service adheres to certain correctness properties, introduce requirements on concrete instantiations, and sketch the three concrete instances dealt with in this thesis.

[3] `http://sfb901.uni-paderborn.de/`

1.2 Assuring Correctness Properties in the Context of Global On-The-Fly Markets

A very simple way to convince the consumer is to implement the software in a safe programming language. The idea of a safe programming language is that programs that do not adhere to certain correctness properties cannot be expressed in the language. For example, type and memory safe programming languages like Java ensure that no type violations or memory access violations may occur during program execution. We think that such an approach works well for low-level correctness properties. However, it is at least difficult to use it for arbitrary safety properties like e.g. the result of the function call *gcd* returns the greatest common divisor of the two input parameters.

A different class of approaches looks at the concrete execution paths of a software program. Techniques like sandboxing [CDK05, p. 2070], which e.g. limits a program's access to certain resources whenever it is executed, or the execution in a virtual machine ensure that a software program does not harm the system or other programs. However, they do not assure functional correctness properties. Other techniques like result checking [WB97] and certifying algorithms [MMNS11] validate that the program's output is compliant with the intention of the program. Runtime verification [LS09] proves that a single execution of a software program adheres to a correctness property (specification). A monitor, which is generated from the property of interest, inspects the run and decides adherence. In online monitoring, the monitor observes the current execution of a program. All these techniques have in common that they detect a violation of a correctness property, but only late at runtime. When no mechanism is applied to recover from a violation, errors will still be observed indirectly by abortion of the program. Furthermore, applying these techniques often results in a runtime overhead. These techniques can be used to protect against harmful software programs, but they are poorly suitable to convince a consumer of the correctness of a software program.

Of course, the consumer could validate the software program himself using e.g. one of the techniques [JM09b, NNH05, Kin76] mentioned earlier. We would prefer automatic, formal verification [DKW08] because these techniques often guarantee that a program adheres to the correctness property of interest. Additionally, they can be used by the consumer, typically a non-expert. Nevertheless, we think that this solution is inappropriate. First, validation is the task of the software producer. For example, a consumer cannot fix a found error. Furthermore, the producer likely uses validation when he wants to build high quality software. Thus, every consumer would repeat the validation process the producer already performed. Second, we already mentioned that validation can be quite time consuming. Such a consumer validation is particularly infeasible in on-the-fly markets, in which a software program is composed with others on demand. Typically, the composed software should be executed almost immediately after the composition. We conclude that the producer and the consumer must cooperate. The producer must convince the consumer that he did the validation and that his validation confirms the correctness of the program. Next, we consider protocols which might be used to convince the consumer.

Digital certificates [DH76] can be used to certify the originator of the software or its integrity. For example, the software producer signs the software with his private key to prove that he is the originator. The consumer can check the origin with the help of the producer's public key. However, digital certificates do not assure that the software adheres to certain correctness properties. The consumer still needs to trust the provider. Thus, digital certificates are not suitable for our context.

The amanat protocol [CSV07, CSV13] is a cryptographic protocol to convince the consumer that the producer's software program adheres to certain correctness properties. The idea is that a third party, an amanat which is controlled by the consumer, validates the software program, builds the executable and signs the executable as well as the validation verdict with a private key only known by the consumer and the amanat. Before the signed executable and validation verdict are sent to the consumer, the producer checks them and only forwards them to the consumer when the amanat does not reveal the source code to the consumer. We think this protocol is difficult to realize in on-the-fly markets. A consumer does not purchase from every producer in the market. Only on demand, he might be interested in a software product of a producer. It does not make sense to establish the amanat protocol among any producer consumer pair separately. This could even be impossible due to the large set of market participants. In contrast, having one amanat for all consumers is likely insecure. The private key must be shared. Thus, getting the private key would become much easier for the producer. Additionally, validation is still performed by the consumer. Hence, we decided against the amanat protocol.

Proof-Carrying Code (PCC) [NL96, Nec97] is a protocol to ensure that a program from an untrusted producer is correct w.r.t. certain correctness properties. In its basic form, the producer performs a formal verification and sends the consumer the software program together with the proof. Then, the consumer checks whether the received proof shows that the received program fulfills the correctness property of interest. We think that this idea is well-suited to assure correctness properties in global on-the-fly markets. The producer does the difficult verification, while the consumer only performs a fast inspection of the proof. The PCC idea is already used in various approaches (see e.g. [NL96, Nec97, App01, WNKN04, MWCG99, CW00, SYY03, Cha06a, Ros03, APH05b, AAPH06, BJP06, HNJ+02, CMZ15, BDDH16, ACAE08]). Nevertheless, the usage of these approaches in on-the-fly markets is limited.

First, some approaches, e.g., PCC [NL96], foundational PCC [App01], and verified PCC [WNKN04], use semi-automatic verification procedures like theorem provers. Verification must be performed by an expert. Reconsidering the heterogeneous class of developers, we cannot expect that all of them are experts. We believe that the producer verification and the consumer validation must be automatic.

Second, PCC approaches are sometimes restricted to certain properties. For example, in the original PCC approach [NL96] the safety property is hard-coded in the verification condition generator, a component of the verification and validation process. In foundational PCC [App01], the safety property is encoded in the program semantics. Similarly, in typed assembly language [MWCG99] the property is incorporated into the type system. We assume that in on-the-fly markets consumer are interested in different correctness properties. We think that providing the infrastructure for all approaches which might be used to assure a property is cumbersome. It is better to have one single approach for all properties.

Third, the verification is often restricted to a certain analysis or analysis type. For example, Henzinger et al. [HNJ+02] use predicate model checking. Rose [Ros03] considers dataflow analyses and Albert et al. [APH05b, AAPH06], Besson et al. [BJP06], or Seo et al. [SYY03] rely on abstract interpretation. However, we suppose that no analysis (type) is always suitable to verify a certain class of properties. The analysis should be interchangeable.

For on-the-fly markets, we require automatic and flexible PCC alike approaches, which cannot only be configured to the property of interest, but must be adaptable to the verification technique most suitable to prove the validity of the property. Our principle

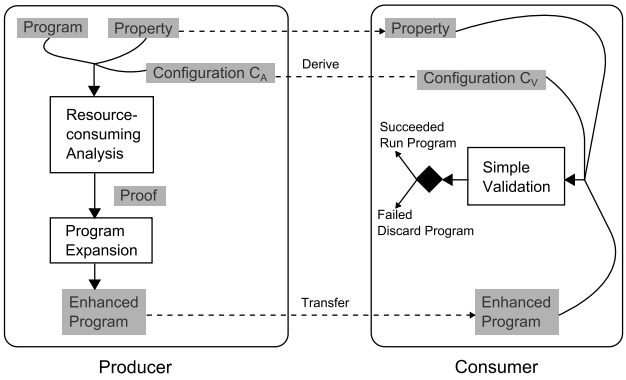

Figure 1.1: Configurable, abstract protocol to convince a consumer that an untrusted program is safe

for asserting correctness properties in on-the-fly markets, which is described in the next section, takes all these aspects into account.

1.2.1 Our Principle

In this section, we describe our high level solution to convince the consumer that a received, untrusted program is safe. We think it is unacceptable that the consumer carries the burden to validate a program that a producer wants to sell or distribute. Hence, like e.g. Proof-Carrying Code [NL96, Nec97] our solution is a protocol between two parties, the producer and the consumer, in which the major workload for program validation is shifted to the producer. From our point of view, such a workload shift is acceptable because the producer directly profits from the salary of the service. Additionally, the producer should validate his program anyway. Furthermore, the producer's validation is not really under time pressure. It can be done offline and needs not be done on demand. In contrast to existing approaches, which are semi-automatic or restricted to certain properties or analyses, we envision a solution which is automatic and broadly applicable. Instead of realizing different protocols for various properties, we have one solution for all properties, which can also be used by a non-expert. To achieve the broad applicability and the automation, we suggest to build our solution on existing, configurable and automatic software analyses [Kil73, CC77, BHT07, BHT08]. Figure 1.1 shows the course of our proposed protocol for the case that the producer verification is successful. So far, this protocol is generic. To instantiate the protocol, the three rectangles with the black borders must be replaced by concrete approaches.

First, the producer and the consumer agree on the correctness property or the producer simply fixes a property. Additionally, the producer must choose a configuration C_A suitable for the analysis of the determined property. Based on the configuration C_A, the producer starts a resource consuming analysis of the program w.r.t. the property. In principle, a protocol instance may use any configurable approach for the resource consuming analysis of the producer that runs automatically and does not simply output true (property satisfied) or false, but also provides some kind of proof for its answer. We do not restrict proofs to be mathematical proofs. Instead, we allow any kind of formal reasoning, e.g., a fixpoint [APH05b], an inductive invariant [MP95, p. 87 f], the precision of the final abstract model [BLN+13], a description or guide for the proof search [NR01, TA10], or a

model of the explored state space [HJMS03]. The only restriction is that the information must be usable to simplify the validation procedure on the consumer side.

Note that the protocol finishes when the producer analysis fails to show correctness of the program w.r.t. the property. In this case, either the program is faulty and must be corrected before the process can be restarted or the configuration is improper to show the validity of the property. The process must be restarted with a more suitable configuration. Figure 1.1 shows the course of the protocol when the producer analysis succeeds. Then, the producer uses a program expansion component which enhances the program, i.e., it adds some information of the proof, which should simplify the program validation. A very common approach of a program expansion is to attach additional information to the program. For example, Proof-Carrying Code [NL96, Nec97] attaches a mathematical proof. Other approaches annotate the program with additional information. For example, one could use JML annotations [LBR99] or construct proof outlines [OG76]. We also allow unorthodox program expansions. In this thesis, we consider a protocol instance which uses the proof to restructure the program and thus implicitly encodes the proof information in the restructured program. The result of the program expansion is the enhanced program, e.g., the program plus attachment, the annotated program, or the restructured program. This enhanced program is shipped to the consumer.

We require that our protocol (instances) can deal with corruption. Corruption may be caused by a malicious attacker or simply by a failure during transportation of the enhanced program. After the consumer receives a possibly corrupted enhanced program, he first derives a configuration C_{V} for his validation, which fits to the producer's analysis configuration C_{A}. Then, the consumer uses a simple validation that is configured by the configuration C_{V}, the enhanced program received by the consumer, and the property the producer and consumer agreed on. Again, different approaches can be used for the validation. Typically, they depend on the type of the enhanced program. Examples are type checking of the proof [NL96, Nec97], (partial) reverification [BLN+13], and checking that the attached proof object fulfills certain properties [HJMS03]. In all cases, the simple consumer validation must run automatically and must have two possible outcomes: succeeded or failed. Note that the outcome failed can have several reasons, e.g., the validation is not able to check the property, the validation takes too long and is aborted due to a timeout, or the validation exceeded its memory. Based on the validation outcome, the consumer decides how to treat the program received by the producer. A successful validation implies that the program is accepted by the consumer, i.e., it is installed or run. In contrast, a program is discarded whenever its validation failed.

Next, we discuss the requirements that each protocol realization should fulfill to become practically usable.

1.2.2 Requirements

In the following, we introduce five important properties each realization of the protocol, the principal discussed in the previous section, should provide. Additionally, we discuss the relevance of the properties. Next, we describe the five properties in alphabetic order. Their ordering does not rank the importance of each individual requirement.

Automation After the property and the analysis configuration are determined, both, the producer process (analysis plus program expansion) and the consumer validation, run fully automatic.

Efficiency The simple, consumer validation is more efficient than the resource consuming analysis of the producer, i.e., the consumer validation consumes less resources than the producer verification. We are especially interested in the resources execution time and memory.

Generality Each protocol instance should be usable to assure various properties. Hence, the instance's process must be configurable to an analysis type and domain which is appropriate to prove the validity of the property.

Relative completeness When the producer's resource consuming analysis generates a valid proof, i.e., a proof that shows that the input program adheres to the given property, then the simple, consumer validation must accept the enhanced program constructed by the producer's expansion step.

Soundness If the simple, consumer validation succeeds, the program that the consumer received will be correct w.r.t. the property that was checked during validation. In Proof-Carrying Code this property is often known as being tamper-proof.

Soundness and *relative completeness* protect the consumer's and producer's interests. The consumer wants that the received program does not violate the safety property. The soundness property guarantees that accepted programs fulfill this property. In contrast, the producer is interested in selling his programs, i.e., the consumer should accept the producer's programs. Especially, if the producer is reliable and does not cheat, i.e., he developed a program that adheres to the property of interest and even proved the program w.r.t. that property, the consumer should accept the producer's program. Relative completeness ensures that well-behaving producers are not discriminated. Thus, these two properties form the basis for the participation of consumers and producers in the process. Additionally, they can be seen as a contract between the consumer and the producer, in which the consumer affirms that he buys the producer's program when the producer adds evidence that the program adheres to the desired property. Hence, they establish trust between consumer and producer.

Efficiency ensures that applying the protocol (instance) is profitable for the consumer. Each approach which realizes the protocol and in which the consumer validation is less or equally efficient than the producer verification becomes unnecessary. In these cases, it is simpler for the consumer to verify the program himself. Moreover, we plan to design approaches for the application in the on-the-fly scenario. In this application scenario, often a software solution is composed on demand. Validating the correctness of the individual components in the composition must be fast and should not take hours or days.

We think that it is common understanding that the consumer, e.g., an end-user, is not an expert in formal methods like verification. Furthermore, we observed that more and more software is developed by individuals or smaller companies instead of large, global players. It cannot be assumed that every producer has expert knowledge in formal methods. To keep an market open to these developers, the protocol instance should be applicable to producers and consumers who are non-experts in formal methods. *Automation* and *generality* aim at making an instance applicable to non-experts. On the one hand, we cannot expect from a non-expert that he guides the analysis or validation. To be applicable for non-experts, analysis and validation must be automatic. On the other hand, non-experts likely lack an overview on existing approaches or the knowledge which approach works best for a specific property and program. For non-experts, it is much easier to use a single approach for all properties. At best, they also should not be bothered with a correct configuration of that approach. Consumers do not need to configure

their validation. The validation configuration is derived from the producer's configuration. For producers, we imagine that they can select the configuration required to analyze a property from an existing set of configurations that fit to that particular property.

1.2.3 Discussed Instantiations

The protocol displayed in Fig. 1.1 allows various design options. For example, different approaches for the resource consuming analysis, the program expansion, and the simple validation can be chosen. However, the selected approaches must be compatible. The proof produced by the resource consuming analysis must fit to the proof format expected by the program expansion instance. Moreover, the output of the program expansion must match the enhanced program format required by the simple validation approach. In this thesis, we propose two instantiations of the protocol, the principle shown in Fig. 1.1: configurable program certification and Programs from Proofs. Furthermore, we discuss how to integrate these two instantiations into a third instantiation.

All three instantiations use the same resource consuming analysis for the producer. This analysis computes an overapproximation of the reachable program states on an abstract level and produces a model of this overapproximation, named abstract reachability graph (ARG). The precision of the overapproximation can be configured by the choice of the abstract domain and the type of the analysis.

Configurable program certification uses the ARG in its program expansion to build a certificate, a witness for program correctness. The certificate contains parts of the proof, the ARG, and is explicitly attached to the program. The combination of the program and the certificate is the enhanced program. The simple consumer validation checks whether the certificate is a valid witness for program correctness w.r.t. the property and the received program.

In contrast, in the *Programs from Proofs* approach the information for a fast and simple validation is implicitly encoded in the program. More concretely, the program expansion uses the ARG to restructure the program s.t. the property can be verified by a simpler, less precise analysis. For example, it excludes infeasible paths or it refines the syntactical partitioning of the program paths, i.e., it syntactically separates some paths. This restructured program is the enhanced program in Fig. 1.1. Given the enhanced program, the simple consumer validation applies the same analysis approach as the producer, but he starts it with the enhanced, i.e., restructured, program and a less precise analysis.

The program expansion in the integration of the two previous instances is a mixture of implicit information encoding and explicit information attachment. Like in the Programs from Proofs approach, the program is restructured. Simultaneously, a certificate in the style of the configurable program certification approach is constructed, which witnesses the correctness of the restructured program. Thus, this certificate and the restructured program become the enhanced program. Now, the simple validation of the consumer utilizes the certificate validation from configurable program certification to validate the attached certificate on the restructured program.

1.3 Thesis Contribution

In this thesis, we design techniques to assure that software programs offered in on-the-fly markets meet their correctness properties. Based on the insights of the previous section, our techniques must be automatic and flexible PCC alike approaches in which property

and analysis (type) can be set. To achieve this goal, all our techniques apply the abstract protocol (Fig. 1.1) proposed in the previous section. More concretely, we develop concrete techniques for the three protocol instances, configurable program certification, Programs from Proofs, and the integration of configurable program certification and Programs from Proofs. All three instances have already been briefly sketched in the former section.

For configurable program certification we develop six variants. We begin with a direct and basic approach. Starting from the basic approach, we consider two, orthogonal lines of optimizations, which we later also combine. The first line of optimizations aims at the reduction of the information attached to the program. We propose two instances, one is generally applicable and the other has a higher reduction. However, the higher reduction comes at the costs that it is only applicable to a smaller, although practically often used class of analyses. In the second optimization, we partition the additional information in such a way that parts can be validated independently of others. This second optimization can be combined with both instances of the first optimization. In contrast to configurable program certification, we consider one efficient technique which realizes the Programs from Proofs approach. The integration of the two approaches is also configurable. Any variant of the configurable program certification can be used in the integration. Furthermore, two different approaches exist for the integration. Both approaches use the same technique for the Programs from Proofs related part, but differ in the construction of the certificate for the restructured program. One transforms the proof generated by the analysis into a proof for the restructured program and applies standard configurable program certification techniques. The other first generates the certificate for the original program and then transforms the certificate into a certificate for the restructured program.

Next to the development of the various techniques, we study their properties. In particular, we look at the five properties, automation, efficiency, generality, relative completeness, and soundness, which every instance of the abstract protocol in Fig. 1.1 should fulfill.

For each of the techniques which we present, we prove that it is sound and relatively complete. For the Programs from Proofs technique we must split the proof of relative completeness into multiple proofs for various classes of analyses. However, relative completeness could only be shown for a large class of standard analyses. Based on the insights of relative completeness and the technique's realization of the abstract protocol from Fig. 1.1, we discuss if or when the technique is fully automatic. Our configurable program certification techniques are only fully automatic for a large class of standard analyses.

To show efficiency and generality of our techniques, we performed an extensive evaluation. All our techniques were performed on a large set of programs mainly taken from well-known software benchmarks. We considered five different analysis types and at least 18 different analyses per technique. During evaluation, we assured various properties. To mention only a few, we checked that no uninitialized variables are used, no assertions are violated, all array accesses are in the bounds, or a program calls external methods in a particular order. Based on the evaluation results, we identified the factors that make our techniques efficient. Furthermore, we used the evaluation to compare the techniques among each other and to compare them with existing state-of-the-art techniques.

1.4 Thesis Outline

The rest of this thesis is structured as follows. In Chapter 2, we introduce the basic concepts like program (model) and its semantics, formalization of properties and correctness of

programs w.r.t. these properties. Moreover, we present the producer's resource consuming analysis, which is built on a configurable, abstract interpretation alike analysis framework. Subsequently, Chapters 3 and 4 describe the configurable program certification instance. We start with a very basic realization in Chapter 3 and discuss optimizations of the basic approach in Chapter 4. Furthermore, we show either with the help of formal proofs or via experimental evaluation that all configurable program certification variants (partially) fulfill the five described properties. Additionally, we compare the configurable program certification instance with existing certification approaches based on witness validation. Thereafter, we present the second instance Programs from Proofs in Chapter 5. Like for the first instantiation, we prove that or experimentally evaluate if and when the five requirements are fulfilled by the Programs from Proofs instance. We also compare the Programs from Proofs instantiation with other approaches including our previous instance configurable program certification. After we considered these two instantiations, in Chapter 6 we explain how to integrate them into a third instance. As before, we use formal proofs and experiments to evaluate the combined instance against the five requirements from above. In the last chapter of our thesis, we conclude this thesis with a comparison of the three instances and suggest future improvements and extensions of our protocol instantiations.

2 Programs and Their Verification

In our approaches, we want to check if a program is correct with respect to some property. Before we can check if a program is correct with respect to some property, we need to understand what this means. Understanding program correctness with respect to a property incorporates at least the comprehension of the following three aspects:

1. the concept of a program, including its representation and semantics, (Section 2.1),
2. the notion of a property (Section 2.2.1), and
3. the meaning of correctness (Section 2.2.2).

As described in the previous chapter, in our approaches the producer and the consumer check if a program is correct with respect to some property. We develop the consumer checking in the subsequent chapters, but for the producer checking we build on existing verification techniques. To apply our approaches in a broad context, we need an adaptable producer verification. Our solution for adaptation lets the producer configure his verification (see Section 2.3) – of course only within a certain design space. For producer checking, the configured verification is executed by a generic analysis (cf. Section 2.4), which also constructs the proof, an abstract reachability graph (see e.g. [BHJM07]) required by the subsequent step of our approaches.

2.1 Programs

Any written definition of a program, e.g., program code, is first of all a sequence of characters. Statements, keywords, variables and operators are all subsequences of characters. However, a sequence of characters is a rather impractical representation of programs. It does not provide any additional information about the structure of a program nor about the expected behavior of the program, which is important for verification. To guarantee more flexibility in the configuration of the producer verification, we use a program model

to describe the structural aspect of a program. On top of the program model, we then define a program's behavior, the program semantics.

2.1.1 Program Model

Various representations for programs exist. All of them were developed for different purposes. To name only a few, abstract syntax trees (ASTs) (see e.g. [ALSU07]) are a hierarchical representation of a program structure typically used in compilers. The AST structure is mainly imposed by the grammar of a programming language. Program dependence graphs [FOW84] provide the control and data dependencies between operations. Control flow graphs [All70] describe the syntactical paths of a program.

To verify program correctness with respect to some property, we want to apply some sort of operational reasoning, a common procedure in verification [KRA09, p. 12]. For operational reasoning, we consider which operations are executed in which order. Hence, we require a representation of the syntactical program paths, which also describes the program flow. Program flows are often represented by graph like structures named e.g. flowcharts [CC77], flow graphs [KU77], control flow graphs [All70], and control flow automaton [BHT07]. Operations or operation sequences are associated with nodes or edges. We decided to follow the notion of the verification framework on which we build the producer verification [BHT07] and use control flow automata to describe programs.

For a broad applicability of our approaches, we do not fix a concrete programming language. In contrast, we want to allow that arbitrary programs[1] can be described by our program model. From now on, let Ops be the set of all program instructions, all operations a program may use. Moreover, let $\mathcal{L}$ be the set of all program locations, e.g., all possible values of the program counter. We define the set $\mathcal{G}$ of all control flow edges to be $\mathcal{G} := \mathcal{L} \times Ops \times \mathcal{L}$. The set $\mathcal{G}$ specifies all statements any program may execute. To describe a program, a *control flow automaton* provides the program's locations as well as its statements, which also model the control flow between the program's location. Additionally, the beginning of a program is defined by a special program location.

Definition 2.1 (Control Flow Automaton (CFA))**.** A *control flow automaton (CFA) modeling program* $P = (L, G_{\text{CFA}}, l_0)$ consists of a set $L \subseteq \mathcal{L}$ of program locations, a set $G_{\text{CFA}} \subseteq L \times Ops \times L$ of control flow edges, and a program entry location $l_0 \in L$.

Figure 2.1 shows our example program `SubMinSumDiv` in pseudocode plus its CFA. We use program `SubMinSumDiv` throughout the thesis to demonstrate all our approaches. Thus, the example program is rather artificial and it intermixes different computations, which will be separated by the Programs from Proofs approach. Depending on the values of variables x and y, program `SubMinSumDiv` subtracts the minimum of x and y from 10 ($x < 0$), sums up the integer values from x to y ($x \geq 0 \wedge y \geq 0$), or when $x \geq 0 \wedge y < 0$ it computes the negated quotient of the euclidean division, which is increased by two if y is not a divisor of x. The following equation shortly summarizes this behavior.

$$z = \begin{cases} -min(x, y) + 10 & \text{if } x < 0 \\ \sum_{i=y}^{x} i & \text{if } x \geq 0 \wedge y \geq 0 \\ -x \text{ div } y & \text{if } x \geq 0 \wedge y < 0 \wedge \frac{x}{y} = \lceil \frac{x}{y} \rceil \\ (-x \text{ div } y) + 2 & \text{else} \end{cases}$$

[1] Note that when we think of a program, we have in mind a program that is written in some imperative language.

```
0 : z:=0;
1 : if(x<0)
2 :   if y<x
3 :     z:=-y;
      else
4 :     z:=-x;
5 :   z:=z+10;
    else
6 :   if y≥0
7 :     s:=1;
      else
8 :     s:=-y;
9 :   while x ≥ y ∧ x ≠ 0
10 :    if y ≥ 0
11 :      z:=z+x;
        else
12 :      z:=z+1;
13 :    x:=x-s;
14 :
```

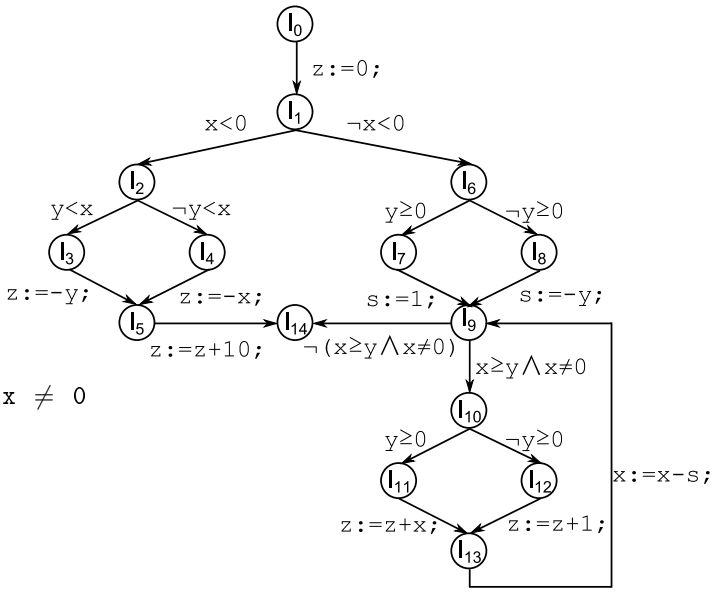

Figure 2.1: Example program `SubMinSumDiv` and its CFA

The CFA of example program `SubMinSumDiv` on the right of Fig. 2.1 provides for every program counter value i – the numbers in front of the pseudocode statements – a control flow location l_i. For each assignment, an edge, which is labeled by that assignment, is contained in `SubMinSumDiv`'s CFA. Furthermore, for each if and while statement, two assume edges are included in `SubMinSumDiv`'s CFA, namely one edge for each evaluation of the condition. The labels of those assume edges describe which evaluation of the condition is true. Hence, one edge is labeled by the condition and the other edge is labeled by the negated condition. Edges start in the control flow location corresponding to the statement's program counter value. They end in the control flow location corresponding to the program counter value of the next statement to be executed.

So far, we introduced the basic program model. For termination of our approaches, this basic model is too general. The main problem – for the consumer checking as well as for the presented producer's verification – is that a program may have infinitely many control flow edges, i.e., the CFA definition is infinite. Typically, the set of program locations is finite when the set of control flow edges is finite[2]. Furthermore, we do not necessarily need a finite set of program locations. Hence, we define a finite CFA based on its set of control flow edges.

Definition 2.2 (Finite CFA)**.** A *control flow automaton* $P = (L, G_{\text{CFA}}, l_0)$ *is finite* if its set of control flow edges is finite, i.e., $\exists n \in \mathbb{N} : |G_{\text{CFA}}| \leq n$.

Next to infinite programs, our basic program model also allows nondeterministic CFAs. For example, given a program location l and an operation op the successor for l and op in a CFA may be ambiguous. Many well-known programming languages like C [ISO11] do not support such nondeterministic specifications. To retranslate a CFA into a programming

[2]Commonly, program locations are the predecessors and successors of control flow edges.

language representation, e.g., like our Programs from Proofs approach does, we need to assure that the CFA is deterministic when the programming language itself does not support nondeterminism.

Definition 2.3 (Deterministic CFA)**.** A *control flow automaton* $P = (L, G_{\text{CFA}}, l_0)$ *is deterministic* if $\forall (l, op, l'), (l, op, l'') \in G_{\text{CFA}} : l' = l''$.

Up to now, we only introduced the program representation, the program syntax. We continue with a program's meaning, its semantics, which describes the behavior of the program.

2.1.2 Program Semantics

The first approaches [McC63, Lan64, Flo67, Hoa69] that formally defined program semantics appeared in the 1960s. Three different types of semantics were proposed.

Denotational semantics was introduced by McCarthy [McC63]. It describes the meaning of a program by a mathematical function that maps input states, the states on which the program is started, to output states, the states after the execution of the program.

Axiomatic semantics was used first by Floyd [Flo67] on flowcharts and two years later it was described Hoare [Hoa69] for programs. It defines the program semantics by pre- and postconditions. If a state s fulfills the precondition before the execution of program P, the state resulting from the execution of program P on state s will fulfill the postcondition. To derive pre- and postconditions, axioms and derivation rules for the different constructs are provided.

Operational semantics determines the meaning of a program by executing it on a virtual machine. The executions' sequences of transition steps determine the program semantics. As an early approach, Landin [Lan64] presented the mechanical, stack-based evaluation of lambda expressions. Later, Plotkin [Plo81] introduced the structural operational semantics for programs.

With our approaches, we want to assure a safe execution of a program, not only a correct result. Denotational semantics is not well-suited for this purpose because it does not provide any information about the program states during execution. Furthermore, in axiomatic semantics many pairs of pre- and postcondition can describe the behavior of a program and it is not clear which pair we should take as basis for our verification. Thus, we use a structural operational semantics for our programs, which defines the execution steps for statements $g \in \mathcal{G}$.

In the previous section, we did not fix the set Ops of statements a program may execute. Hence, we do not provide an exact semantics, but only present those requirements on the semantics which our approaches assume.

Before we consider the semantics of concrete programs, we look at the semantics of single execution steps in arbitrary programs. The semantics for execution steps defines whether in a concrete (program) state a certain program statement, defined by a control flow edge, is enabled and if it is enabled what the resulting state after execution of that program statement will be. We assume that the definition of the semantics for execution steps is given by a labeled transition system $T = (C, \mathcal{G}, \rightarrow)$ on a set C of concrete states with transitions labeled by control flow edges. The set of concrete states contains all

states any program can be in. Furthermore, the transition relation is of the following form $\rightarrow \subseteq C \times \mathcal{G} \times C$. Note that we use $c \xrightarrow{g} c'$ as abbreviation for $(c, g, c) \in \rightarrow$. These assumptions on the program semantics are sufficient for our certification approaches. However, for our Programs from Proofs approach we require further assumptions on the representation of concrete states and the execution of program statements defined by the transition relation. These additional assumptions are presented in the following two paragraphs.

Assumptions on Concrete States: A concrete state can be split into two disjoint parts, the *control state* and the *data state*. The control state contains all information necessary to determine the control flow edge considered next in a program's execution. We assume that the value of the program counter – a location $l \in \mathcal{L}$ – is sufficient for this task. This means, function calls are either inlined or handled as a single operation. In the latter case, our analysis must determine the effect of a function, e.g., by using function summaries first used by Hoare [Hoa71] or other techniques like block abstraction memoization [WW12]. The data state provides all information about data that is manipulated by the execution of an operation $op \in Ops$, e.g., the values of variables. We denote the set of all data states by DS and demand that the set C of concrete states is $C = \mathcal{L} \times DS$. For a concrete state (l, d) we define $\mathrm{cs}((l, d)) = l$ and $\mathrm{ds}((l, d)) = d$ to access the state's control and data state, respectively.

Assumptions on the Transition Relation: We require that the transition relation respects the control flow, i.e., $c \xrightarrow{(l,op,l')} c'$ only if $\mathrm{cs}(c) = l$ and $\mathrm{cs}(c') = l'$. Additionally, a successor's data state is defined by the operation and the predecessor's data state only. We demand that there exists a partial function[3] $\mathrm{succ} : DS \times Ops \rightarrow DS$ that defines the successor's data state. Furthermore, we assume that the transition relation is determined by the control flow edges and the partial function succ. These assumptions give us the following transition relation:

$$\rightarrow := \{(c, (l, op, l'), c') \in C \times \mathcal{G} \times C \mid \mathrm{cs}(c) = l \wedge \mathrm{cs}(c') = l' \wedge \mathrm{ds}(c') = \mathrm{succ}(\mathrm{ds}(c), op)\} \ .$$

Note that these assumptions exclude nondeterministic program behavior, but they do not exclude typical structural operational semantics for deterministic programs like the semantics assumed for abstract interpretation [CC77] or semantics which are presented in standard textbooks like [NNH05, pp. 54ff], [KRA09, pp. 58ff].[4] We mainly provided these assumptions to exclude exotic semantics in which statements can be executed although they are not the next statement in the program description or the evaluation of an expression depends not only on the data but also on the program location.

Next, we describe one semantics for our example program. The data states are all mappings from the set of variables to integers $\mathbb{Z}$. To evaluate boolean and arithmetic expressions $expr$ in such a data state d, denoted by $d(expr)$, first the variables are substituted with their respective integer value in the data state and then the expression is evaluated by standard mathematics (see [NNH05, p. 55]). For an assignment $v := expr$, the partial function $\mathrm{succ} : DS \times Ops \rightarrow DS$ returns for a data state d, the data state $d[v \mapsto d(expr)]$ obtained from d by replacing the value of v by value $d(expr)$ (see [NNH05, p. 56]). Furthermore, for an assume instruction $bexpr$ the partial function $\mathrm{succ} : DS \times Ops \rightarrow DS$ is defined only if $d(bexpr) = true$, and then $\mathrm{succ}(d, bexpr) = d$.

[3] For example, the function may be undefined if the operation is a condition and the input data state does not fulfill the condition.

[4] Note that the representations for the control state vary throughout the presentations of structural operational semantics.

So far, the transitions $c \xrightarrow{g} c'$ of a transition system T define the valid execution steps, but often a program execution consists of multiple, sequential execution steps. Nevertheless, not all sequences of transitions should be proper executions. For example, consider a sequence $c \xrightarrow{g} c', c'' \xrightarrow{g'} c'''$ with $c' \neq c''$. Allowing such a sequence would impose a weird semantics in which a state change may be triggered without actually executing any program statement. We exclude such behavior and restrict the set of proper executions, the set which describes all executions any program may take, to the set of all paths in the transition system T.

Definition 2.4. Let $T = (C, \mathcal{G}, \rightarrow)$ be a transition system. Every concrete state $c \in C$ is a *path of length* 0 in transition system T. A sequence of transitions $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ is a *path of length* n in transition system T if $\forall 1 \leq i \leq n : (c_{i-1}, g_i, c_i) \in \rightarrow$.

Typically, a program should not be able to execute every path in a transition system. Otherwise, we do not need to define programs at all. A program's control locations and control flow edges already provide a convenient way to restrict the paths a program can execute. Hence, we define the executable paths of a program P, its *program paths*, to be all paths in the transition system T that start in a control location of P and that only use the control flow edges defined by P. Since a program usually starts its execution at the program entry location rather than in the middle of the program, not all program paths are of interest. In our approaches, we specify in which concrete states, the so called initial states, a program execution may start. In the following, we are only interested in program paths starting in one of the given initial states.

Definition 2.5 (Program Paths)**.** Let $P = (L, G_{\text{CFA}}, l_0)$ be a program. A path $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ in transition system T is a *program path in* P if $\text{cs}(c_0) \in L$ and $\forall 1 \leq i \leq n : g_i \in G_{\text{CFA}}$. We extend this definition and define the set $paths_P(I)$ of *program paths in* P *starting in a set of states* $I \subseteq C$ to be the set of all program paths $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_m} c_m$ in P with $c_0 \in I$.

After we comprehend the notion of a program, we continue with the specification of properties and the meaning of program correctness with respect to a specified property.

2.2 Properties

To express program properties, different formalisms are available. A very simple form is the direct encoding of properties in the program, e.g., with assertions or specific error labels. Pre- and postconditions proposed by Hoare [Hoa69] can be used to describe the expected relation between the input and output (state) of a program. Temporal logics, like e.g., linear temporal logic (LTL) or computation tree logic (CTL) (see e.g. [CGP02]), specify the allowed temporal ordering of events in program executions. Events may be program operations or atomic propositions on program states. Instead of a temporal logic, also finite state machine based approaches are used to specify (temporal) safety properties. Safety properties "state that something [bad] will not happen" [Lam77]. For example, Schneider [Sch00] uses security automata to define safety properties. The specification language for interface checking (SLIC) [BR02b] uses state machines to describe requirements on the API usage like ordering of method calls. The BLAST query language [BCH+04] provides observer automata to decide whether program executions adhere to certain safety properties. Correctness witness automata [BDDH16] are observer automata for invariant properties.

For our Programs from Proofs approach, we must be able to apply the same property specification on the transformed program. A separation of program and property specification simplifies this task. Furthermore, we already stated that we are interested in program executions. Since the verification of safety and liveness properties requires a different argumentation [AS87], we decided to focus on temporal safety properties. We think that in our scenario it is more important that a program execution is not harmful. Next, we discuss how we describe our safety properties. Then, we explain their semantics.

2.2.1 Property Specification

The most general form to specify a safety property for programs is to describe all safe transition sequences, or dually to define all unsafe transition sequences. We believe that an easily accessible format to specify the (un)safe transition sequences are automata like specifications as supported by SLAM [BR02a] or BLAST [BHJM07]. Hence, we use an automaton based specification which is inspired by the specification language of the analysis tool CPAchecker [BK11b], in which we integrated our approaches.

Our specification should support protocol properties, which define the ordering of program instructions. Furthermore, we also want to express that certain program locations cannot be reached. Similar to an assertion, we also want to assert that if a certain program location is reached, the program state will fulfill some condition. Moreover, our automaton should be able to specify invariants [MP95] on program states. Thus, our specification must be able to monitor operations and program states. Since we are rarely interested that at some point only a single program state can be reached, the input alphabet of our specification considers pairs of operations and sets of (concrete) program states.

In contrast to security automata [Sch00], which describe unsafe behavior by missing transitions, we model unsafe behavior with a special error state. We think this is more intuitive. Furthermore, we require that our transition relation is complete, i.e., it covers the behavior of any execution step. Thus, we avoid that non-defined transitions are misinterpreted, e.g., when our specification is translated in the input language of a tool.[5]

To simplify our proposed verification of programs, we also require that the transition relation is deterministic. Like a nondeterministic finite state machine is as powerful as a deterministic finite state machine [RS59], we believe that our property specification does not become more powerful when we allow nondeterminism. Nevertheless, we left the proof of this claim for future work (see Section 7.2).

The previous considerations let us define a *property automaton*, our formalism to specify properties.

Definition 2.6 (Property Automaton)**.** A *property automaton* $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ consists of a set Q of automaton states, a transition relation $\delta \subseteq Q \times Ops \times 2^C \times Q$, the initial state $q_0 \in Q$, and the error state $q_{\text{err}} \in Q$. In addition, a property automaton must fulfill the following conditions:

- The transition relation is complete, i.e.,

$$\forall q \in Q, c \in C, op \in Ops : \exists (q, op, C_{\text{sub}}, \cdot) \in \delta : c \in C_{\text{sub}} \ .$$

- The transition relation is deterministic, i.e.,

$$\forall (q, op, C_1, q'), (q, op, C_2, q'') \in \delta : C_1 \cap C_2 = \emptyset \vee q' = q'' \ .$$

[5] In contrast to security automata, for the tool CPAchecker a missing transition means no state change.

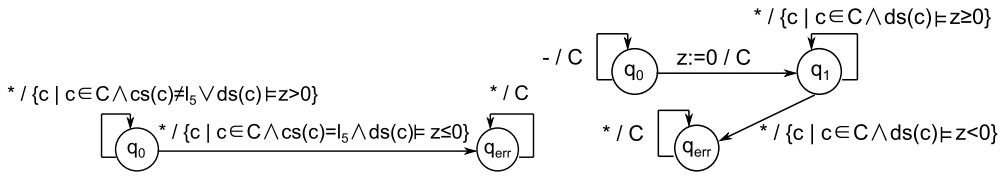

Figure 2.2: Property automata describing properties `pos@l5` and `nonneg`

Figure 2.2 shows two example property automata. The edges are labeled according to the following schema op/C_{sub}, where op is the observable operation and C_{sub} is the set of the accepted concrete states. For the sake of readability, we do not add edges for all operations but use wildcards $*$ and $-$. Wildcard $*$ means any operation op, e.g., the edge $q_{\text{err}} \xrightarrow{*/C} q_{\text{err}}$ represents all edges $(q_{\text{err}}, op, C, q_{\text{err}})$ with $op \in Ops$. Furthermore, we use $-$ to denote all operations which do not occur on an outgoing transition of a state. In our example on the right of Fig. 2.2, $(q_0 \xrightarrow{-/C} q_0)$ represent all edges (q_0, op, C, q_0) with $op \in Ops \setminus \{z := 0\}$ and $q_{\text{err}} \xrightarrow{-/C} q_{\text{err}}$ is equivalent to $q_{\text{err}} \xrightarrow{*/C} q_{\text{err}}$. The left automaton describes the property `pos@l5` which states that if location l_5 is reached, variable z will have a positive value. The right automaton defines the property `nonneg`, which demands that after proper initialization variable z always has a non-negative value.

2.2.2 Correctness Criterion

In the previous section, we introduced our formalism for property specifications. Next, we define its semantics, i.e., what it means that a program is correct with respect to those property specifications. The idea is that the property automaton works like a monitor, which is run in parallel with the program. Then, a program will be correct w.r.t. a property automaton if all its executions never visit the error state q_{err}. Hence, we need to describe how a program path triggers the transitions of the property automaton.

Each execution step $c \xrightarrow{(l,op,l')} c'$ triggers a single transition. We only need to decide whether the property automaton considers c or c'. Since a user, as well as our analysis technique, can control the properties of the input state, we decided that the property automaton monitors successor states. Thus, desired properties can be ensured for all states in a program execution. Now, an execution step $c \xrightarrow{(l,op,l')} c'$ triggers those outgoing transitions $(q, op', C_{\text{sub}}, q')$ of a state q with $op = op'$ and $c' \in C_{\text{sub}}$.

Based on these considerations, we define a *configuration sequence*, a link between the program execution and the property automaton. Mainly, the pairs of program instruction op and successor state of the program execution define the input word of the property automaton. Then, a configuration sequence is simply the product of a program execution path and a run in the property automaton considering the input word determined by the program execution.

Definition 2.7 (Configuration Sequence for Path)**.** Let P be a program, $p \in path_P(C)$ a path, and $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ a property automaton. A sequence $(c_0, q_0) \dots (c_n, q_n)$ of pairs $(c_i, q_i) \in C \times Q$ is a *configuration sequence for p and $\mathcal{A}$* if $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ and there exists a run $r := q_0 \xrightarrow{op_1,C_1} q_1 \dots \xrightarrow{op_n,C_n} q_n$ s.t. $\forall 1 \leq i \leq n : g_i = (\cdot, op_i, \cdot) \wedge c_i \in C_i$.

In the end, we want to use configuration sequences to define when a program is correct

with respect to a property automaton. To ensure that our concept of program safety is well-defined, we show that for each path at least one configuration sequence exists.

Corollary 2.1. *Let P be a program and $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ a property automaton. For every path $p \in path_P(C)$, at least one configuration sequence for p and $\mathcal{A}$ exists.*

Proof. Show by induction that for every path of length i a configuration sequence exists.

Basis Let c_0 be an arbitrary path of length 0. By definition (c_0, q_0) is a configuration sequence.

Step Let $c_0 \xrightarrow{(l_1, op_1, l_1')} c_1 \dots c_{i-1} \xrightarrow{(l_i, op_i, l_i')} c_i$ be an arbitrary path of length i. By induction, there exists a configuration sequence $(c_0, q_0), (c_1, q_1), \dots, (c_{i-1}, q_{i-1})$ for subpath $c_0 \xrightarrow{(l_1, op_1, l_1')} c_1 \dots c_{i-1}$ and hence a run $q_0 \xrightarrow{op_1, C^1\text{sub}} c_1 \dots \xrightarrow{op_{i-1}, C^{i-1}_{\text{sub}}} q_{i-1}$ s.t. $\forall 1 \leq j < i : c_j \in C^j_{\text{sub}}$. Since property automaton $\mathcal{A}$ is complete, there exists $(q_{i-1}, op_i, C^i_{\text{sub}}, q_i) \in \delta$ and $c_i \in C^i_{\text{sub}}$. Hence, $q_0 \xrightarrow{op_1, C^1\text{sub}} q_1 \dots \xrightarrow{op_{i-1}, C^{i-1}_{\text{sub}}} q_{i-1} \xrightarrow{op_i, C^i_{\text{sub}}} q_i$ is a run and by definition $(c_0, q_0), (c_1, q_1), \dots, (c_{i-1}, q_{i-1})(c_i, q_i)$ is a configuration sequence.

□

After we are sure that we can interlink any program path to the property automaton, we now define when a path does not violate the property specification, i.e., when it is safe. Like a nondeterministic automaton accepts an input sequence if a run for the input sequence exists that ends in a final state, a *program path is safe* if a configuration sequence for that path exists that does not visit the error state.

Definition 2.8 (Safe Program Path)**.** Let P be a program and $\mathcal{A}$ be a property automaton. A path $p \in paths_P(C)$ is safe w.r.t. $\mathcal{A}$ if a configuration sequence $(c_0, q_0) \dots (c_n, q_n)$ for p and $\mathcal{A}$ exists s.t. $\forall 0 \leq i \leq n : q_i \neq q_{\text{err}}$.

We understand when a single program path, a single program execution, is safe. Remembering that a program's semantics is defined by its program paths, we naturally derive *program safety*. Since we are only interested in program paths starting in certain initial states, a program is safe with respect to a set of initial states if all program paths starting in those initial states are safe. Considering the program entry location as conventional start of a program, we further define a program to be safe in general if all its executions starting in the program entry location are safe.

Definition 2.9 (Program Safety)**.** Let $P = (L, G_{\text{CFA}}, l_0)$ be a program and $\mathcal{A}$ a property automaton. *Program P is safe w.r.t. property automaton $\mathcal{A}$ and a set of initial states $I \subseteq C$*, denoted by $P \models_I \mathcal{A}$, if every path $p \in paths_P(I)$ is safe w.r.t. $\mathcal{A}$.
Program P is safe w.r.t. $\mathcal{A}$, denoted by $P \models \mathcal{A}$, if $P \models_{\{c | c \in C \wedge cs(c) = l_0\}} \mathcal{A}$.

Note that our example program `SubMinSumDiv` is safe w.r.t. properties `pos@l5` and `nonneg`.

So far, we know what it means that a program is correct, safe, with respect to a property automaton, but not how to actually prove this fact.We continue to explain how a producer verifies that a given program is correct w.r.t. a given property automaton. Next, we present how the producer describes an appropriate analysis for this verification task.

2.3 Analysis Configuration

We already stated that we need an adaptable verification for the producer. To achieve adaptability, we suggested to configure the verification. Another requirement is that our approaches should be available also for producers which are less experienced in verification. Thus, the producer verification must be fully automatic and must not interact with the producer after the verification process is started. Note that we assume that less experienced producers choose from an existing pool of configurations rather than configuring the verification on their own. In the following, we discuss existing configurable verification frameworks before we introduce our configurable framework of choice.

Model checking [JM09b] explores all execution paths of a program to check if a property is satisfied. In the early stages, model checking was defined by the program and the property specification only. For symbolic model checking [BCM+92], this is the same except that the variable ordering considered by the binary decision diagrams can be configured. First, the combination of (data) abstraction and model checking, see e.g. [CGL94], enables adjustable model checking.

Dataflow analyses (DFAs) have their origins in compiler optimization. In contrast to model checking, they are often path-insensitive and combine information for the same program location. Hence, they compute a so called dataflow fact per program location. Dataflow facts describe certain information about program states reachable at the respective location. The first framework for DFAs was suggested by Kildall [Kil73] in 1973. Further frameworks for DFA specification are for example the monotone dataflow analysis framework [KU77] and the IFDS/IDE framework [RHS95]. All these frameworks do not incorporate a semantic meaning of the computed dataflow facts.

Abstract interpretation [CC77] first of all allows to specify an abstract semantics of a program. An abstract semantics overapproximates the concrete semantics if certain conditions are met. In their initial paper, Cousot and Cousot [CC77] use abstract interpretation in a DFA fashion and compute one abstract context (abstract state) per program location.

Beyer et al. introduced a *configurable program analysis* framework [BHT07, BHT08], in which analyses joining information at same program locations as well as model checking based analyses can be specified. Additionally, it allows to directly specify lots of analyses whose precision is in between fully path-sensitive model checking and path-insensitve analyses like DFAs. In principle, a configurable program analysis describes an abstract interpreter plus operators that steer the state exploration.

As a consequence of the convergence between the static analysis and the model checking community, it was shown that model checking, DFAs, and abstract interpretation are equal [Ste91, CC95, Sch98]. Furthermore, techniques like trace partitioning [MR05, RM07] and widening [CC77] allow to specify abstract interpretations or DFAs that behave like the intermediate configurations in the configurable program analysis framework. Hence, we think that from a theoretical perspective all frameworks are equal.

We decided to build our techniques on top of the configurable program analysis framework [BHT07, BHT08]. Due to the built-in specification for the direction of the state space exploration, the specification of analyses is less complex and more comfortable. More importantly, this built-in specification provides us a direct representation of the paths explored during verification. This representation is essential for our Programs from Proofs approach. In the following, we start with the basic concept of a configurable program analysis.

2.3.1 Basic Analysis Configuration

For our producer verification, we utilize the concept of a configurable program analysis [BHT07, BHT08]. A configurable program analysis specifies a custom, abstract interpretation based analysis. On the one hand, the customization allows us to describe the abstraction. The abstraction is defined by an abstract domain and a transfer relation, which describes the abstract semantics of any program. On the other hand, the analysis type, e.g., dataflow analysis or model checking, is determined with the help of the merge operator and the termination check. These two operators steer the state space exploration, a reachability analysis. The merge operator decides if and how to combine an explored state with the already explored states. Dataflow analyses typically merge states with same locations and model checking never merges. The termination check tells the reachability analysis when the exploration of a state can be stopped.

In contrast to the original model of a configurable program analysis [BHT07], a configurable program analysis with precision adjustment [BHT08] also considers precisions. These precisions can be used to adapt the abstraction during program analysis. For this, a precision adjustment operator is provided and all operators of a configurable program analysis also have the precision as an additional input parameter. Nevertheless, Beyer et al. [BHT08] state that theoretically a configurable program analysis with precision adjustment is not more powerful.

For our approaches, we consider a mixture of the original configurable program analysis and the configurable program analysis with precision adjustment. From now on, we refer to this mixture when we use the term configurable program analysis (CPA). We use the precisions from the configurable program analysis with precision adjustment, but the only operator that considers precisions is the precision adjustment operator. Hence, we are able to comfortably specify CPAs that are used in context of lazy refinement [HJMS02]. However, we may keep our approaches simple, i.e., the consumer does not need to deal with precisions. Note that our restrictions are not very restrictive in practice because the analyses implemented in CPAchecker [BK11b], a software verification tool based on the configurable program analysis concept, typically ignore the precision in the transfer relation and the merge operator and do not use the precision in the termination check.

In the following, we formally introduce the definition of a configurable program analysis. Note that a CPA is defined for arbitrary programs and is tailored to concrete programs during its execution. Formally, a CPA $\mathbb{C}$ is a six tuple $\mathbb{C} = (D, \Pi, \rightsquigarrow, \mathsf{prec}, \mathsf{merge}, \mathsf{stop})$. Its six elements are described in the following.

Abstract Domain The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ provides an abstraction for a set C of concrete states. Subsets of concrete states are represented by single abstract states.

The abstract representation is defined by a join-semilattice [DP90, p. 82] $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ on the set E of abstract states with partial order $\sqsubseteq$, least upper bound or join $\sqcup$, a greatest element $\top \in E$, i.e., $\forall e \in E : e \sqsubseteq \top$, and a smallest element $\bot \in E$, i.e., $\forall e \in E : \bot \sqsubseteq e$. The partial order $\sqsubseteq$ describes if an abstract state is more abstract than another. We lift the partial order to set of abstract states and write $S \sqsubseteq S'$ if $S, S' \subseteq E$ and $\forall e \in S : \exists e' \in S' : e \sqsubseteq e'$.

To complete the abstraction, the concretization function $[\![\cdot]\!] : E \to 2^C$ describes for every abstract state $e \in E$ its concrete meaning, i.e., the concrete states it represents.

For our analysis, we need useful abstractions, which respect the subset relation of concrete states. The bottom element must represent the smallest subset of concrete

states, the empty set. The top element must cover the greatest subset, the set of all concrete states. A more abstract state substitutes more concrete states. Abstract domains that fulfill the following equations adhere to these requirements.

$$[\![\bot]\!] = \emptyset \tag{2.1a}$$

$$[\![\top]\!] = C \tag{2.1b}$$

$$\forall e, e' \in E : e \sqsubseteq e' \implies [\![e]\!] \subseteq [\![e']\!] \tag{2.1c}$$

Set of Precisions The set Π of precisions is a non-empty set that defines various abstraction levels for the abstract domain. A single precision defines an abstraction level and describes which abstract information is tracked.

Transfer Relation The transfer relation $\rightsquigarrow \subseteq E \times \mathcal{G} \times E$ describes the abstract semantics of our analysis. For any abstract state and any control flow edge of any program, it defines the abstract successors. If for an abstract state e and a control flow edge g no element (e, g, e') exists in the transfer relation, no abstract successor will exist. This can, e.g., be the case if a branch condition cannot be satisfied or e does not consider states with a control location defined by the predecessor of g. For soundness of our analysis, this may not be the case if a concrete behavior is possible. We require that the transfer relation overapproximates the concrete program behaviors with a finite number of abstract successors. If the semantics defines a concrete successor c' for a concrete state c, which is considered by abstract state e, and edge g, then an abstract successor e' for e and g must exist that considers c'.

$$\forall e \in E, g \in \mathcal{G} : \{c' \mid c \xrightarrow{g} c' \wedge c \in [\![e]\!]\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!] \tag{2.2}$$

$$\forall e \in E, g \in \mathcal{G} : \exists n \in \mathbb{N} : |\{e' \mid (e, g, e') \in \rightsquigarrow\}| = n \tag{2.3}$$

Precision Adjustment The precision adjustment is a total function $\mathsf{prec} : E \times \Pi \times 2^E \to E \times \Pi$. It enforces the analysis precision and adapts the analysis precision to its current needs. Hence, it widens an abstract state considering the current precision and probably the set of already explored states. Moreover, it computes a new precision, which may decrease or increase the analysis precision. If the analysis precision changes during analysis, a non-uniform abstraction will be computed. In our analysis, we use the precision adjustment to enforce the current precision on the currently explored state e and to compute the precision for the next exploration step, the exploration of the successors of e. For (relative) completeness of our approaches we require that the currently explored state is indeed widened.

$$\forall e, e_{\text{prec}} \in E, S \subseteq E, \pi, \pi_{\text{prec}} \in \Pi : \mathsf{prec}(e, \pi, S) = (e_{\text{prec}}, \pi_{\text{prec}}) \Rightarrow e \sqsubseteq e_{\text{prec}} \tag{2.4}$$

Merge Operator The merge operator is a total function $\mathsf{merge} : E \times E \to E$. It defines when and how to combine the information of two abstract states. It does not combine information when the result of merge is the same as the second input parameter. A combination always widens the second parameter, typically an already explored state of our analysis, incorporating partial information of the first parameter or totally

subsuming the first parameter. In our analysis, we use the merge operator to combine the information of the currently explored state with already explored states. The combination of two abstract states in an analysis often results in a less precise but faster analysis. To ensure the widening property and that the combination does not loose information already explored, we require that the result of the operator merge is at least as abstract as its second parameter.

$$\forall e, e' \in E : e' \sqsubseteq \mathsf{merge}(e, e') \tag{2.5}$$

Termination Check The termination check operator $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ is a total function that checks if an abstract state $e \in E$ is covered by a set of abstract states $S \in 2^E$. We use this operator in our analysis to decide when to stop the exploration of an abstract state e. Thus, typically S is the set of already explored states. To guarantee soundness of the analysis, we require that the termination check operator may only return true if the concrete states considered by abstract state e are already taken into account for exploration. All concrete states represented by the checked abstract state e are contained in the concretization of at least one abstract state in the coverage set S.

$$\forall e \in E, S \subseteq E : \mathsf{stop}(e, S) \implies [\![e]\!] \subseteq \bigcup_{e' \in S} [\![e']\!] \tag{2.6}$$

After we presented the concept of a configurable program analysis, we introduce three CPAs, which we use during the explanation of our approaches. To simplify our examples, none of these analyses changes its precision. For the specification of a CPA that adjusts its precision we refer to [BHT08].

The Location CPA $\mathbb{L}$: A location CPA $\mathbb{L} = (D_\mathbb{L}, \Pi_\mathbb{L}, \rightsquigarrow_\mathbb{L}, \mathsf{prec}_\mathbb{L}, \mathsf{merge}_\mathbb{L}, \mathsf{stop}_\mathbb{L})$ is an analysis that examines the syntactical reachability of program locations. The definition of our location CPA is based on a definition of Beyer et al. [BHT08].

The abstract domain $D_\mathbb{L} = (C, \mathcal{E}_\mathbb{L}, [\![\cdot]\!]_\mathbb{L})$ considers a flat lattice on the set $\mathcal{L}$ of all program locations plus top $\top_\mathbb{L}$ and bottom state $\bot_\mathbb{L}$.[6] The concretization function $[\![\cdot]\!]_\mathbb{L}$ maps abstract states $l \in \mathcal{L}$ to all concrete states with control state l, i.e., $[\![l]\!]_\mathbb{L} := \{c \mid c \in C \wedge \mathrm{cs}(c) = l\}$, and the top and bottom state to C and the empty set.

The set of precisions contains a single precision π_{static} defining the single abstraction level that includes all program locations. The transfer relation mimics the syntactical meaning of the control flow edges $\mathcal{G}$, i.e., $(e, g, e') \in \rightsquigarrow_\mathbb{L}$ if $g = (e, \cdot, e') \vee e = e' = \top_\mathbb{L}$. A location CPA never adjusts precisions, $\forall e \in E_\mathbb{L}, S \subseteq E_\mathbb{L}\pi \in \Pi_\mathbb{L} : \mathsf{prec}_\mathbb{L}(e, \pi, S) = (e, \pi)$, nor merges abstract states, $\forall e, e' \in E_\mathbb{L} : \mathsf{merge}_\mathbb{L}(e, e') = e'$. Moreover, it stops exploration if the same or a more abstract state is already explored, $\mathsf{stop}_\mathbb{L}(e, S) := \exists e' \in S : e \sqsubseteq_\mathbb{L} e'$.

The Sign CPA $\mathbb{S}$: A sign CPA $\mathbb{S} = (D_\mathbb{S}, \Pi_\mathbb{S}, \rightsquigarrow_\mathbb{S}, \mathsf{prec}_\mathbb{S}, \mathsf{merge}_\mathbb{S}, \mathsf{stop}_\mathbb{S})$ is an analysis which explores the reachable data states on a coarse level. Following Cousot et al. [CC79], it only investigates the signs of variable values. The lattice of sign values shown in Fig. 2.3 defines the different values considered by the sign CPA. The bottom state $\bot$ of the sign lattice represents no concrete variable value. Abstract value $+$, abstract value $-$, and abstract value 0 mean value greater than zero, less than zero, and zero, respectively.

[6] A flat lattice on a set S with a top $\top$ and bottom element $\bot$ considers a flat ordering of its elements, $\forall e, e' \in S : e \sqsubseteq e' \implies (e = \bot \vee e = e' \vee e' = \top)$.

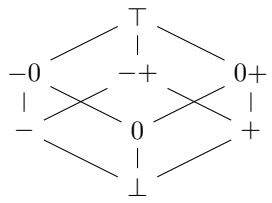

Figure 2.3: Lattice for sign abstract values

Abstract values $+0, -0$, and $+-$ are combinations of the previous abstract values. Their meaning is the union of the elements used for their combination. Finally, the top state $\top$ represents any variable value.

To investigate the signs of variable values, an abstract state of the sign abstract domain $D_{\mathbb{S}} = (C, \mathcal{E}_{\mathbb{S}}, [\![\cdot]\!]_{\mathbb{S}})$ assigns to every variable $v \in V$ in the set V of all variables an abstract value from the lattice of sign values, i.e., $E_{\mathbb{S}} = \{\bot, +, -, 0, +0, -0, +-, \top\}^{|V|}$. The top state $\top_{\mathbb{S}}$ assigns to every variable abstract sign value $\top$. Similarly, in the bottom state $\bot_{\mathbb{S}}$ every variable has abstract sign value $\bot$. The partial order $\sqsubseteq_{\mathbb{S}}$ compares two abstract states per variable, i.e., $(s_1 \dots s_{|V|}) \sqsubseteq_{\mathbb{S}} (s'_1 \dots s'_{|V|})$ if $\forall 1 \leq i \leq |V| : s_i \sqsubseteq s'_i$. The join operator $\sqcup_{\mathbb{S}}$ joins for every variable their abstract values of the two input states, i.e., $(s_1 \dots s_{|V|}) \sqcup_{\mathbb{S}} (s'_1 \dots s'_{|V|}) := (s_1 \sqcup s'_1 \dots s_{|V|} \sqcup s'_{|V|})$. The concretization function $[\![\cdot]\!]_{\mathbb{S}}$ maps an abstract state $(s_1 \dots s_{|V|})$ to all concrete states whose variable values adhere to the abstract sign values of the variables.

The set of precisions contains a single precision π_{static} defining the single abstraction level that includes all program variables. The transfer relation returns the smallest abstract element that covers all concrete successors, $(s, g, s') \in \rightsquigarrow_{\mathbb{S}}$ if $\{c' \mid \exists c \in [\![s]\!]_{\mathbb{S}} : c \xrightarrow{g} c'\} \subseteq [\![s']\!]_{\mathbb{S}}$ and $\forall s'' \sqsubseteq_{\mathbb{S}} s' : \{c' \mid \exists c \in [\![s]\!]_{\mathbb{S}} : c \xrightarrow{g} c'\} \not\subseteq [\![s'']\!]_{\mathbb{S}}$. A sign CPA never adjusts precisions, $\forall e \in E_{\mathbb{S}}, S \subseteq E_{\mathbb{S}}, \pi \in \Pi_{\mathbb{S}} : \mathsf{prec}_{\mathbb{S}}(e, \pi, S) = (e, \pi)$.

The sign CPA merges abstract states computing the join, $\forall e, e' \in E_{\mathbb{S}} : \mathsf{merge}_{\mathbb{S}}(e, e') = e \sqcup_{\mathbb{S}} e'$, and stops if the same or a more abstract state is already explored, $\mathsf{stop}_{\mathbb{S}}(e, S) := \exists e' \in S : e \sqsubseteq_{\mathbb{S}} e'$.

The Predicate CPA $\mathbb{P}$: A predicate CPA $\mathbb{P}_{\mathcal{P}} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{prec}_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}})$ examines the reachable data states based on a set of predicates $\mathcal{P}$. For demonstration, we consider predicate abstraction as suggested by Graf and Saidi [GS97]. In our experiments, we use predicate abstraction with adjustable block encoding [BKW10].

An abstract state of the predicate abstract domain $D_{\mathbb{P}} = (C, \mathcal{E}_{\mathbb{P}}, [\![\cdot]\!]_{\mathbb{P}})$ is a subset of all literals defined by the set $\mathcal{P}$ of predicates. For a set $p \subseteq \mathcal{P}$ of predicates, we define the set of all literals by $\text{lit}(p) := \{lit \mid lit \in p \vee lit = \neg pred \wedge pred \in p\}$. The top state $\top_{\mathbb{P}} := \emptyset$ contains no literal. Similarly, the bottom state $\bot_{\mathbb{P}} := \text{lit}(\mathcal{P})$ contains every literal. The partial order $\sqsubseteq_{\mathbb{P}}$ ensures that a less abstract state only adds more literals, $p \sqsubseteq_{\mathbb{P}} p'$ if $p \supseteq p'$. The join operator $\sqcup_{\mathbb{P}}$ computes the literals common to both states, namely the intersection of the two states, $p \sqcup_{\mathbb{P}} p' := p \cap p'$. The concretization function $[\![\cdot]\!]_{\mathbb{P}}$ maps an abstract state to all concrete states which satisfy all literals of the abstract state, $[\![p]\!]_{\mathbb{P}} = \{c \mid c \in C \wedge \forall lit \in p : c \models lit\}$.

The set of precisions contains a single element, the set of predicates $\mathcal{P}$, $\Pi_{\mathbb{P}} = \{\mathcal{P}\}$. The transfer relation only provides successors if concrete successors exist and determines the greatest set of literals enforced by the concrete successors, $(p, g, p') \in \rightsquigarrow_{\mathbb{P}}$ if

$\{c' \mid \exists c \in [\![p]\!] : c \xrightarrow{g} c'\} \neq \emptyset$, $\forall c' \in C : \exists c \in [\![p]\!] : c \xrightarrow{g} c' \implies \forall lit \in p' : c' \models lit$ and $\neg\exists lit' \in \text{lit}(\mathcal{P}) : lit' \notin p' \wedge [\![\{lit'\}]\!] \supseteq \{c' \mid \exists c \in [\![p]\!] : c \xrightarrow{g} c'\}$. The precision adjustment operator of a predicate CPA never adjusts the precision of an abstrac state, $\forall e \in E_{\mathbb{P}}, S \subseteq E_{\mathbb{P}}, \pi \in \Pi_{\mathbb{P}} : \mathsf{prec}_{\mathbb{P}}(e, \pi, S) = (e, \pi)$.

The predicate CPA never merges abstract states, $\forall e, e' \in E_{\mathbb{P}} : \mathsf{merge}_{\mathbb{P}}(e, e') = e'$, and stops the exploration of an abstract state if the same or a more abstract state is already explored, $\mathsf{stop}_{\mathbb{P}}(e, S) := \exists e' \in S : e \sqsubseteq_{\mathbb{P}} e'$.

Considered separately, the presented CPAs are very coarse and are hardly suited for producer verification. In the subsequent section, we explain how to combine these analyses to get more sophisticated analysis configurations, which may be used for the verification of a program's property.

2.3.2 Combination of Analysis Configurations

Combinations of analyses [CC79, LGC02, GT06] are proposed to obtain more precise information about a program compared to separate analyses. Both configurable program analysis frameworks [BHT07, BHT08] support the combination of analysis configurations. We require the combination of analysis configurations in the Programs from Proofs approach to build the producer analysis. Furthermore, a combination simplifies the construction of more sophisticated analyses. In the following, we explain how to compose two analysis configurations. Our composition mainly follows the composition proposed by Beyer et al. [BHT07, BHT08]. A composition of more than two configurations can be obtained recursively.

The combination of two CPAs $\mathbb{C}_2, \mathbb{C}_1$ results in a *composite CPA*, a CPA $\mathbb{C}_2 \times \mathbb{C}_1 = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{prec}_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$. The composite abstract domain $D_\times = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$ is the product of the components' domains $D_2 = (C, (E_2, \top_2, \bot_2, \sqsubseteq_2, \sqcup_2), [\![\cdot]\!]_2)$ and $D_1 = (C, (E_1, \top_1, \bot_1, \sqsubseteq_1, \sqcup_1), [\![\cdot]\!]_1)$. The join-semilattice $\mathcal{E}_\times = (E_\times, \top_\times, \bot_\times, \sqsubseteq_\times, \sqcup_\times)$ considers the cartesian product of two sets of abstract states, $E_\times = E_2 \times E_1$. Consequently, the top and bottom state are $\top_\times = (\top_2, \top_1)$ and $\bot_\times = (\bot_2, \bot_1)$. The partial order and the join operator are defined per-element basis, i.e., $(e_2, e_1) \sqsubseteq_\times (e_2', e_1')$ if $e_2 \sqsubseteq_2 e_2'$ and $e_1 \sqsubseteq_1 e_1'$, and $(e_2, e_1) \sqcup_\times (e_2', e_1') = (e_2 \sqcup_2 e_2', e_1 \sqcup_1 e_1')$. A combined abstract state (e_2, e_1) represents the set of all concrete states on which its component states e_2 and e_1 agree, i.e., $\forall (e_2, e_1) \in E_\times : [\![(e_2, e_1)]\!]_\times = [\![e_2]\!]_2 \cap [\![e_1]\!]_1$.

For our purpose, it is sufficient that a composite CPA simply adjusts its precision per element[7]. Hence, the set $\Pi_\times$ of composite precisions is the product of the components' precisions, $\Pi_\times = \Pi_2 \times \Pi_1$, and the precision adjustment operator delegates its tasks to the component CPAs $\mathbb{C}_2$ and $\mathbb{C}_1$, i.e., $\mathsf{prec}_\times((e_2, e_1), (\pi_2, \pi_1), S) = ((e^2_{\text{prec}}, e^1_{\text{prec}}), (\pi^2_{\text{prec}}, \pi^1_{\text{prec}}))$ if $\mathsf{prec}_2(e_2, \pi_2, \{e \mid (e, \cdot) \in S\}) = (e^2_{\text{prec}}, \pi^2_{\text{prec}})$ and $\mathsf{prec}_1(e_1, \pi_1, \{e \mid (\cdot, e) \in S\}) = (e^1_{\text{prec}}, \pi^1_{\text{prec}})$.

To profit from the combination, i.e., to obtain more precise analyses, and to allow flexible combinations, a composite CPA provides its own definition of the transfer relation and the merge operator. The transfer relation may consider the pure product combination or it uses a strengthening operator $\downarrow_{E,E'}$ which can improve the result $e \in E$ of a com-

[7] The software analysis tool CPAchecker sometimes uses information from the other component element to compute the adjustment of the component state, e.g., the predicate analysis and the value analysis adjust the state according to the program location. Note that in such a case you cannot use our composition of analyses but you must specify the composed CPA directly – of course you may use all the other composition concepts.

ponent's transfer relation based on the result $e' \in E'$ of the other component's transfer relation. A strengthening operator $\downarrow_{E,E'}: E \times E' \to E$ must not weaken its first parameter, i.e., $\forall e \in E, e' \in E : \downarrow(e, e') \sqsubseteq e$, and cannot be more precise than the composite state, $\forall e \in E, e' \in E' : [\![(e, e')]\!] \subseteq [\![\downarrow(e, e')]\!]$. To be able to configure the analysis technique of the composite analysis, the merge operator $\mathsf{merge}_\times$ must be defined.

Defining the termination check $\mathsf{stop}_\times$ also increases the flexibility of composite analyses. More importantly, an incautious combination of the components' termination check operators likely results in an unsound termination check. Consider the naïve combination of the two operators $\mathsf{stop}_\mathbb{L}$ and $\mathsf{stop}_\mathbb{S}$ that returns true if both operators return true, $\mathsf{stop}_\times((e_2, e_1), S) = \mathsf{stop}_\mathbb{L}(e_2, \{e \mid (e, \cdot) \in S\}) \wedge \mathsf{stop}_\mathbb{S}(e_1, \{e \mid (\cdot, e) \in S\})$. This combination is unsound. For example, $\mathsf{stop}_\times((\top_\mathbb{L}, \top_\mathbb{S}), \{(\bot_\mathbb{L}, \top_\mathbb{S}), (\top_\mathbb{L}, \bot_\mathbb{S})\})$ returns true, although $[\![\top_\mathbb{L}, \top_\mathbb{S}]\!]_\times = C \not\subseteq \emptyset = [\![(\bot_\mathbb{L}, \top_\mathbb{S})]\!]_\times \cup [\![(\top_\mathbb{L}, \bot_\mathbb{S})]\!]_\times$. The reason is that the composite concretization is more restrictive. The meaning of a component abstract state can be restricted by the other component abstract state. The composite termination check analysis must consider the composed abstract state as a whole. Thus, to get a sound composite analysis, we need to define the termination check $\mathsf{stop}_\times$. In practice, we often use the following termination checks $\mathsf{stop}_\times(e, S) := \exists e' \in S : e \sqsubseteq e'$ and $\mathsf{stop}_\times((e_2, e_1), S) := \exists (e_2', e_1') \in S : \mathsf{stop}_2(e_2, \{e_2'\}) \wedge \mathsf{stop}_1(e_1, \{e_1'\})$.

We demonstrate the combination of analysis configurations on two example combinations, which we use during the presentation of our approaches.

The Sign Dataflow Analysis – A Combination of Location and Sign CPA: To obtain a dataflow analysis, which computes sign data flow facts per program location, we need to configure the composite CPA $\mathbb{L} \times \mathbb{S}$ such that it joins abstract states at equal locations and stops exploration if at each program location the dataflow fact cannot be improved, i.e., a fixpoint is reached. This leads to the following definition of the composite CPA $\mathbb{L} \times \mathbb{S}$.

The composite CPA $\mathbb{L} \times \mathbb{S} = (D_{\mathbb{L}\times\mathbb{S}}, \Pi_{\mathbb{L}\times\mathbb{S}}, \rightsquigarrow_{\mathbb{L}\times\mathbb{S}}, \mathsf{prec}_{\mathbb{L}\times\mathbb{S}}, \mathsf{merge}_{\mathbb{L}\times\mathbb{S}}, \mathsf{stop}_{\mathbb{L}\times\mathbb{S}})$ uses the product transfer relation, i.e., $((l, s), g, (l', s')) \in \rightsquigarrow_{\mathbb{L}\times\mathbb{S}}$ if $(l, g, l') \in \rightsquigarrow_\mathbb{L}$ and $(s, g, s') \in \rightsquigarrow_\mathbb{S}$. Abstract states are combined if the location state is the same, $\mathsf{merge}_{\mathbb{L}\times\mathbb{S}}((l, s), (l', s')) := (l, s \sqcup_\mathbb{S} s')$ if $l = l'$ and $\mathsf{merge}_{\mathbb{L}\times\mathbb{S}}((l, s), (l', s')) := (l', s')$ otherwise. The exploration of an element is stopped, when the same or a more abstract state is already explored, $\mathsf{stop}_{\mathbb{L}\times\mathbb{S}}(e, S) := \exists e' \in S : e \sqsubseteq_{\mathbb{L}\times\mathbb{S}} e'$.

The Predicated Sign DFA – A Combination of Predicate and Sign DFA CPA: To increase the path-sensitivity of the sign DFA defined in the previous paragraph, we apply the concept of a predicated DFA [JW15]. We combine the predicate CPA with the sign DFA CPA such that information is only combined if program location and predicate state are equal. Hence, the predicated sign DFA CPA $\mathbb{P}_\mathcal{P} \times (\mathbb{L} \times \mathbb{S})$ is a composite CPA of predicate CPA $\mathbb{P}^\mathcal{P}$, considering a set of predicates $\mathcal{P}$, and the sign DFA CPA $\mathbb{L} \times \mathbb{S}$ with the following properties.

The composite CPA $\mathbb{P}_\mathcal{P} \times (\mathbb{L} \times \mathbb{S}) = (D_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}, \Pi_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}, \rightsquigarrow_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}, \mathsf{prec}_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}, \mathsf{merge}_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}, \mathsf{stop}_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})})$ uses the product transfer relation to compute successors, i.e., $((p, (l, s)), g, (p', (l', s'))) \in \rightsquigarrow_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}$ if $(p, g, p') \in \rightsquigarrow_\mathbb{P}$ and $((l, s), g, (l', s')) \in \rightsquigarrow_{\mathbb{L}\times\mathbb{S}}$. Abstract states are combined if the predicate state and the location state are the same, $\mathsf{merge}_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}((p, (l, s)), (p', (l', s'))) := (p, \mathsf{merge}_{\mathbb{L}\times\mathbb{S}}((l, s), (l', s')))$ if $p = p'$, and otherwise $\mathsf{merge}_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}((p, (l, s)), (p', (l', s'))) := (p', (l', s'))$. The predicated sign DFA stops the exploration of an abstract state, if the same or a more abstract state has been explored already, $\mathsf{stop}_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})}(e, S) := \exists e' \in S : e \sqsubseteq_{\mathbb{P}_\mathcal{P}\times(\mathbb{L}\times\mathbb{S})} e'$.

So far, our analysis configuration only incorporates the abstract model of a program. In the early stages of the CPA concept, this was sufficient because it was used to check reachability properties, especially that certain error locations are not reached. Such a property can easily be decided by an inspection of the abstract program model. Our property specification allows to specify more than simple reachability properties, e.g., protocol properties, which need to consider complete program traces. In the next section, we discuss how to integrate property checking into the configurable analysis process.

2.3.3 Property Integration

The software model checker SLAM [BR02a] encodes a given property into the input program and verifies the instrumented program. Similarly, before BLAST is used to check properties in the Blast Query Language the input program is instrumented [BCH+04]. In these approaches, the consumer must either check that the instrumented program encodes the desired property or he has to instrument the program himself, but in the same way as the producer. Due to these disadvantages, we decided to use a concept similar to an off-line monitor in runtime verification [HG08]. The idea is to run the property automaton in parallel to the analysis[8] and let it monitor the explored program paths.

To integrate the monitoring in an analysis configuration $\mathbb{C}$, we extend the abstract domain of $\mathbb{C}$ with automaton states, i.e., we consider the product of the abstract domain and the flat lattice of automaton states $\mathcal{Q}$ including the automaton states Q plus an additional top $q_\top$ and bottom state $q_\bot$. Furthermore, we integrate the property automaton's transitions into the transfer relation. We start to explain the most precise integration, which we call the most precise enhancement of an analysis configuration $\mathbb{C}$ with a property automaton $\mathcal{A}$.

Definition 2.10 (Most Precise Enhancement)**.** The *most precise enhancement* of a CPA $\mathbb{C} = ((C, \mathcal{E}, [\![\cdot]\!]), \Pi, \rightsquigarrow, \mathsf{prec}, \mathsf{merge}, \mathsf{stop})$ with a property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ is a new CPA $\mathbb{C}^{\mathcal{A}}_{\text{mp}} = (D^{\mathcal{A}}, \Pi, \rightsquigarrow^{\mathcal{A}}_{\text{mp}}, \mathsf{prec}^{\mathcal{A}}_{\text{mp}}, \mathsf{merge}^{\mathcal{A}}_{\text{mp}}, \mathsf{stop}^{\mathcal{A}})$ with

- abstract domain $D^{\mathcal{A}} = (C, \mathcal{E} \times \mathcal{Q}, [\![\cdot]\!]^{\mathcal{A}})$, where $\mathcal{Q}$ is the flat lattice of automaton states Q and additional top state $q_\top$ and bottom state $q_\bot$, and the automaton states do not influence the meaning of an abstract state, i.e., $\forall (e, q) \in E \times Q^\top_\bot : [\![(e, q)]\!]^{\mathcal{A}} = [\![e]\!]$,
- transfer relation $\rightsquigarrow^{\mathcal{A}}_{\text{mp}} \subseteq (E \times Q^\top_\bot) \times \mathcal{G} \times (E \times Q^\top_\bot)$ with $((e, q), (l, op, l'), (e', q')) \in \rightsquigarrow^{\mathcal{A}}_{\text{mp}}$ implies that $(e, (l, op, l'), e') \in \rightsquigarrow$ and either $q' \in Q$ and $\exists (q, op, C_{\text{sub}}, q') \in \delta : [\![e']\!] \subseteq C_{\text{sub}}$ or $q' = q_\top$ and $\neg\exists (q, op, C_{\text{sub}}, q') \in \delta : [\![e']\!] \subseteq C_{\text{sub}}$,
- precision adjustment $\mathsf{prec}^{\mathcal{A}}_{\text{mp}}((e, q), \pi, S) = ((e_{\text{prec}}, q), \pi_{\text{prec}})$ with $(e_{\text{prec}}, \pi_{\text{prec}}) = \mathsf{prec}(e, \pi, \{e' \mid (e', \cdot) \in S\})$,
- merge operator $\mathsf{merge}^{\mathcal{A}}_{\text{mp}}((e, q), (e', q')) = (e', q')$, and
- termination check $\mathsf{stop}^{\mathcal{A}}((e, q), S) = \mathsf{stop}(e, \{e' \mid (e', q') \in S \wedge q \sqsubseteq q'\})$.

The most precise enhancement interferes with our idea of a flexible and general analysis configuration framework. It is too strict. For example, it prohibits dataflow analyses because abstract states are never merged. Additionally, the transfer relation is very precise. On the one hand, this may be expensive to compute. On the other hand, it demands

[8] Note that the tool CPAchecker allows to run the property specification as part of the analysis.

that the automaton state is determined by the complete abstract state $e_{\mathbb{C}}$. However, for our Programs from Proofs approach we require that only one component determines the automaton state. Hence, we use the most precise enhancement as a standard for a correct property integration and accept any enhancement that overapproximates the behavior of the most precise enhancement. Those enhancements can be derived from the most precise enhancement by untightening of the automaton state computation in the transfer relation or relaxing the precision adjustment operator or the merge operator.

Definition 2.11 (Enhancement)**.** Let $\mathbb{C}^{\mathcal{A}}_{\text{mp}} = (D^{\mathcal{A}}, \Pi, \rightsquigarrow^{\mathcal{A}}_{\text{mp}}, \mathsf{prec}^{\mathcal{A}}_{\text{mp}}, \mathsf{merge}^{\mathcal{A}}_{\text{mp}}, \mathsf{stop}^{\mathcal{A}})$ be the most precise enhancement of CPA $\mathbb{C}$ and property automaton $\mathcal{A}$. A CPA $\mathbb{C}^{\mathcal{A}} = (D^{\mathcal{A}}, \Pi, \rightsquigarrow^{\mathcal{A}}, \mathsf{prec}^{\mathcal{A}}, \mathsf{merge}^{\mathcal{A}}, \mathsf{stop}^{\mathcal{A}})$ is an enhancement of CPA $\mathbb{C}$ if

- the transfer relation is less property precise: $((e,q),(l,op,l'),(e',q')) \in \rightsquigarrow^{\mathcal{A}}$ implies that there exists $((e,q),(l,op,l'),(e',q'')) \in \rightsquigarrow^{\mathcal{A}}_{\text{mp}}$ with $q'' \sqsubseteq q'$,
- the precision adjustment may return a more abstract state: $\forall e \in E^{\mathcal{A}}, \pi \in \Pi, S \subseteq E^{\mathcal{A}} : \mathsf{prec}^{\mathcal{A}}_{\text{mp}}(e,\pi,S) = (e^{\text{prec}}_{\text{mp}}, \pi^{\text{prec}}_{\text{mp}}) \wedge \mathsf{prec}^{\mathcal{A}}(e,\pi,S) = (e^{\text{prec}}, \pi^{\text{prec}}) \implies e^{\text{prec}}_{\text{mp}} \sqsubseteq e^{\text{prec}} \wedge \pi^{\text{prec}}_{\text{mp}} = \pi^{\text{prec}}$,
- the merge operator is more abstract: $\forall e, e' \in E^{\mathcal{A}} : \mathsf{merge}^{\mathcal{A}}_{\text{mp}}(e,e') \sqsubseteq \mathsf{merge}^{\mathcal{A}}(e,e')$.

After we understand how to configure the analysis of the producer, we proceed with the execution of a configured analysis.

2.4 Execution of Configured Analyses

To execute arbitrary configurable program analyses, we require a meta algorithm which is steered by the input CPA. From now on, we call this meta algorithm *CPA algorithm*. Since we want to use the CPA algorithm to check that a particular program is safe, we only allow enhanced CPAs $\mathbb{C}^{\mathcal{A}}$ to be input CPAs. The task of the CPA algorithm is to determine for a given input program P, a set of initial states described by an initial abstract state e_0, an initial precision, and an enhanced CPA $\mathbb{C}^{\mathcal{A}}$ whether program P is safe with respect to initial states $[\![e_0]\!]$, and property automaton $\mathcal{A}$, $P \models_{[\![e_0]\!]} \mathcal{A}$. Additionally, the CPA algorithm should construct a witness for each verification result produced. We start to describe the part of the CPA algorithm which does the actual verification.

2.4.1 Examination of Program Safety

Like we adapted the existing concept of a configurable program analysis, we also adapt the corresponding meta reachability algorithm [BHT08] to execute CPAs. Of course, we adapt the algorithm to fit to our definition of a configurable program analysis. Furthermore, we add a test for program safety. Note that we follow the implementation of the CPA algorithm in the tool CPAchecker and adjust a successor's precision directly after we explored the successor.

Algorithm 1 shows the part of the CPA algorithm which is responsible for program verification. The line numbers correspond to the line numbers in the complete CPA algorithm (Algorithm 2). In principal, Algorithm 1 describes a reachability analysis steered by the input CPA. The reachability analysis is similar to the abstract reachability analysis in software model checking [JM09b] and the worklist algorithm [Kil73] for the fixpoint computation in dataflow analyses. It maintains a waitlist, comparable to the worklist in

Algorithm 1: Extract of the CPA algorithm, a modified version of the reachability algorithm for configurable program analyses with dynamic precision adjustment [BHT08]

```
1  waitlist:={(e_0, π_0)}; reached:={e_0};
3  while waitlist≠ ∅ do
4      pop (e, π) from waitlist;
5      for each g ∈ G_CFA do
6          for each e' with (e, g, e') ∈⇝ do
7              (e_prec, π_prec) := prec(e', π, reached);
8              for each e'' ∈ reached do
9                  e_new := merge(e_prec, e'');
10                 if e_new ≠ e'' then
11                     waitlist := (waitlist ∪ {(e_new, π_prec)}) \ {(e'', π) | π ∈ Π};
12                     ... reached := (reached ∪ {e_new}) \ {e''};
18             if ¬stop(e_prec, reached) then
19                 ... waitlist := waitlist ∪ {(e_prec, π_prec)};
                       reached := reached ∪ {e_prec};
29 return (¬∃(·, q) ∈ reached : q = q_err ∨ q = q_⊤, ...)
```

software model checking [JM09b] or dataflow analyses [Kil73], which stores the states that must be explored together with their precision. Additionally, the set reached describes the already explored states and, finally, overapproximates the reachable states of the input program [BHT07, BHT08]. In line 1, the reachability analysis is started in the initial abstract state e_0 with initial precision π_0. For each unexplored state e, Algorithm 1 computes the abstract successors e' respecting the input program. To compute the abstract successors, the CPA's transfer relation $\rightsquigarrow_{\mathcal{A}}$ is used. Line 5 is required to restrict the transfer relation to the input program. Then, the precision of the abstract successor is adjusted. Afterwards, in line 9 the reachability analysis uses the CPA's merge operator and tries to combine the adjusted successor e_{prec} with an already explored state e''. If the merge operator widens the already explored state e'' – this is typically the case when information from different branches is joined –, the state e'' will be replaced by the result e_{new} of the merge. Next, in line 18 it is tested if the adjusted successor is covered by the explored states.[9] Successors which are not covered become part of the explored states and are registered for exploration. Finally, in line 29 it is checked whether the program is proven safe, i.e., the return value becomes $(true, \dots)$. Note that a return value $(false, \dots)$ only means that the given CPA failed to prove program safety, either because the program is unsafe or the CPA is not mature enough.

Originally, CPAs did not incorporate a mechanism for property specification and are used to verify that error locations are unreachable [Thé10]. Thus, Beyer et al. [BHT08] only proved that the set reached of explored states overapproximates the concrete states reachable in the program. However, the producer only wants to build certificates for safe programs. More importantly, in the Programs from Proofs approach also the consumer uses the CPA algorithm to check if a program is safe. Due to the theorem of Rice [Ric53], we know that the CPA algorithm cannot always correctly decide whether a program is safe. The CPA algorithm does not even terminate for any input CPA. Furthermore,

[9] Note that the test will typically return true, if in line 9 the result of different branches is integrated.

the CPA algorithm can only check program safety properly if the initial abstract state considers the initial state of the property automaton. Nevertheless, we require that if the CPA algorithm returns true and the initial abstract state is set up properly, the program will be indeed safe. Hence, the CPA algorithm must be sound.

In the following, we prove soundness of the complete CPA algorithm (Algorithm 2). Note that the complete CPA algorithm is an extension of Algorithm 1, which adds additional constructs to record the explored state space. The extension does not change the behavior of the presented verification procedure. To prove soundness of the CPA algorithm, we start to show that when the CPA algorithm returns true, then for every path a configuration sequence exists that does not consider the error state q_{err}. The idea of the proof is that for every pair of concrete state and automaton state in the configuration sequence, an explored abstract state exists that considers the same automaton state and covers the concrete state.

Lemma 2.2. *If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns $(true, \dots)$, then for every path $p \in paths_P([\![e_0]\!])$ there exists a configuration sequence $(c_0, q_0) \dots, (c_n, q_n)$ for p and $\mathcal{A}$ s.t. $\forall 0 \leq j \leq n : q_j \neq q_{\text{err}}$.*

Proof. See Appendix pp. 249 f. □

Due to the previous lemma, we can now easily apply the definitions for safe program paths and program safety. Thus, we infer that the CPA algorithm succeeds only when the input program is safe with respect to the property automaton considered by the input CPA and the states represented by the initial abstract state, i.e., the CPA algorithm is sound.

Theorem 2.3 (Soundness of CPA Algorithm)**.** *If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns (true,...), then $P \models_{[\![e_0]\!]} \mathcal{A}$.*

Proof. From the previous lemma, we can conclude that for every path $p \in paths_P([\![e_0]\!])$ there exists a configuration sequence $(c_0, q_0) \dots, (c_n, q_n)$ for p and $\mathcal{A}$ s.t. $\forall 0 \leq j \leq n : q_j \neq q_{\text{err}}$. By definition, every path $p \in paths_P([\![e_0]\!])$ is safe. By definition of program safety, it follows that $P \models_{[\![e_0]\!]} \mathcal{A}$. □

As a direct consequence of the above theorem, we conclude that if the initial abstract state considers all concrete states referring to the program entry location and the CPA algorithm succeeds, then the input program P will be safe with respect to the property automaton $\mathcal{A}$ considered by the input enhanced CPA.

Corollary 2.4. *Let $\mathbb{C}^{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$. If Algorithm 2 started with $\mathbb{C}^{\mathcal{A}}$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$, and any precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns (true,...) and $\{c \mid c \in C \wedge cs(c) = l_0\} \subseteq [\![e_0]\!]$, then program $P \models \mathcal{A}$.*

Proof. We assume that Algorithm 2 returns true and $I = \{c \mid c \in C \wedge cs(c) = l_0\} \subseteq [\![e_0]\!]$. From Theorem 2.3, we know that $P \models_{[\![e_0]\!]} \mathcal{A}$. By definition of paths, we know that $paths_P(I) \subseteq paths_P([\![e_0]\!])$. Hence, the definition of safety lets us conclude that $P \models_I \mathcal{A}$. With the definition of I, we infer that $P \models \mathcal{A}$. □

Until now, we only looked at the pure verification aspect of the CPA algorithm. In the following, we continue with the part of the CPA algorithm that generates the witness for a successful verification. We call the generated witness abstract reachability graph.

2.4.2 Abstract Reachability Graphs and Their Properties

Often, verification tools provide an error witness, an unsafe program path, when they fail to prove program safety. Beyer et al. [BDD+15] propose a verifier independent format to describe such error witnesses. In contrast, our approaches require a witness for program safety to construct certificates or generate a new program. Witnesses for program safety do not only represent a single program execution, but must consider the complete state space of the program, i.e., every possible program path.

The set reached is not sufficient to witness program safety. It does not provide any information about the paths explored during verification. Explicit model checking (see e.g. [JM09b]) inspects and constructs the concrete state space. Furthermore, symbolic model checking [BCM+92] encodes the concrete state space with binary decision diagrams, a representation of boolean functions. However, our verification procedure uses abstraction. Correctness witnesses [BDDH16], a special type of automata, are a verifier independent format to describe the (abstract) state space of a program. To model the program paths, the automaton edges are labeled with statements, and boolean expressions on states restrict the possible program states. Instead of boolean expressions, our state space representation should use a representation of states that is closer to the verification procedure. Thus, we want to represent states with the help of abstract states. Dataflow analyses [Kil73] and abstract interpretation [CC77] typically map dataflow facts or abstract states to the program locations. The program model plus the mapping describes the explored, abstract state space. Also, abstract model checking inspects and constructs the abstract state space, e.g., verification tools like CPAchecker [Ker11] or Blast [BHJM07] provide abstract reachability graphs or trees to describe the explored paths. Nodes are abstract states and edges represent possible program execution steps. We decided to use the concept of an *abstract reachability graph* (ARG), which is also used by the verification tool [BK11b] in which we integrate our approaches, as a witness for program safety.

In our setting, the set reached defines the nodes of the ARG. Edges between two nodes exist if the nodes are linked by the transfer relation, possibly indirectly due to a merge in line 9 or a successful termination check in line 18. To relate edges to program statements, we label them by CFA edges. Moreover, we explicitly track those ARG nodes which cover abstract successors, e.g., in case of a successful termination check, but we also incorporate those nodes which have more than one incoming edge. From a structural point of view, an ARG can be described as follows.

Definition 2.12 (Abstract Reachability Graph)**.** An *abstract reachability graph (ARG)* $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *for a program* P *and an enhancement* $\mathbb{C}^\mathcal{A}$ of CPA $\mathbb{C}$ consists of a finite set $N \subseteq E_{\mathbb{C}^\mathcal{A}}$ of nodes, a set G_{ARG} of edges between nodes labeled by control flow edges, $G_{\text{ARG}} \subseteq N \times G_{\text{CFA}} \times N$, a root node $root \in N$, and a subset of covering nodes $N_{\text{cov}} \subseteq N$.

The complete CPA algorithm as presented in Algorithm 2 (see p. 39) shows one possible extension of Algorithm 1 that incorporates the construction of ARGs. To construct ARGs, Algorithm 2 manages two helper variables coverSet and contained plus three additional data structures, a field root for the ARG's root node, a set G_{ARG} maintaining the ARG edges, and a set N_{cov} collecting the ARG's covering nodes. The ARG nodes are represented by the already existing set reached. In the beginning, the root node is the initial abstract state and G_{ARG} and N_{cov} are empty. Whenever the current root node is replaced in the set of nodes in line 12, the root node is adapted in line 15. The same holds for nodes in the set N_{cov}. The adaption of the ARG edges is more difficult. To adapt a successor of an edge due to widening is uncomplicated. A more abstract state covers at least the same parts of

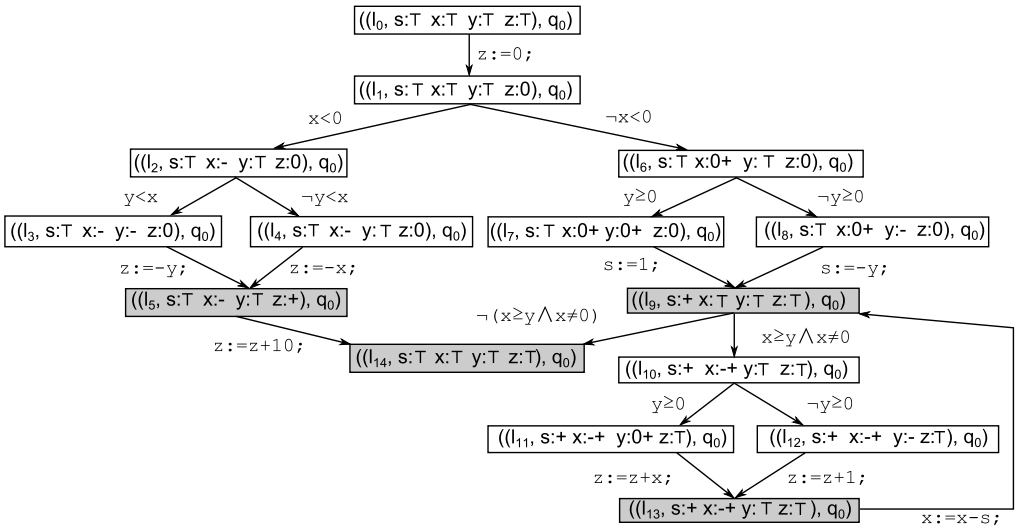

Figure 2.4: ARG constructed by Algorithm 2 when started with CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property `pos@l5`, initial abstract state $((l_0, \top_{\mathbb{S}}), q_0)$, and program `SubMinSumDiv`

the abstract successor. In contrast, if Algorithm 2 replaces the predecessor e'' by e_{new}, it cannot ensure that the set of ARG edges starting in e_{new} covers the abstract successors of e_{new}, even if previously the set of ARG edges starting in e'' covered the abstract successors of e''. Also, such edges do not witness the exploration of e_{new}. Since the abstract state e_{new} is explored anyway, it is added to waitlist, Algorithm 2 deletes all edges for which it might adapt the predecessor. In general, Algorithm 2 deletes edges $(e, \cdot, \cdot)$ when abstract state e is added to waitlist, i.e., it is registered for re-exploration. Algorithm 2 adds a new ARG edge (e, g, e_s) if an adjusted abstract successor e_{prec} of e and g is explored, e is still in the set of explored states, e is not registered for re-exploration, $e_s = e_{\text{prec}}$ and e_{prec} is not covered by the explored states or the termination check considers e_s when detecting coverage of e_{prec}. Nodes that are involved in the covering of an adjusted abstract successor e_{prec} are also added to the set N_{cov} of covering nodes. Moreover, we think that adding a state to reached that is already contained is similar to covering caused by the termination check, in both cases more than one predecessor for this state may exists. Thus, we add those nodes to N_{cov} in lines 17 and 23[10].

Note that the realization of ARGs and their construction in CPAchecker deviates from Algorithm 2. CPAchecker uses a wrapper CPA to construct the ARG [Ker11] and does not extend the CPA algorithm. Furthermore, it uses a different concept of covering. A covered node still becomes an abstract successor and additional unlabeled edges are added that map the covered node to the covering nodes. We chose our model of an abstract reachability graph because it represents the approximated, executable program paths in a more unique way and it does not contain nodes that are not part of reached. Nevertheless, we believe that the ARG constructed by CPAchecker can easily be transformed into our model of an ARG. For our implementation we managed the transformation.

Figures 2.4 and 2.5 show the abstract reachability graphs constructed by the CPA

[10] Note that we did not require that the termination check returns true if the state is contained in its second input parameter.

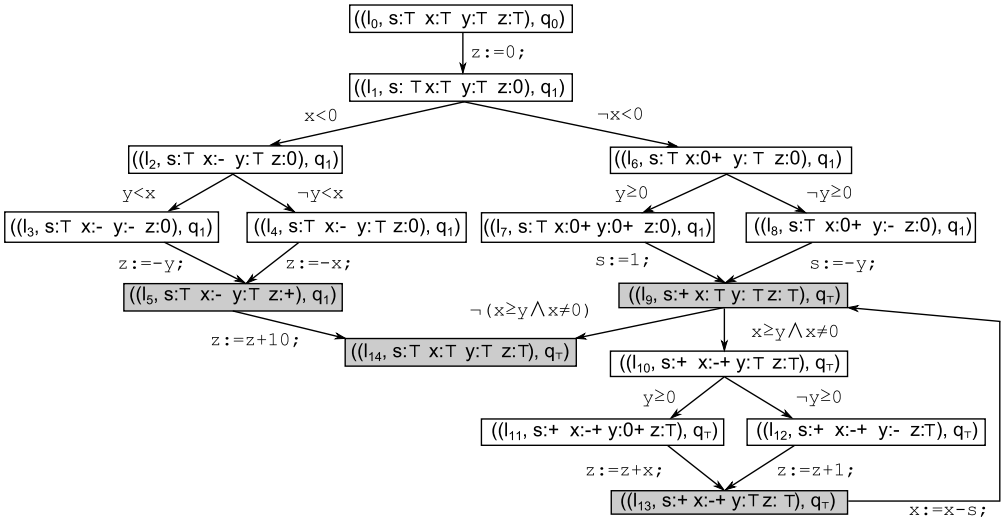

Figure 2.5: ARG constructed by Algorithm 2 when started with CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property `nonneg`, initial abstract state $((l_0, \top_{\mathbb{S}}), q_0)$, and program `SubMinSumDiv`

algorithm during analysis of program `SubMinSumDiv` with CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property automata `pos@l5` and `nonneg`, respectively, when started in the initial abstract state $((l_0, \top_{\mathbb{S}}), q_0)$. The enhanced CPA merges states for same locations and uses the most precise operators for precision adjustment and transfer relation. For the sake of readability, we labeled the edges by program instructions and not by the complete CFA edge. In both cases, the root node is the initial abstract state. ARG nodes contained in the set N_{cov} are highlighted in gray. We observe that property `pos@l5` could be proven (all abstract automaton states are q_0), but the proof of property `nonneg` failed (abstract automaton states $q_\top$ exists).

Before we discuss the properties of ARGs, which our approaches rely on, we first ensure that the structure returned by the CPA algorithm is indeed an abstract reachability graph.

Lemma 2.5. *If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns $(\cdot, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$, then $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an abstract reachability graph for P and $\mathbb{C}^{\mathcal{A}}$.*

Proof. Proof see Appendix pp. 250 f. □

Although we proved that the CPA algorithm returns an ARG, this does not mean that the ARG properly describes the state space exploration. Our definition of an ARG only describes the syntactical appearance. Consider the ARG shown in Fig. 2.6, which is an ARG for our example program `SubMinSumDiv` and CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property automaton `pos@l5`, but it does not track the state space exploration of that analysis on program `SubMinSumDiv` started in the initial abstract state $((l_0, s : \top\ x : \top\ y : \top\ z : \top), q_0)$. First, it does not consider the initial abstract state $((l_0, s : \top\ x : \top\ y : \top\ z : \top), q_0)$. Second, it misses abstract successors. For example, $((l_9, s : +\ x : \top\ y : \top\ z : \top), q_0)$, an abstract successor of $((l_{13}, s : +\ x : -+\ y : \top\ z : \top), q_0)$ along edge $(l_{13}, x := x-s, l_9)$, is not taken into account. Third, the only edge does not describe a part of the state space. The

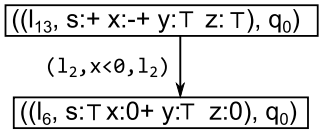

Figure 2.6: ARG for our example program `SubMinSumDiv` and CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property automaton `pos@l5` which is not compliant to the state space exploration

location states do not match the predecessor and successor of the edge. However, during state space exploration only abstract successors are computed that adhere to the control flow. To describe that an ARG properly recorded a state space exploration, we introduce the properties rootedness, completeness, and well-constructedness. Furthermore, we use a safety property to describe that the ARG is constructed during a successful verification. The remaining properties provide further structural details needed by our approaches. For example, determinism and soundness are properties required by our Programs from Proofs approach only.

Definition 2.13 (Properties of Abstract Reachability Graphs)**.** Let $P = (L, G_{\text{CFA}}, l_0)$ be a program, $\mathbb{C}^{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$, $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$ an initial abstract state, and $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$ be an ARG for a program P and enhancement $\mathbb{C}^{\mathcal{A}}$. Following, we define properties an ARG $R^P_{\mathbb{C}^{\mathcal{A}}}$ may fulfill.

Rootedness The initial state is covered by the root, $e_0 \sqsubseteq \mathit{root}$.

Completeness All abstract successors of a node are either covered by a more abstract state or by a subset of the explored states, $\forall n \in N, g \in G_{\text{CFA}} : (n, g, e) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \implies \exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : e \sqsubseteq n' \wedge ((n, g, n') \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S))$.[11]

Well-Coveredness All abstract successors e that are not covered by N_{cov} are covered by a unique non-covering node, formally defined as follows. Let $S_{\text{TCNC}} := \{(n, g, e) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \mid n \in N, g \in G_{\text{CFA}} \wedge \neg\exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : e \sqsubseteq n' \wedge ((n, g, n') \in G_{\text{ARG}} \wedge n' \in N_{\text{cov}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S))\}$ be the set of abstract successor computations which are covered by non-covering nodes. There exists a total, injective function $\text{cov} : S_{\text{TCNC}} \to N \setminus N_{\text{cov}}$ with $\forall (n, g, e) \in S_{\text{TCNC}} : e \sqsubseteq \text{cov}((n, g, e)) \wedge (n, g, \text{cov}((n, g, e))) \in G_{\text{ARG}}$.

Soundness ARG successors are more abstract than abstract successors computed by the transfer relation, $\forall (n, g, n') \in G_{\text{ARG}} : (n, g, n'') \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \implies n'' \sqsubseteq_{\mathbb{C}^{\mathcal{A}}} n'$.

Safety All nodes are safe. They do not reference the error state q_{err} or $q_\top$: $\forall (e, q) \in N : q \neq q_{\text{err}} \wedge q \neq q_\top$.

Well-Constructedness An ARG edge only exists if an abstract successor is present, $\forall (n, g, n') \in G_{\text{ARG}} : g \in G_{\text{CFA}} \wedge \exists (n, g, n'') \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$.

Determinism No two edges exist that have the same predecessor and the same label: $\forall (n, g, n'), (n, g, n'') \in G_{\text{ARG}} : n' = n''$.

[11] Due to the overapproximation of the transfer relation, an ARG edge exists as soon as at least a concrete transition exists, i.e., $\exists c \in [\![n]\!] : c \xrightarrow{g} c'$.

ARGs constructed by the CPA algorithm do not always fulfill all these properties. For example, for an unsafe program the properties safety, soundness, and rootedness cannot be fulfilled at the same time. Some properties of the constructed ARG depend on the input of the CPA algorithm and others do not. We start to show that the properties rootedness, completeness, well-coveredness, and well-constructedness are input insensitive, i.e., the CPA algorithm (Algorithm 2) always returns ARGs that are rooted, complete, well-covered, and well-constructed.

Lemma 2.6. *If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns $(\cdot, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$, then $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is rooted, complete, well-covered, and well-constructed.*

Proof. See Appendix pp. 251 ff. □

For our approaches, the properties rootedness, completeness, well-coveredness, and well-constructedness, which are ensured for any ARG constructed by the CPA algorithm, are mandatory but not enough. They only ensure that all program paths starting in a state represented by the initial abstract state are covered by the ARG, but they do not necessarily witness program safety. Hence, our approaches require ARGs that are safe, too. We call ARGs that fulfill all these five properties *well-formed*.

Definition 2.14 (Well-formedness Criteria for ARG)**.** Let P be a program, $\mathbb{C}^{\mathcal{A}}$ an enhancement of CPA $\mathbb{C}$, and $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$ an initial abstract state. An *abstract reachability graph $R^P_{\mathbb{C}^{\mathcal{A}}}$ for P and $\mathbb{C}^{\mathcal{A}}$ is well-formed for e_0* if $R^P_{\mathbb{C}^{\mathcal{A}}}$ is rooted, complete, well-covered, safe, and well-constructed.

Well-formed ARGs provide a nice property that turns them into a witness for program safety. For every path p starting in a state represented by the initial abstract state, an ARG path exists with the following two properties. First, the ARG path covers p. Second, assuming a proper initial abstract state is used, i.e., the initial abstract state considers the initial automaton state q_0, the pairwise combination of p's concrete states and the abstract automaton states of the ARG path yield a configuration sequence for p that does not consider the error state q_{err}, i.e., the ARG path is a witness for the safety of program path p. In the following, we consider a weaker concept of a witness, which is used by our certification approaches. Instead of an ARG path, we only require a sequence of ARG nodes whose abstract automaton states can be combined with p's concrete states to form a configuration sequence for p that does not consider the error state q_{err}. Nevertheless, we use the ARG paths to prove that well-formed ARGs fulfill the requirement of the weaker concept of a witness. The following lemma only shows that for every path p starting in a state represented by the initial abstract state, a configuration sequence for p exists that is covered by a sequence of ARG nodes. We can easily conclude the remaining part of the witness requirement from the safety property.

Lemma 2.7. *Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph for program P and enhancement $\mathbb{C}^{\mathcal{A}}$ of CPA $\mathbb{C}$ with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ s.t. $R^P_{\mathbb{C}^{\mathcal{A}}}$ is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$. Then, for all paths $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P([\![root]\!])$ there exists a configuration sequence $(c_0, q_0), \ldots, (c_n, q_n)$ for p and $\mathcal{A}$ with $\forall 0 \leq i \leq n : \exists (e, q) \in N : c_i \in [\![e]\!] \wedge q_i \sqsubseteq q$.*

Proof. See Appendix pp. 255 f. □

The safety property of a well-formed ARG and the previous lemma guarantee us that a well-formed ARG witnesses program safety. Our approaches rely on a witness for program safety in form of a well-formed ARG. We propose that the CPA algorithm indeed constructs well-formed ARGs whenever program verification succeeds. From Lemma 2.6, we already know that every ARG constructed by the CPA algorithm is rooted, complete, well-covered, and well-constructed. To prove our claim, we only need to show the safety property.

Proposition 2.8. *If Algorithm 2 started with enhancement $\mathbb{C}^{\mathcal{A}}$ of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns (true, $\mathsf{reached}$, $R^P_{\mathbb{C}^{\mathcal{A}}}$), then $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is well-formed for e_0.*

Proof. From Lemma 2.6, we know that $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is rooted, complete, well-covered, and well-constructed. Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$. We only need to show safety. Since in line 29 $N' = \mathsf{reached}$ and Algorithm 2 returns true $(\neg\exists(\cdot, q) \in \mathsf{reached} : q = q_{\text{err}} \vee q = q_\top)$, the ARG $R^P_{\mathbb{C}^{\mathcal{A}}}$ is safe. □

For our certification approaches, well-formed ARGs are sufficient. However, our Programs from Proofs approach demands two further properties. To ensure that no new behavior is introduced in the generated program, the ARG must be sound. Furthermore, to retranslate the CFA representation of the generated program into a program written in the same programming language as the original program, we require that the constructed ARG is deterministic. The CPA algorithm ensures these two additional properties in case of a successful verification if the analysis is of a particular shape. The transfer relation must be a function, i.e., per abstract state and statement at most one abstract successor may exists. Additionally, state exploration may only be stopped if the state is covered by a more abstract state. The analyses considered by our Programs from Proofs approach meet these requirements. The following proposition states that an ARG constructed by the CPA algorithm fulfills the two additional properties when the input CPA's transfer relation and termination check meet the previous two requirements.

Proposition 2.9. *Let CPA $\mathbb{C}^{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ s.t. $\forall e \in E_{\mathbb{C}^{\mathcal{A}}}, S \subseteq E_{\mathbb{C}^{\mathcal{A}}} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(e, S) \implies \exists e' \in S : e \sqsubseteq_{\mathbb{C}^{\mathcal{A}}} e'$ and $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ is a function. If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns $(\cdot, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$, then $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is sound and deterministic.*

Proof. See Appendix pp. 256 f. □

After we introduced all fundamentals required to understand our approaches, we continue with the description of the first approach, the basic configurable certification approach.

Algorithm 2: CPA algorithm, a modified version of the reachability algorithm for configurable program analysis with dynamic precision adjustment[BHT08]

Input: A property enhanced CPA $\mathbb{C}^{\mathcal{A}} = ((C, (E, \top, \bot, \sqsubseteq, \sqcup), [\![\cdot]\!]), \Pi, \rightsquigarrow, \mathsf{prec}, \mathsf{merge}, \mathsf{stop})$, initial abstract state $e_0 \in E$, initial precision $\pi_0 \in \Pi$, program $P = (L, G_{\mathrm{CFA}}, l_0)$

Output: A boolean value indicating if no unsafe automaton state is reached, a set of reachable abstract states and an abstract reachability graph for P and $\mathbb{C}^{\mathcal{A}}$

Data: A set reached of elements of E, a set waitlist of elements of $E \times \Pi$, a root node from E, a set of edges G_{ARG} of elements of $E \times G_{\mathrm{CFA}} \times E$, a set N_{cov} of elements of E, a set coverSet of elements of E, boolean variable contained

1 waitlist:=$\{(e_0, \pi_0)\}$; reached:=$\{e_0\}$;
2 $root := e_0$; $G_{\mathrm{ARG}} := \emptyset$; $N_{\mathrm{cov}} := \emptyset$;
3 **while** waitlist$\neq \emptyset$ **do**
4 pop (e, π) from waitlist;
5 **for** each $g \in G_{\mathrm{CFA}}$ **do**
6 **for** each e' with $(e, g, e') \in \rightsquigarrow$ **do**
7 $(e_{\mathrm{prec}}, \pi_{\mathrm{prec}}) := \mathsf{prec}(e', \pi, \mathsf{reached})$;
8 **for** each $e'' \in \mathsf{reached}$ **do**
9 $e_{\mathrm{new}} := \mathsf{merge}(e_{\mathrm{prec}}, e'')$;
10 **if** $e_{\mathrm{new}} \neq e''$ **then**
11 $\mathsf{waitlist} := (\mathsf{waitlist} \cup \{(e_{\mathrm{new}}, \pi_{\mathrm{prec}})\}) \setminus \{(e'', \pi) \mid \pi \in \Pi\}$;
12 contained:=$e_{\mathrm{new}} \in$ reached;$\mathsf{reached} := (\mathsf{reached} \cup \{e_{\mathrm{new}}\}) \setminus \{e''\}$;
13 $G_{\mathrm{ARG}} := (G_{\mathrm{ARG}} \cup \{(e_{\mathrm{p}}, g, e_{\mathrm{new}}) \mid (e_{\mathrm{p}}, g, e'') \in G_{\mathrm{ARG}}\}) \setminus \{(e_{\mathrm{p}}, g, e_{\mathrm{s}}) \mid (e_{\mathrm{p}}, g, e_{\mathrm{s}}) \in (G_{\mathrm{ARG}} \cup \{(e_{\mathrm{p}}, g, e_{\mathrm{new}}) \mid (e_{\mathrm{p}}, g, e'') \in G_{\mathrm{ARG}}\}) \wedge (e_{\mathrm{p}} = e'' \vee e_{\mathrm{s}} = e'' \vee e_{\mathrm{p}} = e_{\mathrm{new}})\}$;
14 **if** $e'' = root$ **then**
15 $root := e_{\mathrm{new}}$;
16 **if** $e'' \in N_{\mathrm{cov}} \vee$ contained **then**
17 $N_{\mathrm{cov}} := (N_{\mathrm{cov}} \cup \{e_{\mathrm{new}}\}) \setminus \{e''\}$;
18 **if** $\neg\mathsf{stop}(e_{\mathrm{prec}}, \mathsf{reached})$ **then**
19 contained:= $e_{\mathrm{prec}} \in$ reached;$\mathsf{waitlist} := \mathsf{waitlist} \cup \{(e_{\mathrm{prec}}, \pi_{\mathrm{prec}})\}$; $\mathsf{reached} := \mathsf{reached} \cup \{e_{\mathrm{prec}}\}$;
20 $G_{\mathrm{ARG}} := G_{\mathrm{ARG}} \setminus \{(e_{\mathrm{prec}}, \cdot, \cdot) \in G_{\mathrm{ARG}}\}$;
21 **if** $e \in \mathsf{reached} \wedge \neg\exists(e, \dot{}) \in \mathsf{waitlist}$ **then**
22 **if** contained **then**
23 $N_{\mathrm{cov}} := N_{\mathrm{cov}} \cup \{e_{\mathrm{prec}}\}$;
24 $G_{ARG} := G_{ARG} \cup \{(e, g, e_{\mathrm{prec}})\}$;
 else
25 **if** $e \in \mathsf{reached} \wedge \neg\exists(e, \cdot) \in \mathsf{waitlist}$ **then**
26 $\mathsf{coveringSet} := \min\{S \subseteq \mathsf{reached} \mid \mathsf{stop}(e_{\mathrm{prec}}, S)\}$;
27 $N_{\mathrm{cov}} := N_{\mathrm{cov}} \cup \{\mathsf{coveringSet}\}$;
28 $G_{ARG} := G_{ARG} \cup \{(e, g, e_{\mathrm{r}}) \mid e_{\mathrm{r}} \in \mathsf{coveringSet}\}$;
29 **return** $(\neg\exists(\cdot, q) \in \mathsf{reached} : q = q_{\mathrm{err}} \vee q = q_{\top}, \mathsf{reached}, (\mathsf{reached}, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}}))$

3 Configurable Program Certification – The Basic Idea

Generally, all our approaches aim to reduce the validation costs of the consumer. To achieve this goal, our first approach, the configurable program certification approach, builds on a phenomenon that is well-known for NP complete problems. For NP complete problems L, when given a suitable witness w one can utilize w to check in polynomial time whether an input word x is an element of L [Hro11, pp. 232-238]. However, assuming $P \neq NP$ constructing a witness or equivalently inspecting whether $x \in L$[1] is non-polynomial, i.e., inefficient in practice. A similar observation applies to program verification, which is in general undecidable [Ric53], and thus much harder than NP problems. Verifying program correctness w.r.t. a property, i.e., finding a proof that shows that a program is correct w.r.t. a property, is difficult. Yet, checking whether a proof shows that a program is correct w.r.t. a property is much simpler.

In 1996, Necula and Lee introduced the Proof-Carrying Code (PCC) framework [NL96] which uses the above observation to ensure that code from untrusted providers does not violate the consumer's safety policy. The provider attaches a proof in logic that the code does not violate the given safety policy and the consumer only checks the proof. At the beginning, proofs were constructed semi-automatically with theorem provers [NL96, Nec97, AF00]. PCC research focused on proof sizes, see e.g. [NL98a, NR01, WAS03], and a small trusted computing base [AF00, App01, WAS03].

Later, also automatic verification tools are used to construct proofs. For example, Henzinger et al. [HJMS02] use model checking with predicate abstraction to show that an error location is unreachable. The proof for the verification condition is constructed from the abstract reachability tree built during model checking. As another example, Seo et al. [SYY03] execute an abstract interpreter and use its result to construct a Hoare proof.

With the use of automatic verification tools, the proposed PCC techniques deviate from the strict concept of a mathematical proof, too. To mention only a few, Kupferman

[1] Any inspection which shows that $x \in L$ or $x \notin L$ is a witness.

and Vardi [KV04] use a ranking function to certify that a system fulfills a LTL formula. The model checker SLAB [DKFW10] describes its verification diagram in SMT code. Many approaches, e.g. [Ros03, HJMS03, APH05b, BJP06], utilize (parts of) the explored state space as proofs. Beyer et al. [BDDH16] suggest a correctness witness automaton whose edges are labeled with statements and its states are labeled with C expressions representing local invariants.

To construct a logic proof from the analysis result, an ARG, of an arbitrary configurable program analysis, the abstract states must be transformed into an equivalent logical representation. Of course, one could use the concretization of the abstract state. However, in this case one would loose the advantage of the abstraction. Furthermore, abstract states like the sign or predicate abstract state, which only restrict the numerical values of variables could be easily represented by logic formulae and are understood by logic theories. Finding a logic theory that is able to express that a variable is initialized or points to a valid memory address is more difficult. Thus, we think that it is difficult to provide a generic approach that constructs a logical proof from the analysis result of an arbitrary configurable program analysis. The same holds true for a specific proof format like SMT. Hence, we follow a typical certificate format and use parts of the state space, represented by the ARG, as certificate in our configurable program certification approach.

PCC approaches put different emphasize on the trusted computing base. The trusted computing base of an approach contains all those elements the consumer must trust if he trusts the outcome of his validation procedure. Approaches like foundational PCC [AF00, App01], verified PCC [WNKN04, WN05] or the approach by Besson et al. [BJP06] try to reduce the trusted computing base to a minimum. For example, they restrict the number of axioms that can be used in the proof or they verify or certify components used in the consumer validation. In other approaches, e.g., [Ros03, HJMS03, APH05b], there is almost no difference between the trusted computing base for verification and consumer validation. In favor of generality and efficiency, we decided not to focus on a small trusted computing base. In our approaches, the consumer's trusted computing base is only a little bit smaller than the producer's trusted computing base.

Next, we describe the details of our configurable program certification approach. We start with a general overview of configurable program certification.

3.1 Overview of Configurable Program Certification

Configurable program certification (CPC) was first proposed by us [JW14] and later also optimized by us [Jak15]. In the following two chapters, we present an improvement of the original configurable program certification and its optimizations. This improvement supports a broader class of program analyses and safety properties.

Figure 3.1 shows the general process of our configurable program certification approach. It is an instance of our generic solution presented in the introduction. The producer starts to verify programs as presented in the previous chapter. He uses the concept of a configurable program analysis enhanced with a property (automaton) to configure his verification. The verification is executed with the CPA algorithm, the resource consuming analysis. After a successful verification (result true), the producer generates a certificate from the abstract reachability graph, the proof. The certificate generation (the program expansion) as well as the certificate, which is used to enhance the program, differ across the basic and optimized instances of the configurable program certification approach. In all cases, the certificate is a witness for program safety, which uses ARG

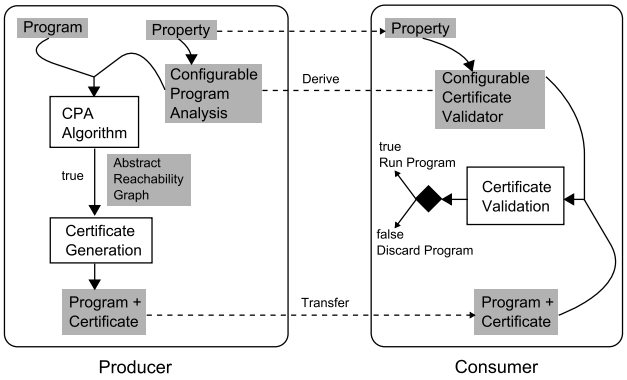

Figure 3.1: Overview of configurable program certification

nodes to provide some information about the state space explored during verification. Finally, the producer transfers the program plus the certificate to the consumer.

The consumer receives a program and a certificate from the producer. We assume that both can be corrupted during the transfer. To get a configuration that fits to the certificate, the consumer derives his configuration from the producer. Putting it simple, the configurable certificate validator adopts a subset of the CPA configuration. Finally, the consumer uses the certificate validation, his simple validation procedure, to check whether the received certificate witnesses safety of the received program. If the validation succeeds, the program can be executed. Otherwise the received program will be discarded. We want that the validation only succeeds when the program is safe w.r.t. the provided property. Additionally, validation must discard a program if the validation procedure fails to validate the certificate, e.g., when the program or the certificate are corrupted, no matter whether the program is safe or not.

Note that for the sake of comprehensibility, we left out the initial abstract state and the initial precision in our overview. As known from the previous chapter, the CPA algorithm requires these two elements as additional inputs. The certificate validation expects only an initial abstract state. Furthermore, two additional constraints must be considered. As known from verification, the initial abstract states must represent the initial automaton state. Additionally, the consumer must not consider more behavior, i.e., his initial abstract state must be less than or equal to the producer's initial abstract state. A consumer must derive his initial abstract state from the producer's initial abstract state. Of course, the initial abstract state could be part of the negotiations between the producer and the consumer.

In the remaining part of this chapter, we present the basic, non-optimized version of our configurable program certification approach. Following the configurable program certification procedure, we start to explain the producer's task.

3.2 Producer Verification and Certificate Construction

The overview of the configure program certification process revealed that the producer uses the CPA algorithm and an appropriate enhanced CPA to verify program safety. We already discussed this kind of program verification in the previous chapter. Thus, in this section we focus on the certificate construction.

The producer constructs certificates after a successful verification, i.e., when the CPA algorithm returned true. From the previous chapter, we remember that the CPA algorithm reports true/false based on its state space exploration. A straightforward solution is to use the explored state space as a certificate. A (partial) representation of the explored state space is a regularly used type of certificate for safety properties, see e.g. [HJMS03, Ros03, APH05b]. To validate a certificate, the consumer must check two aspects. First, the certificate must describe a correctly explored state space, e.g., Abstraction-Carrying Code[APH05b] checks whether it is a fixpoint and the approach by Henzinger et al. [HJMS03] inspects if all successors of a state are considered. Second, the state space must imply program safety.

Recall that the CPA algorithm records its state space exploration in the abstract reachability graph. In contrast to e.g. Henzinger et al. [HJMS03], who use the complete ARG as certificate, we decided that we do not want to store how the state space is constructed. In our basic version of the configurable program certification, we only store the explored states, the nodes of the ARG. The advantage is that our certificates become smaller, but probably at the cost of an increase in validation costs. During validation we cannot utilize the local successor relation and always need to consider all ARG nodes.

From a syntactical point of view, our *certificates* are sets of abstract states. To our mind, a syntactical criterion is a proper choice to define certificates. Parsers easily determine whether an input file/stream adheres to a syntactical format. Moreover, like verifiers do not check syntactically incorrect programs, we do not want to validate certificates which are syntactically incorrect, i.e., do not describe a set of abstract states. Furthermore, we cannot assume that we get a certificate that could have been constructed by a process conformant producer. A constructed certificate may be corrupted while it is transferred to the consumer. Additionally, the producer may be malicious and does not follow our proposed certificate construction. Finally, checking only certificates for which their definition already states that they witnesses program safety is wasted effort. Hence, from now on we only consider syntactically correct certificates and assume that before validation the consumer refuses programs that are not enhanced with syntactically correct certificates.

The producer considers only a single CPA and, thus, a single abstract domain during verification. Hence, his certificates consider only abstract states of a single abstract domain. To exclude sets of abstract states that mix abstract states from different domains, we reuse the definition of a certificate in our previous paper [JW14] and define a certificate w.r.t. an abstract domain. The abstract domain is given by an enhanced CPA. Now, a certificate is any subset of the set of abstract states considered by that enhanced CPA.

Definition 3.1 (Certificate)**.** Let $\mathbb{C}_{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ considering the set of abstract states $E_{\mathbb{C}\mathcal{A}}$. A *certificate* $\mathcal{C}_{\mathbb{C}\mathcal{A}}$ is a set of abstract states, $\mathcal{C}_{\mathbb{C}\mathcal{A}} \subseteq E_{\mathbb{C}\mathcal{A}}$.

A syntactically correct certificate does not automatically witness program safety. For example, look at the following set of abstract states:

$$\{((l_0, s : \top\ x : \top\ z : \top), q_0), ((l_1, s : \top\ x : \top\ z : 0), q_{\text{err}})\}\ .$$

This set is a certificate for the sign dataflow analysis enhanced with the property automaton for `pos@l5`, one of our example analyses. However, it does not witness safety of program `SubMinSumDiv` w.r.t. property `pos@l5` and the initial states considering the initial program location. First, the behavior of the property automaton is not considered completely. Look at a path $c_0 \xrightarrow{(l_0, z:=0;, l_1)} c_1$ of program `SubMinSumDiv` that starts in the initial

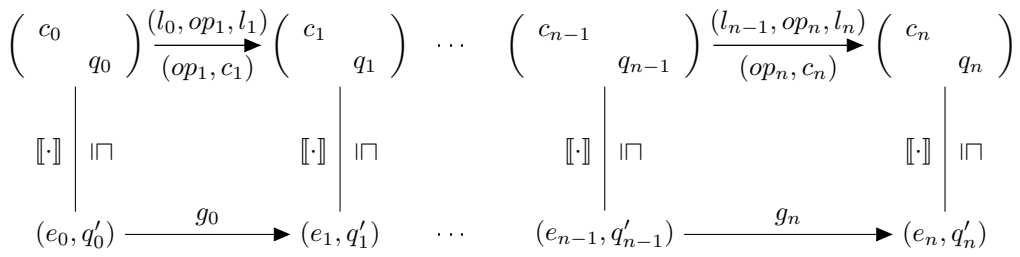

Figure 3.2: Correspondence between ARG path and configuration sequence

program location, $\text{cs}(c_0) = l_0$. From our example semantics for program `SubMinSumDiv`, we infer that such a path exists and that $\text{cs}(c_1) = l_1$. The property automaton `pos@l5` always starts in q_0 and the transition triggered in q_0 by $c_0 \xrightarrow{(l_0, z:=0;, l_1)} c_1$ ends in q_0. State c_1 is related to automaton state q_0, but the only abstract state $((l_1, s : \top\ x : \top\ z : 0), q_{\text{err}})$ that covers c_1 contains automaton state $q_{\text{err}} \neq q_0$. Second, it misses program behavior. For example, consider a path $c_0 \xrightarrow{(l_0, z:=0;, l_1)} c_1 \xrightarrow{(l_1, x<0, l_2)} c_2$ of program `SubMinSumDiv` that starts in the initial program location, $\text{cs}(c_0) = l_0$. From our example semantics for program `SubMinSumDiv`, we infer that such a path exists. The certificate's abstract states cover c_0 and c_1. In contrast, c_2 is not covered because none of the abstract states considers location l_2, which is the control state of c_2 (program semantics). Third, the state $((l_1, s : \top\ x : \top\ z : 0), q_{\text{err}})$ considers the error state of the property automaton, i.e., the certificate is not safe. Note that after a successful verification the set of explored states does not fulfill any of the three properties. After we presented reasons why a certificate is not a witness, we continue to discuss the properties a certificate should fulfill to be a witness.

Looking at the definition of program safety (Definition 2.9), we observe that a program is unsafe iff a program path exists such that all corresponding property automaton runs end in the error state. Hence, a witness must convince the consumer that for every program path a configuration sequence exists that does not contain the error state.

Reconsider that an enhanced CPA defines how to explore the program paths while concurrently monitoring the explored paths, i.e., it runs the associated property automaton in parallel. Assuming that the CPA algorithm is started with a proper initial abstract state, after termination of the CPA algorithm the ARG paths do not only overapproximate the program paths but also the corresponding automaton runs. The ARG paths cover the configuration sequences of the program paths. Figure 3.2 visualizes this coverage. For any explored program path $c_0 \xrightarrow{(l_0, op_1, l_1)} c_1 \dots c_{n-1} \xrightarrow{(l_{n-1}, op_n, l_n)} c_n$ and any corresponding configuration sequence $(c_0, q_0)(c_1, q_1) \dots (c_{n-1}, q_{n-1})(c_n, q_n)$ an ARG path of length n exists s.t. for all pairs (c_i, q_i) in the configuration sequence the ith ARG node (e_i, q'_i) covers (c_i, q_i), i.e., $c_i \in [\![(e_i, q'_i)]\!]$, equivalently $c_i \in [\![e_i]\!]$, and $q_i \sqsubseteq q'_i$.

Since we never used the ARG edges, we conclude that a sequence of ARG nodes that covers a configuration sequence is sufficient to convince the consumer that this configuration sequence was considered during verification. With this insight, we claim that the abstract states in a valid certificate must cover at least one configuration sequence per path to convince the consumer that for every program path a corresponding configuration sequence is considered during verification. To ensure that the inspected configuration sequences for all paths do not contain the error state, we require that no certificate state

includes the error state q_{err}, i.e., the certificate is safe.

We call a certificate that fulfills these requirements *a valid certificate*. In the following, we define certificate validity w.r.t. a set of initial states because we already restrict program safety to a set of initial states.

Definition 3.2 (Valid Certificate)**.** A *certificate* $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ *is valid* for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$ if

- it covers one configuration sequence per path, $\forall p := c_0 \dots c_n \in paths_P(I) : \exists$ configuration sequence $(c_0, q_0), \dots, (c_n, q_n)$ for p and $\mathcal{A} : \forall 0 \leq i \leq n : \exists (e, q) \in \mathcal{C}_{\mathbb{C}^{\mathcal{A}}} : c_i \in [\![e]\!] \wedge q_i \sqsubseteq q$, and
- the certificate is safe, i.e., $\forall (e, q) \in \mathcal{C}_{\mathbb{C}^{\mathcal{A}}} : q \neq q_{\text{err}} \wedge q \neq q_{\top}$.

After we defined when a certificate becomes a valid witness for program safety w.r.t. a certain property and a given set of initial states, we need to prove that our definition of valid certificates indeed ensures the witness property. If the certificate is valid for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$, the program P must be safe w.r.t. $\mathcal{A}$ and I. This is stated by the following lemma.

Lemma 3.1. *If certificate* $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ *is valid for program* P*, property automaton* $\mathcal{A}$*, and a set of initial states* $I \subseteq C$*, then* $P \models_I \mathcal{A}$*.*

Proof. Let $p \in paths_P(I)$ be an arbitrary path. We need to show that p is safe w.r.t. $\mathcal{A}$. Since certificate $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ is valid, there exists a configuration sequence $(c_0, q_0) \dots (c_n, q_n)$ for p and $\mathcal{A}$ s.t. $\forall 0 \leq i \leq n : \exists (\cdot, q) \in \mathcal{C}_{\mathbb{C}^{\mathcal{A}}} : q_i \sqsubseteq q$, $q_{\text{err}} \neq q$ and $q_{\top} \neq q$. We can conclude that $q_i \neq q_{\text{err}}$. It follows that $P \models_I \mathcal{A}$. □

The previous two definitions establish the frame conditions for the certificate constructed by the producer. With these frame conditions in mind, we explain how the producer builds his certificate from the result of the CPA algorithm. The CPA algorithm returns three components which are already certificates: the set reached, the set of ARG nodes, and the set of covering nodes. From the CPA algorithm and the definition of an ARG, we know that the set reached and the set of ARG nodes are identical and the set of covering nodes is a subset of those two. Consider the ARG in Fig. 3.4. We know this ARG, which is generated by one of our example analyses, from the previous section.

We notice that its set of covering nodes shown in Fig. 3.3 (the gray nodes in Fig. 3.4) do not cover a configuration sequence for each path of program `SubMinSumDiv` starting in the initial program location, although this ARG results from a successful verification of program `SubMinSumDiv` w.r.t. property `pos@l5`. For example, consider arbitrary path c of length 0 with $c \in C$ and $\text{cs}(c) = l_0$. This path is a path of program `SubMinSumDiv` starting in the initial program location. By definition, the only valid configuration sequence is (c, q_0). Since all covering nodes refer to a concrete location that is different from l_0, none of them covers c. Thus, no configuration sequence for path c is covered.

$$\left\{ \begin{array}{l} ((l_5, s : \top\ x : -\ y : \top\ z : +\ ), q_0), ((l_9, s : +\ x : \top\ y : \top\ z : \top\ ), q_0), \\ ((l_{13}, s : +\ x : -+\ \ y : \top\ z : \top\ ), q_0), ((l_{14}, s : \top\ x : \top\ y : \top\ z : \top\ ), q_0) \end{array} \right\}$$

Figure 3.3: The set of covering nodes of the ARG shown in Fig. 3.4

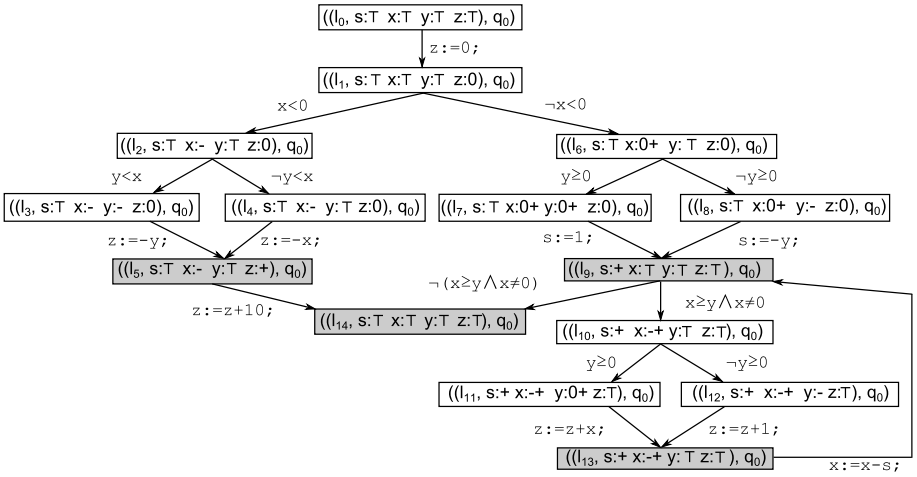

Figure 3.4: ARG constructed by the CPA algorithm during the analysis of program `SubMinSumDiv` with CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property automata `pos@l5` when started in initial abstract state $((l_0, \top_{\mathbb{S}}), q_0)$

In contrast, we know that if the producer verification succeeds, the ARG will be well-formed (Proposition 2.8). Thus, the set of ARG nodes, which is equivalent to reached, will cover one configuration sequence for each program path that starts in a state represented by the root node (Lemma 2.7). Moreover, in a well-formed ARG the initial abstract state is less abstract than the root. Hence, we infer from the definition of paths that the set of ARG nodes of a well-formed ARG also covers one configuration sequence for each program path that starts in a state represented by the initial abstract state. Furthermore, in a well-formed ARG the set of ARG nodes is safe. Thus, if the producer uses the set reached, equivalently the set of ARG nodes, as certificate, he will generate proper certificates. To be consistent with the following approaches, the producer constructs his *certificate from the ARG* and not from the set reached. Now, the certificate construction is quite simple. After a successful verification, the producer adds all nodes of the abstract reachability graph to the certificate.

Definition 3.3 (Certificate from ARG)**.** Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph. The *certificate from ARG* $R^P_{\mathbb{C}^{\mathcal{A}}}$ is $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}}) = N$.

For the ARG shown in Fig. 3.4 the producer constructs the following certificate. As we will explain, the certificate is valid because it is constructed from an ARG generated during a successful verification, i.e., a well-formed ARG.

$$\left\{ \begin{array}{l} ((l_0, s: \top\ x: \top\ y: \top\ z: \top), q_0), ((l_1, s: \top\ x: \top\ y: \top\ z: 0), q_0), \\ ((l_2, s: \top\ x: -\ y: \top\ z: 0), q_0), ((l_3, s: \top\ x: -\ y: -\ z: 0), q_0), \\ ((l_4, s: \top\ x: -\ y: \top\ z: 0), q_0), ((l_5, s: \top\ x: -\ y: \top\ z: +), q_0), \\ ((l_6, s: \top\ x: 0+\ y: \top\ z: 0), q_0), ((l_7, s: \top\ x: 0+\ y: 0+\ z: 0), q_0), \\ ((l_8, s: \top\ x: 0+\ y: -\ z: 0), q_0), ((l_9, s: +\ x: \top\ y: \top\ z: \top), q_0), \\ ((l_{10}, s: +\ x: -+\ y: \top\ z: \top), q_0), ((l_{11}, s: +\ x: -+\ y: 0+\ z: \top), q_0), \\ ((l_{12}, s: +\ x: -+\ y: -\ z: \top), q_0), ((l_{13}, s: +\ x: -+\ y: \top\ z: \top), q_0), \\ ((l_{14}, s: \top\ x: \top\ y: \top\ z: \top), q_0) \end{array} \right\}$$

In general, from the definition of an abstract reachability graph (Definition 2.12) it only follows that the producer constructs syntactically correct certificates. However, we already demonstrated, not all syntactically correct certificates are valid. As we will see, also not all certificates constructed from an arbitrary ARG are always valid.

Reconsider the ARG, which is not compliant with the state space exploration (see Fig. 2.6). The certificate constructed from that ARG is not valid for program `SubMinSumDiv`, property automaton `nonneg`, and initial states $I = \{c \in C \mid \text{cs}(c) = l_0\}$. For example, for all program paths $c_0 \in paths_P(I)$ it does not cover the configuration sequence (c_0, q_0), the only configuration sequence for c_0. Moreover, the certificate constructed from the ARG constructed by the CPA algorithm during the analysis of program `SubMinSumDiv` with CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property automata `nonneg` (ARG in Fig. 2.5) is not safe, it contains abstract states considering automaton abstract state $q_\top$.

To apply configurable program certification in arbitrary settings, we need to guarantee that a producer generates valid certificates in case he sticks to our process for configurable program certification (cf. Fig. 3.1). In the process of configurable program certification, the producer only generates a certificate when the CPA algorithm successfully verified program P with CPA $\mathbb{C}^{\mathcal{A}}$. After a successful verification, the producer uses the ARG generated by the CPA algorithm to construct the certificate. Due to Proposition 2.8, we know that the ARG used by the producer for certificate construction has a special form, namely it is well-formed. Hence, to show that the producer part of our configurable program certification approach is thought through, we prove that certificates generated from well-formed ARGs are valid.

Proposition 3.2. *Let ARG $R^P_{\mathbb{C}^{\mathcal{A}}}$ for program P and enhancement $\mathbb{C}^{\mathcal{A}}$ of CPA $\mathbb{C}$ with property automaton $A = (Q, \delta, q_0, q_{\text{err}})$ be well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$. The certificate $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is valid for program P, property automaton $\mathcal{A}$, and initial states $[\![e_0]\!]$.*

Proof. Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$. By definition of $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$, we know that $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}}) = N$. Since $R^P_{\mathbb{C}^{\mathcal{A}}}$ is rooted (well-formed), we know that $e_0 \sqsubseteq root$. By meaning of $\sqsubseteq$, we know that $[\![e_0]\!] \subseteq [\![root]\!]$. By definition of paths, $paths_P([\![e_0]\!]) \subseteq paths_P([\![root]\!])$. From Lemma 2.7, $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}}) = N$, and $paths_P([\![e_0]\!]) \subseteq paths_P([\![root]\!])$, we know that certificate $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ covers at least one configuration sequence for each path $p \in paths_P([\![e_0]\!])$ and $\mathcal{A}$. Since $R^P_{\mathbb{C}^{\mathcal{A}}}$ is well-formed implies that $R^P_{\mathbb{C}^{\mathcal{A}}}$ is safe, by definition of $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ and definition of safety of $R^P_{\mathbb{C}^{\mathcal{A}}}$, it follows that $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is safe. Hence, $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$. □

So far, we presented the details of the producer part of the configurable program certification approach. In the next section, we continue with the certificate validation, which is performed by the consumer.

3.3 Consumer Certificate Validation

In our approach, the consumer inspects the certificate generated by the producer to decide whether a program is safe w.r.t. a property automaton and a set of initial states. We know that valid certificates guarantee program safety. Hence, the consumer tries to find out whether a given certificate fulfills the two properties of a valid certificate.

Recapture the proof of Lemma 2.7 (see Appendix pp. 255 f), which proved that the nodes of a well-formed ARG cover a configuration sequence per path. We remember that the inductive proof relies on the following two properties: all initial states are considered by the ARG nodes and the ARG nodes are closed under abstract successor computation.

We conclude that if the certificate covers the initial abstract state and if the certificate is closed under (abstract) successor computation, we can use the induction principle to show that a certificate covers a configuration sequence per path. Thus, to check the first property, the consumer examines the two presented conditions. Note that from the producer's point of view, the consumer checks the following. The producer explored the program behavior determined by the initial states and the producer properly stopped the exploration of states, i.e., the producer performed a legal state space exploration (our proposed consumer check). Therefore, the major part of the validation still matches our previous certificate validation [JW14] for a more restrictive concept of a CPA . To ensure the second property of a valid certificate, the consumer simply needs to inspect the states in the certificate, which is the same as the inspection of the set reached on producer side.

Like the producer, the consumer should examine program safety on the basis of abstraction. We think this will be cheaper. In contrast to the producer, the abstraction level of each validation is dictated by the certificate. For the validation, the consumer requires a validation configuration that provides the necessary operators, abstract successor computation and coverage check, for the certificate's abstraction and an algorithm that executes the different steps of our proposed certificate validation with the help of a suitable validation configuration.

3.3.1 Validation Configuration

From the previous considerations, we know that a validation configuration must provide three elements: a description of the abstraction in terms of an abstract domain, an operator for abstract successor computation, and a coverage check. We start with the definition and discussion of the *coverage check*, which inspects whether an abstract state is represented by the certificate.

Definition 3.4 (Coverage Check)**.** Let $D^{\mathcal{A}}$ be an abstract domain enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$. A function $\mathsf{cover}^{\mathcal{A}} : E^{\mathcal{A}} \times 2^{E^{\mathcal{A}}} \to \mathbb{B}$ is a *coverage check* if $\forall (e,q) \in E^{\mathcal{A}}, S \subseteq E^{\mathcal{A}} : \mathsf{cover}^{\mathcal{A}}((e,q),S) \implies [\![(e,q)]\!]^{\mathcal{A}} \subseteq \bigcup_{(e',q') \in S \wedge q \sqsubseteq q'} [\![(e',q')]\!]^{\mathcal{A}}$.

Next, we consider how a consumer can automatically derive such a coverage check from the verification configuration, an enhanced CPA. Every CPA enhanced with a property automaton $\mathcal{A}$ already provides an operator, the termination check operator, that looks similar to a coverage check. The termination check's task is to check when the exploration of a state may be stopped. Like a coverage check, it examines if a state is covered by a given set of states. The termination check seems to be a suitable coverage check. The following corollary confirms our observation and proves that a termination check of an enhanced CPA is also a coverage check.

Corollary 3.3. *Let $\mathbb{C}^{\mathcal{A}} = (D^{\mathcal{A}}, \cdot, \cdot, \cdot, \cdot, \mathsf{stop})$ be a CPA enhanced with property automaton $\mathcal{A}$. Then, the termination check* stop *is a coverage check.*

Proof. Let $(e,q) \in E_{\mathbb{C}^{\mathcal{A}}}$ be an arbitrary abstract state and $S \subseteq E_{\mathbb{C}^{\mathcal{A}}}$ be an arbitrary set of abstract states. Assume $\mathsf{stop}((e,q),S) = \mathit{true}$. By definition of an enhancement $\mathbb{C}^{\mathcal{A}}$ of a CPA $\mathbb{C}$, we know that $\mathsf{stop}((e,q),S) = \mathsf{stop}_{\mathbb{C}}(e, \{e' \mid (e',q') \in S \wedge q \sqsubseteq q'\}) = \mathit{true}$. Soundness of the termination check plus the definition of the concretization function let us infer that $[\![(e,q)]\!] = [\![e]\!] \subseteq \bigcup_{(e',q') \in S \wedge q \sqsubseteq q'} [\![e']\!] = \bigcup_{(e',q') \in S \wedge q \sqsubseteq q'} [\![(e',q')]\!]$. The termination check is a coverage check. □

The previous corollary tells us that it is sound to use the termination check. This does not mean that any termination check is is appropriate for certificate validation. Already in our previous work [JW14], we observed that in theory not all termination checks are mature enough.

For example, let us look at a variant $\widehat{\mathbb{L} \times \mathbb{S}}$ of the sign dataflow analysis $\mathbb{L} \times \mathbb{S}$. Its precision adjustment uses the original, static precision adjustment $\mathsf{prec}_{\widehat{\mathbb{L}\times\mathbb{S}}}(e, \pi, S) = \mathsf{prec}_{\mathbb{L}\times\mathbb{S}}(e, \pi, S)$ except for one abstract state, $\mathsf{prec}_{\widehat{\mathbb{L}\times\mathbb{S}}}((l_5, s : \top\ x : -\ y : -\ z : +), \pi, S) = ((l_5, s : \top\ x : -\ y : \top\ z : +), \pi)$. Furthermore, the termination check only returns true when the abstract state plus the set S of abstract states are of a particular shape and the abstract state is covered by a more abstract state, $\mathsf{stop}_{\widehat{\mathbb{L}\times\mathbb{S}}}((l, s), S) := s(y) \neq - \wedge (l \neq l_5 \vee |S| \leq 6) \wedge \mathsf{stop}_{\mathbb{L}\times\mathbb{S}}(e, S)$[2].

Assume the producer enhances $\widehat{\mathbb{L} \times \mathbb{S}}$ with the property automaton `pos@l5` instead of $\mathbb{L} \times \mathbb{S}$. Then, an exploration order of waitlist exists such that the CPA algorithm started with that variant of the enhanced CPA and initial abstract state $((l_0, \top_{\mathbb{S}}), q_0)$ successfully verifies program `SubMinSumDiv` and generates the ARG shown in Fig. 3.4. Hence, the certificate from the ARG shown in Fig. 3.4 is still a valid certificate for the CPA variant. Nevertheless, the stricter termination check cannot be used to detect that this (producer) certificate is valid. The reason is that the stricter termination check is not mature enough to detect that the certificate is closed under abstract successor computation. It fails to discover that all abstract successors are covered by the certificate. Next, we explain the reasons why the termination check fails to detect coverage.

First, the termination check $\mathsf{stop}_{\widehat{\mathbb{L}\times\mathbb{S}}}$ is not consistent with the partial order. For example, consider the abstract state associated with location l_8. The termination check $\mathsf{stop}_{\widehat{\mathbb{L}\times\mathbb{S}}}$ does not detect that the transfer relation successor $((l_9, s : +\ x : 0+\ y : -\ z : 0), q_0)$ of state $((l_8, s : \top\ x : 0+\ y : -\ z : 0), q_0)$ and CFA edge $(l_8, s := -y; , l_9)$ is covered by the set of ARG nodes, the certificate, i.e., $\mathsf{stop}_{\widehat{\mathbb{L}\times\mathbb{S}}}$ returns false, although a more abstract state $((l_9, s : +\ x : \top\ y : \top\ z : \top), q_0)$ in the set of ARG nodes exists.

Second, the termination check is not monotonic. For example, consider the abstract state associated with location l_3. The termination check does not return true when considering the transfer relation successor $((l_5, s : \top\ x : -\ y : -\ z : +), q_0)$ of state $((l_3, s : \top\ x : -\ y : -\ z : 0), q_0)$ and CFA edge $(l_3, z := -y; , l_5))$ plus the set of ARG nodes, the certificate. However, during verification the termination check $\mathsf{stop}_{\widehat{\mathbb{L}\times\mathbb{S}}}$ returned true for the abstract state $((l_5, s : \top\ x : -\ y : \top\ z : +), q_0)$ obtained by precision adjustment of that abstract successor and the set S that contains the states of the certificate that are associated with locations $l_0, l_1, l_2, l_3, l_4, l_5$. On the one hand, the termination check fails because the certificate contains more states, i.e., the certificate is more abstract than S. The termination check does not support the increase of the abstract state space during verification, which is reflected by the set reached, the set which finally defines the set of ARG nodes. On the other hand, the termination check even does not return true for the abstract successor $((l_9, s : +\ x : 0+\ y : -\ z : 0), q_0)$ and the set S that is sufficient to accept the more abstract state obtained after precision adjustment of that abstract successor. The more concrete abstract sign value $-$ for variable y prohibits the termination check operator from returning true. This termination check operator is vulnerable to precision adjustment effects.

We conclude that termination checks which are used for certificate validation should fulfill the two properties, consistency with partial order and monotonicity. In general, we call coverage checks which meet the above criteria *well-behaving*.

[2] We use $s(y)$ to denote the abstract sign value of variable y in sign abstract state s

Definition 3.5. Let $\mathsf{cover}^{\mathcal{A}} : E^{\mathcal{A}} \times 2^{E^{\mathcal{A}}} \to \mathbb{B}$ be a coverage check for abstract domain $D^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A}$. The *coverage check* $\mathsf{cover}^{\mathcal{A}}$ *is well-behaving* if it is

- consistent with the partial order, i.e., $\forall e \in E^{\mathcal{A}}, S \subseteq E^{\mathcal{A}} : \exists e' \in S : e \sqsubseteq e' \implies \mathsf{cover}(e, S)$, and
- monotonic, i.e., $\forall e, e' \in E^{\mathcal{A}}, S, S' \subseteq E^{\mathcal{A}} : e \sqsubseteq e' \wedge S \sqsubseteq S' \wedge \mathsf{cover}(e', S) \implies \mathsf{cover}(e, S')$.

Requiring a well-behaving coverage check is first of all a limitation of the generality of our approach. Validation configurations can only be derived automatically when the termination check is well-behaving, too. To support termination checks that are not well-behaving, we support a semi-automatic derivation of the validation configuration. In this semi-automatic derivation, the consumer must provide a coverage check, which extends the termination check such that it becomes well-behaving. However, such an extension may be less efficient. It may drop the performance of the validation procedure. Nevertheless, we claim that the well-behaving requirement is a minor restriction for the practical applicability of our configurable program certification approach.

For standard termination checks that are not well-behaving, we imagine that extensions for the termination checks are provided, which make them well-behaving. We even think that these standard termination checks can be replaced automatically with their extended version. Furthermore, we argue that termination checks that are not well-behaving are not very common in practice. To the best of our knowledge, only a few termination checks in the tool CPAchecker exist which are not well-behaving. We list them in the following paragraphs.

Some abstract domains may be used together with a termination check that always returns false. A proper extension of this termination check is the following coverage check $\mathsf{cover}(e, S) := \exists e' \in S : e \sqsubseteq e'$, which is identical to the most common termination check and for which we show that it is well-behaving.

The termination check of the `ARGCPA` does not fulfill the well-behaving property in case the `isTarget` property of an abstract state is not consistent with the partial order. For example, there exists abstract states e, e' with $e \sqsubseteq e'$ but only for e the `isTarget` property is true. Our implementation of the configurable program certification framework in CPAchecker only use the `ARGCPA` on the producer side to construct the ARG. For the validation configuration, the well-behaving property of the `ARGCPA`'s termination check is not important.

Also the termination checks of the `AssumptionStorageCPA` and the `MonitorCPA` are not well-behaving. In the first case, the abstract domain does not support a partial order. In the second case, the termination check is nondeterministic. It depends on the exploration order used for the second parameter S. Both CPAs do not adhere to our formal definition of a CPA and are currently not used by our configurable program certification approach.

In practice, the most commonly used termination check operator checks if an abstract state is covered by a more abstract state, i.e., $\mathsf{stop}(e, S) := \exists e' \in S : e \sqsubseteq e'$. The following corollary states that this standard termination check meets our well-behaving conditions.

Corollary 3.4. *Let $D^{\mathcal{A}}$ be an abstract domain enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$. The function $\mathsf{cover}^{\mathcal{A}} : E^{\mathcal{A}} \times 2^{E^{\mathcal{A}}} \to \mathbb{B}$ with $\mathsf{cover}(e, S) := \exists e' \in S : e \sqsubseteq e'$ is a well-behaving coverage check.*

Proof. Let $(\widehat{e}, q) \in E^{\mathcal{A}}$ be an arbitrary abstract state and $S'' \subseteq E^{\mathcal{A}}$ be an arbitrary set of abstract states. Assume $\mathsf{cover}((\widehat{e}, q), S'') = \mathit{true}$. From $\mathsf{cover}((\widehat{e}, q), S'') = \mathit{true}$, it

follows that there exists $(\widehat{e}', q') \in S$ with $(\widehat{e}, q) \sqsubseteq (\widehat{e}', q')$. It follows that $q \sqsubseteq q'$ and due to the meaning of $\sqsubseteq$ it holds that $[\![(\widehat{e}, q)]\!] \subseteq [\![(\widehat{e}', q')]\!] \subseteq \bigcup_{(e*,q*)\in S \wedge q \sqsubseteq q*} [\![(e*, q*)]\!]$. Hence, cover is a coverage check. By definition of cover the function cover is consistent with the partial order. Assume $e'' \sqsubseteq e$ and $\mathsf{cover}(e, S) = \mathit{true}$. From $\mathsf{cover}(e, S) = \mathit{true}$, it follows that there exists $e' \in S$ with $e \sqsubseteq e'$. Using transitivity of partial order $\sqsubseteq$, we get $e'' \sqsubseteq e'$. Hence, we conclude $\mathsf{cover}(e'', S) = \mathit{true}$ (definition of cover). Assume $S \sqsubseteq S'$ and $\mathsf{cover}(e, S) = \mathit{true}$. From $\mathsf{cover}(e, S) = \mathit{true}$, it follows that there exists $e' \in S$ with $e \sqsubseteq e'$. From $S \sqsubseteq S'$, we conclude that there exists $e''' \in S'$ with $e' \sqsubseteq e'''$. Thus, we get $\mathsf{cover}(e, S') = \mathit{true}$ (definition of cover). Operator cover is monotonic. □

Theoretically, the most commonly used termination check operator is well-behaving, but we observed two implementational issue. We noticed the first issue for the default configuration of the `CallstackCPA`. Due to performance reasons, in the Java implementation of the `CallstackCPA` two abstract states are only equal in case they are the same object and not when they represent the same data, i.e., the implementation uses the `equals()` method inherited from its superclass `Object`. In general, the problem occurs whenever a CPA uses a flat lattice and does not override the `equals()` method in the implementation of the abstract state. One problem is that the initial abstract state of the producer and the consumer can be different although they want to use the same state. The reason is that the initial abstract state contained in the certificate is read from disk and the initial abstract state provided to the validation algorithm is constructed independently. Another problem is that if the transfer relation computes the abstract successors of an abstract state and a CFA edge twice, the results may be different. The termination check already fails to detect that an abstract successor has already been computed, a property that must be recognized during certificate validation. To overcome the previous problem in the `CallstackCPA`, we allow the consumer to configure the abstract domain with a flat lattice that considers the data of an object to decide equality.

A second related issue concerns the `PredicateCPA`. Again, due to performance – cf. comments in the implementation – the implementation of the partial order of the abstract domain sometimes uses a fast check to decide whether a state is less abstract than another. This fast check is sufficient for verification, but neither trustworthy for the consumer nor does it consider all relations in the partial order. For the consumer, we implemented a coverage check that behaves as it was intended for the termination check.

Up to now, we discussed the problems regarding a reuse of the termination check. In contrast, a reuse of the transfer relation configuration for abstract successor computation is uncomplicated. Furthermore, we think that producers and consumers should at least agree on the abstract domain. The certificate fixes the abstract states anyway. Moreover, standard abstract domains like e.g. the domain for constant propagation [NNH05, p. 72], interval analysis [CC77], or predicate abstraction [GS97] have a widely accepted meaning. It would be inconvenient if the producer or the consumer interpret these abstract states differently. Generally, we assume that the producer and the consumer agree on the different meaning of the abstract states. We have everything at hand to describe how to derive the validation configuration. Nevertheless, before we explain how to derive the validation configuration from the verification configuration, namely from a CPA, we generally describe the structure of and the requirements on a validation configuration.

A validation configuration is related to a set of abstract states and can only inspect certificates that are consistent with those abstract states. It is given by a *configurable certificate validator* (CCV). As mentioned earlier a CCV must provide three elements: an enhanced abstract domain, which also considers the property automaton states, a

transfer relation, and a coverage check. For the enhanced abstract domain, we require the same constraints as for an enhanced abstract domain used by the producer. Furthermore, the transfer relation must overapproximate the semantics of the execution steps and the behavior of the property automaton.

Definition 3.6 (Configurable Certificate Validator)**.** A *configurable certificate validator* (CCV) $\mathbb{V}^{D^{\mathcal{A}}} = (D^{\mathcal{A}}, \rightsquigarrow^{\mathcal{A}}, \mathsf{cover}^{\mathcal{A}})$ for an abstract domain $D^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ consists of

- an enhanced abstract domain $D^{\mathcal{A}} = (C, \mathcal{E} \times \mathcal{Q}, [\![\cdot]\!]^{\mathcal{A}})$ ensuring Eq. 2.1,
- a transfer relation $\rightsquigarrow^{\mathcal{A}} \subseteq E^{\mathcal{A}} \times \mathcal{G} \times E^{\mathcal{A}}$ for abstract successor computation which ensures that $\forall (e,q) \in E^{\mathcal{A}}, (l, op, l') \in \mathcal{G} : \forall c \in [\![(e,q)]\!]^{\mathcal{A}} : c \xrightarrow{(l,op,l')} c' \implies \exists ((e,q),(l,op,l'),(e',q')) \in \rightsquigarrow^{\mathcal{A}}\colon c' \in [\![(e',q')]\!]^{\mathcal{A}} \wedge (\exists C_{\text{sub}} \subseteq C : (q, op, C_{\text{sub}}, q') \in \delta \wedge c' \in C_{\text{sub}} \vee q' = q_\top)$, and
- a coverage check $\mathsf{cover}^{\mathcal{A}} : E^{\mathcal{A}} \times 2^{E^{\mathcal{A}}} \to \mathbb{B}$.

Our definition of a CCV allows the consumer to specify his validation configuration independently of the producer. Though we think that it is inefficient, the consumer should successfully validate producer certificates when he uses the most precise transfer relation and coverage check. We suggest a different strategy to adjust the validation configuration to the certificate generated by the producer. As in our previous work [JW14], we (semi-)automatically derive the validation configuration from the producer's CPA. Semi-automation takes into account that the termination check is not always mature enough for certificate validation. Note that a derivation does not mean that the consumer must trust parts of the producer's implementation. The consumer may use his own or any trusted implementation of the relevant concepts for which he believes that they are compliant with the requirements of a CCV or CPA. Additionally, a consumer might even verify that a CCV or CPA implementation adheres to the requirements, which we imposed on them. A further option is presented by Besson et al. [BJP06]. Besson et al. propose to use the PCC idea for the validation of the consumer's certificate checker – the CCV in our case.

In the previous paragraphs, we already sketched the semi-automatic derivation, which results in a so called *configurable certificate validator for a CPA*. Given a CPA and a coverage check, the derived CCV consists of the CPA's abstract domain and transfer relation plus the provided coverage check. In a fully automatic derivation, which is used when the termination check is already well-behaving, the termination check is used as coverage check.

Definition 3.7 (Configurable Certificate Validator for CPA)**.** Let $\mathbb{C}^{\mathcal{A}} = (D_{\mathbb{C}^{\mathcal{A}}}, \Pi_{\mathbb{C}^{\mathcal{A}}}, \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}, \mathsf{prec}_{\mathbb{C}^{\mathcal{A}}}, \mathsf{merge}_{\mathbb{C}^{\mathcal{A}}}, \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ be an enhancement of CPA $\mathbb{C}$ and $\mathsf{cover} : E_{\mathbb{C}^{\mathcal{A}}} \times 2^{E_{\mathbb{C}^{\mathcal{A}}}} \to \mathbb{B}$ be a coverage check which is as least as precise as the termination check, i.e., $\forall e \in E_{\mathbb{C}^{\mathcal{A}}}, S \subseteq E_{\mathbb{C}^{\mathcal{A}}} : \mathsf{stop}(e,S) \implies \mathsf{cover}(e,S)$. The *configurable certificate validator for* $\mathbb{C}^{\mathcal{A}}$ *and* cover is $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover}) = (D_{\mathbb{C}^{\mathcal{A}}}, \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}, \mathsf{cover})$.

The previous definition describes how to (semi-)automatically build the validation configuration from the producer's verification configuration, his CPA. In the following, we ensure that our derivation of the validation configuration from a CPA constructs a proper validation configuration. We show that a configurable certificate validator for a CPA is indeed a CCV.

Algorithm 3: Validation algorithm for certificates

Input: A CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}} = ((C, (E, \top, \bot, \sqsubseteq, \sqcup), [\![\cdot]\!]), \rightsquigarrow, \mathsf{cover})$, initial abstract state $e_0 \in E$, certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}} \subseteq E$, program $P = (L, G_{\text{CFA}}, l_0)$
Output: Boolean indicator, if certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is valid

```
1 if ¬cover(e_0, C_{C^A}) then
2     return false
3 for each e ∈ C_{C^A} do
4     for each g ∈ G_CFA do
5         for each (e, g, e') ∈⇝ do
6             if ¬cover(e', C_{C^A}) then
7                 return false
8 return ¬∃(·, q) ∈ C_{C^A} : q = q_err ∨ q = q_⊤
```

Corollary 3.5. *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ enhanced with property automaton $\mathcal{A}$ and* cover *be a coverage check for abstract domain $D_{\mathbb{C}^\mathcal{A}}$. $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ is a configurable certificate validator for abstract domain $D_{\mathbb{C}^\mathcal{A}}$.*

Proof. By definition of CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, the abstract domain of the CCV is the abstract domain $D_{\mathbb{C}^\mathcal{A}}$ of CPA $\mathbb{C}^\mathcal{A}$. By definition of a CPA, $D_{\mathbb{C}^\mathcal{A}}$ fulfills Eq. 2.1. The CCV's transfer relation is the CPA's transfer relation. Let $(e, q) \in E^\mathcal{A}$ be an arbitrary abstract state, $(l, op, l') \in \mathcal{G}$ be an arbitrary edge, and $c \in [\![(e, q)]\!]$ be an abstract state. Assume that there exists a transition $c \xrightarrow{(l,op,l')} c'$. By overapproximation of the transfer relation (Eq. 2.2), there exists $((e, q), (l, op, l'), (e', q')) \in \rightsquigarrow$ with $c' \in [\![(e', q')]\!]$. By definition of enhancement, either $q' = q_\top$ or $(q, op, C_{\text{sub}}, q') \in \delta$ and $c' \in [\![(e', q')]\!] = [\![e']\!] \subseteq C_{\text{sub}}$. Hence, $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ is a configurable certificate validator for abstract domain $D_{\mathbb{C}^\mathcal{A}}$. □

We continue to describe how the consumer uses a CCV to check that a certificate is valid, i.e., it witnesses program safety.

3.3.2 Validation Algorithm

Like we require a meta algorithm to execute arbitrary configurable program analyses, we need a meta algorithm, the so called *validation algorithm*, to inspect arbitrary certificates. Our validation algorithm, which is a variant of our previous validation algorithm [JW14], is shown in Algorithm 3. The task of the validation algorithm is to determine for a given input program P, a certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$, an initial abstract state e_0, and a CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ whether certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is valid for program P, property automaton $\mathcal{A}$, and the set of initial states $[\![e_0]\!]$.

Hence, two aspects must be checked by our validation algorithm. Lines 1-7 check the first property: for each program path a configuration exists that is covered by the certificate. We already discussed that for this inspection two aspects must be examined: coverage of the initial states and closure under successor computation. The first two lines inspect if the initial states are considered by the certificate. In lines 3-7 the validation algorithm checks if the certificate is closed under successor computation. Since we only examine if a certificate is a witness for the input program, it is sufficient that the successor computation is closed for all successors of the input program. Hence, in line 4 we restrict

the transfer relation to the control flow edges of the input program. Finally, line 8 inspects whether the certificate is safe, the second property of a valid certificate.

Technically, we presented all details to get our configurable program certification ready to use. Yet, we have not convinced consumers that their validation process is tamper-proof (sound), i.e., it will only accept programs if they are indeed safe. Furthermore, producers only participate in the configurable program certification process when it ensures that if his process works out as intended, the consumer will accept the program, i.e., our configurable program certification is relatively complete. Next, we inspect these properties.

3.4 Properties of the Consumer Certificate Validation

We already stated in the introduction that our approaches must be sound and relatively complete. Soundness can only be established on the consumer side and, thus, by the validation algorithm. To ensure relative completeness, we already proved that a cooperative producer constructs valid certificates. It remains to be shown that if the consumer considers the same certificate and program as the producer, a cooperative consumer will accept the producer's certificate. More concretely, the validation algorithm, which performs the certificate validation, must accept the producer's certificate, i.e., it must be relatively complete. Next, we discuss when our approach guarantees these two properties.

First, we discuss soundness. Like the producer needs to consider a proper initial abstract state for verification, namely a state which consider the initial automaton state q_0, we require that the consumer also provides such a proper initial abstract state to the validation algorithm. Based on this assumption, we designed our validation algorithm s.t. it should accept valid certificates only. Due to Lemma 3.1, we know that a valid certificate witnesses program safety. Hence, to ensure soundness, we prove that the validation algorithm accepts only valid certificates. It only returns true when the certificate is valid for input program P, property automaton $\mathcal{A}$ considered by the abstract domain of the CCV, and initial states $[\![e_0]\!]$, the states represented by the initial abstract state e_0.

A valid certificate fulfills two properties: it covers one configuration sequence per path and it is safe. The following lemma states the first property: if Algorithm 3 returns true and it is started with a proper initial abstract state that considers the initial automaton state q_0, for each path at least one configuration sequence will be covered.

Lemma 3.6 (Configuration Sequence Coverage)**.** *If Algorithm 3 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ returns true, then certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ covers at least one configuration sequence per path.*

Proof. See Appendix pp. 257 f. □

Line 8 of the validation algorithm (Algorithm 3) checks safety of the input certificate. With the previous lemma, we now easily infer that if a proper initial abstract state is provided, the validation algorithm will accept only valid certificates.

Theorem 3.7. *If Algorithm 3 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ returns true, then $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$.*

Proof. Due to Lemma 3.6, certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ covers at least one configuration sequence per path. Since Algorithm 3 returns true and it may only return true in line 8, safety of certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ follows. By definition, certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$. □

Due to Lemma 3.1, we already know that a valid certificate is a witness for program safety. Thus, from the previous theorem we simply conclude that the certificate validation on the consumer side is sound.

Corollary 3.8 (Soundness). *If Algorithm 3 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ returns true, then $P \models_{[\![e_0]\!]_{\mathbb{C}^\mathcal{A}}} \mathcal{A}$.*

Proof. From the previous theorem, we get that $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is a valid certificate for P, $\mathcal{A}$, and $[\![e_0]\!]$. From Lemma 3.1, it follows that $P \models_{[\![e_0]\!]} \mathcal{A}$. □

Soundness of the validation algorithm is mandatory to get a reliable configurable program certification. However, to ensure applicability, the validation algorithm must accept the certificates generated by a producer who sticks to the configurable program certification process. Propositions 2.8 and 3.2 guarantee us that certificates constructed by a process conformant producer are valid. It is riskless to accept these certificates as long as the producer and the consumer consider the same program. Hence, a sound validation algorithm may accept these certificates. In the following, we show that the validation algorithm accepts those producer certificates in case the consumer configures the validation algorithm as proposed by the configurable program certification approach.

First, we ensure that the certificate validation algorithm terminates. We assume that the transfer relation always terminates for any input and the coverage check terminates when the elements in the certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ are finite. Then, the validation algorithm will terminate if all loops are bounded and the certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is finite. The producer uses the ARG nodes as certificate. Since the set of ARG nodes is finite by Definition 2.12, the certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is finite. Hence, the coverage check terminates and the number of iterations of the outermost loop are bounded. Furthermore, the innermost loop terminates because the transfer relation of any CPA provides only finitely many elements $(e, g, \cdot)$ for any abstract state e and control flow edge g. The loop in the middle is only bounded when the number of control flow edges of input program P is finite, i.e., program P is finite. Note that the CPA algorithm (Algorithm 2) only terminates when the input program P is finite. Thus, in the configurable program certification approach the producer constructs certificates for finite programs only. Assuming that input program P is finite does not restrict the approach. With these arguments at hand, we show that the validation algorithm terminates for finite programs and certificates.

Lemma 3.9 (Termination). *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *and program $P = (L, G_{\text{CFA}}, l_0)$ be finite. Then, Algorithm 3 started with $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, P, initial abstract state $e_0 \in E_{\mathbb{C}^\mathcal{A}}$, and finite certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ terminates.*

Proof. Algorithm 3 terminates when the certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ is finite, the number of program edges is finite, and for each pair $(e, g) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}} \times G_{\text{CFA}}$ only finitely many elements $(e, g, e') \in \rightsquigarrow$ exists.[3] By assumption, the certificate and the number of program edges are finite (finite certificate and definition of finite program). Since the transfer relation is a transfer relation of a CPA (definition of $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$), it follows from Eq. 2.3 that $\forall (e, g) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}} \times G_{\text{CFA}} : \exists n \in \mathbb{N} : |\{(e, g, e') \in \rightsquigarrow\}| \leq n$. Algorithm 3 terminates. □

[3] We assume that the result of each CPA operator can be computed within a fixed time bound when its result is a single element or a finite set.

The previous lemma ensures that the validation algorithm terminates. Now, we need to show that the validation algorithm indeed accepts the certificate constructed by a cooperative producer. Of course, the consumer must not use any configurable certificate validator that fits to the certificate's domain. For example, a CCV is allowed to use a transfer relation that maps any pair of abstract predecessor and control flow edge to the top state. Yet, such a transfer relation is not helpful for validation. This can easily be observed when considering the certificate constructed from the ARG in Fig. 3.4. Similarly, a coverage check that always returns false is valid, but not valuable. To ensure that the CCV is mature enough for certificate validation, we proposed to derive the CCV from the CPA used by the producer during verification. Nevertheless, we already discussed in Section 3.3.1 that in some cases we need to extend the producer's termination check to get an appropriate coverage check, namely a well-behaving coverage check, for validation. Furthermore, a certificate cannot cover more program behavior than considered by its construction. Hence, validation will likely fail if the consumer uses an initial abstract state that is more abstract than the one used by the producer, but the consumer may always use a more concrete one. In the following, we ensure that in case the consumer adheres to the previous restrictions plus the producer and the consumer consider the same program, then the validation algorithm will accept the certificate produced by a cooperative producer. The validation algorithm is relatively complete.

Theorem 3.10 (Relative Completeness)**.** *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving,* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^{\mathcal{A}}$ *of CPA* $\mathbb{C}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. Algorithm 3 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and certificate* $\mathrm{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

Proof. See Appendix p. 258. □

Until now, we only considered the semi-automatic validation. The consumer must provide a coverage check, which extends the producer's termination check to include the well-behaving properties. We already proved that every termination check operator is also a coverage check operator. Hence, if the termination check operator is well-behaving, e.g., like the frequently used termination check $\mathsf{stop}(e, S) := \exists e' \in S : e \sqsubseteq e'$, we can automatically derive the complete validation configuration from the producer's verification configuration. In this case, the automatically derived configuration is sufficient for a successful consumer validation in our configurable program certification approach.

Corollary 3.11. *Let* $\mathbb{C}^{\mathcal{A}}$ *be a CPA,* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ *be a configurable certificate validator for* $\mathbb{C}^{\mathcal{A}}$ *and* $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}$ *which is well-behaving,* P *be a program, and* $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$ *be an initial abstract state. If Algorithm 2 started with CPA* $\mathbb{C}^{\mathcal{A}}$*, initial abstract state* e_0*, initial precision* $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$*, and program* P *returns* $(\mathit{true}, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$*, then Algorithm 3 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$*, initial abstract state* e_0*, certificate* $\mathrm{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$*, and program* P *returns true.*

Proof. From Corollary 3.3, we know that $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}$ is a coverage check. Hence, $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ is a CCV. From Proposition 2.8, we know that $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is well-formed for e_0. Since Algorithm 2 terminates, we conclude that P is finite. The corollary follows from the previous theorem. □

Summing up, if the producer successfully checks a program P with CPA $\mathbb{C}^{\mathcal{A}}$ that uses a well-behaving termination check and a proper initial abstract state e_0, then the consumer can automatically derive the CCV from the producer's CPA, his validation of the

producer's certificate for program P considering e_0 succeeds and guarantees safety of program P w.r.t. $\mathcal{A}$ and $[\![e_0]\!]$. Our proposed configurable program certification approach meets the automation, soundness, and relative completeness requirement from the introduction.

Note that these results can easily be transferred to program safety. Simply add another restriction to the initial abstract state that ensures that the initial abstract state represents all concrete states whose control state is the initial program location. Next, we study the effectiveness of our configurable program certification approach.

3.5 Evaluation

In the previous section, we proved soundness and relative completeness. These two properties are mandatory for the application of configurable program certification, but they do not reveal whether a consumer profits from certificate validation. To profit from configurable program certification, certificate validation must be efficient w.r.t. execution time and memory consumption. However, it is difficult to determine this property theoretically. On the one hand, when the merge operator never combines an abstract state during combination, the size of the certificate corresponds with the outermost loop in the verification algorithm (Algorithm 2) and certificate validation becomes similar to verification. Thus, we think that the worst case bounds for validation and verification might be identical or at least very similar. On the other hand, we think that reading the certificate is an important aspect of validation in practice. Since the speed of storage devices and processing units differ in order of magnitudes, we think it is difficult to derive a reliable, theoretical statement about efficiency. That is why we decided to compare validation with verification in a practical experiment.

In the following, we study if and when is certificate validation efficient, i.e., it (significantly) outperforms verification w.r.t. execution and memory consumption. Furthermore, we try to predict the performance of certificate validation upon our insights from verification. To apply configurable program certification in practice, two further aspects must be considered. First, certificates should not be much larger than the program. The size of a certificate is important because it must be stored together with the program. Second, consumers observe the execution times of the tool they run instead of the time for the verification algorithm and the validation algorithm plus certificate reading. Thus, our configurable program certification approach should still perform well when the complete tool execution times of verification and validation are compared. Hence, we also study these two further aspects. Finally, we examine whether parallelization can be used to improve certificate validation. Moreover, we identify which operations of the certificate validation should be improved. Before we present our evaluation results, we introduce the evaluation setting.

3.5.1 Evaluation Setting

In this section, we present the verification tasks, describe the infrastructure used for evaluation, and briefly mention how we generated the certificates for the verification tasks and how we derived the validation configuration from a verification task. We start with the presentation of the verification tasks. Each verification task consists of an analysis configuration, a program, and a property. Note that we do not configure initial abstract states because this is currently not supported by the tool CPAchecker. In CPAchecker the

analysis configuration determines the initial abstract state, typically our configurations provide the most abstract state exclusively considering the initial program location.

Configurations Since configurable program certification is a general approach applicable to various configurations (CPAs), we want to evaluate our basic CPC approach with different abstract domains and analysis types. For our configurations, we can select from the following seven basic abstract domains:

- an interval domain $\mathbb{I}$ [CC77] abstracting a variable's value by a lower and an upper bound on the possible concrete values,
- an octagon domain $\mathbb{O}$ which is for example used by the industrial analyzer Astrée – we use a realization based on the Apron library [JM09a] –,
- a predicate domain $\mathbb{P}$, an abstract domain used by several participants of the software verification competition, we use predicate abstraction with adjustable block encoding [BKW10] and utilize the SMT solver SMTInterpol [CHN12] version 2.1-238-g1f06d6a-comp for abstraction computation and interpolation,
- a reaching definition domain $\mathbb{R}$ [NNH05, p. 42], well-known from compiler optimization, which remembers for each variable the possible definition points of the most recent definition,
- the sign domain $\mathbb{S}$ already known from Chapter 2,
- an uninitialized variable domain $\mathbb{U}$ tracking which variables have not been initialized, and
- a value abstract domain $\mathbb{V}$ used in constant propagation [NNH05, p. 72] or explicit model checking [DKW08, BL13], which tracks a concrete value per variable.

To study combined abstract domains and to more easily configure intermediate analyses with precisions between dataflow analysis and model checking, we also looked at two combined abstract domains $\mathbb{SI}$ and $\mathbb{VR}$, standard product combinations of two of the above domains. Note that the main purpose of these combined domains is to allow an easy configuration of intermediate analyses. We are not claiming that the domains are necessarily useful. As soon as we configure flow-sensitive analyses, e.g., dataflow analyses, intermediate analyses, or model checking based analyses, we always combine the mentioned abstract domains with the location domain $\mathbb{L}$ (see p. 25) and a callstack domain. On a very abstract level the callstack domain remembers for all active subroutines (called functions) where to continue after the subroutine is finished.

Next to the different domains, we consider five different classes of analyses. Our coarsest analysis is flow-insensitive. This analysis computes a single abstract state. Hence, its merge operator always computes the join of two abstract states and the termination check operator returns true if the abstract state is covered by a more abstract state. Due to the restrictions imposed by the implementations of the abstract domains, we could only use the reaching definition domain to build a flow-insensitive analysis.

We use the concept of a dataflow analysis to configure analyses that are only flow-sensitive. Now, the merge operator only merges abstract states when they agree on their location and callstack states. The termination check remains the same as for flow-insensitive analyses. In our experiments we consider dataflow analyses for all abstract

domains except for the predicate and octagon domain. The latter domains are solely used to experiment with non-static precision adjustment.

We use a special form of trace partitioning [MR05, RM07] to configure intermediate analyses, which are more precise than the previous dataflow analyses, but which are not fully path-sensitive. As already mentioned, we use the two combined abstract domains $\mathbb{SI}$ and $\mathbb{VR}$ to build these intermediate analyses. To get the intermediate analysis, we only need to adapt the merge operator of the dataflow analysis. Now, abstract states are only combined, i.e., joined, if additionally the first elements, either the sign abstract states or the value abstract states, are identical.

The most precise analysis type is fully path-sensitive and never merges different abstract states. We consider two types of fully path-sensitive analyses: model checking and CEGAR model checking. Both never combine abstract states and stop when an element is covered by a more abstract state. The difference between these two types is that the model checking analyses like all previous analyses never adjust precisions. In contrast, in CEGAR model checking the precision adjustment operator often weakens its input abstract state. Additionally, a technical difference exists. The CEGAR model checking analyses use multi-edges, i.e., the program model already groups sequences of edges. To properly examine properties beyond non-reachability of error locations or assertion failure, model checking and less precise analyses use the concept of dynamic multi-edges.

Again, we exclude the predicate and the octagon domain from model checking. For each of the remaining domains we used its model checking configuration for evaluation. CEGAR model checking is only available for predicate, octagon, and value analysis. The reason is that for simplicity we utilize counterexample guided abstraction refinement [CGJ+00] with lazy refinement [HJMS02] to determine the behavior of the precision adjustment and such a refinement is only implemented for these three domains. Although refinement does not directly fit into our framework, we are confident that we could have modeled an analysis in our framework that produced the same result as the respective analysis with refinement.

For the predicate domain, the refinement determines the set of predicates considered to compute the predicate abstraction at loop heads. Refinement for the value domain $\mathbb{V}$ and the octagon domain $\mathbb{O}$ identifies the variables whose values must be tracked. Furthermore, for these two domain we apply refinement selection [BLW15b] and path prefix slicing [BLW15c]. For the predicate analysis we do not use path prefix slicing, and thus refinement selection, because it is not compatible with adjustable block encoding. To properly configure the refinement selection, we performed a pre-evaluation which runs each analysis with a set of promising selection strategies. We considered the following, promising refinement selection strategies: `default`, `none`, `domain_min`, `length_min`, `length_max`, and the configuration used in the SV-COMP. Based on the pre-evaluation, for the explicit value analysis we select the strategy `domain_min`, which performed best w.r.t. speed and the number of analysis tasks that could be verified. For the octagon analysis, we choose `length_max` which is the best compromise w.r.t. number of solved tasks and speed.

In total, we evaluate with 20 CPA configurations.

Programs and Properties To evaluate our configurable program certification approach, we select a subset of the programs from the well-established software verification competition (SV-COMP) benchmark, which can be handled by all of the selected CPAs[4]. In detail, we chose the categories `ControlFlow`, `ECA`[5], `ProductLines`, and the subcate-

[4]Some CPAs are less mature than others and e.g. cannot deal with pointers.

[5]Event-Condition-Action Systems

Table 3.1: Number of Verification Tasks per CPA

Flow-Insensitive Analysis												
		$\mathbb{R}$	1078									
Dataflow Analysis												
$\mathbb{I}$	1073	$\mathbb{R}$	990	$\mathbb{S}$	1138	$\mathbb{U}$	1028	$\mathbb{V}$	1151	$\mathbb{SI}$	1072	$\mathbb{VR}$	1042
Intermediate												
										$\mathbb{SI}$	964	$\mathbb{VR}$	536
Model Checking												
$\mathbb{I}$	789	$\mathbb{R}$	287	$\mathbb{S}$	853	$\mathbb{U}$	1087	$\mathbb{V}$	634	$\mathbb{SI}$	746	$\mathbb{VR}$	476
CEGAR Model Checking												
				$\mathbb{O}$	645	$\mathbb{P}$	503	$\mathbb{V}$	588				

gory `loops` from category `Loops` as considered in the 2016th SV-COMP[6] [Bey16]. Since certification requires that the program is proven to be correct w.r.t. a property, we only select those programs for which the encoded property is known to be true. Furthermore, we use our example program `SubMinSumDiv`. In theory, we may evaluate each of the 20 configurations on all of the 1151 programs. In practice, we excluded for each configurations those programs that cannot be verified in 15 minutes. Additionally, we excluded the `Problem05*` programs from model checking with the abstract domain $\mathbb{SI}$ since certificate construction failed after 20 minutes, a timeout one third longer than the official timeout. Furthermore, we do not consider the `Problem03*` programs in the predicate analysis because the certificates become too large (several 100 MB). Table 3.1 gives an overview of the number of programs considered per configuration.

Next, we describe the properties checked on the selected programs. For our example program and the sign analysis, we check the property `pos@l5`. Analyses based on the uninitialized variables domain $\mathbb{U}$ check that only initialized variables occur in expressions. For the analyses applying CEGAR model checking, we considered the SV-COMP property `Error Function Unreachability`, i.e., all calls to `__VERIFIER_error(int)` are not reachable. The analyses for the combined domains simply check the combination of the properties for the single properties. For the remaining analyses, it was difficult to get realistic properties, thus, we check more or less artificial properties. The reaching definition based analyses inspect that a certain variable is initialized at most once. Similarly, other analyses ensure that at a certain location the abstract value of a certain variable is upper bounded by a given abstract value.

Execution Set Up Our configurable program certification approach is integrated into the software analysis tool CPAchecker. For evaluation, we used the CPAchecker revision 23042 available in the runtime_verification branch[7]. All experiments are executed on machines with Intel Xeon E5-2650 v2 CPUs at 2.6 GHz and with 135 GB of RAM. The Java version was Java HotSpot(TM) 64-Bit Server VM 1.8.0_101. Each verification and validation task is restricted to only use 2 CPU cores and 15 GB of RAM. The time limit was set to 15 minutes of CPU time. We used the benchmarking evaluation framework BenchExec [BLW15a] to enforce these restrictions.

All certificates were constructed once before the evaluation. Certificate generation was simple, we just had to write the reached set (the ARG node set) to disk. This took never

[6] `https://github.com/sosy-lab/sv-benchmarks/tree/svcomp16`
[7] `https://svn.sosy-lab.org/software/cpachecker/branches/runtime_verification/`

Table 3.2: Extract of the efficiency examination of the certificate validation w.r.t. verification which looks at the best and total improvement of execution times and memory consumption

CPA	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{R}$	342.95	2.16	159.05	3706	1311.7	0.35
	13384.58	480.65	27.85	494847.8	321089.4	0.65
$\mathbb{I}$	308.01	0.15	2071.32	3496.8	212.6	0.06
	18626.48	18510.39	1.01	879860.9	811305.6	0.92
$\mathbb{SI}$	403.09	0.17	2392.25	3498	206.2	0.06
	32917.31	30960.39	3.21	1097817.7	1057072.2	0.96
$\mathbb{SI}$	1.74	0.14	11.98	2163	583.6	0.27
	19488.76	37146.18	0.52	1225311.2	1325337.7	1.08
$\mathbb{VR}$	336.23	63.26	5.32	3202	344.9	0.11
	9332.39	24637.32	0.38	564549.8	703871.4	1.25
$\mathbb{O}$	63.10	3.80	16.61	3939.8	1135.5	0.29
	36293.32	63677.21	0.57	795084.3	684812.4	0.86
$\mathbb{V}$	484.96	0.95	510.59	4304.9	270.8	0.06
	23505.46	7655.15	3.07	686101.9	448909.4	0.65

more than 190 s. For comparison, the slowest verification required about 735 s.

The consumer's validation configuration are almost automatically derived from the verification configuration. Only for the callstack CPA and the predicate CPA more precise coverage check components must be configured. Note that the producer verification only constructs abstract reachability graphs when considering one of the three CEGAR based model checking analyses. For the remaining configurations, the abstract reachability graph is not necessary for verification and we use a more efficient variant of the CPA algorithm (cf. Algorithm 1), which does not generate the ARG.[8] In the following, we examine the average of 10 experimental executions.

3.5.2 RQ 1: How Does the Basic CPC Certificate Validation Perform Compared to Verification?

One requirement stated in the introduction was efficiency. Certificate validation should be (significantly) faster than the verification and should require less memory. In the following, we study if and when validation is efficient. We compare the validation time, certificate reading plus the time for the execution of the validation algorithm, with the verification time, the time for the execution of the CPA algorithm. Since a consumer typically observes the execution time of the analysis tool, the total time, we also compare these total times. Additionally, we compare the memory consumption, i.e., the heap and non-heap memory used by the software analysis tool when verifying the program or validating the certificate.

We start to study the best improvement each analysis can achieve as well as the overall improvement. Table 3.2 shows the verification time V_P, the certificate validation

[8] Since the ARG nodes are identical with the final reached set of the CPA algorithm and the certificate from ARG becomes the set of ARG nodes, Algorithm 1 is sufficient.

time V_C, their speed-up $\frac{V_P}{V_C}$, the memory M_P used in verification, the memory usage M_C of certificate validation, and the memory decrease $\frac{M_C}{M_P}$. Times are given in seconds and memory usage is given in MB. For each analysis in which certificate validation can be at least 5-times faster than verification, Tab. 3.2 compares the fastest validation time achieved for the analysis with the respective verification time and the smallest amount of memory consumed by certificate validation of that analysis with the memory consumption of the respective verification. Additionally, it shows the sums of the times and memory for each of these analyses. Double lines are used to separate the analysis types. Analysis types are presented in the same order as in Tab. 3.1. The complete table, which display these numbers for all analyses, can be found in the appendix (see Tab. B.1).

Looking at Tab. 3.2, we observe that tasks for all analyses types except for model checking exist that achieve significant speed-ups and a significant memory decrease. However, not all analyses and domains contain tasks for which validation is significantly more efficient. The extreme cases are the two analyses considering the uninitialized variables domain $\mathbb{U}$. For these analyses, the certificate validation is never faster than verification. Furthermore, when considering the sums overall tasks of each analysis, the results rarely indicate that certificate validation improves significantly on verification. Sometimes verification even becomes better than validation.

In Tab. 3.2, we experienced a big difference between what is possible and what is achieved in sum. In principle, a few extremes could have caused this big difference. Thus, we look at the results per task in more detail.

First, we look at the validation and verification times in more detail. Figure 3.5 shows the comparison of the verification and validation times for the flow-insensitive tasks, the dataflow analysis tasks, the model checking tasks, and the CEGAR model checking tasks. We left out the diagram for the intermediate tasks, which can be found on the left of Fig. B.1, because it is similar to the diagram of the dataflow analysis tasks. All times are given in seconds. The black line is the identity function and the dashed line marks a speed-up of 10, i.e., it is the function $V_C = 0.1 \cdot V_P$.

Looking at Fig. 3.5, many data points are close to the solid line or are above the line. In these cases, certificate validation is not much faster or even slower than verification. Solely, in the diagrams for the flow-insensitive analysis and CEGAR model checking tasks many data points are significantly below the solid line and sometimes even close to or below the dashed line, the marker for a speed-up of 10. For flow-insensitive and CEGAR model checking tasks, validation is regularly much faster. This is supported by a detailed inspection of the results. Only 60 tasks are 10-times or more times faster than verification, i.e., achieve a significant speed-up. Most of the tasks (36) belong to the flow-insensitive reaching definition analysis and another 18 tasks use CEGAR model checking.

Unfortunately, certificate validation is not always faster than verification. Thus, we like to know when is validation faster and how to predict this from the verification. Let us start with the flow-insensitive analysis tasks. Inspecting our results, we observe that the speed-up correlates with the number of merges as well as with the time the analysis spent on merging. The computed correlation factors are 0.987 and 0.998, respectively. Often, we get a good underapproximation of the speed-up when dividing the ratio of analysis time to merge time by 10.

For the dataflow analyses, we observed a correlation between speed-up and the ratio of the number of computed transfer successors to the size of the reached set, the number of ARG nodes. We computed a correlation factor of 0.977. Dividing the ratio of the number of computed transfer successors to the number of ARG nodes by a factor of 3.1 gives in most of the tasks a good underapproximation of the observed speed-up. We think

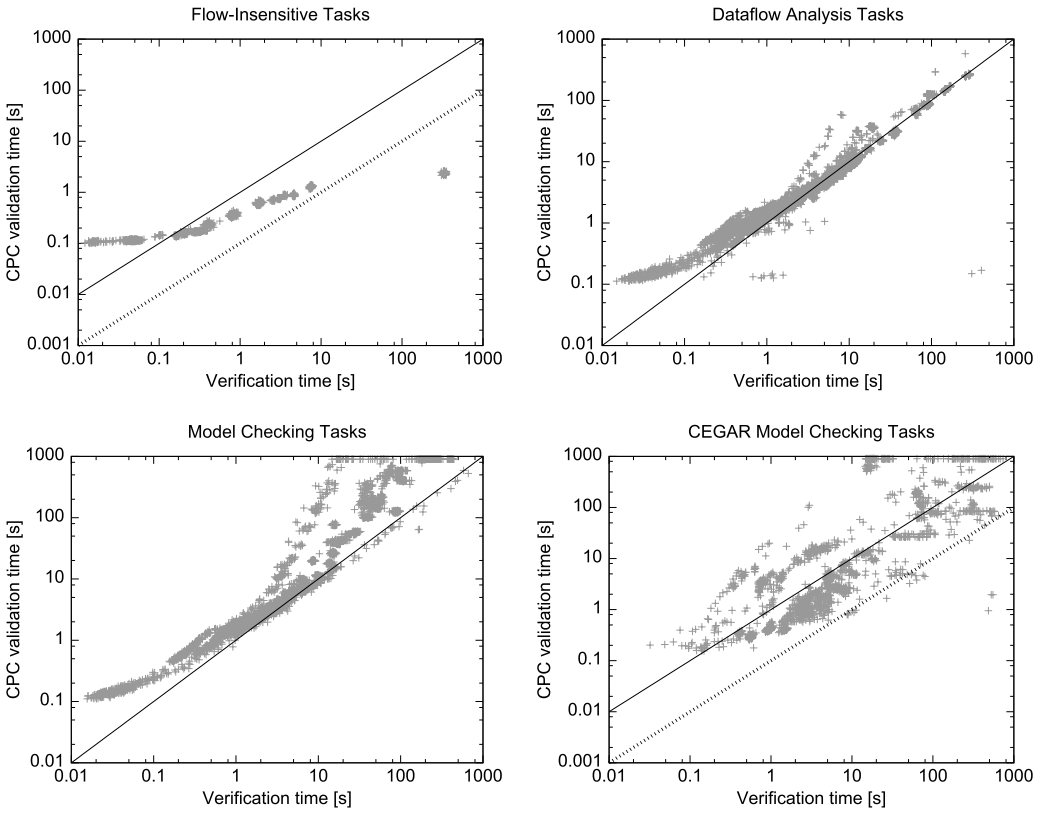

Figure 3.5: Comparison of the producer verification time with the consumer time for validation of the certificate from the producer's ARG

that the reason why the mentioned ratio is appropriate is that it captures the additional computations performed by the analysis. On the one hand, it includes the number of merges. After a merge, transfer successors are recomputed. On the other hand, it also tracks how many recomputations are caused by a merge.

The same ratio also correlates with CEGAR model checking except for the predicate analysis, in which we have to use an expensive extension of the termination check operator. For the set of octagon and value CEGAR model checking tasks, the correlation factor is 0.945. In case of CEGAR model checking, the ratio estimates the additional load caused due to lazy refinement, i.e., the part of the explored state space that is deleted during refinement. Dividing the ratio by 30 often worked for our CEGAR model checking tasks to underapproximate the real speed-up.

When recapturing the execution of the validation and the verification algorithm for model checking, which never adjusts precisions and never merges, we recognize that more or less these algorithms perform the same tasks. Their execution is nearly identical. Since validation must also read certificates, it is obvious that validation is likely slower than verification. Thus, we decided not to further investigate on a predictor for the validation behavior w.r.t. model checking tasks.

Unfortunately, we did not find a relation between verification and validation for the intermediate tasks. We think that a prediction depends on whether the intermediate task

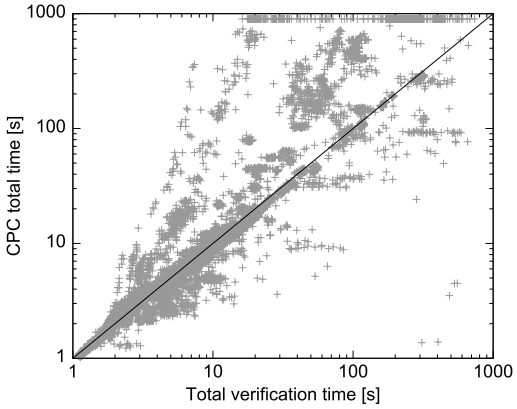

Figure 3.6: Comparing the total execution time of the verification with the total execution time of the respective certificate validation

acts similar to a dataflow analysis or similar to model checking. This seems to be difficult to predict from the verification itself and may depend on the program and the analysis.

Next, we consider the total times. Figure 3.6 compares for all tasks the total execution time of the software analysis tool CPAchecker for the verification run with that time of the certificate validation run. As always, times are given in seconds.

Like in the comparison of the verification and validation times, we see that the data points are widely spread in the diagram in Fig. 3.6. Some points are far above or below the line and others are close to the line. Since some points are significantly below the line, significant speed-ups can still be achieved. Additionally, we notice that the diagram starts with 1 s instead of 0.01 s or even 0.001 s. We conclude that setting up the infrastructure for analysis or validation, e.g., parsing the program, takes at least about 1 s. When the validation or verification takes less or similarly long as the set up, a potential speed-up is likely destroyed by the additional set up time. In general, the speed-ups for the total times are lower.

Finally, we study the relation of the memory consumption. Figure 3.7 depicts the comparison of the memory consumption of the verification and the certificate validation for the flow-insensitive tasks, the dataflow analysis tasks, the model checking tasks, and the CEGAR model checking tasks. Again, the diagram for the intermediate tasks can be found on the right of Fig. B.1. It is similar to the diagram of the dataflow analysis tasks.

Considering the top left diagram in Fig. 3.7, all data points are close to the line or significantly below. For flow-insensitive analyses memory consumption is typically reduced by certificate validation, sometimes even extremely. Getting to the dataflow analysis tasks (top right diagram), we see that the data points are often either closely below the line or above the line. Significant improvements do not occur that often and sometimes the memory usage is even worse for certificate validation. A similar observation can be made for the intermediate analysis tasks. For the model checking tasks, memory consumption is rarely decreased by certificate validation. Most of the data points in the bottom left diagram of Fig. 3.7 are above the line. The picture for CEGAR model checking is more diverse. A data point can be extremely above or below the solid line. Hence, memory consumption for CEGAR model checking tasks is sometimes efficiently reduced by certificate validation, but need not be. In general, we observe that improvements of the execution time and memory usage are related. Often, either both are decreased or

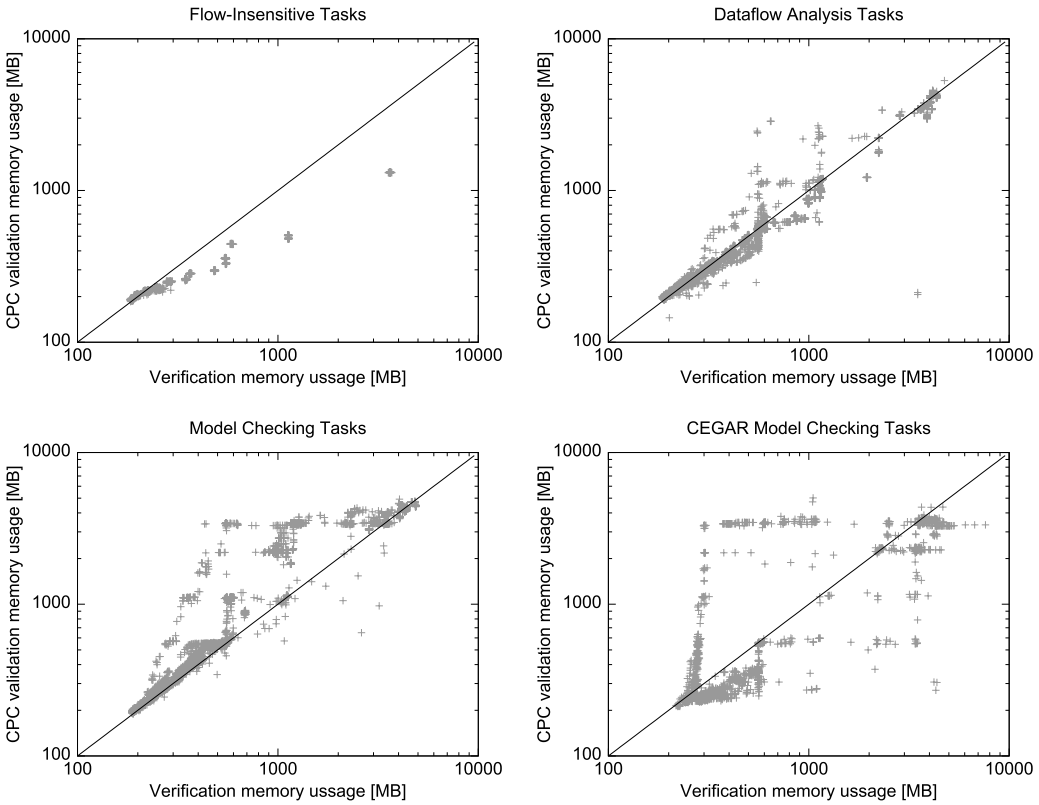

Figure 3.7: Comparison of the memory consumption of the producer verification with the memory consumption of the validation of the certificate from the producer's ARG

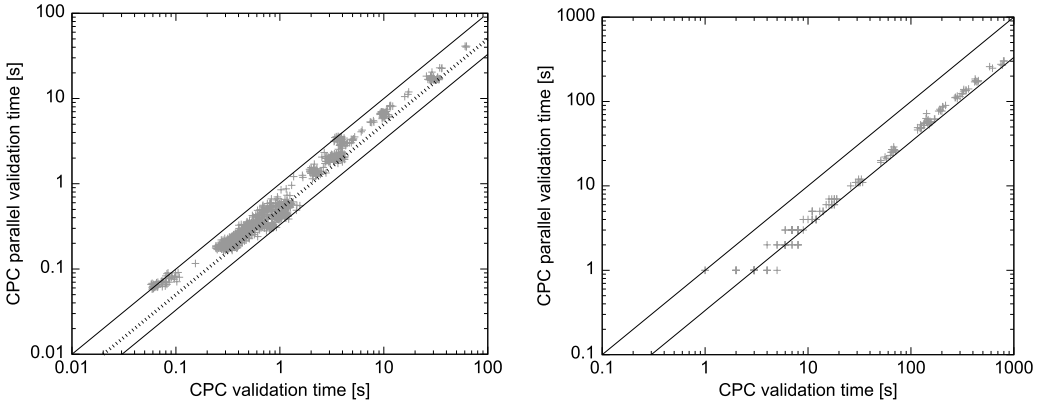

Figure 3.8: Comparing the validation times of sequential certificate validation and parallel validation with three threads. The left diagram shows the results for the reaching definition dataflow analysis tasks and the right one those for the reaching definition model checking tasks.

none of them.

All in all, the consumer part, the certificate validation, can be extremely efficient in the basic configurable program certification approach. Especially, flow-insensitive analysis tasks and CEGAR model checking tasks seem to profit most from certificate validation. However, certificate validation is not very efficient in general.

3.5.3 RQ 2: Does Parallelization Help?

Many of today's devices provide more than one processor unit. To efficiently utilize the available compute power, sequential execution is no longer sufficient. Algorithms must become ready for parallel execution. In this section, we want to examine whether we can profit from parallel certificate validation.

Reconsidering the certificate validation algorithm (Algorithm 3), we observe that its three loops are excellent candidates for parallelization. So far, we only provide an implementation which parallelizes the outermost for loop of the certificate validation algorithm. Unfortunately, many implementations of the different analysis configurations do not support parallel validation. In most of the configurations, the implemented transfer relation is a subclass of the `ForwardingTransferRelation`. This `ForwardingTransferRelation` stores common information required by various subroutines of the transfer successor computation in object variables instead of passing it via parameters. Additionally, the transfer relation computing the successor automaton state uses timers not designed for parallel access. Furthermore, the SMT solver used e.g. in the transfer relation of the predicate analysis does not cope well with parallel access. These are more or less practical issues that restrict our evaluation to the reaching definition dataflow analysis and to reaching definition model checking, the only two analysis configurations that support a parallel certificate validation in practice.

For our experiments, we restricted the CPU time and not the wall time. Parallelization distributes the CPU time among multiple CPU (cores), but does not reduce the complete CPU time, the sum of the CPU times overall used CPU (cores) remains the same. Validation tasks that timed out for the sequential validation likely time out for the parallel validation. Thus, in the following we only consider tasks for which neither sequential nor parallel validation times out. Furthermore, due to the very limited task set we experimented less thoroughly and compare only the results of a single experimental execution run on an Intel Core i5-2400 at 3.10 GHz. As before, the sequential validation was restricted to 2 CPU cores. For the parallel validation, we utilized 4 CPU cores, three threads were dedicated to the validation algorithm only.

Figure 3.8 compares the sequential validation time with the parallel validation time. Both times are given in seconds and represent the wall times for the complete validation process starting with certificate reading and ending with the complete execution of the validation algorithm. The left diagram shows the results for the reaching definition dataflow analysis tasks and the right diagram presents the results for reaching definition model checking.

We observe that in the left diagram most of the data points are between the upper solid line and the dashed line. For the reaching definition dataflow analysis tasks, the speed-up of parallelization is often at most two. In contrast, in the right diagram most of the data points are close to the lower solid line. In case of reaching definition model checking, parallelization often comes close to the theoretical optimum of three[9]. Hence,

[9] Note that we assume that an improvement beyond the theoretical bound of three (lower solid line) is caused by deviations in the experimental executions.

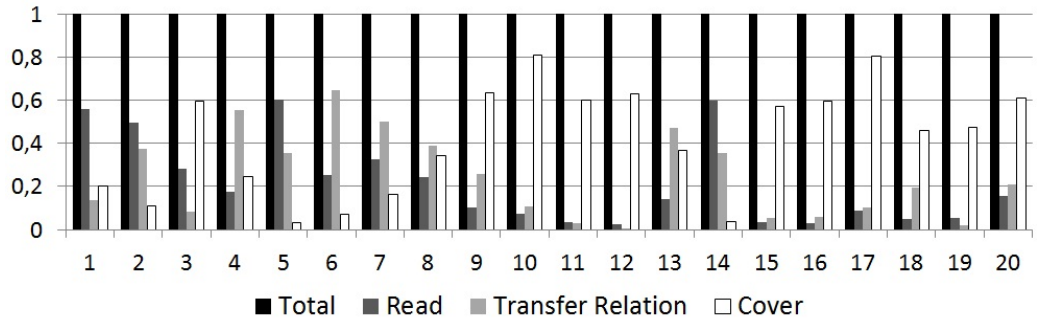

Figure 3.9: Proportion of total validation time spent on certificate reading, transfer relation, and coverage check on per analysis basis

parallelization of the larger model checking related tasks seems to be better. We think that the reason is that the fraction of certificate reading, the non-parallelizable part, to the complete validation time is lower. Despite the better improvement of the model checking related tasks, for them we rarely observed an additional improvement over the verification. We noticed only three additional cases. In contrast, after parallelization the validation of the dataflow analysis tasks beat verification in additional 501 cases.

Parallelization improves certificate validation well. However, for the model checking tasks, we would require a large amount of CPU cores, theoretically sometimes already more than 40, to let certificate validation beat verification. Hence, we still need to improve certificate validation.

3.5.4 RQ 3: Where Does Validation Spent Its Major Time?

In the previous sections, we found out that for some analysis tasks and types certificate validation does not perform well compared to verification and parallelization is rarely an option in practice. To overcome the performance problem, we want to improve certificate validation in the next chapter. For a significant improvement, we need to know first which operations are worth to improve. Principally, certificate validation performs four types of operations: certificate reading, computation of transfer successors, coverage check, and inspection of the safety property. Looking at our results, we noticed that the time spent for the inspection of the safety property is negligible. In the following, we examine the time spent on certificate reading, the time required to compute transfer successors, and the time spent with coverage checking. Figure 3.9 shows for all analysis configurations the sum of all these times in relation to the sum of the total time for certificate validation, the time for certificate reading plus the execution time of the validation algorithm. All sums are normalized w.r.t. the sum of the respective total time.

The validation of the coarsest analysis (1) suffers most from certificate reading. The behavior of the dataflow analysis tasks $(2-10)$ is diverse. In one case, the coverage check is the largest cost factor. For some analysis configurations, the major cost is certificate reading. For others, the transfer relation requires the major time and sometimes two or more operators require similar costs. Looking at the intermediate, model checking, and CEGAR model checking tasks $(9-20)$, we observe that certificate validation for these tasks is mainly dominated by the time spent on the coverage check. The only exceptions are sign model checking (13) and uninitialized variables model checking (14), for which

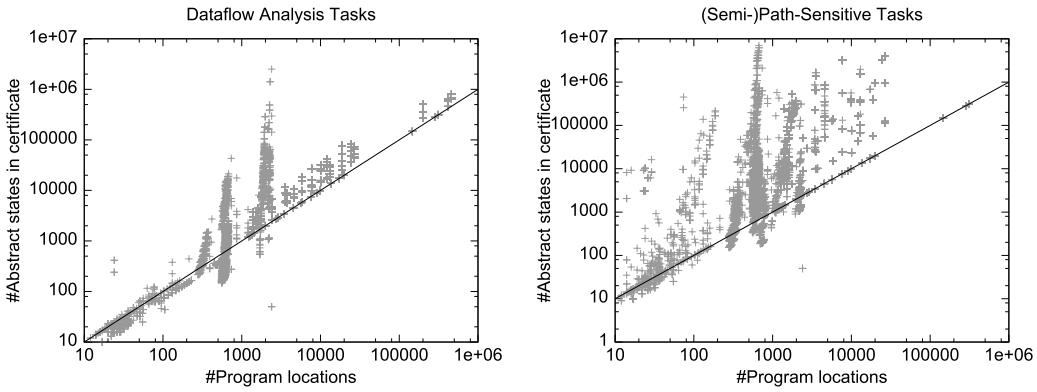

Figure 3.10: Relation of the certificate size (number of abstract states) to the number of program locations for dataflow analysis tasks (left) and (semi-)path-sensitive tasks (right)

the time spent on successor computation or certificate reading is the main issue.

Based on these insights, we come to the conclusion that we should definitely improve the coverage check operator. As we will see, our certificates become quite large. Thus, it would also be beneficial to reduce the certificate size and, hence, improve certificate reading. In contrast, we decided no to optimize the transfer relation because we expect that a generic improvement is difficult. The transfer relation is applied rather locally and it is the component one definitely needs to construct the abstract state space.

3.5.5 RQ 4: How Big Are Certificates?

The size of a certificate plays an important role in the practical application of configurable program certification. On the one hand, it influences the performance of the validation. Reading larger certificates is more costly and certificate reading, which does not appear in the verification process, is an extra cost factor. Additionally, the size of the certificate influences the memory consumption. Currently, our validation algorithm must keep the complete certificate in main memory to check the safety property. On the other hand, the certificate must be transferred to and stored by the user. Thus, optimally its size should be at most in the order of magnitude of the program size. In the following, we consider two metrics to study the certificate size: the number of stored abstract states and its file size.

We start with the more abstract and less physical metric, the number of abstract states in the certificate. To get a better feeling for the meaning of these sizes, we relate them to the number of program locations, which we think is an adequate, abstract measure for the program size.

Our coarsest analysis is flow-insensitive. During a flow-insensitive analysis, the reached set (the set of ARG nodes) always contains at most a single element. In the current metric, all certificates for flow-insensitive tasks have size one. We conclude that the certificate size is smaller than the program size.

Figure 3.10 shows the comparison of the number of stored abstract states (certificate size) and the number of program locations for the remaining tasks. The left diagram describes the results for the dataflow analyses, the flow-sensitive only analyses. On the right of Fig. 3.10, we see the values for all tasks which are (semi-)path-sensitive, i.e.,

the tasks related to intermediate, model checking, and CEGAR model checking analysis configurations.

Looking at Fig. 3.10, we see that in both diagrams the data points are often above the solid line. This means the certificate contains more abstract states than the corresponding program locations. For flow-sensitive or more precise analyses, the certificates become likely much larger than the program. For the dataflow analysis tasks, more than one state per location exists only because methods are inlined. The more precise analyses in the right diagram often separate different abstract states per program location. Thus, it is no surprise that the data points in the right diagram are often farther away from the solid line. We conclude that the more precise an analysis the larger its certificate.

The relation of the number of stored abstract states to the number of program locations might give us a first impression of the physical storage size of the certificate compared to the program size. However, all our certificates are automatically ZIP-compressed during generation while programs are uncompressed. Furthermore, program locations do not reflect the physical storage size of a program properly, e.g., they do not consider program statements. Similarly, we have no idea how large a representation of an abstract state might become. Potentially, certificate compression, abstract state or statement size may reverse our previous observation on certificate sizes.

Studying our results (see Fig. B.2 in the appendix), we notice that the trend observed for the previous metric continues for the file size metric. Passing the threshold of one kilobyte, certificates generated for flow-insensitive analysis tasks have smaller file sizes than programs. More precise analyses often generate certificates that are larger than the program. Again, certificates generated by the less precise dataflow analysis tasks seem to be smaller than those generated by the more precise, (semi-)path-sensitive analyses. For the details we refer the reader to Fig. B.2 in the appendix.

In a nutshell, the more precise the analysis the larger its certificate and the likelier it is that its certificate is orders of magnitudes larger than the program. Thus, for most of our tasks the certificate is (much) larger than the program.

3.5.6 Summary

Certificate validation can significantly outperform verification. In practice, certificate validation rarely significantly outperforms verification and verification is often still more efficient than validation, especially for particular analyses or analysis types like e.g. model checking. Thus, we described how to estimate the performance of the validation based on the verification results. Furthermore, we observed that parallelization can speed up certificate validation well, but it is not sufficient to effectively outperform verification. For further improvements of the validation, we identified that certificate reading and the coverage check are good candidates. Additionally, the certificate size should be improved. It is often much larger than the program itself.

3.6 Discussion

Our basic configurable program certification approach is a general framework. In our evaluation, we tested it with seven different domains and even 20 different CPAs. From a theoretical point of view, it performs well. We proved the desired theoretical properties soundness and relative completeness. Furthermore, the consumer's trusted computing base does not contain the merge operator of the analysis. Likely, the consumer's trusted

computing base is smaller. Moreover, the trusted computing base can be further reduced by the configurable certificate validator when one uses a certified CPA. For example, similar to the approach [BJP06] suggested by Besson et al., one could extend our approach with a preceeding PCC phase in which the producer sends the consumer its CPA and a correctness proof.

The only theoretical drawback is that validation cannot be guaranteed to be automatic. Except for the coverage check, all components of the configurable certificate validator can always be derived from the analysis configuration. To ensure relative completeness, we require a well-behaving coverage check. While some termination checks are well-behaving, others must be adapted by the consumer to become well-behaving. Thus, full automation is restricted to analysis configurations with a well-behaving termination check. Since we proved that a standard termination check is well-behaving, in practice automation is rarely a big issue. Often, we only had to deal with a technical problem w.r.t. equivalence of callstack states. For validation, we need to exchange standard, syntactical equivalence by semantical equivalence. All configurations using the callstack CPA require this change, which practically means just to change a configuration parameter in CPAchecker. Principally, the change of the configuration parameter can always be automatically added.

Our evaluation with a large set of programs and analysis configurations demonstrates the practical feasibility of our framework. Nevertheless, we would like to mention that the programs are often simplified, and thus not realistic. Due to the immaturity of some of the analysis domains, many of our programs do not consider certain language features like pointers. We also want to remark that the evaluation is rather sensitive to the different speeds of persistent storage access and CPU.

Considering the certificates, we perceived that the certificate generation is simple, but certificates are typically too large. Often, they are much larger than the program. Note that a similar observation was made for the original PCC approach by Necula (see e.g. [NL98a, NR01]).

Additionally, we noticed that often the performance of the validation is not significantly better than verification. Sometimes it is even worse. However, except for model checking, validation significantly outperforms verification on all other analysis types with varying domains for at least some programs. Furthermore, we explained how to predict the performance, especially the validation time, from the knowledge gained during the verification. We also tried to overcome the performance issue with parallelization. Parallelization worked well, but it is not enough to overcome the performance problem and the implementations were rarely prepared for parallelization. Moreover, we found out that an improvement of the certificate reading time and especially the coverage check can help to overcome the performance problem.

In short, the basic configurable program certification approach works well in theory, but our evaluation revealed that it often lacks practical applicability. Certificates are too large and validation is only seldom much better than verification. Thus, we should definitely improve our basic configurable program certification approach. In the next chapter, we propose two techniques to improve the basic configurable program certification approach. Both techniques try to improve the observed time spent on certificate reading and the coverage check.

4 Optimization of Configurable Program Certification

The basic configurable program certification approach presented in the previous chapter fulfills both theoretical requirements we had on approaches aiming at a fast validation at the consumer side. It is sound and it is relatively complete. However, its evaluation revealed that its validation is not always efficient and that the approach would benefit much from an improvement of the coverage check. Additionally, certificates are often much larger than the program itself. In this chapter, we want to tackle these two drawbacks, but with the main emphasize on improving the coverage check.

To develop an idea how to improve the coverage check, we first need to take a closer look at how it works. In theory, the coverage check is a function. However, we think that it is unlikely that one has the complete function table, especially because the number of abstract states can be infinite. The coverage check is more of a computational check than a constant look up. For example, it could compute the set of concrete states represented by the set of abstract states (the second parameter) or an abstract underapproximation of it and check whether it covers the input abstract state (first parameter). Another possibility is to subsequently test if a subset of the covering candidates, possibly consisting of a single abstract state, covers the input abstract state. In all cases, we assume that the computational check becomes more expensive the larger the set of covering candidates becomes. In practice, all of our coverage checks work similar. They iterate over the set of states, the second parameter, which is already restricted to states with the same location as considered by the first parameter, and check whether the state considered in the current iteration covers the first parameter. The iteration stops when all states are considered or one covers the first parameter. We conclude that on average the coverage check should become much faster when it has to consider less abstract states. The main goal should be to reduce the number of abstract states provided to the coverage check.

A straightforward idea, which can easily be integrated into the existing validation, is to reduce the number of stored states in the certificate. The simplest solution would be

that the second parameter of the coverage check may only consider abstract states from such a smaller certificate. While we would get the smallest certificate when we never store an abstract state, in general certificate validation would end up in a re-execution of the verification. Hence, the smallest certificate cannot be used to restrict the number of abstract states considered by the coverage check and we again need to merge during validation, i.e., we must include the merge operator into the trusted computing base. Thus, a smaller certificate must store some, but not all states of the basic certificate.

Another idea is to include the knowledge about the ARG successors, the nodes that cover a particular transfer successor, in the certificate. This idea likely restricts the coverage check to the smallest number of abstract states required for a successful validation. However, much more information must be stored. The chance that the increased time for certificate reading counterbalances the improvement on the coverage check is high. That it is why we decided to mimic the effect of storing the successor relation. Basically, we want to partition the certificate's abstract states s.t. when inspecting one partition element the coverage check may only consider abstract states from the same partition element and unrelated abstract states with same locations likely belong to different partition elements. Moreover, in this partitioning approach we plan to structure the certificate in such a way that single partition elements can be checked independently. Parallel reading and validation of a certificate becomes feasible.

In the following, we propose optimizations based on certificate reduction and partitioning. We start to explain the reduction of the certificate size. Then, we introduce our partitioning approach and the combination of our two optimizations. Thereafter, we examine the practical performance of the proposed optimizations. Subsequently, we discuss the results of configurable program certification. Finally, we present related work.

4.1 Reduction of the Certificate Size

In principle, two orthogonal opportunities exist to reduce the certificate size. First, some approaches aim at reducing the size of the representation of a single abstract state. Giacobazzi et al. [GR10, GR14] propose to exchange the abstract domain by a simpler one, which shows the same approximate behavior on the input program. The idea is that the simpler abstract domain considers less abstract values and can be encoded more efficiently. In contrast, Besson et al. [BJT07] and Seo et al. [SYYH07] use the observation that the representation of a more abstract state is often smaller than a more precise one. Based on this observation, they suggest to remove information that is not relevant to prove a property and, thus, replace single abstract states by suitable, but more abstract ones. Furthermore, Besson et al. [BJT07] additionally applies the second approach to reduce the certificate size and stores only a subset of the explored states plus a reconstruction strategy for the complete certificate. Also other approaches, e.g., [Ros03, AAPH06, BJP06, BJT07, AMA07], only partially store the explored state space, i.e., a subset of the explored states. However, not all of those approaches need a reconstruction strategy.

We believe that in practice a consumer will have certificate validators for standard configurable program analyses, but it is unlikely that he will support certification for all slight variants. Moreover, we think that without further information it is difficult to generally decide for an arbitrary abstraction, an arbitrary set of abstract states, which information are not needed to prove the property and can be removed safely. The first approach is not well-suited for our general configurable program certification framework. Hence, we also would like to apply the second idea and only store parts of our basic

certificate, which represents the explored state space.

Existing approaches [Ros03, APH05b, BJP06, BJT07, AMA07] typically look at analyses that compute one abstract state per program location. Often, they use the program structure to define which states must be stored. In contrast, our approach must work on the abstract reachability graph and a more general class of program analyses. Additionally, our certificate reduction must not rely on a reconstruction strategy and parallel validation should still be possible. We continue to discuss why we nevertheless think that it is feasible and beneficial for certificate validation in our configurable program certification framework when only a subset of the basic certificate is given to the consumer.

4.1.1 Foundation For Abstract State Deletion

The major part of the certificate validation consists of the computation of transfer successors and checking their coverage. If a transfer successor is contained in the certificate, then certificate validation should reproduce that abstract state anyway. The storage and the coverage check of such an abstract state would be redundant. Hence, to successfully apply the second reduction technique and to only store a subset of the basic certificate, we require that some abstract states are transfer successors of others.

Since the CPA algorithm does not directly deal with transfer successors, but uses the abstract state e_{prec} obtained after precision adjustment of the transfer successor, we first need to be sure that sometimes the precision adjustment does not weaken the transfer successor. Looking at precision adjustment operators in practice, we make the following observations:

- Some analyses like the sign dataflow analysis never adjust precisions.
- Even if an analysis adjusts precisions, it will not necessarily adjust precisions all the time. For example, consider the implementation of predicate abstraction with adjustable block encoding [BKW10]. In case a block end is reached, the new abstraction formula ψ is not computed in the transfer relation, but in the precision adjustment. However, typical block ends are loop heads or the beginning or end of a function. Precisions are adjusted rarely.
- Moreover, an analysis may use the form of the reached set to adjust precisions. Based on the number of different values for a particular variable observable in the reached set, the analysis may decide to no longer track the concrete values for that variable.
- Furthermore, a precision adjustment may not lead to a more abstract state in general. If the predecessor of a transfer successor is already coarse enough, the computed transfer successor will already meet the desired precision, e.g., it does not consider concrete values for variables that are not supposed to be tracked by the current precision.

In these cases, the abstract state resulting from the precision adjustment is the same as the transfer successor. Often, the transfer successor and the abstract state obtained after precision adjustment are identical.

It remains to be shown that such transfer successors are added to the reached set and are never merged after they are inserted into the reached set. Flow-sensitive analyses, a widely-used analysis type in practice, merge at most when the control-flow joins, i.e.,

the CFA node has more than one incoming edge, or the transfer relation is not a function. Many flow-sensitive analyses use transfer functions. In sequential code parts, those analyses may only merge with abstract states explored in previous iterations. Since the certificate will be computed from the final fixpoint, the transfer successor will be the last successor being explored. We only need to argue that these transfer successors may be added. When we merge the transfer successor with an abstract state explored in a previous iteration, it is likely that the abstract state from the previous iteration is subsumed by the current successor. Transfer relations are often monotonic and abstract states are only reexplored in case they are more abstract. Furthermore, we think that it is common that the merged state is identical with the subsuming state, e.g., when the analysis computes a join. The transfer successor will be added to the reached set. Moreover, the transfer successor will be added to the reached set when it is not covered by the current reached set. In flow-sensitive analyses, abstract states are covered by abstract states referring to the same location information. The final reached set will contain at least one abstract state for most of the program locations and – as discussed above – for some of them it is likely that they are transfer successors of other states in the reached set.

Hence, it will be likely that at least some abstract states are transfer successors of others. We believe that our configurable program certification framework may benefit from abstract state deletion. In the following, we discuss how the producer constructs certificates that contain only a subset of the final reached set, what must be considered during certificate construction to meet our requirements, and how the consumer checks such certificates. Additionally, we prove that the optimization presented in the following remains sound and relative complete.

4.1.2 Construction of Reduced Certificates

The producer verification is the same as in the basic configurable program certification approach. Thus, we directly start to explain how the producer generates a smaller witness, a *reduced certificate*, after a successful verification. To get a smaller witness, we follow a common idea, see e.g. [Ros03, BJP06, AAPH06, BJT07, AMA07, Jak15], and store only a subset of the explored state space, i.e., a subset of the states of the original certificate.

We first discuss the syntactical appearance of a reduced certificate, which again allows us to reject programs enhanced with an improper syntactical format. The original certificate contains abstract states of a single abstract domain given by an enhanced CPA. Hence, a reduced certificate must at least contain a set of abstract states. We follow our previous definition of a reduced certificate [Jak15] and also add the size of the basic certificate. The provided size allows us to eventually stop recomputation of the basic certificate in case of a malicious reduced certificate. Still, the termination of our validation algorithm only depends on the input program and the validation configuration. Syntactically, a reduced certificate is a pair of abstract states and a natural number, typically the size of the basic certificate.

Definition 4.1 (Reduced Certificate)**.** Let $\mathbb{C}_{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ considering the set of abstract states $E_{\mathbb{C}^{\mathcal{A}}}$. A *reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}}$ is a pair of a set of abstract states and a natural number, $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} \in 2^{E_{\mathbb{C}^{\mathcal{A}}}} \times \mathbb{N}$. A *reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}}$ *is finite* if the set of abstract states is finite.

Similar to a syntactically correct certificate, a syntactically correct reduced certificate does not automatically witness program safety. Consider for example the following syntactically correct reduced certificates $(\emptyset, 0)$ and $(\top_{\mathbb{C}^{\mathcal{A}}}, 1)$. The reduced certificate $(\emptyset, 0)$ claims

that considering zero abstract states, i.e., the empty set, is sufficient to show program safety. However, for our example program `SubMinSumDiv` the empty set is not sufficient to show correctness w.r.t. property `pos@l5`. The empty set does not overapproximate `SubMinSumDiv`'s state space. In contrast, the reduced certificate $(\top_{\mathbb{C}^{\mathcal{A}}}, 1)$ overapproximates any program's state space but the overapproximation does not guarantee safety. In both cases, we cannot reconstruct a valid certificate with the provided size.

According to our conception, a reduced certificate should store a subset of the basic certificate and the size of the basic certificate. With this idea in mind, we say that a *reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}, n)$ *is valid* if the set $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ is a subset of a valid certificate $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ and this certificate $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ has size n.

Definition 4.2 (Valid Reduced Certificate)**.** A *reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}, n)$ *is valid* for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$ if the set of states $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ can be extended to a certificate $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ s.t.

- $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq \mathcal{C}_{\mathbb{C}^{\mathcal{A}}} \subseteq E_{\mathbb{C}^{\mathcal{A}}}$,
- $|\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}| \leq n$ and
- $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ is valid for P, $\mathcal{A}$, and I.

From our definition of a valid reduced certificate, it easily follows that a valid reduced certificate witnesses program safety w.r.t. a certain property and a given set of initial states. The third property together with the witness property of a valid certificate $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$, which we showed in the previous chapter, is sufficient to show the witness property of a valid reduced certificate.

Corollary 4.1. *If* reduced certificate $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}}$ is valid *for program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$, then $P \models_I \mathcal{A}$.*

Proof. By definition, there exists a certificate $\mathcal{C}_{\mathbb{C}^{\mathcal{A}}}$ which is valid for P, $\mathcal{A}$, and I. Now, Lemma 3.1 lets us infer that $P \models_I \mathcal{A}$. □

Next, we discuss how the producer builds his reduced certificate from the generated ARG. The definition of a valid reduced certificate tells us that any subset of a valid certificate together with the size of that certificate is a proper reduced certificate. From the previous chapter, we know that after a successful verification the set N of ARG nodes is a valid certificate. Hence, any certificate $(N_{\text{s}}, |N|)$ with $N_{\text{s}} \subseteq N$ could be a reasonable choice.

When we use reduced certificate $(N, |N|)$, we mainly get our basic configurable program certification scenario. We do not improve the basic configurable program certification presented in the previous chapter. Furthermore, if we provide the reduced certificate $(\emptyset, |N|)$ to the consumer, the consumer must redo the complete verification. In this case, the consumer does not benefit from the configurable program certification approach. He has the same effort as the producer. Additionally, the effort is the same as without the configurable program certification approach. From these two extreme cases, we conclude that at best the reduced certificate generated by the producer should contain less abstract states than the basic certificate. Simultaneously, the validation effort of a reduced certificate should not be higher than for the basic certificate.

The previous two requirements on a reduced certificate ensure that our certificate reduction indeed improves the basic configurable program certification approach. Next to these requirements, we demand two further properties, which are not considered by other certificate reduction approaches. First, we want to reuse the configurable certificate

validator for the validation configuration. Contrary to, e.g., lightweight bytecode verification [Ros03] or reduced certificates in Abstraction-Carrying Code [AAPH06], we will not combine abstract states during validation. This requirement ensures that the trusted computing base remains the same for the consumer. Second, the inspection of an abstract state in the reduced certificate should not require an abstract state recomputed during the inspection of another state in the reduced certificate. In contrast to, e.g., lightweight bytecode verification [Ros03], reduced certificates in Abstraction-Carrying Code [AAPH06], or the approaches by Besson et al. [BJP06, BJT07] and Amme et al. [AMA07], the validation of a reduced certificate should be independent of the course of the verification, the structure of the program or the ARG. The last property allows us a validation implementation that can use any inspection order on the abstract states. Especially, a parallelization of the validation does not need to consider dependencies between the inspection of certificate states. Parallelization stays as simple and efficient as in the basic approach.

Remember that our overall goal is the reduction of the time required for the coverage check and reading the certificate. Additionally, the coverage check will only consider states stored in the certificate to detect coverage. Hence, we want to delete as many elements from the basic certificate as possible. Based on the four presented requirements, we must answer the question: Which states do we need to keep of the basic certificate, i.e., which ARG nodes do we need to add to the reduced certificate?

First, we observe that it is problematic when we delete the root node. On the one hand, we will add the initial abstract state e_0 to the recomputed certificate if e_0 is not covered by the reduced certificate. A problem occurs when the consumer uses a more precise initial abstract state, a scenario supported by the basic approach, and the transfer relation is not monotonic. The successors computed for e_0 could be less precise than those computed for the root node and the validation of the certificate may fail, although it succeeds in the basic approach. On the other hand, if we only delete the root node when we can recompute it, we always need to check coverage of the root node on the recomputed certificate. However, restricting the coverage check of the root node to the states in the reduced certificate is faster. We decided to always keep the root node. Thus, we profit from a faster coverage check and avoid the problems of the first variant.

Second, we examine which abstract states, ARG nodes, are recomputed during validation and whether they can be deleted. The basic validation uses the transfer relation to compute for every abstract state (ARG node) in the certificate its transfer successors. Since the producer uses complete (well-formed) ARGs to construct the certificate, the transfer successor is either covered by ARG successors contained in N_{cov} or by an ARG successor which is at least as abstract as the transfer successor. In the following, we discuss for which of these ARG successors it is okay to use the transfer relation to recompute them. Next to ARG successors in N_{cov}, we distinguish between ARG successors which are more abstract and those which are equal to the transfer successor.

ARG successors in N_{cov} We know that in a well-formed ARG, which the producer uses for reduced certificate construction, nodes in N_{cov} may partially cover transfer successors. Especially, the number of transfer successors covered by a particular N_{cov} is likely greater one. To get an order independent validation, we need to keep any node from N_{cov}, even if it could be recomputed.

ARG successor more abstract than transfer successors In the previous item, we already discussed ARG nodes in N_{cov}. Therefore, we restrict our considerations to ARG successors that are not part of N_{cov} and cover more precise transfer successors. Since the producer uses a well-formed ARG to construct the certificate, we can assign

to each transfer successor that is only covered by an ARG node from $N \setminus N_{\text{cov}}$ a unique ARG successor from $N \setminus N_{\text{cov}}$. Hence, deleting such a unique ARG successor of a transfer successor does not affect the examination of other transfer successors. If we want to delete ARG successors that cover more precise transfer successors, during validation it must be acceptable to add a more precise abstract state instead of the deleted state. Let us have a look at the case that we add a more precise abstract state e_{mp} instead of the deleted state e_{d}. If the transfer relation is not monotonic, two problems may occur. First, a more precise abstract state e_{mp} may provide more transfer successors. If we also deleted (some of) the successors of e_{d}, then we could add more successors for e_{mp} than we had for e_{d}. The reconstructed certificate can become too large and is no longer valid. Second, the computed transfer successors may become more abstract and are no longer covered by the successors of e_{d}. Then, the reconstructed certificate either may not be closed under successor computation or it may become too large. In both cases, the reconstructed certificate is not valid. Even with monotonic transfer relations a deletion can be improper. For example, the computation of the transfer successors of a less precise abstract state may be more costly. If these additional costs are not compensated by the saved reading costs, the validation costs may be higher than for the original certificate. We violate the performance requirement. In summary, deleting these kinds of ARG successors should be an alternative for CPAs with monotonic transfer relations.

ARG successors identical with transfer successors Similar to the second item, we only consider ARG successors that are not contained in N_{cov}. In contrast to the previous item, we consider ARG successors that solely cover transfer successors that are identical with them. These ARG nodes will be recomputed during inspection of any of its ARG predecessors since during inspection of an abstract state its transfer successors are computed. The order of the inspection and, thus, the validation order does not matter. We only have to look at the special cases of self-loops. If the ARG node has a predecessor that is different from itself, the inspection of this predecessor will recompute the ARG node. In the other case, we cannot reach the ARG node from the root node or the ARG node is the root node. From the proof of Theorem 3.10[1] (see p. 258), we remember that only ARG nodes that can be reached from the root node are important to ensure the validity of the basic producer certificate. We do not need to keep ARG nodes with self-loops which do not have further predecessors and are not the root node. Since we store the root node anyway, we do not need to bother any further about an ARG node with a self-loop. Summing up, ARG successors that are identical with all respective transfer successors may be deleted.

So far, we considered ARG nodes that cover transfer successors of other ARG nodes and the root node. At last, we must decide how to handle ARG nodes that are neither the root node nor cover transfer successors. From the proof of Theorem 3.10[2] (see p. 258), we remember that only the root node and ARG nodes that are either in N_{cov} or that cover a transfer successor are important to ensure the validity of the basic producer certificate. Hence, ARG nodes that are neither the root node nor cover transfer successors can be safely removed even if they cannot be recomputed.

From the previous considerations, we infer that the reduced certificate should contain the root node and all nodes from N_{cov}. ARG nodes that are either ARG and transfer

[1] Proof of relative completeness of the basic configurable program certification approach
[2] Proof of relative completeness of the basic configurable program certification approach

successors of other ARG nodes only or do not have an ARG predecessor do not need to become part of the reduced certificate. Deletion of further ARG nodes, namely ARG nodes that cover more precise transfer successors, can become problematic for arbitrary CPAs. With this in mind, we define the *reduced node set*, the subset of ARG nodes stored in a reduced certificate when no information about the verification configuration, a CPA, is known.

Definition 4.3. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph for enhanced CPA $\mathbb{C}^\mathcal{A}$. The *reduced node set* of ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is $N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}}) = \{n \in N \mid n = root \vee n \in N_{\text{cov}} \vee (\exists (n', g, n) \in G_{\text{ARG}} : \exists (n', g, e) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}} \wedge e \sqsubset n)\}$.

The following set describes the reduced node set for our example ARG shown in Fig. 2.4. We observe that the set consists of the root node plus the gray nodes, i.e., all nodes contained in N_{cov}. In our example, the remaining nodes are transfer successors of ARG nodes and can be deleted safely.

$$\left\{ \begin{array}{l} ((l_0, s : \top\ x : \top\ y : \top\ z : \top), q_0), ((l_5, s : \top\ x : -\ y : \top\ z : +), q_0), \\ ((l_9, s : +\ x : \top\ y : \top\ z : \top), q_0), ((l_{13}, s : +\ x : -+\ \ y : \top\ z : \top), q_0), \\ ((l_{14}, s : \top\ x : \top\ y : \top\ z : \top), q_0) \end{array} \right\}$$

A reduced node set is the basis for a reduced certificate built for an arbitrary configurable program analysis. However, we observed that in practice the transfer relations of many CPAs are monotonic. One exception is the implementation of the predicate analysis with adjustable block encoding [BKW10] in CPAchecker. Due to the incomplete implementation of the partial order, some cases, e.g., whether a non-abstraction state covers an abstraction state, are not checked. Hence, monotonicity cannot be detected properly. Furthermore, although in theory the abstract states $(l, false, l', true)$ and $(l, x > 0, l'', x = 0)$ are equivalent, the transfer relation does not provide a successor in the first case while it could in the second.

The following definition describes our formal idea of a *monotonic transfer relation*. It states that a transfer relation is monotonic if whenever a predecessor e' is more abstract than another predecessor e, then for every program statement $g \in \mathcal{G}$ we can uniquely map the transfer successors of e and g to more abstract successors of e' and g.

Definition 4.4 (Monotonic Transfer Relation)**.** Let $\mathbb{C} = (D, \Pi, \rightsquigarrow, \mathsf{prec}, \mathsf{merge}, \mathsf{stop})$ be a CPA. The transfer relation $\rightsquigarrow$ is monotonic if $\forall e, e' \in E, g \in \mathcal{G} : e \sqsubseteq e' \implies \exists$ total, injective function $f\colon \{(e, g, \cdot) \in \rightsquigarrow\} \to \{(e', g, \cdot) \in \rightsquigarrow\}, \forall (e, g, e_s) \in \rightsquigarrow\colon f((e, g, e_s)) = (e', g, e'_s) \implies e_s \sqsubseteq e'_s$.

We already discussed that ARG nodes that are not in N_{cov} and are more abstract than the transfer successor can be deleted when the transfer relation is monotonic. Hence, we decided to offer an alternative to the reduced node set, the *highly reduced set*, which can be used whenever a CPA's transfer relation is monotonic. The highly reduced set is a subset of the reduced node set. It removes all states that can be removed from the original certificate while meeting the four presented requirements. Additionally, it is easier to compute for the producer than the reduced node set. The producer only needs to consider the syntactical structure of the ARG. Formally, the highly reduced node set is defined as follows.

Definition 4.5. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph for enhanced CPA $\mathbb{C}^\mathcal{A}$. The *highly reduced node set* of $R^P_{\mathbb{C}^\mathcal{A}}$ is $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}) = \{root\} \cup N_{\text{cov}}$.

Coincidentally, for our example ARG, the ARG shown in Fig. 2.4, the highly reduced node set is identical with the reduced node set shown above. Note that this is not always the case for ARGs generated for any configuration. However, it is likely the case when the precision adjustment operator never changes the explored transfer successors, the transfer relation is a function, and transfer successors are only merged into at most one element.

The construction of a proper subset of the ARG nodes, either the reduced or highly reduced node set, is the major part of the reduced certificate generation. Afterwards, the reduced certificate generation becomes simple. To built the reduced certificate, the producer only needs to combine one of the two node sets with the size of the set of the ARG nodes. Depending on which node set he chooses, he either builds a *reduced certificate* or a *highly reduced certificate* from a given ARG.

Definition 4.6 ((Highly) Reduced Certificate from ARG)**.** Let $\mathbb{C}^{\mathcal{A}}$ be an enhanced CPA and $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph for $\mathbb{C}^{\mathcal{A}}$. The *reduced certificate from ARG* $R^P_{\mathbb{C}^{\mathcal{A}}}$ is $\text{cert}_{\text{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) = (\mathcal{C}^{\text{R}}_{\mathbb{C}^{\mathcal{A}}}, |N|)$ s.t. $\mathcal{C}^{\text{R}}_{\mathbb{C}^{\mathcal{A}}} = N_{\text{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$. The *highly reduced certificate from ARG* $R^P_{\mathbb{C}^{\mathcal{A}}}$ is $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) = (\mathcal{C}^{\text{hR}}_{\mathbb{C}^{\mathcal{A}}}, |N|)$ s.t. $\mathcal{C}^{\text{hR}}_{\mathbb{C}^{\mathcal{A}}} = N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$.

We already saw that for our example, the ARG shown in Fig. 2.4, the highly reduced and the reduced node set are the same. Thus, also the highly reduced certificate and the reduced certificate are the same. Below we can see the highly reduced certificate and the reduced certificate for our example.

$$\left(\left\{ \begin{array}{l} ((l_0, s : \top\ x : \top\ y : \top\ z : \top), q_0), ((l_5, s : \top\ x : -\ y : \top\ z : +), q_0), \\ ((l_9, s : +\ x : \top\ y : \top\ z : \top), q_0), ((l_{13}, s : +\ x : -+\ y : \top\ z : \top), q_0), \\ ((l_{14}, s : \top\ x : \top\ y : \top\ z : \top), q_0) \end{array} \right\}, 15 \right)$$

Compared to the basic certificate generated for our example, the certificate generated from the ARG shown in Fig. 2.4, the (highly) reduced certificate generated for that example only contains 5 out of 15 abstract states. With our certificate reduction, we were able to reduce the size of the certificate by $\frac{2}{3}$.

Up to now, we only indicated that the producer will generate valid reduced certificates if he sticks to the process of configurable program certification. Following the configurable program certification approach, the producer only constructs a reduced certificate after a successful verification. After a successful verification, we know that the generated ARG is well-formed and the set of ARG nodes is a certificate. From the definition of a valid reduced certificate we conclude that any subset of the set of ARG nodes, like the (highly) reduced node set, together with the size of the set of ARG nodes is a valid reduced certificate. As stated by the following corollary, we conclude that the (highly) reduced certificate from the generated, well-formed ARG is a valid reduced certificate.

Corollary 4.2. *Let $R^P_{\mathbb{C}^{\mathcal{A}}}$ be an abstract reachability graph for program P and enhanced CPA $\mathbb{C}^{\mathcal{A}}$ which is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$. Then, certificates $\text{cert}_{\text{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ and $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ are valid reduced certificates for program P, property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and initial states $[\![e_0]\!]$.*

Proof. Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$. By definition, $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}}) = N$. From Proposition 3.2, we know that $\text{cert}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$. By definition of an ARG, $N_{\text{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, and $N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, we get $N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq N_{\text{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq N$. From the definition of a reduced and highly reduced certificate from ARG, we conclude that $\text{cert}_{\text{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ and $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ are valid reduced certificates for P, $\mathcal{A}$, and $[\![e_0]\!]$. □

Algorithm 4: Validation algorithm for reduced certificates

Input: A CCV $\mathbb{V}^{D_{\mathbb{C}\mathcal{A}}} = ((C, (E, \top, \bot, \sqsubseteq, \sqcup), [\![\cdot]\!]), \rightsquigarrow, \mathsf{cover})$, initial abstract state $e_0 \in E$, reduced certificate $\mathcal{RC}_{\mathbb{C}\mathcal{A}} = (\mathcal{C}^{\text{sub}}_{\mathbb{C}\mathcal{A}}, n) \in 2^E \times \mathbb{N}$, program $P = (L, G_{\text{CFA}}, l_0)$
Output: Boolean indicator, if reduced certificate $\mathcal{RC}_{\mathbb{C}\mathcal{A}}$ is valid
Data: A set reached of elements of E, a set waitlist of elements of E

```
1  if ¬cover(e_0, C^sub_{ℂ𝒜}) then
2      return false
3  reached := C^sub_{ℂ𝒜}; waitlist := C^sub_{ℂ𝒜};
4  while waitlist ≠ ∅ ∧ |reached| ≤ n do
5      pop e from waitlist;
6      for each g ∈ G_CFA do
7          for each (e, g, e') ∈⇝ do
8              if ¬cover(e', C^sub_{ℂ𝒜}) ∧ e' ∉ reached then
9                  reached := reached ∪ {e'}; waitlist := waitlist ∪ {e'};
10 return |reached| ≤ n ∧ (¬∃(·, q) ∈ reached : q = q_err ∨ q = q_⊤)
```

So far, we introduced the concept of a reduced certificate and explained how the producer generates these reduced certificates. Next, we continue with the validation of a reduced certificate and the properties of that validation.

4.1.3 Validation of Reduced Certificates

The consumer validates reduced certificates to check whether a program is safe w.r.t. a property automaton and a set of initial states. Since valid reduced certificates ensure program safety, the consumer only needs to prove whether a reduced certificate is valid. To validate arbitrary reduced certificates, we require a validation configuration that fits to the reduced certificate, especially to its abstract domain, and a meta algorithm that is steered by the validation configuration to check the validity of a reduced certificate. We already mentioned that we want to reuse the validation configuration from the basic approach, a configurable certificate validator, to inspect a reduced certificate. It remains to be explained how the meta validation algorithm employs the CCV to investigate whether an input reduced certificate is valid w.r.t. the input program, the property automaton $\mathcal{A}$ included in the CCV, and a set of initial states given by the input abstract state e_0. From the definition of a valid reduced certificate, we infer that the meta validation algorithm must be able to reconstruct a valid certificate. Algorithm 4 shows this meta validation algorithm, an adaption of a previous version of a validation algorithm for reduced certificates [Jak15]. Its basic idea is to try to restore the original certificate and simultaneously examine the validity of the restored certificate.

In principle, Algorithm 4 is a variant of the validation algorithm for certificates (Algorithm 3). Like the basic validation algorithm, Algorithm 4 checks if the restored certificate covers the initial state and is closed under successor computation. In the beginning, Algorithm 4 uses the abstract states provided by the reduced certificate as a first version of the restored certificate. If the currently restored certificate does not cover a successor, it will simply be added to the restored certificate. However, whenever the currently restored certificate becomes too large, the validation stops.

To store the restored certificate, Algorithm 4 maintains an additional data structure reached. Furthermore, it utilizes a waitlist to track the states of the currently restored certificate for which successors must still be checked and, thus, enables an incremental inspection of the restored certificate even when this grows.

In the first two lines, Algorithm 4 checks if the initial abstract state is covered by the restored certificate. In contrast to the basic valdiation algorithm, it may consider only a subset of the restored certificate. However, we generate our reduced certificates such that they contain the initial abstract state that is considered during the construction. Line 3 sets up the initial restored certificate and registers its states for successor computation. As long as the currently restored certificate reached is not too large ($|\mathsf{reached}| \leq n$), lines 4-9 check if the restored certificate is closed under successor computation, enlarge the currently restored certificate where required, and register the newly added states for successor computation. In each iteration, a state e from the restored certificate for which the transfer successors have not been inspected is chosen from the waitlist. Then, for each program edge the transfer successors are computed, and it is checked if they are covered by the currently restored certificate. Since we demand that the validation is order independent, it should be sufficient to consider the set $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$, a subset of the restored certificate, to detect coverage. Due to performance reasons, we decided to only consider $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ to detect coverage. In case of coverage, the validation proceeds as in the basic case. When a transfer successor is not covered, we assume that the element(s) that covered that transfer successor in the original certificate was (were) deleted. To restore the original certificate, Algorithm 4 adds the transfer successor to the currently restored certificate reached.[3] Since Algorithm 4 added the transfer successor to the restored certificate, the restored certificate definitely covers the transfer successor. Note that Algorithm 4 checks that it does not add states to reached that are already contained. By this, it avoids that states for which successors are already computed are not re-added to waitlist and enables its termination. Finally, line 10 checks if the certificate is fully reconstructed from the reduced certificate ($|\mathsf{reached}| \leq n$) and if the reconstructed certificate is safe.

To use the presented validation procedure in the configurable program certification setting, it must fulfill two properties: soundness and relative completeness. After having understood how reduced certificates are validated, we continue to discuss when our reduced variant of the configurable program certification approach fulfills these properties.

4.1.4 Properties of Reduced Certificate Validation

Like the basic configurable program certification approach, the reduced variant must provide certain properties to guarantee practical applicability. First, the validation procedure may only return true if the reduced certificate witnesses program safety, i.e., it must be tamper-proof (sound). Second, the validation procedure should accept reduced certificates from a process conformant producer, i.e., it will return true if the producer constructs a (highly) reduced certificate from the ARG constructed during a successful verification. Hence, it must be relatively complete. We start with soundness.

Again, we will ensure soundness only in case the consumer uses a proper initial abstract state, which includes the initial automaton state q_0. For this case, we show that Algorithm 4, which performs the validation of a reduced certificate $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}}$, will only return true, i.e., it accepts the input reduced certificate $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}}$, if input program P is safe w.r.t. the property automaton $\mathcal{A}$ considered by the input CCV and initial states I given by

[3] Note that in case of a highly reduced certificate, the validation algorithm may reconstruct a more precise certificate, but it still reconstructs a valid certificate.

initial abstract state e_0. Recall that we designed Algorithm 4 in such a way that it should solely accept valid reduced certificates. Furthermore, Corollary 4.1 guarantees us that if a reduced certificate is valid w.r.t. a program P, a property automaton $\mathcal{A}$, and a set of initial states I, then program P will be safe w.r.t. $\mathcal{A}$ and I. Hence, to show soundness, we prove that if Algorithm 4 returns true, then the input reduced certificate will be valid w.r.t. input program P, property automaton $\mathcal{A}$ considered by the input CCV, and the set of initial states I provided by the initial abstract state e_0.

Remember that a reduced certificate will be valid if it can be extended to a valid certificate of a specific size. After termination of Algorithm 4, the set reached will contain the extension of the reduced certificate. By construction of reached, we know that the abstract states in the reduced certificate become part of reached, and reached only contains abstract states of the same domain as the certificate[4]. Moreover, Algorithm 4 may only return true in line 10. Line 10 checks that the extension has the required size and it is safe. The only property that remains to be shown for a valid reduced certificate is that the extension reached covers one configuration sequence per path. The following lemma ensures this remaining property.

Lemma 4.3. *If Algorithm 4 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and reduced certificate $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ returns true, then the reached set at the state of termination of Algorithm 4 covers at least one configuration sequence per path.*

Proof. See Appendix pp. 258 f. □

With the previous lemma at hand, we can conclude that the validation algorithm for reduced certificates (Algorithm 4) started with a proper initial abstract state only will return true if the given reduced certificate is valid.

Theorem 4.4. *If Algorithm 4 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and reduced certificate $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ returns true, then the reduced certificate $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$.*

Proof. Let $\mathsf{reached}'$ denote the reached set at the state when Algorithm 4 terminates. From the previous lemma, we get that $\mathsf{reached}'$ covers at least one configuration sequence per path. Since Algorithm 4 returns true, we know that $\mathsf{reached}'$ is safe. It follows that $\mathsf{reached}'$ is a valid certificate for P, $\mathcal{A}$, and $[\![e_0]\!]$. Since $\mathcal{C}^{\text{sub}}_{\mathbb{C}^\mathcal{A}} \subseteq \mathsf{reached}'$ (construction of $\mathsf{reached}'$) and $|\mathsf{reached}'| \leq n$ (Algorithm 4 returns true), we conclude that $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$ (definition of valid reduced certificate). □

Corollary 4.1 already proved that every valid reduced certificate is a witness for program safety. The previous theorem and Corollary 4.1 let us easily conclude the desired soundness property of our reduced configurable program certification approach.

Corollary 4.5 (Soundness)**.** *If Algorithm 4 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and reduced certificate $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ returns true, then $P \models_{[\![e_0]\!]} \mathcal{A}$.*

Proof. From the previous theorem, we get that $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$. Now, Corollary 4.1 lets us conclude that $P \models_{[\![e_0]\!]} \mathcal{A}$. □

[4] In the validation algorithm, we assume that the reduced certificate considers the same abstract domain as the CCV. Note that this assumption can easily be checked syntactically and if it is not met the validation would fail similar to the case of a syntactically incorrect certificate.

So far, we showed that also our first optimization, the certificate reduction approach, is reliable. We continue with the second property, relative completeness. This property guarantees practical applicability of our certificate reduction approach. We need to show that if the producer and the consumer stick to the reduced configurable program certification approach and the producer and the consumer consider the same program and initial abstract state, then the consumer validation will accept the reduced certificate generated by the producer.

A process conformant consumer derives his configurable certificate validator from the producer's CPA and may provide his own, well-behaving coverage check. Thus, his validation configuration fits to the generated reduced certificate. Moreover, a process conformant producer executes the CPA algorithm with an appropriate analysis configuration (CPA). After a successful verification, he uses the ARG constructed during verification to build a reduced certificate. Depending on the properties of the CPA, the producer constructs a reduced certificate from the ARG or chooses between two alternatives, the reduced certificate or the highly reduced certificate. Due to Corollary 4.2, any certificate provided by a process conformant producer is a valid reduced certificate. Our sound validation procedure may accept them. In the following, we show that the validation algorithm indeed accepts those certificates when the producer and the consumer adhere to the reduced configurable program certification process.

Again, we begin our proof steps with the termination aspect of the validation algorithm. Like in the basic approach, we assume that the operators of the CCV terminate for any finite input. Since the node set considered by a reduced certificate constructed by a process conformant producer is finite (the set of ARG nodes is always finite and the respective node set is a subset of the ARG nodes), all CCV operators always terminate. For termination, we only need to guarantee that all loops terminate, i.e., they are bounded. To ensure termination of the loops, we additionally require that the CFA edges are finite, i.e., the input program P is finite. This assumption is valid, because the producer's analysis only terminates for finite programs. Assuming finite programs, we can now state termination of the validation algorithm.

Lemma 4.6 (Termination)**.** *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover*, and let program* $P = (L, G_{\text{CFA}}, l_0)$ *be finite. Then, Algorithm 4 started with* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*,* P*, initial abstract state* $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$*, and finite reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}, n)$ *terminates.*

Proof. See Appendix pp. 259 f. □

After we proved that the validation algorithm terminates for a (highly) reduced certificate constructed by a process conformant producer, we continue to show that it really accepts a producer's (highly) reduced certificate. Next, we show relative completeness for the general case in which the producer generates the reduced certificate based on the reduced node set. More concretely, we show that the validation algorithm accepts any reduced certificate that contains at least the reduced node set, but no abstract states that are no ARG nodes. This allows us to reuse the relative completeness result, when we combine our certification approaches with our second technique Programs from Proofs.

First, we prove that when the validation algorithm executes line 10, the only line which may return true, the extension reached of the reduced certificate is of a particular form. We claim that our validation algorithm produces an extension reached of the reduced certificate that is a subset of the ARG nodes. Note that we cannot guarantee a complete reconstruction of the original certificate. As discussed earlier, we also delete ARG nodes

without predecessors, although we may not be able to reconstruct them. Hence, the validation algorithm may reconstruct a smaller subset of the original certificate.

Lemma 4.7. *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving,* $R^{P}_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^{\mathcal{A}}$ *of CPA* $\mathbb{C}$*, and* $R^{P}_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. If Algorithm 4 starts with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}, |N|)$ *s.t.* $N_{\mathrm{R}}(R^{P}_{\mathbb{C}^{\mathcal{A}}}) \subseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq N$*, then at line 10* $\mathsf{reached} \subseteq N$*.*

Proof. See Appendix p. 260. □

From the previous lemma, we know that if Algorithm 4 reaches line 10 in a process conformant course of our reduced certification approach, then reached will be a subset of the ARG nodes. Since the second element of a reduced certificate constructed from an ARG is the size of the set N of ARG nodes, the size of reached is smaller or equal to the second element of the reduced certificate. Furthermore, a process conformant producer constructs his certificate from a well-formed ARG. The set of ARG nodes and, thus, reached is safe (safety property of well-formed ARGs). If Algorithm 4 reaches line 10, it will return true. It remains to be shown that the validation algorithm reaches line 10.

Having in mind that (1) the reduced node set contains the root node, (2) the root node is at least as abstract as the initial abstract state e considered by the producer, (3) the consumer uses an initial abstract state e_0 which is at least as precise as e, (4) the partial order is transitive, and (5) a well-behaving coverage check is consistent with the partial order, we know that the check in line 1 succeeds. Due to proven termination of Algorithm 4, we get that Algorithm 4 reaches line 10. With this observations in mind, we propose the subsequent lemma. Given a well-formed ARG and a reduced certificate that contains at least the reduced node set, but no abstract states that are not ARG nodes, e.g., a reduced certificate that a process conformant producer constructs, then the validation algorithm started by a process conformant consumer will accept this certificate. Hence, the lemma states relative completeness of the general reduced certification approach applicable with any analysis configuration.

Lemma 4.8. *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving,* $R^{P}_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^{\mathcal{A}}$ *of CPA* $\mathbb{C}$*, and* $R^{P}_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. Algorithm 4 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and reduced certificate* $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}, |N|)$ *s.t.* $N_{\mathrm{R}}(R^{P}_{\mathbb{C}^{\mathcal{A}}}) \subseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq N$ *returns true.*

Proof. See Appendix pp. 260 f. □

Until now, we only discussed relative completeness for the general case. However, for CPAs with monotonic transfer relations we introduced an alternative, the highly reduced certificate. We proceed to show that when for CPAs with monotonic transfer relations the highly reduced certificate is used, the reduced certification approach is still relatively complete. Similarly to the general case, we first prove that at line 10 of the validation algorithm the set reached is of a specific form. Since in a highly reduced certificate additionally abstract states, for which only a less abstract state can be recomputed, are deleted, the extension reached of the certificate is no longer a subset but a set that is at least as abstract as the set of ARG nodes. Hence, we claim that at line 10, the set reached contains at most as many elements as the set N of ARG nodes, and it is at most as abstract as the ARG nodes.

Lemma 4.9. *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover}) = (D_{\mathbb{C}^{\mathcal{A}}}, \rightsquigarrow, \mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving,* $\rightsquigarrow$ *be monotonic,* $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^{\mathcal{A}}$ *of CPA* $\mathbb{C}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. If Algorithm 4 starts with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and certificate* $\mathrm{cert}_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$*, then at line 10* $\mathsf{reached} \sqsubseteq N$ *and* $|\mathsf{reached}| \leq |N|$*.*

Proof. See Appendix pp. 261 f. □

From the previous lemma, we know that in a process conformant course of our reduced certification approach in line 10 of the validation algorithm, reached is smaller than or equal to and at most as abstract as the set of ARG nodes. The construction of a highly reduced certificate lets us conclude that the size of reached is smaller than or equal to the second element of the highly reduced certificate. Furthermore, a process conformant producer constructs his certificate from a well-formed ARG. The set of ARG nodes and, thus, reached is safe (safety property of well-formed ARGs). If Algorithm 4 reaches line 10, it will return true. Following an argumentation similar to the general case, we can show that the check in line 1 succeeds. Hence, the validation algorithm indeed reaches line 10. These observations let us formulate the following lemma. In principle, the lemma states relative completeness for the special case of CPAs with monotonic transfer relations.

Lemma 4.10. *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover}) = (D_{\mathbb{C}^{\mathcal{A}}}, \rightsquigarrow, \mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving,* $\rightsquigarrow$ *be monotonic,* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^{\mathcal{A}}$ *of CPA* $\mathbb{C}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. Algorithm 4 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and certificate* $\mathrm{cert}_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

Proof. See Appendix pp. 262 f. □

The previous lemmas assure that the presented approach is relatively complete in case the producer constructs reduced certificates from the ARG or he generates highly reduced certificates and the transfer relation is monotonic. We unite these insights into a single theorem, which states that our reduced configurable program certification approach is relatively complete.

Theorem 4.11 (Relative completeness)**.** *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving,* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^{\mathcal{A}}$ *of CPA* $\mathbb{C}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*.*

1. *Algorithm 4 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and certificate* $\mathrm{cert}_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*
2. *If the transfer relation* $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ *of CPA* $\mathbb{C}^{\mathcal{A}}$ *is monotonic, then Algorithm 4 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and certificate* $\mathrm{cert}_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

Proof. The theorem follows from Lemma 4.8, definition of $\mathrm{cert}_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, and Lemma 4.10. □

Our previous considerations included semi-automatic validation of reduced certificates. The consumer may provide his own well-behaving coverage check. Finally, we consider

when the certificate reduction approach becomes fully automatic. Like in the basic approach, the producer's verification configuration must already use a well-behaving termination check. Additionally, we do not allow more precise initial abstract states for the consumer. The following corollary states that if these conditions are met, the certificate reduction approach will be fully automatic, too.

Corollary 4.12. *Let $\mathbb{C}^{\mathcal{A}}$ be a CPA, $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ be a configurable certificate validator for CPA $\mathbb{C}^{\mathcal{A}}$ and $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}$ which is well-behaving, P be a program, and $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$. If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$, initial abstract state e_0, initial precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$, and program P returns $(true, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$,*

1. *then Algorithm 4 started with $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, program P, initial abstract state e_0, and certificate $\mathrm{cert}_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ returns true.*

2. *and $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ is monotonic, then Algorithm 4 started with $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, program P, initial abstract state e_0, and certificate $\mathrm{cert}_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ returns true.*

Proof. From Corollary 3.3, we know that $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}$ is a coverage check. Hence, $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ is a CCV. From Proposition 2.8, we know that $R^P_{\mathbb{C}^{\mathcal{A}}}$is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is well-formed for e_0. Since Algorithm 2 terminates, we conclude that P is finite. Now, the corollary follows from the previous theorem. □

Summing up, also the first optimization of the basic approach meets our theoretical requirements presented in the introduction. Whenever the producer successfully checks a program P using a CPA $\mathbb{C}^{\mathcal{A}}$, and a proper initial abstract state[5] e_0, then the consumer validation using the producer's reduced certificate, e_0, and the CCV automatically derived from CPA $\mathbb{C}^{\mathcal{A}}$ succeeds and guarantees program safety. Additionally, we can use the same restriction of the initial abstract state as in the basic approach to transfer these results to program safety. The next section continues with the second, orthogonal optimization of our configurable program certification.

4.2 Certificate Partitioning

In the previous section, we reduced the number of states stored in a certificate to improve certificate validation. Now, we want to consider a recent trend in computer hardware development: processor speed is no longer increased, but the number of cores in computing devices increases. Hence, to make a program faster today, one should efficiently use parallelism. The goal of our certificate partitioning optimization is to make the complete certificate validation process ready for parallelism. While our previous certificate validation algorithms are already well-suited for parallelization, the loops can easily be parallelized, certificate reading is not considered. For the approaches to work out, the certificate must be read sequentially in advance. Our certificate partitioning approach will eliminate this downside, providing a way to read and validate the certificate in parallel. The general idea is to partition the certificate into several pieces such that a piece can be validated immediately after it has been read. This has the advantage that the wall time for certificate validation decreases and at best it is close to the time required by the validation algorithm.

[5] An initial abstract state which considers the initial automaton state.

So far, parallelization is considered rarely in PCC approaches and proof checking in general. Search-Carrying Code [TA10] allows the consumer to partition the search script of an explicit model checking run, the certificate, and to check each partition element of the search script in parallel. Computation certification [KH13] is a technique based on checkpoints that supports the verification of the integrity of a computation result in parallel. In principle, all recomputations starting in a checkpoint i and inspecting whether checkpoint $i + 1$ can be reached may be executed in parallel. Karsten Klohs discusses incremental validation [Klo09] to check a fixpoint of a interprocedural dataflow analysis in his PhD thesis. The idea is that a method can be checked as soon as the fixpoint information for all called methods is available. Similarly, incremental checking [Stu09], a technique for SMT proof checking, parses and checks the proof in an interleaved way.

Before we come to the details of our certificate partitioning approach, we first discuss why parts of a certificate can be validated although the complete certificate is not yet available.

4.2.1 Foundation for Partitioning

Reading parts of the certificate while parallely checking already read parts, the idea behind certificate partitioning, is only feasible when certificate validation does not always fully rely on the complete certificate. In the following, we discuss which parts of the certificate validation are local and to which extend.

First, let us look at the coverage of the initial abstract state. We know that the producer uses a well-formed ARG to construct his certificate and the ARG's root node is always contained in the producer's certificate. Due to the rootedness property, the initial abstract state is more abstract than the root node. A well-behaving coverage check should be able to detect coverage of the initial abstract state by any set of abstract states that contains the root node. If we divide the certificate's abstract states into several subsets, we may locally check whether the initial abstract state is covered by the current subset. For producer certificates, we know that the check returns true for one of the subsets and coverage of the initial abstract state is surely detected.

Second, we consider the major part of the certificate validation, the computation of transfer successors and checking their coverage. To compute a transfer successor, we only require the single abstract state for which we want to compute the transfer successor. Transfer successor computation is completely local. Next, let us come to the coverage check of the transfer successor. For the basic, non optimized certificate, we know that all abstract states are ARG nodes of a well-formed ARG. The validation recomputes transfer successors of ARG nodes only. We conclude from the completeness property (well-formed ARG) that for each transfer successor of an ARG node we can detect coverage of that transfer successor when we consider any subset of the ARG nodes that includes at least all successors of that node in the ARG. The coverage check of transfer successors of ARG nodes is local to the ARG node's successors. The properties of a well-behaving coverage check guarantees us that the validation may detect coverage locally. Hence, the computation of transfer successors and their coverage check can be performed locally for the basic certificate. In case the producer constructs a reduced certificate, transfer successors may or may not be covered. Whenever they are covered, the same observations as for the basic certificate apply. When they are not covered, we assume that they are removed from the certificate and we just re-add them. Since we may only re-add finitely many abstract states, after a finite sequence of transfer successor applications, we either reach an abstract state without a transfer successor or the transfer successor is

covered and for this transfer successor the observations from before remain valid. Thus, the computation of transfer successors and their coverage check remain local for reduced certificates.

Finally, to inspect safety, for each abstract state that is saved in the certificate or that is recomputed, the validation checks that it does not consider the automaton's error state. This check is purely local and can also be done when an abstract state is explored.

At first sight, it seems that certificate validation is completely local and we never require the complete set of abstract states at once. However, we will later see that we require the complete set of abstract states for integrity checks on the partitioned certificate, e.g., to ensure that the complete state space of the program is considered by the certificate. Nevertheless, the major part, the computation of transfer successors and checking their coverage, can be checked independently part by part. Since we require the complete set of abstract states anyway, we decided to simplify validation of a partitioned certificate and check coverage of the initial abstract state only once with the complete set of abstract states. Furthermore, we do not inspect safety individually per state because practically we implemented some safety policies that require the complete set of states.

Next, we continue with a brief discussion of the basic partitioning approach that looks at the partitioning of the basic, non-optimized certificate only.

4.2.2 Overview of the Partitioning Approach

Partitioning the basic certificate is a special case of partitioning a reduced certificate. Hence, we directly incorporate the combination with the reduction approach, when we discuss the details of the partitioning approach. However, we think that the ideas behind the partitioning approach will be much easier to understand if we leave out the reduction aspect. Next, we discuss these ideas.

The process of the partitioning approach is the same as in the basic approach, but like in the reduction approach the certificate generation and validation differs. Following the process, we first discuss the certificate generation. To read and check a certificate in parallel, we require that the certificate is constructed in a way that when a part is checked, all information required to check that part has already been read. From the previous considerations, we know that the inspection of a part only depends on the abstract states in the part and their successors in the ARG. A first naïve idea is to organize the abstract states in the certificate s.t. ARG successors are read and checked before their predecessors. Remember that our example ARG shown in Fig. 2.4 contains a loop. For this example, we cannot find such an ordering of the ARG nodes, the abstract states of the certificate. Thus, for at least some parts we need to attach additional information, i.e., namely certain successors of the abstract states, which have not been read and checked before. To be able to check multiple parts in parallel, we decided to add all information required to check a part independently from the remaining parts. A part does not only contain the abstract states checked in that part, but additionally those successors that are not checked in the part.

After we got an idea of the partitioned certificate, we continue with the second open question: its validation. Essentially, we keep the validation procedure as is and mainly adapt the meta validation algorithm. The adapted algorithm checks coverage of the initial abstract states after it checked that the certificate is closed under successor computation. Thus, the coverage check is done after all states are read. For checking closure under successor computation, the validation algorithm inspects each part individually. As soon as a part is read, the validation algorithm may start with its inspection. During inspection of

a single part, we assume that for those successors that are solely added to the part to ensure that it can be checked independently no successors exists. Based on this assumption, the validation algorithm checks that each part is closed under successor computation. Of course, these successor states may have transfer successors. Thus, after the inspection of all parts, we need an integrity check that examines if the exploration of these successors is considered in a different part. At the end, the validation algorithm performs the same safety test as the basic validation algorithm.

Summing up, the advantage of the partition approach, i.e., reading and checking in parallel, comes at the cost of larger certificates and additional integrity checks. Next, we come to the details of the partitioning approach.

4.3 Combination of Reduction and Partitioning

First, we describe the details of our complete optimization, the combination of certificate reduction and partitioning. Thereafter, we investigate whether configurable program certification also fulfills the necessary properties stated in the introduction section when it applies all proposed optimizations. Like in the certificate reduction approach, the producer verification remains the same as in the basic configurable program certification approach. Thus, we can directly move on to describe how the producer constructs the certificate when he uses certificate partitioning possibly combined with the previous approach of certificate reduction. Since the construction, but already the definition, of those certificates is more complicated than in the previous approaches, we split the definition of those certificates from their construction. We start to explain what the producer must construct.

4.3.1 Reduced, Partitioned Certificates: Structure and Validity

The key idea of certificate partitioning is that parts of the certificate are read and at the same time other parts, which were already read, are checked. We already observed that due to circular dependencies it is impossible to always order the elements in a certificate s.t. the following property holds: Whenever we inspect element i, then the first i elements are sufficient to guarantee a successful inspection. Hence, if the consumer defines the parts, e.g., after reading k elements he decides that one part ends and the next part starts, during inspection of a part the consumer cannot distinguish whether a missing element must be re-added or it has just not been read. To decide this, the consumer must always wait until the complete certificate is read. The advantage of reading and checking in parallel is gone. During certificate construction, the producer must already take care that a part can be checked independently of the remaining parts. Thus, the partitioned certificate must make the division into parts explicit and should group certificate elements, which are supposed to be read and inspected altogether.

The producer partitions the basic certificate or the (highly) reduced certificate in case he combines the approaches. A part contains at least a subset of the states stored in such a certificate, the so called *set of partition nodes*. Syntactically, basic certificates and (highly) reduced certificates are subsets of abstract states. From a syntactical point of view, a set of partition nodes is a subset of abstract states. To ensure that a part can be checked independently from the others, the set of partition nodes is not sufficient. Often, all elements in the certificate are interconnected. Note that the edges in the ARG that is used to construct the certificates represent this interconnection. Remember that

for example during inspection of the original certificate, we use the ARG successors of a certificate element to check that all its transfer successors are considered. For each part, we do not only need to store the partition nodes, but also further elements of the certificate, the so called *boundary nodes*. Boundary nodes are not inspected in the part, another part is responsible for their inspection, but they contain all those nodes that might be read later, i.e., they are in another part, and are required to successfully inspect the part. Due to the boundary nodes, the consumer can easily distinguish if a missing element must be re-added or it has just not been read.

Summing up, a part must be represented by a pair of a set of partition nodes and a set of boundary nodes. From now on, we call such a pair a *partition element*. A partition element is finite if both sets are finite.

Definition 4.7 (Partition Element)**.** Let $\mathbb{C}^{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ considering the set $E_{\mathbb{C}^{\mathcal{A}}}$ of abstract states. A *partition element* $\text{part}_{\mathbb{C}^{\mathcal{A}}} = (pn, bn)$ is a pair of a set $pn \subseteq E_{\mathbb{C}^{\mathcal{A}}}$ of partition nodes and a set $bn \subseteq E_{\mathbb{C}^{\mathcal{A}}}$ of boundary nodes. A partition element is finite if the set of partition nodes and the set of boundary nodes are finite.

After having defined the syntax of a single partition element, we use the concept of a partition element to describe the syntax of a *partitioned certificate*. When a partitioned certificate considers only a single partition element, the validation first reads, than checks the single partition element, and thereafter finishes. Validation would be sequential. To read and check a partitioned certificate in parallel, we require multiple partition elements. A partitioned certificate must contain a set of partition elements. Furthermore, we do not only want to construct partitioned variants of the basic certificate, but also of the (highly) reduced certificates. Similar to a reduced certificate, we also add the size of the basic, unpartitioned certificate. Thus, we are able to eventually stop validation of partitioned certificates even if the partitioned certificate got corrupted or is not constructed appropriately. From a syntactical point of view, a partitioned certificate consists of a set of partition elements and a natural number. The natural number typically represents the size of the basic certificate and, thus, restricts the number of abstract states that may be re-added during validation. A partitioned certificate is finite if it contains a finite set of finite partition elements.

Definition 4.8 (Partitioned Certificate)**.** Let $\mathbb{C}_{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ considering the set $E_{\mathbb{C}^{\mathcal{A}}}$ of abstract states. A *partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is a pair of a set $parts_{\mathbb{C}^{\mathcal{A}}}$ of partition elements and a natural number, $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}} \in 2^{2^{E_{\mathbb{C}^{\mathcal{A}}}} \times 2^{E_{\mathbb{C}^{\mathcal{A}}}}} \times \mathbb{N}$. A *partitioned certificate is finite* if the set of partition elements is finite and every partition element in that set is finite.

As before, a syntactically correct partitioned certificate is not automatically a witness for program safety. In previous work [Jak15], we used a similar concept of a partitioned certificate and were satisfied when sequential validation of the partitioned certificates would be feasible., i.e., the set of all partition nodes could be extended to a valid certificate. Hence, for the definition of a valid partitioned certificate, we totally ignored the boundary nodes. Since parallel reading and checking is the standard validation of a partitioned certificate, for this thesis we decided to include this aspect of parallelity into the definition of a valid partitioned certificate. To infer the requirements on a valid partitioned certificate,

let us look at the following partitioned certificate $\mathcal{PC}_{(\mathbb{L}\times\mathbb{S})^{\mathcal{A}}} =$

$$\left(\left\{\begin{pmatrix} \{((l_{14}, s:\top\ x:\top\ y:\top\ z:\top), q_\top)\} & , & \emptyset &), \\ \left(\left\{\begin{array}{l}((l_9, s:+\ x:\top\ y:\top\ z:\top), q_0),\\ ((l_{13}, s:+\ x:-+\ y:\top\ z:\top), q_0)\end{array}\right\}, \{((l_5, s:\top\ x:-\ y:\top\ z:+), q_0)\}\right) & & & \end{pmatrix}\right\}, 6\right).$$

Let us assume the partitioned certificate should witness program safety for our example program `SubMinSumDiv`, property `pos@l5`, and all initial states considering the initial program location l_0. We observe that for the first partition element we can successfully check that all successors are considered by the partitioned certificate. No CFA edge leaves location l_{14} and, thus, no successors exist. However, the first partition element contains an abstract state that considers abstract automaton state $q_\top$. The partitioned certificate does not witness program safety. Now, let us take a look at the second partition element. During inspection of the second partition element, we explore abstract successors $((l_{10}, s:+\ x:-+\ y:\top\ z:\top), q_0)$, $((l_{11}, s:+\ x:-+\ y:0+\ z:\top), q_0)$, $((l_{12}, s:+\ x:-+\ y:-\ z:\top), q_0)$, and $((l_{14}, s:+\ x:\top\ y:\top\ z:\top), q_0)$. The last successor is covered by a partition node of the first partition element, but this partition node is not contained in the boundary nodes of the second element. We cannot independently check the second partition element. The other three successors were removed during certificate construction. Since the partitioned certificate, more concretely the set of all partition nodes, contains only three nodes of the basic certificate and the certificate claims that the basic certificate has size 6, we can securely re-add these successors. For program safety, we need to ensure that all program paths that should be considered for program safety are safe. In our case, we e.g. need to witness program safety for paths consisting of a single concrete state c with control state $\mathrm{cs}(c) = l_0$. None of the existing partition nodes considers those concrete states. Furthermore, after we checked the second partition element, we reached the upper limit of states we are allowed to add. No new states should be added. Hence, the partitioned certificate does not guarantee that all program paths necessary for ensuring program safety are considered. Once again, the partitioned certificate cannot be a proper witness.

Some of the above insights are specific for partitioned certificates and others also apply to the previous certification approaches. In the following, we formalize the requirements a partitioned certificate must fulfill to be a proper witness, i.e., it is valid. Like for reduced certificates, we infer validity for the combination of reduction and partitioning from simple partitioning of the basic certificate. For simple partitioning of the basic certificate, we know that no states were removed, the partitioned certificate must be sufficient to witness program safety. The set of partition elements of a partitioned certificate is the component that witnesses program safety. Following the definition of program safety, the set of partition elements must witness that for every relevant program path a configuration sequence exists that does not consider the property automaton's error state.

Next, we define the concept of a *safe overapproximation*. The requirements of a safe overapproximation on a set of partition elements should guarantee the witness property discussed above. To ensure that for every relevant program path a configuration sequence exists, we must find a sequence of partition nodes that covers the configuration sequence. We use the observation that program paths and corresponding configuration sequences are inductively defined. If we extend a path, we can always extend a corresponding configuration sequence to get a proper configuration sequence for the extended path.

Our first requirement ensures that always a configuration sequence for the smallest paths p_s exists, namely those that only consider a state from the set of initial states I,

$p_s := c_0$ and $c_0 \in I$. The only possible configuration sequence is (c_0, q_0). Hence, each such (c_0, q_0) must be considered by at least one partition node.

The second and third requirement guarantee that if we may extend a program path, we can also extend a covered configuration sequence s.t. the extended configuration sequence is configuration sequence for the extended path and is covered by the partition nodes. In the second requirement, we assume that if we consider boundary nodes for the coverage of the extended configuration sequence, then the coverage of the extended configuration sequence will continue in another partition element. Let us consider an arbitrary, relevant program path and a corresponding configuration sequence that is covered by the partition elements. Furthermore, let us assume that the last element (c, q) of that configuration sequence is covered by a partition node of partition element pe. We know that c is the last state of the path we want to extend and any extension by one step adds something of the form $c \xrightarrow{(l,op,l')} c'$. Since the transition relations of all property automata are deterministic and complete, a unique automaton state q' exists, which must be used to extend the configuration sequence to a configuration sequence of the extended path. To guarantee that the extended configuration sequence is covered, it is sufficient that the state c' is covered by a partition node that respects the automaton state q'. However, we want to have independently checkable parts. Coverage must be detectable within the partition element. Based on our assumption, it is adequate to require that the state c' must be covered by a partition or boundary node that respects the automaton state q' and belongs to the same partition. To assure that our assumption that a configuration sequence is continued in another partition element is indeed true, our third requirement demands that each boundary node must be covered by the set of all partition nodes that respect the boundary node's automaton state.

So far, we only guarantee that for each relevant program path a configuration sequence exists. To witness program safety, we also need to know that the covered configuration sequences do not contain the error state q_{err}. Our last requirement demands that all partition nodes are safe, i.e., they do not contain the abstract automaton states q_{err} or $q_\top$. Hence, the partition nodes can only cover configuration sequences which do not contain the error state q_{err}. The following definition of a safe overapproximation summarize the discussed requirements.

Definition 4.9. A set $parts_{\mathbb{C}^\mathcal{A}}$ of partition elements *is a safe overapproximation* for program $P = (L, G_{\text{CFA}}, l_0)$, property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and a set of initial states $I \subseteq C$ if

- the initial states are covered by partition nodes considering the initial automaton state, $I \subseteq \bigcup_{(e,q_0) \in pn \wedge (pn,\cdot) \in parts_{\mathbb{C}^\mathcal{A}}} [\![(e, q_0)]\!]_{\mathbb{C}^\mathcal{A}}$,
- concrete successor configurations (c', q') of a partition node are considered by the same partition element as the partition node, $\forall (pn, bn) \in parts_{\mathbb{C}^\mathcal{A}} : \forall (e, q) \in pn :$ $\forall c \in [\![(e, q)]\!]_{\mathbb{C}^\mathcal{A}}, (l, op, l') \in G_{\text{CFA}} : c \xrightarrow{(l,op,l)} c' \implies (q' = q_\top \vee q' \in Q \wedge \exists C_{\text{sub}} \subseteq C :$ $c' \in C_{\text{sub}} \wedge (q, op, C_{\text{sub}}, q') \in \delta) \wedge \exists (e', q'') \in (pn \cup bn) : q' \sqsubseteq q'' \wedge c' \in [\![(e', q'')]\!]_{\mathbb{C}^\mathcal{A}}$,
- boundary nodes are covered by partition nodes considering the same or a more abstract automaton abstract state, $\forall (pn, bn) \in parts_{\mathbb{C}^\mathcal{A}} : \forall (e, q) \in bn : [\![(e, q)]\!]_{\mathbb{C}^\mathcal{A}} \subseteq$ $\{[\![(e', q')]\!]_{\mathbb{C}^\mathcal{A}} \mid q \sqsubseteq q' \wedge \exists (pn, \cdot) \in parts_{\mathbb{C}^\mathcal{A}} : (e', q') \in pn\}$, and
- the partition nodes are safe, $\forall (e, q) \in E^\mathcal{A}, (pn, bn) \in parts_{\mathbb{C}^\mathcal{A}} : (e, q) \in pn \implies q \neq$ $q_{\text{err}} \wedge q \neq q_\top$.

We claimed that a safe overapproximation guarantees that for every program path starting in an initial state $c_0 \in I$ a configuration sequence exists that witnesses safety of that path. Next, we show that these conditions on a safe overapproximation for program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$ are indeed sufficient to guarantee that every path $p \in paths_P(I)$ is safe w.r.t. $\mathcal{A}$.

Lemma 4.13. *Let $parts_{\mathbb{C}^\mathcal{A}}$ be a set of partition elements which is a safe overapproximation for program $P = (L, G_{\text{CFA}}, l_0)$, property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and a set of initial states $I \subseteq C$. Then, every path $p \in paths_P(I)$ is safe w.r.t. $\mathcal{A}$.*

Proof. See Appendix p. 263. □

Due to the previous lemma, we know that to guarantee program safety it would be sufficient that the set of partition elements in the partitioned certificate can be extended to a safe overapproximation. However, we already mentioned that a valid partitioned certificate should also support parallel validation. When more than one partition element is contained in the partitioned certificate, during validation partition elements are read and at the same time other partition elements, which were already read, are checked. We think that it is unintuitive that during validation more parts (partition elements) must be checked than are actually read. Hence, we require that the extension of the set of partition elements only extends the partition elements in the set, but does not add new partition elements. Nevertheless, the extension of two different partition elements may result in the same extension. Furthermore, we added the size of the basic certificate, which we want to use to abort certificate validation when the recomputed certificate becomes larger than the basic certificate, i.e., certificate reconstruction failed. Hence, an extension of the set of partition elements must exist that respects this upper bound. Since the set of all partition nodes in the extension reflects the recomputed certificate, this set must not be larger than the saved size of the basic certificate. These considerations lead us to the subsequent definition of a *valid partitioned certificate*.

Definition 4.10 (Valid Partitioned Certificate)**.** Let $\mathcal{PC}_{\mathbb{C}^\mathcal{A}} = (parts_{\text{sub}}, n)$ be a partitioned certificate. *Partitioned certificate $\mathcal{PC}_{\mathbb{C}^\mathcal{A}}$ is valid* for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$ if the set $parts_{\text{sub}}$ of partition elements can be extended to a set $parts_{\mathbb{C}^\mathcal{A}}$ of partition elements s.t.

- $parts_{\mathbb{C}^\mathcal{A}}$ is a safe overapproximation for P, $\mathcal{A}$, and I,
- the extension is restricted to the extension of partition elements, $\exists$ total surjective function $m : parts_{\text{sub}} \to parts_{\mathbb{C}^\mathcal{A}} : \forall (pn, bn) \in parts_{\text{sub}} : m((pn, bn)) = (pn', bn') \implies pn \subseteq pn' \wedge bn \subseteq bn'$, and
- the number of extended partition nodes is bounded by n, $|\bigcup_{(pn,\cdot)\in parts_{\mathbb{C}^\mathcal{A}}} pn| \leq n$.

After having defined when a partitioned certificate is valid, we want to ensure that this definition guarantees us that a valid partitioned certificate is a proper witness for program safety. We need to prove that if a partitioned certificate is valid for program P, property automaton $\mathcal{A}$, and initial states I, then P will be safe w.r.t. $\mathcal{A}$ and I. Due to the definition of a valid partitioned certificate and Lemma 4.13, we already know that all paths of program P starting in the set I of initial states are safe. Based on the definition of program safety, we can simply infer the desired witness property: valid partitioned certificates witness program safety.

Proposition 4.14. *If a partitioned certificate $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is valid for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$, then $P \models_I \mathcal{A}$.*

Proof. We need to show that all paths $p \in paths_P(I)$ are safe w.r.t. $\mathcal{A}$. Let $p \in paths_P(I)$ be an arbitrary path. Since $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is valid for P, $\mathcal{A}$, and I, there exists a set $parts_{\mathbb{C}^{\mathcal{A}}}$ of partition elements which is a safe overapproximation for program P, $\mathcal{A}$, and I. From Lemma 4.13, we conclude that p is safe w.r.t. $\mathcal{A}$. Hence, $P \models_I \mathcal{A}$. □

In the following, we will show that the certificates constructed by a process conformant producer are valid w.r.t. the states represented by the root node of the producer's ARG, which was constructed during successful producer verification. However, the consumer is typically interested in validity w.r.t. the states represented by the initial abstract state with which the producer analysis is started. Generally, the ARG's root node and the initial abstract state of the producer's analysis may not be identical. We only know that the set of concrete states represented by the initial abstract state is a subset of the concrete states represented by the root node. Thus, we need to be sure that a valid partitioned certificate remains valid after we shortened the set of initial states.

Corollary 4.15. *Let $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}} = (parts_{\text{sub}}, n)$ be a partitioned certificate which is valid for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$ and let $I_{\text{sub}} \subseteq I$ be a subset of the initial states. Then, $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is also valid for P, $\mathcal{A}$, and I_{sub}.*

Proof. See Appendix pp. 263 f. □

Until now, we know that valid partitioned certificates are proper witnesses for program safety. Hence, the producer should construct valid partitioned certificates. Next, we proceed to explain how the producer builds valid partitioned certificates.

4.3.2 Construction of Reduced, Partitioned Certificates

Remember that the goal of the partitioning approach is to divide the original certificate or the (highly) reduced certificate and read and check the certificate in parallel. We also discussed that the certificate must explicitly describe the split of the certificate, i.e., the division of the abstract states in the certificate. In a first step, the producer needs to decide how to divide the abstract states occurring in such a certificate. On the one hand, the consumer should not inspect abstract states multiple times and on the other hand, each element must be inspected. Hence, we require a partition [CLRC07, p. 1073] of the abstract states in the certificate. The following definition generally introduces the concept of a partition of a set of elements.

Definition 4.11 (Partition)**.** Let S be a non-empty set. A *partition of S* is a set $\text{partition}(S) = \{p_1, \ldots, p_k\}$ of non-empty, disjoint subsets p_i of S s.t. each element of S is contained in one of the subsets, formally, $\forall 1 \leq i \leq k : p_i \neq \emptyset \wedge \forall 1 \leq j \leq k : i \neq j \implies p_i \cap p_j = \emptyset$ and $\bigcup_{1 \leq i \leq k} p_i = S$.

Reconsider the highly reduced node set that we used to construct an example for a highly reduced certificate. This highly reduced node set was computed from the ARG shown in the top of Fig. 4.1, the ARG, which we obtained after the verification of our example program `SubMinSumDiv` with the sign dataflow analysis and property `pos@l5`. Next, we see a partition of that highly reduced node set. The partition consists of two subsets. The

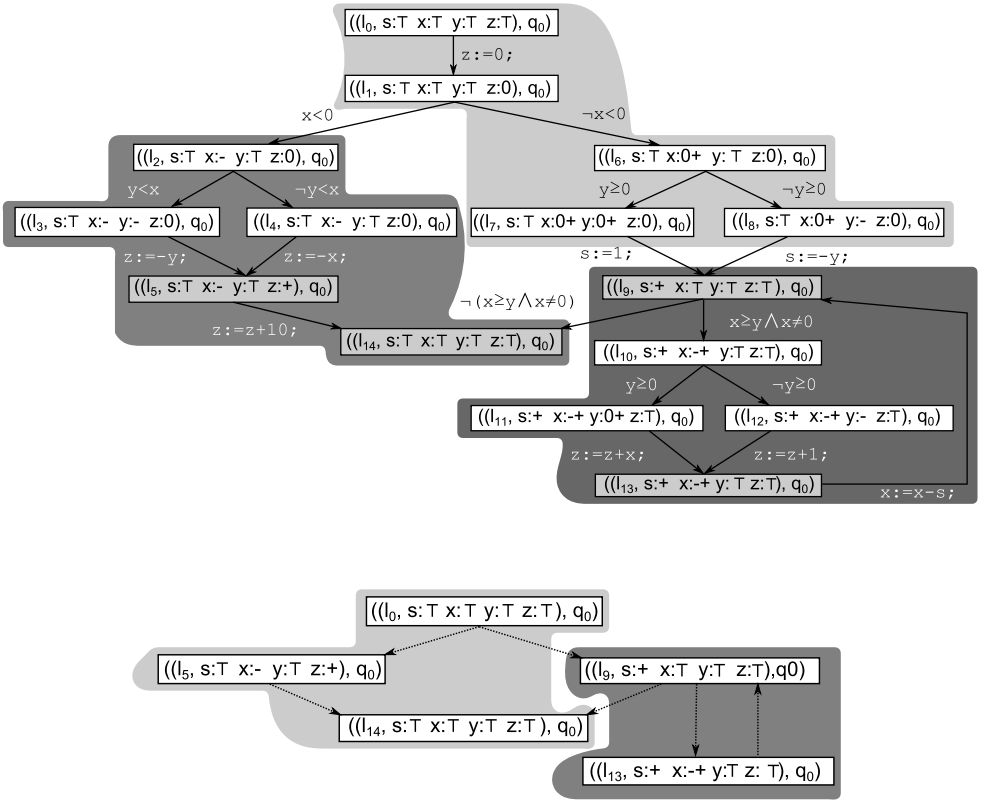

Figure 4.1: Top: ARG constructed by the CPA algorithm during the analysis of program `SubMinSumDiv` with CPA $\mathbb{L} \times \mathbb{S}$ enhanced with property automata `pos@l5` started in the initial abstract state $((l_0, \top_{\mathbb{S}}), q_0)$
Bottom: Vertex contraction of the above ARG to its highly reduced node set
Shaded areas represent our example partition of the respective node set.

first subset contains the covering nodes associated with the while loop and the second one consists of the root node and the remaining covering nodes.

$$\left\{ \left\{ \begin{array}{l} ((l_9, s:+\ x:\top\ y:\top\ z:\top), q_0), \\ ((l_{13}, s:+\ x:-+\ y:\top\ z:\top), q_0) \end{array} \right\}, \left\{ \begin{array}{l} ((l_0, s:\top\ x:\top\ y:\top\ z:\top), q_0), \\ ((l_5, s:\top\ x:-\ y:\top\ z:+), q_0), \\ ((l_{14}, s:\top\ x:\top\ y:\top\ z:\top), q_0) \end{array} \right\} \right\}$$

Throughout this subsection, we assume that the producer already computed a partition of the abstract states, which should become part of the certificate. In the next subsection (Sec. 4.3.3), we discuss how to compute a good partition.

When the producer has computed a proper partition, he uses the subsets in the partition to construct his certificate. Each subset in the partition defines one part of the constructed partitioned certificate. The subsets in the partition become the sets of partition nodes. However, we already discussed that the sets of partition nodes alone are not sufficient. Each set of partition nodes must be extended to a partition element, which stores additional information that allows the independent inspection of the part.

For the moment, let us assume that we have a directed graph [KKB05, p. 121] whose edges encode the interconnections between the abstract states in the certificate. Then, each edge in the graph represents the fact that for the independent inspection of the predecessor we require knowledge about the successor of that edge. The directed graph in the bottom of Fig. 4.1 encodes the dependencies among the highly reduced node set considered in the above example partition. The shaded areas describe the partition of the highly reduced node set presented above. We later explain how to generally construct a directed graph like the one in Fig. 4.1, which encodes the dependencies.

Since the dependencies among the abstract states are encoded in the directed graph, we only have to look at the graph to compute the additional information, the set of boundary nodes, necessary to independently check a part. Remember we want to inspect each partition element, i.e., each set of partition nodes, independently. To independently check a single abstract state in a set of partition nodes, we must know all its successor nodes in the directed graph. Hence, all its successors must be available in the partition element. If a successor node is part of the same set of partition nodes, it will be available during inspection. In the other case, we need to make it available, i.e., we must add the successor to the set of boundary nodes of the respective partition element. Generally speaking, for a directed graph modeling the dependencies among certain abstract states and a subset of these abstract states, the *boundary nodes of that subset in the directed graph* are all successors external to the subset.

Definition 4.12. Let $DG = (V, G_V)$ be a directed graph and $V_{\text{sub}} \subseteq V$ a subset of nodes. The *boundary nodes* $\text{bound}(V_{\text{sub}}, DG)$ *of* V_{sub} *in* DG are all successor nodes of nodes in V_{sub} which are not contained in V_{sub}, $\text{bound}(V_{\text{sub}}, DG) = \{v_{\text{bn}} \mid (v, v_{\text{bn}}) \in G_V \wedge v \in V_{\text{sub}} \wedge v_{\text{bn}} \notin V_{\text{sub}}\}$.

Let us look at the first subset of the presented partition and the directed graph shown in the bottom of Fig. 4.1. The first subset is the set of all nodes in the right shaded area of that graph. We observe that only one edge leaves the right shaded area. The successor node of that edge is the only boundary node for that subset in that directed graph. Hence, the subsequent equation displays the boundary nodes of the first subset of our example partition and the directed graph shown in the bottom of Fig. 4.1.

$$\{((l_{14}, s : \top\ x : \top\ y : \top\ z : \top), q_0)\}$$

If we want to use the previous definition to construct the sets of boundary nodes for the producer's certificate, we cannot use any arbitrary directed graph. To construct syntactically correct certificates, we require that the nodes are abstract states of the same abstract domain as the states in the certificate. Like we construct certificates for an enhanced configurable program analysis, we want to use directed graphs for the same enhanced CPA during their construction. The subsequent definition states when a directed graph is a *directed graph for an enhanced CPA*. In principle, the nodes must be abstract states considered by the enhanced CPA.

Definition 4.13. Let $\mathbb{C}^{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ considering the set $E_{\mathbb{C}^{\mathcal{A}}}$ of abstract states. A *directed graph* $DG_{\mathbb{C}^{\mathcal{A}}} = (V, G_V)$ *for* $\mathbb{C}^{\mathcal{A}}$ consists of a set of nodes $V \subseteq E_{\mathbb{C}^{\mathcal{A}}}$ and a set of edges $G_V \subseteq V \times V$.

Given a directed graph $DG_{\mathbb{C}^{\mathcal{A}}}$ for an enhanced CPA whose edges encode the dependencies among the certificate states and a subset p_i from the partition of certificate states, we can use the definition of boundary nodes of p_i in $DG_{\mathbb{C}^{\mathcal{A}}}$ to construct a partition element.

As already stated, the subset from the partition becomes the set of partition nodes of the partition element. The set of boundary nodes is computed according to the previous definition of boundary nodes of p_i in $DG_{\mathbb{C}^\mathcal{A}}$. We call such a partition element a *partition element for $DG_{\mathbb{C}^\mathcal{A}}$ and p_i*.

Definition 4.14. Let $DG_{\mathbb{C}^\mathcal{A}} = (V, G_V)$ be a directed graph for enhanced CPA $\mathbb{C}^\mathcal{A}$ and $V_{\text{sub}} \subseteq V$ a subset of the nodes. A *partition element* $\text{part}(V_{\text{sub}}, DG_{\mathbb{C}^\mathcal{A}})$ *for* V_{sub} *and* $DG_{\mathbb{C}^\mathcal{A}}$ consists of partition nodes V_{sub} and boundary nodes $\text{bound}(V_{\text{sub}}, DG_{\mathbb{C}^\mathcal{A}})$.

Revisit the first subset in the partition, for which we already computed the set of boundary nodes according to the directed graph shown in the bottom of Fig. 4.1. Below, we present the partition element for this set and that directed graph. We observe that the set of partition nodes corresponds to the first subset in the partition and the set of boundary nodes is the same as before.

$$\left(\left\{ \begin{array}{l} ((l_9, s : + \; x : \top \; y : \top \; z : \top), q_0), \\ ((l_{13}, s : + \; x : - + \; y : \top \; z : \top), q_0) \end{array} \right\} \; , \{((l_{14}, s : \top \; x : \top \; y : \top \; z : \top), q_0)\} \right)$$

Now that we know how to construct a single partition element, the construction of the set of partition elements is straightforward. The partition describes how to divide the abstract states in the certificate. Furthermore, we want to inspect each abstract state in the certificate exactly once. Thus, we use the definition from above to construct the partition element for each subset in the partition. The set of the constructed partition elements is the desired set of partition elements. The following definition formally defines this *set of partition elements for a partition and a directed graph for an enhanced CPA*.

Definition 4.15. Let $DG_{\mathbb{C}^\mathcal{A}} = (V, G_V)$ be a directed graph for enhanced CPA $\mathbb{C}^\mathcal{A}$ and let $\text{partition}(V) = \{p_1, \ldots, p_k\}$ be a partition of the nodes V. The *set of partition elements for* $\text{partition}(V)$ *and directed graph* $DG_{\mathbb{C}^\mathcal{A}}$ is $\text{parts}(\text{partition}(V), DG_{\mathbb{C}^\mathcal{A}}) = \{\text{part}(p_i, DG_{\mathbb{C}^\mathcal{A}}) \mid p_i \in \text{partition}(V)\}$.

For our example partition and the directed graph shown in the bottom of Fig. 4.1, we get the following set of partition elements. The set contains two partition elements. The first partition element is the one from above. The second one is the partition element for the second set in the example partition. Looking at the directed graph, we observe that only one edge in the graph starting in one of the nodes of the second set of the partition does not end in this set. The set of boundary nodes of the second partition element consists of the successor of that edge.

$$\left\{ \begin{array}{l} \left(\left\{ \begin{array}{l} ((l_9, s : + \; x : \top \; y : \top \; z : \top), q_0), \\ ((l_{13}, s : + \; x : - + \; y : \top \; z : \top), q_0) \end{array} \right\} \; , \{((l_{14}, s : \top \; x : \top \; y : \top \; z : \top), q_0)\} \right) \\ \left(\left\{ \begin{array}{l} ((l_0, s : \top \; x : \top \; y : \top \; z : \top), q_0), \\ ((l_5, s : \top \; x : - \; y : \top \; z : +), q_0), \\ ((l_{14}, s : \top \; x : \top \; y : \top \; z : \top), q_0) \end{array} \right\} \; , \{((l_9, s : + \; x : \top \; y : \top \; z : \top), q_0)\} \right) \end{array} \right\}$$

Given a partition of the states in that certificate that one wants to partition and a directed graph that properly encodes the dependencies between these states, we just explained how to construct the main part of the partitioned certificate, the set of partition elements. We continue to describe how to build a directed graph that encodes the dependencies between the states.

To be able to describe all dependencies, the nodes of the directed graph must be the set of all abstract states in that certificate that one wants to partition, i.e., the set of

all partition nodes or equivalently all nodes occurring in the partition. To model the dependencies, i.e., build the edges, we need to know when the inspection of an abstract state depends on another abstract state. Like in the previous approaches, the producer wants to use the ARG obtained after successful verification to generate the partitioned certificate and a subset of the ARG nodes becomes the set of all abstract states considered in the certificate. Hence, we must be able to infer the dependencies from the ARG.

Remember that we observed that checking of a single abstract state in the ARG is local to its successors. Since the producer's certificate does not always consider all abstract states, during inspection of an abstract state the validation will recompute deleted descendants and should stop as soon as we come to a descendant that is contained in the certificate. Thus, when we inspect an abstract state we may not always depend on its successors in the ARG, but on particular descendants. However, if we do not delete any nodes , i.e., we do not combine the two optimizations and use the complete set of ARG nodes, the structure of the ARG will already describe the required dependencies. To model the dependencies in general, for each abstract state considered by the certificate one needs to identify all descendants in the ARG that are also considered by the certificate and that can be reached on a path without other certificate states in between. Additionally, one needs to add an edge from those abstract state to each of their identified descendants. In principle, one can compare the described construction with a vertex contraction that contracts each abstract state in the certificate with all descendants that are recomputed during the inspection of that abstract state. That is why, we call the directed graph obtained in the described way a *vertex contraction of the ARG*. Next, we formally describe the graph construction discussed above.

Definition 4.16. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ and $N_{\text{sub}} \subseteq N$ a subset of nodes. The *vertex contraction of* $R^P_{\mathbb{C}^\mathcal{A}}$ *to vertex set* N_{sub} is a directed graph $VCG(N_{\text{sub}}, R^P_{\mathbb{C}^\mathcal{A}}) = (N_{\text{sub}}, G_{N_{\text{sub}}})$ for $\mathbb{C}^\mathcal{A}$ with edges $G_{N_{\text{sub}}} = \{(n, n') \in N_{\text{sub}} \times N_{\text{sub}} \mid \exists n = n_1, n_2, \ldots, n_m = n' : m > 1 \wedge \forall 2 \leq i \leq m-1 : n_i \notin N_{\text{sub}} \wedge \forall 1 \leq i \leq m-1 : \exists (n_i, \cdot, n_{i+1}) \in G_{ARG}\}$.

Based on the previous definition, we no longer need to believe, but can now see that the directed graph shown in the bottom of Fig. 4.1 properly models the dependencies among the nodes in the highly reduced node set. We only have to check that an edge between two nodes exists iff in the ARG displayed above a path from the predecessor to the successor exists s.t. all intermediate nodes on the path do not belong to the highly reduced node set.

With the definition of a vertex contraction, we have everything at hand to build a producer's partitioned certificate. Before we come to the producer's partitioned certificates, we generally describe how to construct a *partitioned certificate from an ARG and a partition of a subset of the ARG nodes*. The set of partition elements becomes the set of partition elements derived from the vertex contraction of the ARG to a subset of the ARG nodes and the partition of that subset. For the upper bound on partition nodes, the maximal number of partition nodes that are stored or recomputed, we use the same number as in the reduced certificate and add the size of the set of ARG nodes. This leads us to the definition of a *partitioned certificate from a partition and an ARG*.

Definition 4.17. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ and $\text{partition}(N_{\text{sub}}) = \{p_1, \ldots, p_k\}$ a partition of $N_{\text{sub}} \subseteq N$. The *partitioned certificate from* $\text{partition}(N_{\text{sub}})$ *and* $R^P_{\mathbb{C}^\mathcal{A}}$ is $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}}) = (\text{parts}(\text{partition}(N_{\text{sub}})), VCG(N_{\text{sub}}, R^P_{\mathbb{C}^\mathcal{A}})), |N|)$.

Next, we present the partitioned certificate from the example partition and the ARG shown in the bottom of Fig. 4.1. We observe that it contains the set of partition elements seen before plus the number 15, the number of nodes in that ARG.

$$\left\{\left\{\left(\left\{\begin{array}{l}((l_9, s: + \; x: \top \; y: \top \; z: \top), q_0),\\ ((l_{13}, s: + \; x: - + \; y: \top \; z: \top), q_0)\end{array}\right\}, \{((l_{14}, s: \top \; x: \top \; y: \top \; z: \top), q_0)\}\right)\right\}, 15\right\}$$
$$\left\{\left\{\left(\left\{\begin{array}{l}((l_0, s: \top \; x: \top \; y: \top \; z: \top), q_0),\\ ((l_5, s: \top \; x: - \; y: \top \; z: +), q_0),\\ ((l_{14}, s: \top \; x: \top \; y: \top \; z: \top), q_0)\end{array}\right\}, \{((l_9, s: + \; x: \top \; y: \top \; z: \top), q_0)\}\right)\right\}, 15\right\}$$

Essentially, the producer will not consider arbitrary subsets of the ARG nodes. Since we aim at the partitioning of basic or (highly) reduced certificates, the producer may select one out of the following three subsets: (1) the complete set of ARG nodes, (2) the reduced node set, or (3) the highly reduced node set. The producer must select the complete set if he does not combine the two optimizations. Whenever the producer wants to combine the two optimizations, the producer's choice depends on the capabilities of the analysis. If the transfer relation is monotonic, he may either choose the reduced or the highly reduced node set. In the other case, he must always use the reduced node set. To easily refer to one of the producer's options, the subsequent definition fixes the names for the three types of certificates a producer can construct.

Definition 4.18. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ and partition(N_{sub}) a partition of $N_{\text{sub}} \subseteq N$. The partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ is

- a *full, partitioned certificate from ARG* $R^P_{\mathbb{C}^\mathcal{A}}$ if $N_{\text{sub}} = N$,
- a *reduced, partitioned certificate from ARG* $R^P_{\mathbb{C}^\mathcal{A}}$ if $N_{\text{sub}} = N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}})$, and
- a *highly reduced, partitioned certificate from ARG* $R^P_{\mathbb{C}^\mathcal{A}}$ if $N_{\text{sub}} = N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$.

Previously, we displayed a partitioned certificate constructed from a partition of the highly reduced node set of the ARG shown in the top of Fig. 4.1. According to our definition, the presented certificate is a highly reduced, partitioned certificate. Remember that for our example (the ARG shown in Fig. 4.1), the reduced and the highly reduced node set are identical. The presented certificate is also a reduced, partitioned certificate. Next, we show a full, partitioned certificate from the same ARG. The full, partitioned certificate is constructed from the partition of the ARG nodes indicated by the shaded areas in Fig. 4.1.

$$\left\{\left\{\left(\left\{\begin{array}{l}((l_0, s: \top \; x: \top \; y: \top \; z: \top), q_0),\\ ((l_1, s: \top \; x: \top \; y: \top \; z: 0), q_0),\\ ((l_6, s: \top \; x: 0 + \; y: \top \; z: 0), q_0),\\ ((l_7, s: \top \; x: 0 + \; y: 0 + \; z: 0), q_0),\\ ((l_8, s: \top \; x: 0 + \; y: - \; z: 0), q_0)\end{array}\right\}, \left\{\begin{array}{l}((l_9, s: + \; x: \top \; y: \top \; z: \top), q_0),\\ ((l_2, s: \top \; x: - \; y: \top \; z: 0), q_0)\end{array}\right\}\right)\right\}\right.$$
$$\left.\left\{\left(\left\{\begin{array}{l}((l_2, s: \top \; x: - \; y: \top \; z: 0), q_0),\\ ((l_3, s: \top \; x: - \; y: - \; z: 0), q_0),\\ ((l_4, s: \top \; x: - \; y: \top \; z: 0), q_0),\\ ((l_5, s: \top \; x: - \; y: \top \; z: +), q_0),\\ ((l_{14}, s: \top \; x: \top \; y: \top \; z: \top), q_0)\end{array}\right\}, \emptyset\right)\right\}, 15\right\}$$
$$\left\{\left\{\left(\left\{\begin{array}{l}((l_9, s: + \; x: \top \; y: \top \; z: \top), q_0),\\ ((l_{10}, s: + \; x: - + \; y: \top \; z: \top), q_0),\\ ((l_{11}, s: + \; x: - + \; y: 0 + \; z: \top), q_0),\\ ((l_{12}, s: + \; x: - + \; y: - \; z: \top), q_0),\\ ((l_{13}, s: + \; x: - + \; y: \top \; z: \top), q_0)\end{array}\right\}, \{((l_{14}, s: \top \; x: \top \; y: \top \; z: \top), q_0)\}\right)\right\}, 15\right\}$$

Until now, we focused on how the producer generates partitioned certificates. Of course, we aimed at the construction of valid partitioned certificates. Nonetheless, we need to guarantee that within the certificate partitioning approach the producer constructs valid partitioned certificates. Like in the basic configurable program certification process, we assume that the producer generates certificates after a successful verification. Thus, the producer constructs partitioned certificates from well-formed ARGs. To show that the producer generates valid partitioned certificates, we prove that full, partitioned, reduced, partitioned, and highly reduced, partitioned certificates built from well-formed ARGs are valid.

To prove validity of these producer's certificates, we first of all need a proper extension of the respective set of partition elements. Due to the construction of the producer's certificates, an extension $\{(N, N)\}$, which only considers the single pair of ARG nodes, would be proper if it was a safe overapproximation. The next lemma guarantee us that such a set of partition elements is a safe overapproximation whenever the ARG is well-formed.

Lemma 4.16. *Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ which is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$. The set $parts_{\mathbb{C}^\mathcal{A}} := \{(N, N)\}$ of partition elements is a safe overapproximation for P, $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and $[\![root]\!]$.*

Proof. See Appendix p. 264. □

Since the producer uses a well-formed ARG during certificate construction, we conclude from the previous lemma that the set of partition elements $\{(N, N)\}$ is a proper extension. To prove validity, it remains to be shown that the extension $\{(N, N)\}$ is sufficient to prove the remaining properties of a valid certificate. Instead of showing these properties for each kind of producer certificate individually, we look at a broader class of certificates, which includes the producer's certificates, namely at all partitioned certificates $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ from a well-formed ARG $R^P_{\mathbb{C}^\mathcal{A}}$ and a partition $\text{partition}(N_{\text{sub}})$ of a subset N_{sub} of the ARG nodes. Based on the construction of those certificates $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, we easily infer the remaining properties of a valid certificate and conclude that those certificates are valid. This leads us to the subsequent lemma.

Lemma 4.17. *Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ which is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and $\text{partition}(N_{\text{sub}})$ a partition of $N_{\text{sub}} \subseteq N$. Then, the partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ from $\text{partition}(N_{\text{sub}})$ and $R^P_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and $[\![root]\!]$.*

Proof. From the definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, we conclude that partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}}) = (parts_{\text{sub}}, |N|)$. Let $parts_{\mathbb{C}^\mathcal{A}} := \{(N, N)\}$. From the previous lemma, we know that $parts_{\mathbb{C}^\mathcal{A}}$ is a safe overapproximation for P, $\mathcal{A}$, and $[\![root]\!]$. Let $m : parts_{\text{sub}} \to parts_{\mathbb{C}^\mathcal{A}}$ be a total function with $m((pn, bn)) = (N, N)$ for all $(pn, bn) \in parts_{\text{sub}}$. By definition of partition, $parts_{\text{sub}}$, and m, m is surjective. By definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, we get that $\forall (pn, bn) \in parts_{\text{sub}} : pn \subseteq N \wedge bn \subseteq N$. Thus, $\forall (pn, bn) \in parts_{\text{sub}} : m((pn, bn)) = (N, N) \implies pn \subseteq N \wedge bn \subseteq N$. We compute that $\left| \bigcup_{(pn,\cdot) \in parts_{\mathbb{C}^\mathcal{A}}} \right| = |N|$. Hence, $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ is valid for P, $\mathcal{A}$, and $[\![root]\!]$. □

We know that the producer constructs full, partitioned, reduced, partitioned, and highly reduced, partitioned certificates from the well-formed ARG obtained after successful verification. Based on the definition of these partitioned certificates and the previous lemma,

we simply conclude the desired property: the producer constructs valid partitioned certificates.

Proposition 4.18. *Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root)$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ which is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$. Every full, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, every reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, and every highly reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and $[\![root]\!]$.*

Proof. From definitions, we get $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N$. Hence, from the definition of full, reduced and highly reduced certificates, we know that for every full, partitioned certificate, every reduced, partitioned certificate, and every highly reduced certificate a set $N_{\text{sub}} \subseteq N$ and a partition partition(N_{sub}) of that set exist s.t. the partitioned certificate from partition(N_{sub}) and $R^P_{\mathbb{C}^\mathcal{A}}$ is the full, partitioned certificate, the reduced, partitioned certificate, and the highly reduced certificate, respectively. Thus, the previous lemma lets us conclude that every full, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, every reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, and every highly reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![root]\!]$. □

The previous theorem stated that the producer generated partitioned certificates are valid w.r.t. the states represented by the root node of the well-formed ARG used in certificate construction. We believe that the consumer is typically interested in validity w.r.t. the initial abstract state used during producer verification, i.e., the abstract e_0 the producer's ARG is well-formed for. The following corollary claims that the producer certificates are also valid w.r.t. the concrete states represented by the initial abstract state.

Corollary 4.19. *Let $R^P_{\mathbb{C}^\mathcal{A}}$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ which is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$. Every full, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, every reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, and every highly reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and $[\![e_0]\!]$.*

Proof. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root)$. From the previous proposition, we know that every full, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, every reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, and every highly reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and $[\![root]\!]$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is well-formed, we get $e_0 \sqsubseteq root$. Due to the requirements on a CPA's abstract domain, we have $[\![e_0]\!] \subseteq [\![root]\!]$. From Corollary 4.15, we conclude that every full, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, every reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$, and every highly reduced, partitioned certificate from ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$b and $[\![e_0]\!]$. □

So far, we know that the producer may use any partition to construct his ((highly) reduced) partitioned certificates. However, some partitions are more appropriate than others. For example, the partition determines the number of boundary nodes, the additional overhead in the certificate. Of course, a partition that consists of a single element provides the smallest overhead, but reading and checking of the certificate cannot be parallelized anymore. Furthermore, in case the partition is unbalanced or the partition elements are too large, during validation we waste more processing time than necessary waiting that a partition element is read and thus becomes ready for inspection. We continue to discuss in detail what the producer should consider to determine an appropriate partition and how the producer can construct an appropriate partition.

4.3.3 Finding a Good Partition of the Set of Partition Nodes

Reconsidering the construction of the producer's certificate, we observe that the partition of the set of all partition nodes, i.e., the partition of the nodes obtained by the vertex contraction of the ARG, determines the shape of that certificate and its size. In turn, the shape of the certificate influences the performance of the certificate validation.

For example,the certificate validation must read the first partition element before it can start the certificate inspection. The size of the first partition element compared to the size of the complete certificate influences the sequential proportion of certificate validation. Thus, its size should be small. To fully exploit parallel reading and checking of the certificate, reading the next partition element should never outlast the ongoing parallel inspection of already read partition elements.

Taking into account that more than one pair of two threads, namely one thread for reading and one for inspecting partition elements, may be used, we no longer know how partition elements are spread among the threads. To equally distribute the workload among the threads, we require that the inspection of the partition elements is balanced, i.e., the time required for the inspection of each partition element is roughly the same. Since we no longer know when and where a partition element is read, the time required for reading each partition element should be similar. Hence, the size of the partition elements must be equal. Furthermore, each partition element could be the first partition element read by a thread. All partition elements should be small.

So far, we discussed the objectives w.r.t. certificate validation, but we also want to optimize the size of the certificate. The overhead for certificate partitioning should be as small as possible, i.e., we want to store few boundary nodes. A minimal overhead can always be achieved with a partition that consists of a single set. However, in such a partition the partition elements are not small. The optimization criteria on the partition are conflicting. Thus, we need to find a good compromise among the criteria.

Of course, we could try to find a good compromise while considering the real numbers for the storage size, the reading time, and the inspection time for each node in the vertex contraction of the ARG. While the storage size can be determined exactly, the reading and inspection time cannot be that easily determined. Likely, these two times must be measured experimentally, but then they depend on the underlying hardware. Another opportunity is to estimate the times. The inspection time of a node could be estimated based on the number of ARG descendants of that node that are reached from that node before another node in the vertex contraction of the ARG is reached. Since the reading time is mainly prescribed by the storage size of the partition and boundary nodes in a partition element, one could try to estimate the reading time based on the storage size. However, we still have to consider lots of data to compute a good partition. Thus, we decided to use a less precise, but also simple and less laborious estimation. We use the number of abstract states to approximate the storage size, the reading time, and the inspection time.

With this approximation in mind, we can reformulate our optimization goal from above. To achieve an equal workload distribution for the inspection of the partition elements, the sets of partition nodes must have equal sizes. The partition should be balanced. Furthermore, we want to minimize the total number of boundary nodes to reduce the storage overhead. We think that often the minimization indirectly balances the reading time. On the one hand, a partition element hopefully contains significantly more partition nodes than boundary nodes. On the other hand, we can imagine that the number of boundary nodes may be almost equally spread among the partition elements.

To get a small reading time for the partition elements, the partition must divide all partition nodes into many, small sets – in our experiments, we fixed the size of these sets. These considerations lead us to the following optimization problem.

Definition 4.19. Let $VCG(N_{\text{sub}}R_P^{\mathbb{C}_\mathcal{A}}) = (N_{\text{sub}}, G)$ be a vertex contraction of ARG $R_P^{\mathbb{C}_\mathcal{A}}$ to vertex set N_{sub} and $k \in \mathbb{N}^+$. A k-partition $p = \{p_1, \ldots, p_k\}$ of N_{sub} is optimal if it is balanced, $\forall 1 \leq i \leq k : |p_i| \leq \left\lceil \frac{|N_{\text{sub}}|}{k} \right\rceil$, and its partitioned certificate has the smallest overhead,

$$\begin{aligned}
&\sum_{i=1}^{k} |\text{bound}(p_i, VCG(N_{\text{sub}}R_P^{\mathbb{C}_\mathcal{A}}))| = \sum_{i=1}^{k} |\{v \mid (v', v) \in G \wedge v' \in p_i \wedge v \notin p_i\}| \\
&= \min_{\{p'_1,\ldots,p'_k\}} \sum_{i=1}^{k} |\text{bound}(p'_i, VCG(N_{\text{sub}}R_P^{\mathbb{C}_\mathcal{A}}))| \\
&= \min_{\{p'_1,\ldots,p'_k\}} \sum_{i=1}^{k} |\{v \mid (v', v) \in G \wedge v' \in p'_i \wedge v \notin p'_i\}|
\end{aligned}$$

with $\{p'_1, \ldots, p'_k\}$ partition of N_{sub} and $\forall 1 \leq i \leq k : |p'_i| \leq \left\lceil \frac{|N_{\text{sub}}|}{k} \right\rceil$.

In this thesis, we neither discuss the complexity of this optimization problem nor present an algorithm that computes the optimal solution. For our purposes, the construction of a certificate for an efficient consumer validation, it is sufficient that in practice we can compute good, nearly balanced partitions with few overhead. At first [Jak15], we used a random partition computation. Later, we observed that our optimization problem is related to graph partitioning [GJ79, p. 209], especially *balanced graph partitioning* [AR04], for which we know that it is NP-complete [AR04, GJ79, p. 209] and that an approximate solution bounded by a constant factor cannot be computed in polynomial time unless $P = NP$ [AR04]. Based on this insight, Henrik Bröcher [Brö16] implemented, adapted, and evaluated existing heuristics known for graph partitioning. In his bachelor thesis [Brö16], he considered the following heuristics: (1) greedy graph growing [KK98], a global, greedy approach, which consecutively builds the partition creating the sets p_i one after another always adding the vertex with the best gain until the current p_i is full, (2) a local optimization of an existing partition with the help of the Fiduccia und Mattheyses approach [FM82], and (3) multi-level partitioning [KK98]. Different variants can be configured in all three cases. For example, in greedy graph growing the selection criterion, i.e., the definition of the gain, can be exchanged. The initial partition and the optimization goal can be configured in the optimization approach. Similarly, the partitioning and the optimization strategy can be selected for the multi-level partitioning. Thus, in our experiments we can select from various heuristics and we experimentally tried out which seems to be best for a certain set of verification tasks.

Computing a good partition of all partition nodes was the last aspect in the construction of a partitioned certificate. Next, we continue with the inspection of partitioned certificates.

4.3.4 Validation of Partitioned Certificates

The consumer validates partitioned certificates to determine whether a program is safe w.r.t. a property automaton and a set of initial states. Following the previous approaches, we use a meta validation algorithm that can be configured by a CCV that fits to the

Algorithm 5: Validation algorithm for partitioned certificates

Input: A CCV $\mathbb{V}^{D_{\mathbb{C}^{\mathcal{A}}}} = ((C, (E, \top, \bot, \sqsubseteq, \sqcup), [\![\cdot]\!]), \rightsquigarrow, \mathsf{cover})$, initial abstract state $e_0 \in E$, partitioned certificate $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}} = (parts_{\text{sub}}, n) \in 2^{2^E \times 2^E} \times \mathbb{N}$, program $P = (L, G_{\text{CFA}}, l_0)$
Output: Boolean indicator, if partitioned certificate $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is valid
Data: A set reached of elements of E, a set waitlist of elements of E

```
1  reached := ∅; waitlist := ∅;
2  for each (pn, bn) ∈ parts_sub do
3      reached := reached ∪ pn; waitlist := pn;
4      while waitlist ≠ ∅ ∧ |reached| ≤ n do
5          pop e from waitlist;
6          for each g ∈ G_CFA do
7              for each (e, g, e') ∈⇝ do
8                  if ¬cover(e', pn ∪ bn) ∧ e' ∉ reached then
9                      reached := reached ∪ {e'}; waitlist := waitlist ∪ {e'};
10 if ¬cover(e_0, ⋃_{(pn,·)∈parts_sub} pn) then
11     return false
12 return |reached| ≤ n ∧ ¬∃(·, q) ∈ reached : (q = q_err ∨ q = q_⊤)
           ∧ ⋃_{(·,bn)∈parts_sub} bn ⊆ ⋃_{(pn,·)∈parts_sub} pn
```

partitioned certificate being inspected. Algorithm 5, an adaption of the second validation algorithm presented in our previous paper [Jak15], displays our meta validation algorithm for partitioned certificates. Next to the partitioned certificate and the CCV, the validation algorithm also requires an abstract state, describing the concrete set of initial states for which program safety should be examined, and the program itself.

Note that Algorithm 5 only describes how to validate a partitioned certificate, but not how to read and check it in parallel. Since Algorithm 5 first inspects all partition elements in the for loop and in each iteration of the for loop it only refers to information saved in the inspected partition element or to information already explored by the validation algorithm, achieving parallel reading and checking is straightforward. One uses two threads, one reads the partitioned certificate and the other executes the validation algorithm. Of course, the maximal number n of partition nodes must be read before any partition element and must be given to the validation algorithm before it actually starts the validation. Furthermore, one either needs to tell the validation algorithm in advance how many partition elements exist or that all partition elements are read. Moreover, one could use a synchronized data structure in which a partition element is put directly after it is read and from which the validation algorithm takes the next partition element for checking. Additionally, one needs to assure that when not all partition elements have been read and the validation algorithm has checked all available partition elements, the validation algorithm waits, e.g., blocks, until the next partition element is read and put into the shared data structure. In the following, we explain how Algorithm 5 uses a partitioned certificate to determine program safety.

From previous considerations, we know that a valid partitioned certificate guarantees program safety and that a well-acting producer generates valid partitioned certificates.

Furthermore, it is totally fine for our certification approach that the consumer validation only accepts certificates of well-acting producers. Based on these insights, we designed a validation algorithm that checks that the input certificate is a valid partitioned certificate a well-acting producer could have constructed.

To examine validity, the validation algorithm requires an extension of the certificate's set of partition elements $parts_{\text{sub}}$. The validation algorithm uses the following extension which it never stores and only implicitly refers to. Each abstract state that is added to reached and waitlist in line 9 while the validation algorithm considers partition element (pn, bn) belongs to the extension of pn. Furthermore, we add e' to the set of boundary nodes bn when the branch condition in line 8 evaluates to false because $e' \in$ reached. The extension does not add new partition elements. Note that this is sufficient because the producer constructs his certificate such that at most partition nodes are removed and we added the extension for boundary nodes and the check $e' \notin$ reached solely to guarantee the termination of the validation algorithm. Based on the described extension, the validation algorithm decides whether the given partitioned certificate is valid.

Due to the construction of the extension, we already now that the extension is restricted to partition elements. Looking at Algorithm 5, we further observe that little by little all partition nodes of the extension are added to reached and at line 12 the data structure reached contains all partition nodes of the extension. Thus, Algorithm 5 either returns false or in line 12 the size of the reached set, i.e., the number of partition nodes, is less or equal than n.

It remains to be explained how the validation algorithm examines if the extension is a safe overapproximation. In line 10, the validation algorithm checks if the partition nodes cover the initial abstract state. We assume that the initial abstract state describes all initial states for which program safety should be determined and also contains the initial state of the property automaton. Due to the requirements on the coverage check, we know that if that check does not fail, the initial states considered for program safety will be covered by appropriate partition nodes. Note that it is sufficient to look at the partition nodes in the partitioned certificate instead of all partition nodes occurring in the extension because the producer puts the ARG root node, which covers the initial abstract state, into one of the constructed sets of partition nodes.

Looking at the for loop in the validation algorithm, we observe the following. In each iteration, all partition nodes that belong to the extended partition element obtained by the extension of the partition element inspected in the current iteration are added to the data structure waitlist. We abort the inspection of the partition nodes, when $|\text{reached}| > n$, i.e., the partitioned certificate cannot be valid. Note that abortion is needed to guarantee that our validation algorithm terminates on any certificate, no matter if it is valid or not. Furthermore, for each abstract state in the waitlist, we either compute its abstract transfer successors in the program or the number of partition nodes is already too high for a valid certificate ($|\text{reached}| > n$). In lines 8 and 9, we ensure that each explored abstract successor is covered by the set of partition and boundary nodes, it is already contained in reached, i.e., it is covered by the extended set of boundary nodes, or we will add the abstract successor to reached in line 9, and, thus, to the set of extended partition nodes. Due to the requirements on the transfer relation, the requirements on the coverage check, and the construction of the extension of the set of partition elements, we conclude the following. Every concrete successor configuration of a partition node that is considered in the current iteration of the for loop is considered by the same partition element as the partition node or the validation algorithm already failed to prove the validity of the partitioned certificate, i.e., $|\text{reached}| > n$. We do not add new partition elements to the

extension. Abstract states belonging to the extension are explored and inspected in the same iteration in which we examine the partition nodes of the partition element that these abstract states extend or the inspection of the partitioned certificate is aborted because we already failed to prove its validity. Hence, the validation algorithm determines that concrete successor configurations of a partition node are considered by the same partition element as the partition node or validation will fail. To check that the boundary nodes are covered by partition nodes, the validation algorithm simply examines whether the set of all boundary nodes is a subset of the set of all partition nodes. This simplification is sufficient because the producer only uses boundary nodes that are also partition nodes and we only add additional boundary nodes for which we already know that they are partition nodes. Finally, the validation algorithm either returns false or at line 12 the validation algorithm recognizes that the reached set, the set of all partition nodes in the extension, is safe. The validation algorithm returns false or all four conditions on a safe overapproximation are examined successfully.

Summing up, we designed the validation algorithm in such a way that it inspects the validity of a partitioned certificate and should successfully inspect the producer's partitioned certificates. If we were careful enough with our design, the consumer analysis should fulfill the two important properties, soundness and relative completeness, which we require to safely and successfully apply our certificate partitioning approach. Next, we study whether we really achieved these goals.

4.3.5 Properties of Reduced, Partitioned Certificate Validation

Like the approaches before, our partitioning approach must ensure the two properties, soundness and relative completeness, mentioned in the introduction. We start to discuss soundness. Remember that soundness means that the validation algorithm for partitioned certificates (Algorithm 5) may only return true if the partitioned certificate witnesses program safety. To provide partitioned certificates that witness program safety, the producer generates valid partitioned certificates, for which we know that they are proper witnesses. Furthermore, we designed our validation algorithm in such a way that it accepts the producer's certificates. Similar to the previous approaches, we show that the validation algorithm will only return true if the input partitioned certificate is valid.

In its main part, the validation algorithm implicitly constructs an extended set of partition elements and simultaneously checks whether that set is a safe overapproximation. Note that the validation algorithm should fail if that set is not a safe overapproximation. However, the validation algorithm never explicitly stores the extended set of partition elements we had in mind when we designed the validation algorithm. Thus, we use the observation that if we have a safe overapproximation, we can unite all partition and boundary nodes in a single set S and the resulting set of partition elements $\{(S, S)\}$ is again a safe overapproximation. Since the validation algorithm checks that the set of all boundary nodes is a subset of the set of all partition nodes, we know that if the validation algorithm returns true, at the end of the validation algorithm $\mathsf{reached}$ will be the union of all partition and boundary nodes contained in the extended set of partition elements the algorithm implicitly considered. To ensure that the validation algorithm constructs a safe overapproximation during validation, we simply prove that the set of partition elements $\{(\mathsf{reached}', \mathsf{reached}')\}$, $\mathsf{reached}'$ denoting the final version of the set $\mathsf{reached}$, is a safe overapproximation whenever the validation algorithm returns true.. The following lemma claims that the set of partition elements $\{(\mathsf{reached}', \mathsf{reached}')\}$ is a safe overapproximation if the validation algorithm returns true.

Lemma 4.20. *Let* $\mathsf{reached}'$ *denote the reached set at the state of termination of Algorithm 5. If Algorithm 5 started with configurable certificate validator* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}$ *for abstract domain* $D_{\mathbb{C}^{\mathcal{A}}}$ *enhanced with property automaton* $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$*, program* P*, initial abstract state* $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$*, and partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *returns true, then the set* $parts_{\mathbb{C}^{\mathcal{A}}} := \{(\mathsf{reached}', \mathsf{reached}')\}$ *of partition elements is a safe overapproximation for* P*,* $\mathcal{A}$*, and* $[\![e_0]\!]$*.*

Proof. See Appendix p. 265. □

To prove soundness, the previous lemma is sufficient. Similar to the previous two configurable program certification instances, we are not only interested in soundness. Additionally, the validation algorithm should accept valid partitioned certificates only. In the proof, we again avoid to use the implicitly constructed extension of the set of partition elements. Once more, we use the set of partition elements $\{(\mathsf{reached}', \mathsf{reached}')\}$ instead. Note that this is possible because when the validation algorithm returns true, it ensures that the set of all boundary and partition nodes is equivalent with the set of partition nodes. Now, we prove validity analogous to how we proved that a certificate constructed from a well-formed ARG and a partition of a subset of the ARG nodes is valid. Based on the previous lemma and the construction of $\mathsf{reached}'$, we show that the partitioned certificate can be extended to the safe overapproximation $\{(\mathsf{reached}', \mathsf{reached}')\}$ s.t. the extension fulfills the other constraints required for a valid certificate.

Theorem 4.21. *If Algorithm 5 started with configurable certificate validator* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}$ *for abstract domain* $D_{\mathbb{C}^{\mathcal{A}}}$ *enhanced with property automaton* $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$*, program* P*, initial abstract state* $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$*, and partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *returns true, then* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *is valid for* P*,* $\mathcal{A}$*, and* $[\![e_0]\!]$*.*

Proof. Let $\mathsf{reached}'$ be the reached set at the state of termination of Algorithm 5 and $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}} = (parts_{\text{sub}}, n)$. Let $parts_{\mathbb{C}^{\mathcal{A}}} := \emptyset$ if $parts_{\text{sub}} = \emptyset$, $parts_{\mathbb{C}^{\mathcal{A}}} := \{(\mathsf{reached}', \mathsf{reached}')\}$ otherwise. From Lemma 4.20, $\{(\mathsf{reached}', \mathsf{reached}')\}$ is a safe overapproximation for P, $\mathcal{A}$, and $[\![e_0]\!]$. Due to Algorithm 5, $parts_{\text{sub}} = \emptyset$ implies $\mathsf{reached}' = \emptyset$. By definition, if $(\emptyset, \emptyset)$ is a safe overapproximation for P, $\mathcal{A}$, and $[\![e_0]\!]$, so is $\emptyset$. Thus, $parts_{\mathbb{C}^{\mathcal{A}}}$ is a safe overapproximation for P, $\mathcal{A}$, and $[\![e_0]\!]$. Furthermore, if $parts_{\text{sub}} = \emptyset$, let $m : parts_{\text{sub}} \to parts_{\mathbb{C}^{\mathcal{A}}}$ be arbitrary. If $parts_{\text{sub}} \neq \emptyset$, let $m : parts_{\text{sub}} \to parts_{\mathbb{C}^{\mathcal{A}}}$ be a total function with $m((pn, bn)) = (\mathsf{reached}', \mathsf{reached}')$ for all $(pn, bn) \in parts_{\text{sub}}$. By definition, m is surjective. Since Algorithm 5 explores all partition elements $(pn, bn) \in parts_{\text{sub}}$, added pn to $\mathsf{reached}$ in line 3 for each (pn, bn), and never removes a state from $\mathsf{reached}$, we get that $\forall (pn, bn) \in parts_{\text{sub}} : pn \subseteq \mathsf{reached}'$. Due to line 12 and Algorithm 5 returning true, $\forall (pn, bn) \in parts_{\text{sub}} : bn \subseteq \bigcup_{(pn', \cdot) \in parts_{\text{sub}}} pn' \subseteq \mathsf{reached}'$. Thus, $\forall (pn, bn) \in parts_{\text{sub}} : m((pn, bn)) = (pn', bn') \implies pn \subseteq pn' \wedge bn \subseteq bn'$. Since Algorithm 5 returns true, $\mathsf{reached}' = \bigcup_{(pn, \cdot) \in parts_{\mathbb{C}^{\mathcal{A}}}} pn \leq n$. Hence, $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$. □

Since valid partitioned certificates witness program safety, we easily conclude soundness.

Corollary 4.22 (Soundness)**.** *If Algorithm 5 started with configurable certificate validator* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}$ *for abstract domain* $D_{\mathbb{C}^{\mathcal{A}}}$ *enhanced with property automaton* $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$*, program* P*, initial abstract state* $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$*, and partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *returns true, then* $P \models_{[\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}}} \mathcal{A}$*.*

Proof. From the previous theorem, we know that $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ is valid for P, $\mathcal{A}$, and $[\![e_0]\!]$. Now, Proposition 4.14 lets us conclude that $P \models_{[\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}}} \mathcal{A}$. □

Now that we have shown the soundness property, we continue with the second property, relative completeness. Relative completeness of the partitioning approach means that the validation algorithm for partitioned certificates must return true if the producer sticks to the certificate partitioning approach and the validation algorithm inspects the producer's certificate – either a full, partitioned, a reduced, partitioned or a highly reduced, partitioned certificate –, and the same program and the same set of initial states as the producer. We examine relative completeness in two major steps. First, we prove termination of the validation algorithm. Second, we show that if the validation algorithm terminates, it will return true.

The producer analysis, i.e., the CPA algorithm, may only terminate if the input program is finite. Since for any ARG its set of ARG nodes is finite, we conclude from the certificate construction process applied by a process conformant producer that the constructed partitioned certificate is finite. We only need to prove termination of the validation algorithm when its input program and partitioned certificate are finite. The following lemma shows termination of the validation algorithm in this case.

Lemma 4.23 (Termination)**.** *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *and program* $P = (L, G_{\text{CFA}}, l_0)$ *be finite. Then, Algorithm 5 started with* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*,* P*, initial abstract state* $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$*, and finite partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *terminates.*

Proof. See Appendix pp. 265 f. □

Termination of the validation algorithm is guaranteed. Next, we examine whether the validation algorithm indeed returns true. Four reasons exist why the validation algorithm might not return true.

1. The initial abstract state is not covered by the partition nodes. Some relevant program paths might be missed.
2. The size of reached is greater than n. The validation algorithm stopped before it explored all relevant program paths.
3. Some abstract states in reached consider the automaton's error state. The program may be unsafe.
4. The set of all boundary nodes is not a subset of the set of all partition nodes. Some boundary nodes are not explored in other partitions. The state space exploration could be incomplete.

Due to the construction of the producer's certificate, we know that the root node of the producer's ARG is contained in one set of partition nodes and all boundary nodes occur in a set of partition nodes. Thus, the fourth case can never become true. Furthermore, the producer uses a well-formed ARG to construct its certificate. The rootedness property of a well-formed ARG, the requirements on $\sqsubseteq$, and the root node being part of the partition nodes let us conclude that the partition nodes cover the initial abstract state used by the producer. The consumer either uses the same or a less abstract initial abstract state. The set of partition nodes also covers the consumer's initial abstract state. As already discussed in the basic configurable program certification approach, the consumer must not use an arbitrary configurable certificate validator, but one that is derived from the producer's configuration and that uses a well-behaving coverage check. Since the coverage check is well-behaving, the coverage check will detect that the consumer's initial abstract

state is covered by the partition nodes. The first cause cannot occur. We only need to examine the second and third cause. Similar to the certificate reduction approach, we separately discuss the case of a highly reduced, partitioned certificate afterwards.

Since the producer uses a well-formed ARG $R^P_{\mathbb{C}^\mathcal{A}}$ to construct his certificate, the set N of ARG nodes is safe. Furthermore, the size of the set N of ARG nodes becomes the value of n in the producer's certificate, $|N| \leq n$. For the set N, the second and third cause do not apply. We already discussed for our reduced certification approach that we might not be able to fully reconstruct the set of ARG nodes. Every reduced certificate $\text{cert}_\text{R}(R^P_{\mathbb{C}^\mathcal{A}}) = (\mathcal{C}^\text{R}_{\mathbb{C}^\mathcal{A}}, |N|)$ can be transformed into a reduced, partitioned certificate $(\{(\mathcal{C}^\text{R}_{\mathbb{C}^\mathcal{A}}, \emptyset)\}, |N|)$ that will be constructed by the producer if he considers the partition $\text{partition}(\mathcal{C}^\text{R}_{\mathbb{C}^\mathcal{A}}) = \{\mathcal{C}^\text{R}_{\mathbb{C}^\mathcal{A}}\}$. If we combine reduction with partitioning, we also cannot always reconstruct the complete set of ARG nodes. Reconsidering the second and third reason, we observe that they also do not apply for any subset of N. When we can show that at line 12 the computed set reached is a subset of N, we are almost done. To include the full, partitioned certificate, the reduced, partitioned certificate, but also any certificate we obtain when we combine our certification approach with the Programs from Proofs approach, we want to prove that reached is a subset of N for any partitioned certificate that is constructed from a well-formed ARG and a partition of a subset of the ARG nodes that includes at least the reduced node set. We claim that at line 12 reached will be a subset of the ARG nodes, if the validation algorithm for partitioned certificates is started with such a certificate plus a validation configuration, initial abstract state, and finite program that fits to the well-formed ARG used for certificate construction. Note that these condition are always met when the producer and the consumer stick to the partitioning process and they check the same program. The following lemma supports our claim about reached.

Lemma 4.24. *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_\text{ARG}, root, N_\text{cov})$ be an ARG for finite program P and $\mathbb{C}^\mathcal{A}$, and $R^P_{\mathbb{C}^\mathcal{A}}$ be well-formed for $e \in E_{\mathbb{C}^\mathcal{A}}$. Furthermore, let $N_\text{R}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N_\text{sub} \subseteq N$ and $\text{partition}(N_\text{sub})$ a partition of N_sub. If Algorithm 5 started with CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_\text{sub}), R^P_{\mathbb{C}^\mathcal{A}})$ from $\text{partition}(N_\text{sub})$ and $R^P_{\mathbb{C}^\mathcal{A}}$, then at line 12* reached $\subseteq N$.

Proof. See Appendix pp. 266 f. □

Based on the previous lemma and our previous observations, we simply conclude that the validation algorithm returns true whenever the validation algorithm for partitioned certificates is started with a partitioned certificate from a well-formed ARG that considers at least the reduced node set and with a validation configuration, initial abstract state, and finite program that fits to the well-formed ARG used for certificate construction.

Lemma 4.25. *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_\text{ARG}, root, N_\text{cov})$ be an ARG for finite program P and $\mathbb{C}^\mathcal{A}$, and $R^P_{\mathbb{C}^\mathcal{A}}$ be well-formed for $e \in E_{\mathbb{C}^\mathcal{A}}$. Furthermore, let $N_\text{R}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N_\text{sub} \subseteq N$ and $\text{partition}(N_\text{sub})$ a partition of N_sub. Algorithm 5 started with CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_\text{sub}), R^P_{\mathbb{C}^\mathcal{A}})$ from $\text{partition}(N_\text{sub})$ and $R^P_{\mathbb{C}^\mathcal{A}}$ returns true.*

Proof. See Appendix p. 267. □

In principle, we showed relative completeness except for the special case of a highly reduced, partitioned certificate. We proposed highly reduced, partitioned certificates as a

likely more compact alternative, which may only be used if the CPA's transfer relation is monotonic. As discussed in the reduction approach, we must delete abstract states for which only less abstract states can be recomputed if we want an alternative certificate that contains less abstract states. Hence, we cannot ensure that we recompute a subset of the ARG nodes. To exclude the second and third failure cause, we once again want to show that the validation algorithm recomputes a set reached that is less abstract than the set of ARG nodes and that is not larger than the set of ARG nodes. Thus, we explicitly show that the second cause cannot become true. Based on the definition of $\sqsubseteq$ and the fact that the set of ARG nodes in a well-formed ARG is safe, we can conclude that a set that is less abstract than the set of ARG nodes is also safe. In case, we can show that reached fulfills the above requirements, the third cause does not apply, too. The next lemma states that the validation algorithm guarantees the above requirements when the validation algorithm is started with a highly reduced, partitioned certificate from a well-formed ARG considering a CPA with a monotonic transfer relation, and with a validation configuration, initial abstract state, and finite program that fits to the well-formed ARG used for certificate construction.

Lemma 4.26. *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving, let transfer relation* $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ *be monotonic,* $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for finite program* P *and* $\mathbb{C}^{\mathcal{A}}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. If Algorithm 5 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and highly reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ *from* $\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}))$ *and* $R^P_{\mathbb{C}^{\mathcal{A}}}$*, then at line 12* $\mathsf{reached} \sqsubseteq N$ *and* $|\mathsf{reached}| \leq |N|$*.*

Proof. See Appendix pp. 268 f. □

Due to the previous lemma and our previous observations, we easily infer that the validation algorithm (Algorithm 5) returns true when the validation algorithm is started with a highly reduced, partitioned certificate from a well-formed ARG for a CPA considering a monotonic transfer relation and with a validation configuration, initial abstract state, and finite program that fits to the well-formed ARG used for certificate construction.

Lemma 4.27. *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving, let transfer relation* $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ *be monotonic,* $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for finite program* P *and* $\mathbb{C}^{\mathcal{A}}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*. Algorithm 5 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and highly reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ *from* $\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}))$ *and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *returns true.*

Proof. See Appendix p. 270. □

The previous lemmas state that the partitioning approach is relatively complete whenever the producer constructs full, partitioned or reduced, partitioned certificates, or highly reduced, partitioned certificates and the transfer relation considered by the producer's CPA is monotonic. We integrate the results of the lemmas in a single theorem, which states relative completeness of our certificate partitioning approach.

Theorem 4.28 (Relative Completeness)**.** *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *which is well-behaving. Furthermore, let* $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for finite program* P *and* $\mathbb{C}^{\mathcal{A}}$*, and* $R^P_{\mathbb{C}^{\mathcal{A}}}$ *be well-formed for* $e \in E_{\mathbb{C}^{\mathcal{A}}}$*.*

1. *Algorithm 5 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and full, partitioned* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N), R^P_{\mathbb{C}^{\mathcal{A}}})$ *or reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

2. *If the transfer relation* $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ *of CPA* $\mathbb{C}^{\mathcal{A}}$ *is monotonic, then Algorithm 5 started with CCV* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and highly reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

Proof. The theorem directly follows from Lemma 4.25, the definition of full, partitioned and reduced, partitioned certificates, and Lemma 4.27. □

Our theorem of relative completeness still considers semi-automatic configurations for the consumer analysis. The coverage check is not automatically derived and the consumer's initial abstract state is not fixed. As before, the previous theorem reveals us when the approach becomes fully automatic. Namely, the termination check operator used in the producer configuration must be a proper coverage check, i.e., it must be well-behaving, and the consumer must reuse the producer's initial abstract state. The following corollary states that these conditions are sufficient to get a fully automatic partitioning approach.

Corollary 4.29. *Let* $\mathbb{C}^{\mathcal{A}}$ *be a CPA,* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}$ *which is well-behaving,* P *be a program, and* $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$*. If Algorithm 2 started with CPA* $\mathbb{C}^{\mathcal{A}}$*, initial abstract state* e_0*, initial precision* $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$*, and program* P *returns* $(\mathit{true}, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$*,*

1. *then Algorithm 5 started with configurable certificate validator* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* e_0*, and full, partitioned* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N), R^P_{\mathbb{C}^{\mathcal{A}}})$ *or reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

2. *and* $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ *is monotonic, then Algorithm 5 started with configurable certificate validator* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*, program* P*, initial abstract state* e_0*, and highly reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ *returns true.*

Proof. From Corollary 3.3, we know that $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}$ is a coverage check. Hence, $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}})$ is a CCV. From Proposition 2.8, we know that $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is well-formed for e_0. Since Algorithm 2 terminates, we conclude that P is finite. Now, the corollary follows from the previous theorem. □

We assume that the producer constructs a highly reduced, partitioned certificate only when the transfer relation used during verification is monotonic. Thus, whenever the producer successfully checks a program P using a CPA $\mathbb{C}^{\mathcal{A}}$ and a proper initial abstract state e_0, he can construct any of the three certificates which is appropriate and the consumer validation started with one of these certificates, e_0, and the CCV automatically derived from CPA $\mathbb{C}^{\mathcal{A}}$ finishes with outcome program is safe and can be executed. Again, all results can easily be transferred to program safety. As before, we only need to ensure that the producer and the consumer consider a proper initial abstract state that includes all concrete states with a control state that refers to the initial program location. In summary, no matter whether we solely use the second optimization or combine our two optimizations the partitioning approach always fulfills the requirements stated in the introduction. The next section describes our experimental evaluation of our two, orthogonal optimizations.

4.4 Evaluation

Up to now, we described the basic configurable certification approach and several variants, which should improve the basic approach. However, only the basic configurable certification approach has already been evaluated. In this section, we carry on with our evaluation of the configurable program certification approach. We want to compare the different variants of our configurable program certification approach to find out which variant is best. Furthermore, we would like to show that the variants discussed in this chapter improve on the basic configurable program certification. Additionally, we continue to study when our configurable program certification outperforms verification (significantly), especially considering the certification techniques presented in this chapter. Finally, we want to check our configurable program certification approach against existing certification approaches. Before we examine these aspects, we first describe the evaluation setting.

4.4.1 Evaluation Setting

To examine our research questions, we reuse the verification tasks and configurations already known from the evaluation of the basic approach. Furthermore, for our validation approaches we also reuse the validation configurations. In the following, we describe the remaining components for our experiments: the certificates, the validation competitors, and the experimental set up.

Certificate Generation For our evaluation, we generated for each tasks all five certificates considered in this chapter, namely a reduced certificate, a highly reduced certificate, a full, partitioned certificate, a reduced, partitioned certificate, and a highly reduced, partitioned certificate. In contrast to verification (except for CEGAR model checking) and the construction of the basic certificate, we require the ARG for the construction of these five certificates. For each pair of verification task and certificate type, we run the verification including ARG construction once and generate the respective certificate afterwards.

As already mentioned, the implemented ARG (structure) deviates from our formalization. Whenever certificate generation utilizes the covered node set N_{cov}, our implementation uses all ARG nodes with more than one incoming edge. For standard analyses, which we use in our experiments, the two sets are identical.

To construct the three types of partitioned certificates, we must determine a partition of the set of partition nodes first. Previously, we discussed how one can compute such a partition. To get a good partition, we did a pre-evaluation based on the four different partition element sizes 10, 100, 1000, and 10,000, and the three strategies greedy graph growing (BEST_FIRST), Fiduccia und Mattheyses optimization (FM K-WAY), and multi level partitioning considered in Henrik Bröcher's bachelor thesis [Brö16]. For pre-evaluation the complete CPC process of a single tasks, i.e., verification, proof generation, and proof validation, is executed in one run of CPAchecker. Then, for each analysis configuration and certificate type we considered all pre-evaluation results of all programs for which all combinations of partition element size and strategy finished validation. Based on the validation times, for each analysis configuration and partitioned certificate type we selected that combination that performed best w.r.t. the number of times the combination outperformed the others and the sum of all validation times. The results of this pre-evaluation can be found in the appendix (see Tables B.2, B.3, B.4). When we failed to construct

a certificate in 25 minutes with that combination, we generated the certificate with the same partition element size but the random strategy.

Generally considering the time for certificate generation, we observed that the construction of the full, partitioned certificate is most expensive and can take much longer than verification. At longest, its generation took about 1891 s which is maybe a bit too much for the producer. We think the main reason is the computation of the partitioning. On the one hand, the graphs for partitioning can become quite large. On the other hand, the partitioning implementation has not been tuned for performance. In contrast, the construction of the highly reduced certificate is with at most 162 s definitely feasible. Its largest generation time is even lower than that of the basic approach. The construction of reduced certificates and certificates for the combination of the two optimization approaches takes at most about 455 s. This worst time is significantly higher than the one for the highly reduced certificate. In case of the reduced certificate, the construction does not only rely on structural information, but must also compute the transfer successors and compare them with the ARG successors. In case of the (highly) reduced, partitioned certificate, the partitioning is one factor. We already observed that partitioning can be very costly. Additionally, like during the construction of the reduced certificate, during the construction of the reduced, partitioned certificate also transfer successors must be computed. However, generating a reduced certificate or a certificate for the combined approach can still be expected of the producer.

Validation Competitors To find out whether our configurable program certification approach is also a practical alternative to existing configurable PCC approaches, we want to compare it with existing techniques. Different underlying verification principles make a comparison challenging. It is hard to detect if a performance difference is already caused by the difference in verification. Thus, we decided to restrict our comparison to approaches that can deal with our verification technique. Additionally, for some of our abstract domains we are unaware of a straightforward translation to e.g. logic. A validation competitor must store information as produced by our verification. Based on these conditions, we could compete with Abstraction-Carrying Code [APH05b], approaches storing a subset of a fixpoint, a subset of abstract states, plus optionally some additional information [Ros03, BJP06, BJT07, AMA07, AAPH06], an approach checking the conformance of an ARG [HJMS03], and precision reuse [BLN+13].

We excluded Abstraction-Carrying Code [APH05b] because when transferred into our framework it would be identical to our basic approach. Among the approaches storing at least a subset of abstract states, we chose to compete with the idea presented in lightweight bytecode verification [Ros03]. This approach, we call it backwards strategy, is easy to integrate. We do not need to guide the exploration during validation and we only need to store a set of abstract states. In our implementation, we extended lightweight bytecode verification [Ros03] to arbitrary analyses types. Extending the idea to arbitrary analyses types, we store all ARG nodes which can be reached via backward edges in the ARG. For validation, we use a modified version of the lightweight verification algorithm, Algorithm 1, which does not construct an ARG. Our modification is restricted to the first line in which we also add the certificate states to the reached set and the waitlist. Technically, we also need to change the organization of the reached set from `PARTITIONED` to `LOCATIONMAPPED`. Thus, we ensure that the termination check properly detects coverage even when a state is covered by abstract states stored in the certificate.

We adapted the ARG validation suggested by Henzinger et al. [HJMS03] to deal with arbitrary analyses, not only predicate model checking. Based on the idea of temporal

safety proofs [HNJ+02], we check that the root node covers the initial state. For each ARG node, we compute the transfer successors and check if each such successor is covered by the ARG successors that are reachable from the ARG node along edges labeled with the same control-flow edge used to compute the transfer successor. Finally, we check that each ARG node is safe.

For the CEGAR model checking approaches we additionally compete with precision reuse [BLN+13]. Precision reuse stores the abstraction level. Instead of iteratively computing a suitable abstraction level, the verification is started with the stored abstraction level. Thus, precision reuse saves the iterative refinement of the abstraction.

Experimental Set Up The execution set up also remains the same as in the evaluation of the basic configurable program certification technique. All experiments are executed with Java HotSpot(TM) 64-Bit Server VM 1.8.0_101. Whenever our analyses rely on a SMT solver, we use SMTInterpol [CHN12] version 2.1-238-g1f06d6a-comp. Our experiments are performed with the benchmarking evaluation framework BenchExec [BLW15a] on machines with Intel Xeon E5-2650 v2 CPUs at 2.6 GHz and 135 GB of RAM. We use BenchExec to restrict each verification task to 15 GB of RAM and 15 minutes of CPU time. Furthermore, for the competitive PCC approach based on ARG validation we increased the stack size of the Java virtual machine from 1024 kB to 51200 kB to enable ARG reading. As before, we use the software analysis tool CPAchecker in the version available in the runtime_verification branch[6] revision 23042 and repeat each experiment 10 times. In the following, we study the average of these 10 repetitions.

After we know the evaluation setting, we continue to systematically study the results of our experiments. First, we investigate which configurable program certification technique performs best. Then, we compare the best technique with verification. Finally, we examine if our best technique is competitive with the validation competitors presented above.

4.4.2 RQ 1: Which Reduced Certificate Should One Use?

During the presentation of the certificate reduction technique, we proposed an improvement of the generated reduced certificate, the highly reduced certificate. Each analysis with a monotonic transfer relation can make use of this improvement. Except for the predicate analysis, all our analysis configurations fulfill the requirement for the application of the improvement. In the following, we want to use our analyses configurations to study if the suggested improvement is really an improvement.

Remember that the improvement is to store a subset of the abstract states in the reduced certificate. Obviously, the size of a highly reduced certificate is smaller than or equal to the size of the reduced certificate. In practice, the highly reduced certificate is typically smaller. For details, we refer the reader to Fig. B.3 in the appendix. This figure compares the number of abstract states[7] stored in both types of certificates. Next, we focus on less obvious results. We study the performance, validation time and memory consumption, of both certificate validations.

Figure 4.2 contrasts the validation of the reduced certificate with the validation of the highly reduced certificate. The left diagram sets the validation time of the reduced certificate against the validation time of the highly reduced certificate. As always, the validation times are given in seconds and describe the sum of the execution time of the

[6] `https://svn.sosy-lab.org/software/cpachecker/branches/runtime_verification/`

[7] The only elements that can cause the highly reduced certificate to be smaller than the reduced certificate.

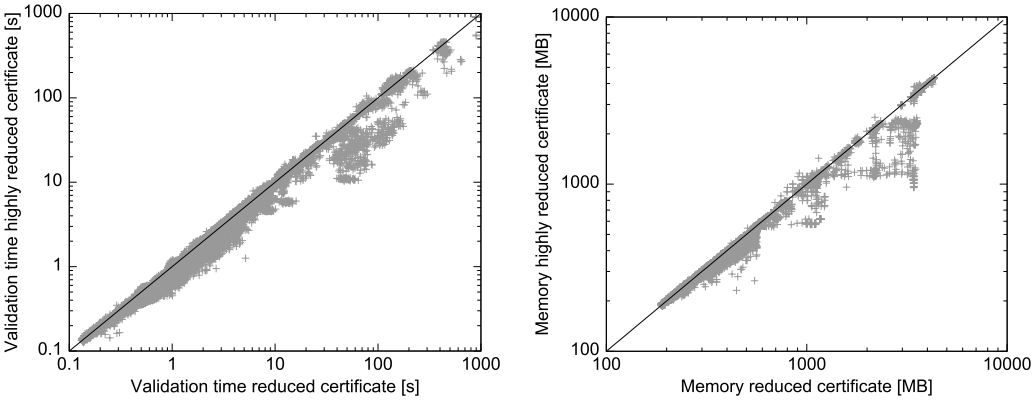

Figure 4.2: Comparison of validation times and memory consumption measured for the validation of reduced and highly reduced certificates

validation algorithm and the time for certificate reading. The right diagram depicts the comparison of the memory consumption, the sum of used heap and non-heap in MB.

Looking at the left diagram of Fig. 4.2, which compares the validation times, we notice that most of the data points are below the line. Hence, the validation of the highly reduced certificate is often faster than the validation of the reduced certificate. Indeed, for 64% of the validation tasks the validation of the highly reduced certificate took less time. In total, the validation of the highly reduced certificate was about 14% faster than the validation of the reduced certificate. At a first glance, reading those abstract states that are more abstract than the transfer successor and that are not contained in N_{cov} is more expensive than recomputing (sequences of) more precise abstract states. Reconsidering our analysis configurations, we come to the conclusion that in most of the analyses the reduced certificate contains more states because we failed to detect that the transfer successor is identical with the ARG successor. The reason is that in our implementation we use the termination check operator to detect identity, but remember that we must adapt the termination check to properly deal with the abstract states of the callstack CPA. Only in case of CEGAR model checking, abstract states are deleted that are more abstract than the transfer successor. For these analyses, we observed that validation of a highly reduced certificate is better in 84% of the validation tasks, but the complete validation for those tasks was only 3% faster. The observation that reading additional abstract states is more costly than recomputing more precise successors remains true. However, the difference between the two variants is smaller.

Similar to the left diagram, most of the data points in the right diagram are below the line, i.e., the validation of the highly reduced certificate often needs less memory than the validation of the corresponding reduced certificate. We think that better performance w.r.t. memory consumption is a direct implication of smaller certificates and faster validation.

Due to certificate construction, the highly reduced certificate is always smaller. Moreover, the validation of the highly reduced certificate is never significantly slower or consumes much more memory than the validation of the corresponding reduced certificate. In contrast, the validation of the highly reduced certificate often performs slightly better. Whenever the highly reduced certificate validation is applicable (the transfer relation is monotonic), the highly reduced certificate should be preferred to the reduced certificate.

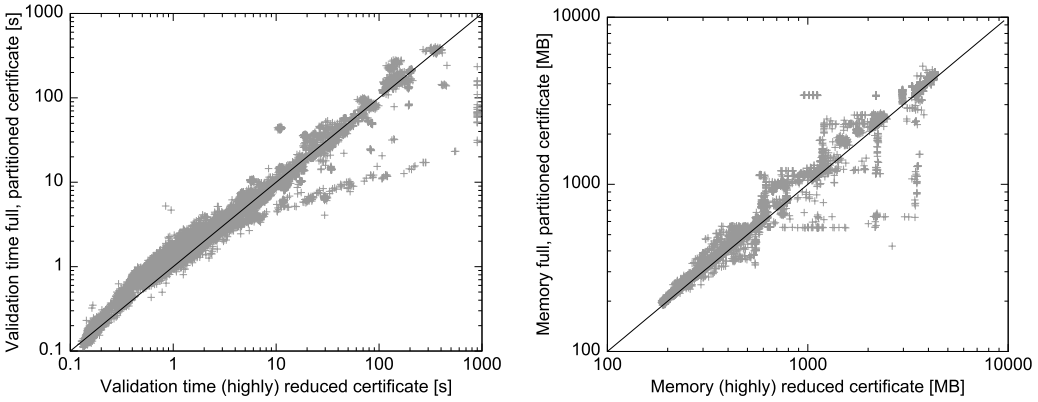

Figure 4.3: Comparison of validation times and memory consumption of the validation of full, partitioned certificates and best certificates generated by the reduction approach

4.4.3 RQ 2: Which Optimization Approach is Better?

In this chapter, we discussed two orthogonal optimizations for the basic certification approach, a reduction and a partitioning approach. Now, we would like to find out which of these two approaches performs better. Since the SMT solver used in the predicate analysis does not cope well with concurrent parsing and creation of new formulae, we use all our previous tasks except for those relying on the predicate analysis to answer the previous question. In more detail, for each task we compare the validation of the full, partitioned certificate[8] with that certificate variant of the reduction approach whose validation was faster.

Recapturing the construction of both types of certificates, we know that the reduction approaches store a subset of the ARG nodes plus a number and the partitioning approach stores all ARG nodes, some even multiple times, plus a number. From a theoretical point of view the reduction approach should generate the smaller certificates. Figure B.4 in the appendix, which compares the number of states stored in the two types of certificates, supports this theoretical observation. As in the previous section, we continue to compare the performance.

Figure 4.3 presents two diagrams. The left diagram contrasts the validation times, the time spans starting with certificate reading and ending after the validation algorithm finished, of the two optimization approaches. The right diagram relates the memory, heap plus non-heap memory in MB, consumed during certificate validation.

First, let us inspect the left diagram, which compares the validation times. We observe that some data points are below and others are above the line. Especially, the data points far below the line are remarkable. In general, none of the approaches is always better. However, for some tasks the partitioning approach significantly outperforms the reduction approach. Having a detailed look at our experimental results, we recognized that a significantly better performance of the partitioning approach can mainly be observed for tasks related to reaching definition model checking. For these tasks, we determined that the reduction approach still stores a lot more states than program locations exist, typically at least 50-times as many and even up to 500-times as many. In contrast, the partitioning approach stores 100 states per set of partition nodes. Due to the partition construction

[8] This is the certificate generated by the pure partitioning approach.

strategy, we used the Fiduccia und Mattheyses approach to improve an initial partition computed with greedy graph growing, it is unlikely that a partition element contains many different abstract states per program location. Often, the number of abstract states considered in a call to the coverage check should be much smaller in the partitioning approach. We think that this is the reason why the partitioning approach performs clearly better. Studying our experimental results, we also tried to find out when which validation approach is faster. We observed that for the different analysis types between 52% to 61% of the validation tasks were solved faster by the reduction approach. For example, the reduction approach solved 61% of the model checking as well as of the CEGAR model checking tasks faster. Since for the different analysis types both approaches perform almost equally well, we next investigate their behavior per analysis configuration and program class. Sometimes we could observe that one approach is more suitable for some program classes or analysis configurations. For example, the partitioning approach seems to be well-suited for sign model checking and `Problem15` tasks of the `ECA` category. In contrast, the reduction approach performs well with the uninitialized variables domain and the elevator tasks from the `ProductLine` category. Generally, the performance of the approaches depends on the analysis configuration and the program.

Next, we study the difference in memory consumption depicted in the right diagram. Like in the left diagram, there is no clear tendency for memory consumption. Sometimes, the data points are below the line, i.e., the partitioning approach requires less memory. Similar to the validation times, sometimes it requires much less memory. This is the case when validation with the partitioning approach is much faster. We think that the more expensive coverage check, which has to consider a lot more abstract states per call in the reduction approach, is responsible for the larger memory consumption of the reduction approach. However, much more often the data points are above the line. The reduction approach is faster. Studying the concrete numbers, we found out that in about 80% of the tasks the reduction approach requires less memory.

Summing up, the reduction approach generates smaller certificates and often requires less memory for validation. However, we observed that certificate validation benefits significantly from the partitioning approach whenever the reduction approach fails to store only a handful of abstract states per program location. This is the only case in which one should use the partitioning approach. In all other cases, the performance of the reduction approach is better or almost equally good. Thus, one should prefer the reduction approach.

4.4.4 RQ 3: Do We Profit from a Combination?

Next to the two optimization approaches examined in the previous sections, we also suggested to combine these two, orthogonal approaches. In this section, we want to find out whether the combined approach improves on both optimization approaches. To investigate this question, we use the same tasks as in the previous section and the following certificates. For the pure optimization approaches, we selected for each task that certificate among the reduced, the highly reduced, and the full, partitioned certificate whose validation was fastest. During construction of the certificate for the combined approach, we used the highly reduced node set for all tasks of an analysis whenever for the tasks of that analysis the validation of the highly reduced certificate was more often faster than the validation of the corresponding reduced certificate. Otherwise, we constructed a reduced, partitioned certificate. In the following, we use these certificates to evaluate if the validation of the combined certificate performs better, i.e., is faster and consumes less

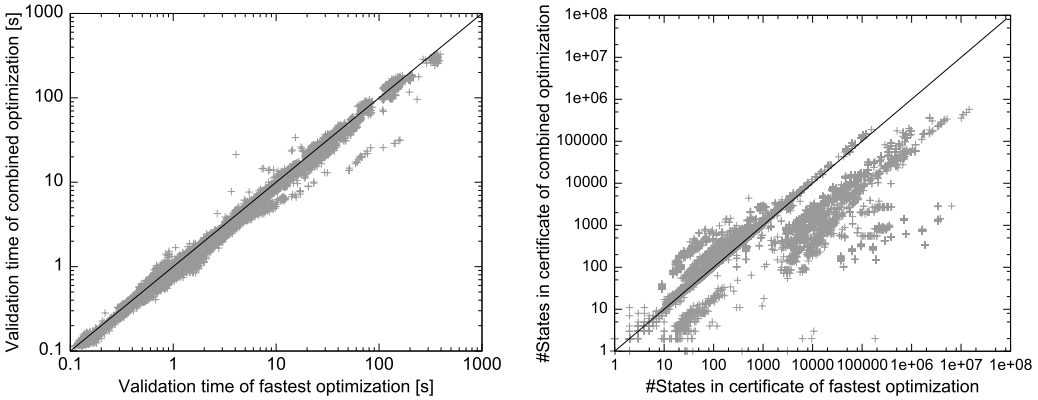

Figure 4.4: Comparing the validation time of the fastest, optimized configurable program certificate approach with the combination of the two optimization approaches (left), as well as the size of the validated certificates (right)

memory than the validation of the corresponding certificate that was selected to represent the pure optimization approaches. Additionally, we compare their sizes in terms of stored abstract states.

Figure 4.4 presents two diagrams. The left diagram relates the certificate validation time. The validation times are given in seconds. Certificate sizes are compared in the right diagram.

Starting with the left diagram of Fig. 4.4, we observe that many data points are closely below the line. It seems that the validation of the combined certificate slightly improves on the validation of the best optimization approach. Indeed, we counted that per analysis type the combined certificate validation is faster in 77% to 91% of the tasks. Furthermore, over the sum of all validation tasks the combined certificate validation was about 9% faster. While for the sum of all intermediate analysis tasks we nearly do not observe any improvement, for the complete set of CEGAR model checking tasks we calculated an improvement of about 27%. Next, we continue with the memory consumed during certificate validation.

The diagram for the memory consumption (see Fig. B.5 in the appendix) looks quite similar to the diagram for the validation times. It indicates that often the combination of the approaches consumes less memory, i.e., it uses fewer heap or non-heap memory during certificate validation. However, a more detailed inspection revealed that only in about 48% of the tasks the memory consumptions of the combined approach is smaller or equal. Typically, memory consumption is smaller when the combined approach competes with the pure partitioning approach.

In contrast to memory consumption, the results for the certificate size in terms of the number of stored abstract states already look more diverse. On the one hand, there is a large number of tasks in the right diagram of Fig. 4.4 for which the combined approach stores more abstract states. These are those tasks in which we check the combined approach against the reduction approach. Hence, the certificate of the combined approach stores the abstract states of the reduced certificate plus additional boundary nodes. In case the combined approach uses a partition of size one, this can be the case when the set of partition nodes is rather small, the certificate sizes are identical. On the other hand, we also observe a large set of data points that are below the line. The certificate of the

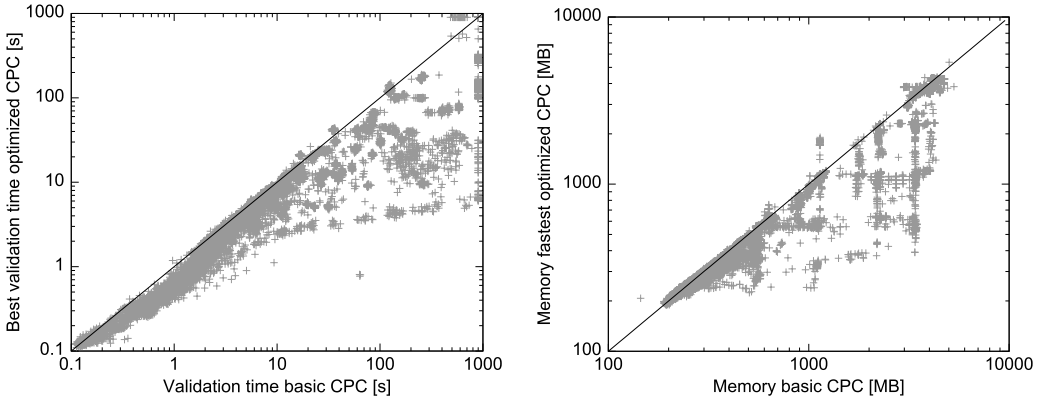

Figure 4.5: Comparison of the validation time and memory consumption of the optimized certificate for which validation is fastest with the validation time and memory consumption of the basic certificate validation

combined approach contains less abstract states, it is smaller. Typically, these are all tasks in which the combined certificate is compared with the full, partitioned certificate.

In a nutshell, the combined certificate is always bigger than the certificate of the reduction approach, but smaller than the certificate of the partitioning approach. Furthermore, memory consumption is mainly smaller when we compare the combined approach with the partitioning approach. In contrast, the combined optimization speeds up validation in many cases. Hence, one should definitely use the combined approach whenever one would prefer the partitioning approach over the reduction approach. For all remaining cases, the combined approach is worth a try when the main goal is a fast validation.

4.4.5 RQ 4: Do Our Optimizations Outperform the Basic Approach?

The goal of this chapter was to develop variants of the basic configurable program certification approach that perform better, i.e., validate faster, require less memory during validation, and generate smaller certificates. In this section, we want to check whether we have achieved this goal. Since we are mainly interested in a fast validation, for each task we compare the basic certificate validation with that optimized certificate validation that was fastest for that specific task. Note that for the predicate analysis we always use the validation of the reduced certificate, the only optimized validation that works with the predicate analysis. In the following, we use the three criteria validation time, memory consumption during validation, and certificate size for performance comparison. As always, validation time describes the time duration from the start of certificate reading until the end of the validation algorithm and the memory consumption is the sum of used heap and non-heap memory. To relate the certificate sizes, we use the number of abstract states stored in the certificates because the abstract states are the main ingredients of each certificate and in contrast to the file size this metric is not affected by compression.

Figure 4.5 contains two diagrams. The left diagram contrasts the validation time of the basic validation with the fastest, optimized validation. Both validation times are given in seconds. The second performance measure, memory consumption in MB, is compared in the right diagram.

First, let us look at the comparison of the validation times. We observe that may data points in the left diagram are below the line. Indeed, in 95% (14851 tasks) the optimization is faster. Except for the predicate analysis, which uses an expensively extended coverage check, the optimization approach never times out. Apparently, the predicate analysis is also responsible for the new timeouts, the data points at the right of the left diagram in Fig. 4.5. Moreover, we found out that the optimization becomes more efficient when the underlying analysis type is more precise. For example, the optimization seems to improve worst for the dataflow analysis tasks. In contrast, many of the data points that are significantly below the line belong to (CEGAR) model checking tasks. For (CEGAR) model checking, for more than 94% of the tasks the validation with the optimized approach is at least 10% faster than the validation performed by the basic approach. Additionally, in 9% of the model checking tasks we detected that the validation of the best optimization approach is at least 10-times faster than the validation of the basic approach. The reason for the better performance is that for the coarsest analyses we mostly profit from the reduction of the time solely spent with reading the certificate. In contrast, for more precise analysis our optimizations also improve on the coverage check, because it restricts the number of covering candidates. However, for the dataflow analyses we always have at most one candidate per program location. The more precise an analysis is typically the higher the number of explored abstract states and often also the lower the number of explored states that cover more than one abstract successor, i.e., the smaller the size of N_{cov}, are. Thus, more precise analyses seem to have more optimization potential.

Next, we continue with the memory consumption. Inspecting the top right diagram in Fig. 4.5, we see that often the data points are below the line. In numbers, in 72% or 11185 of the tasks memory consumption of the optimized approach is lower. Furthermore, the memory consumption is almost never much larger than in the basic approach; in 15480 cases (99%) it is less than 10% larger. Rarely, when the best optimization uses the full, partitioned certificate, we observe a significant increase. In these cases, a fast validation comes at the price of more memory usage. We found out that the reduction approaches would have used less memory. However, remembering that the reduction and partitioning approach performs almost equally well, we conclude that these are exceptions and the combined optimization performs well, too.

At last, we study the development of the certificate size. Figure 4.6 relates the number of abstract states stored in the basic certificate to the number of abstract states stored in that optimized certificate whose validation is fastest.

The diagram in Fig. 4.6 indicates that the optimized certificate is often smaller than the basic certificate. We found out that in 14727 cases (94%) the optimized certificate requires less abstract states. Typically, the optimized certificate contains at most 30% of the states of the basic certificate, i.e., the data points are below the dashed line. In 856 of the cases (about 5%), the optimized certificate stores more states, but it stores at most 1.5 times as many abstract states as the basic certificate. Likely, the 856 cases are those in which the full, partitioned certificate is validated fastest. However, since the reduction and partitioning approach perform equally well, more than 856 tasks must be validated by the partitioning approach. We conclude that most likely for the other tasks validated by the partitioning technique the combination of the reduction and partitioning approach is used. Thus, the certificate sizes of the combined approach are also much better.

For most of the analysis tasks, we achieved our goal. Faster validation and a reduction of the certificate size are easier to achieve than a reduction of the memory consumption. Sometimes, a faster validation comes at the cost of larger certificates or higher memory consumption.

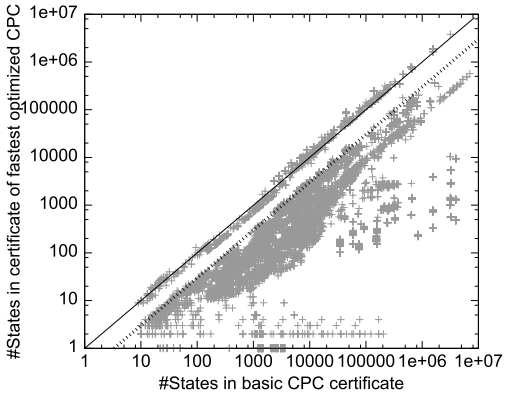

Figure 4.6: Comparing the number of abstract states stored in the basic certificate and in that optimized certificate whose validation is fastest

4.4.6 RQ 5: Do Our Optimizations Lead to a Success Story of Configurable Program Certification?

An improvement over the basic approach does not automatically guarantee that the optimizations perform better than verification. Remember that we observed that the validation of the basic certificate is often more costly than verification. However, one important aspect for the success of our configurable certification approach is that certificate validation significantly outperforms verification. For practical applicability, it is furthermore important that certificate validation still significantly outperforms verification when considering the total times including set up or program parsing times. Additionally, certificates should not become too large. Ideally, their size is in the order of magnitude of the program size or below.

First, let us compare the performance of the verification with the validation performed by the optimized approaches. For each task, we compare the verification with that optimization approach whose certificate was validated fastest for that particular task. We are most interested in execution time. Table 4.1 shows the best speed-up, i.e., the largest value for verification time divided by validation time, achieved for every analysis configuration (CPA). For comparison, the second line depicts the best speed-up achieved for the basic approach.

First, we observe that now for every CPA the certificate validation can be faster than verification. Furthermore, more CPAs achieved a significant speed-up (≥ 10). Instead of 22 tasks, with optimizations 104 tasks achieved a speed-up of at least 10.[9] By far, the largest increase of such tasks can be observed for CEGAR model checking. Generally, we recognized that the number of tasks that achieved such a significant speed after optimization is larger for more precise analyses. This fits to the observation that the largest speed-up of most of the non-dataflow analyses is larger than 10. Next, we take a deeper look at all tasks.

Figure 4.7 relates the execution times of verification and validation. The left diagram compares the verification time, the time required for the CPA algorithm, with the validation time, the duration from the beginning of certificate reading until the end of the

[9]Note that for these number we only considered tasks whose verification configuration was at least flow-sensitive. These are the tasks that our optimization approaches aim to improve.

Table 4.1: Per analysis configuration best speed-up, $\frac{\text{verification time}}{\text{validation time}}$, achieved by the optimization approaches and the basic approach

Dataflow Analysis						
$\mathbb{I}$	$\mathbb{R}$	$\mathbb{S}$	$\mathbb{U}$	$\mathbb{V}$	$\mathbb{SI}$	$\mathbb{VR}$
2419.53	2.73	2.20	1.31	2.72	2653.68	2.27
2071.32	1.56	1.38	0.98	1.41	2392.25	1.46
Intermediate						
					$\mathbb{SI}$	$\mathbb{VR}$
					26.54	416.59
					11.98	5.32
Model Checking						
$\mathbb{I}$	$\mathbb{R}$	$\mathbb{S}$	$\mathbb{U}$	$\mathbb{V}$	$\mathbb{SI}$	$\mathbb{VR}$
18.09	38.46	68.82	1.14	211.28	36.76	213.17
1.22	1.27	1.18	0.86	2.62	1.41	2.57
CEGAR Model Checking						
		$\mathbb{O}$	$\mathbb{P}$	$\mathbb{V}$		
		25.62	2.68	978.93		
		16.61	2.62	510.59		

respective validation algorithm. In contrast, the right diagram considers the total times, the time it takes to run the analysis tool either in verification or validation mode. Thus, total times also include set up times like program parsing or analysis set up. In both diagrams, times are given in seconds.

Looking at the left diagram in Fig. 4.7, many data points are below the line. For these data points, validation outperforms verification. It seems that validation outperforms verification much more often than it did in the basic approach. From a detailed inspection of our results, we know that for 78% of the tasks validation is faster than verification. When looking at the sum of all times of one analysis, except for the predicate analysis and the two analyses considering the uninitialized variables domain, the sums of the validation times are smaller than the sums of verification times.

Still, validation is not always faster and especially not significantly faster. Thus, we discuss how one could predict the validation speed-up based on the observations during verification. Unfortunately, for the intermediate tasks we failed to detect a predictor.

Like in the basic approach, for dataflow analyses we computed a correlation of 0.999 with the relation of number of transfer successors computed during verification to the final size of the set of ARG nodes. Multiplying the fraction with a factor of 0.45 is good underapproximation of the speed-up. Thus, the most important factors for dataflow analyses are the number of merges and how many recomputations they cause. Additionally, the verification should take at least 0.1s to achieve a speed-up of at least one.

For model checking tasks, we observed a correlation of 0.961 with the relation of analysis (verification) time to the time for transfer successor computation. Multiplying the analysis time with 0.33 and dividing it by the time for the transfer successor computation is a reliable underapproximation of the speed-up for our evaluation results. Moreover, we observed that for a speed-up of at least 1 the verification must take more than 0.25 s. Note that when a verification task is faster than 0.25 s, it is sufficient for the prediction to multiply the analysis time with the factor 0.5.

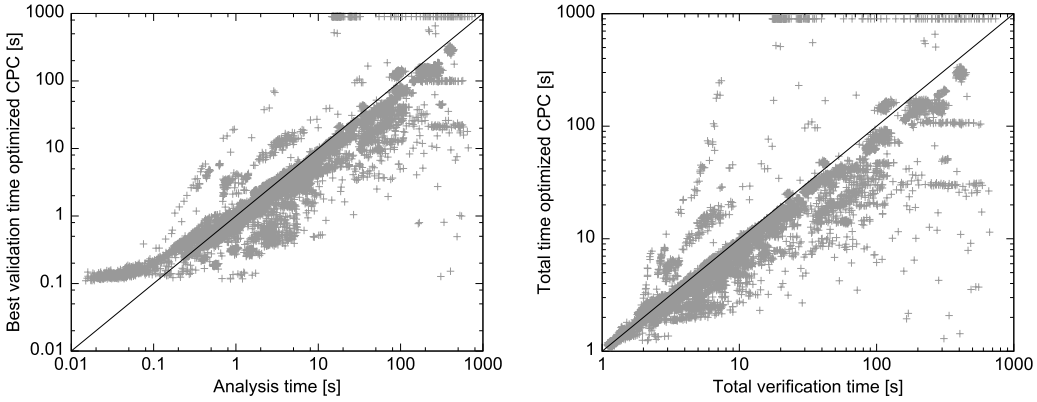

Figure 4.7: Per task comparing the execution times of the fastest certificate validation with verification: concept/algorithm (left) and total tool times (right)

Finally, we discuss the predictor for CEGAR model checking tasks. We were not able to find a predictor for the predicate analysis. The predictor applicable to the other two analyses does not fit. Once more, we think the reason is that we must adapt the termination check not only for the callstack CPA, but also for the predicate component. For the octagon and the value analysis, we noticed that a speed-up above 1 can only be achieved when the verification took at least 0.13 s. Furthermore, we recognized a correlation of 0.939 to the following expression $\frac{\#\text{transfer successors}}{|N|} + \frac{\text{analysis time}}{\text{transfer successor time}}$. To get a reliable underapproximation of the speed-up, one should multiply the previous expression with 0.05. Note that in contrast to the basic approach, the speed-up does not only depend on the costs caused by refinement, but also on how much one can benefit from an improvement of the coverage check. The last aspect is the main reason why validation for model checking improved well.

In reality, users do not only observe validation and verification times, respectively. Normally, they notice the total time for running the tool including program parsing or setting up of the analysis. Now, we want to examine whether users will recognize a difference between our optimized validation and verification. Looking at the right diagram in Fig. 4.7, we observe that many data points are below the line. Even when we consider the complete running time of the software analysis tool CPAchecker, certificate validation regularly outperforms verification. Comparing the right diagram in Fig. 4.7 with the left diagram, we see that the total times are larger than 1 s, while verification and validation times can be significantly below 1 s. Thus, it is no surprise that in the lower left corner of the right diagram validation is no longer significantly faster than verification. For the tasks in the lower left corner of the left diagram, the set up time becomes the larger part and nearly destroys the advantage that validation is faster than verification. Furthermore, as the two diagrams indicate, when total verification took at least 1 s, the total validation time remains faster than the total verification time for all cases in which the validation time already outperformed the verification time.

Next, we look at the memory consumption of the optimized certificate validation and verification. Figure 4.8 depicts the comparison of the memory consumption of the fastest, optimized validation with verification. Memory consumption represents the sum of heap and non-heap memory used during validation and verification, respectively, and is given in MB.

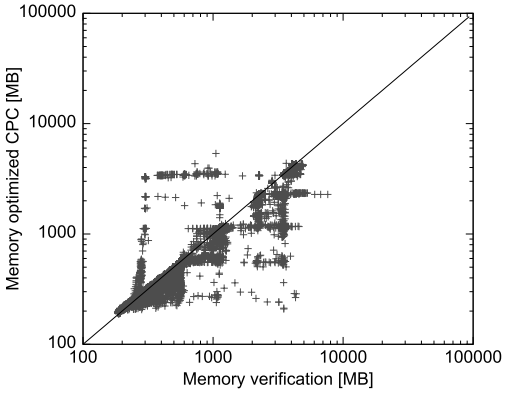

Figure 4.8: Comparison of the memory consumption, used heap plus non-heap memory, of verification and validation of that optimized certificate whose validation is fastest

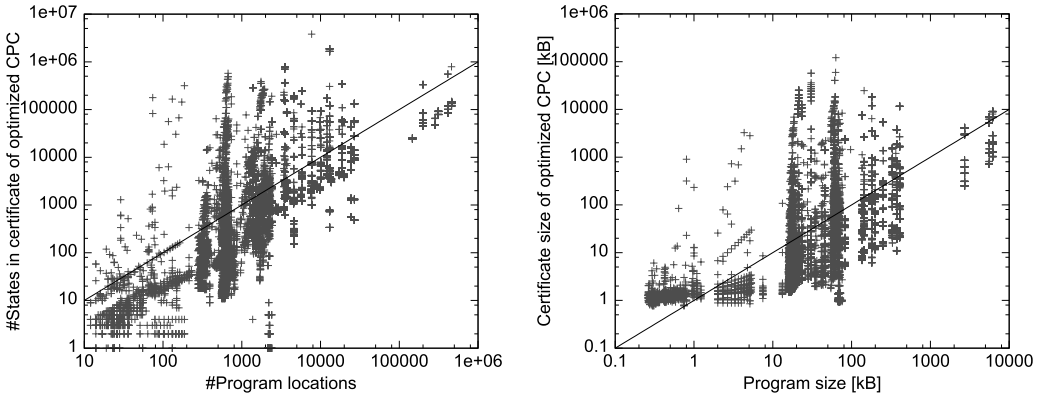

Figure 4.9: Comparing the number of states stored in the optimized certificate whose validation is fasted with the number of program locations (left) and relating the file size of that certificate (zip compressed) to the file size of the uncompressed program (right)

Looking at the diagram in Fig. 4.8 we see that most of the data points are below the line. Often, the memory consumption of our optimized validation is lower than the memory consumption of verification. Inspecting our results in detail, we observe that compared to the basic approach especially for the intermediate and model checking tasks the memory consumption of the validation is much more often smaller than the memory consumption of the verification. Furthermore, we notice that the complete memory consumption for all tasks is now lower than the corresponding memory consumption for verification. However, mostly the memory consumption of the validation is smaller when validation is faster than verification, but it is not always smaller when validation is faster.

Now, we come to the certificate size. As before, we consider two metrics: number of elements and file size. Figure 4.9 shows the the comparison of the certificate size and the program size based on these two metrics. The left diagram relates the number of abstract states stored in that optimized certificate whose validation was fastest to the respective number of program locations. Furthermore, the right diagram compares the file size of that certificate with the file size of the program.

Despite our optimization, we observe that many data points are still significantly above the line, i.e., the certificate is much larger than the program. However, compared to the basic approach we notice an improvement. The decrease in the number of stored abstract states, which we often observed when comparing the basic and optimized certificates, is the reason why the number of abstract states stored in the optimized certificate is much more often smaller than the number of program locations.

In sum, certificate sizes are still an issue for many tasks. Furthermore, we seldomly observed a significant performance improvement. However, significant performance improvements are achieved for many different CPAs – a proof that our framework is general. Moreover, often our optimized validation is at least a little better than verification. Additionally, we described how to use the verification to predict the validation performance.

4.4.7 RQ 6: Does Parallelization Help To Further Improve Validation?

Despite optimization, the performance of the best optimization approach is not always better, e.g., faster than verification. We already studied parallelization of certificate validation for the basic approach and found out that it improved validation time well. Now, we would like to know if we can experience a similar effect for our optimized approaches. Looking at the two validation algorithms for the optimized approaches, we observe that the loops are again good candidates for parallelization. Similar to the basic approach, we parallelize the outermost loop, the while loop, of Algorithm 4, the validation algorithm for reduced certificates. Since we require that also multiple partition elements can be read in parallel when we successfully want to parallelize the outermost loop in the validation algorithm for partitioned certificates, we decided to parallelize the inspection of a single partition element. Thus, our implementation parallelizes the while loop of Algorithm 5.

Like in the evaluation of the basic approach, we considered only tasks related to the reaching definition dataflow analysis and reaching definition model checking. Again, we run our experiments on an Intel Core i5-2400 CPU at 3.10 GHz utilizing 2 CPU cores for sequential validation and 4 CPU cores for parallel validation, while three threads are exclusively used for the parallelization of the validation algorithm. Note that we had to fix a bug in the parallel validation of partitioned certificates. Thus, for this evaluation we used revision 23749 of the runtime_verification branch of CPAchecker. In the following, we compare the results of a single experimental execution for each of the four optimization approaches excluding tasks for which the sequential validation already timed out.

Figure 4.10 compares the sequential validation time and the parallel validation time for all optimization approaches. The top diagrams compare the validation times for the reduced (left) and the highly reduced certificate (right). At the bottom, the left diagram considers the full, partitioned certificates and the right diagram considers the certificates generated for the combination of the two optimization approaches. In all four diagrams, the validation times are given in seconds.

Looking at Fig. 4.10, we observe that in all four diagrams the data points are never above the upper line and mostly below that line. Parallelization likely improves the validation time. Additionally, we see that the two partitioning approaches, considered in the bottom diagrams, perform worse than the reduction approach, especially when validation takes longer (validation times become larger). We think the reason is that the number of partition elements in the corresponding certificates is high, often several thousands, and the partition elements are small. Each set of partition nodes was restricted

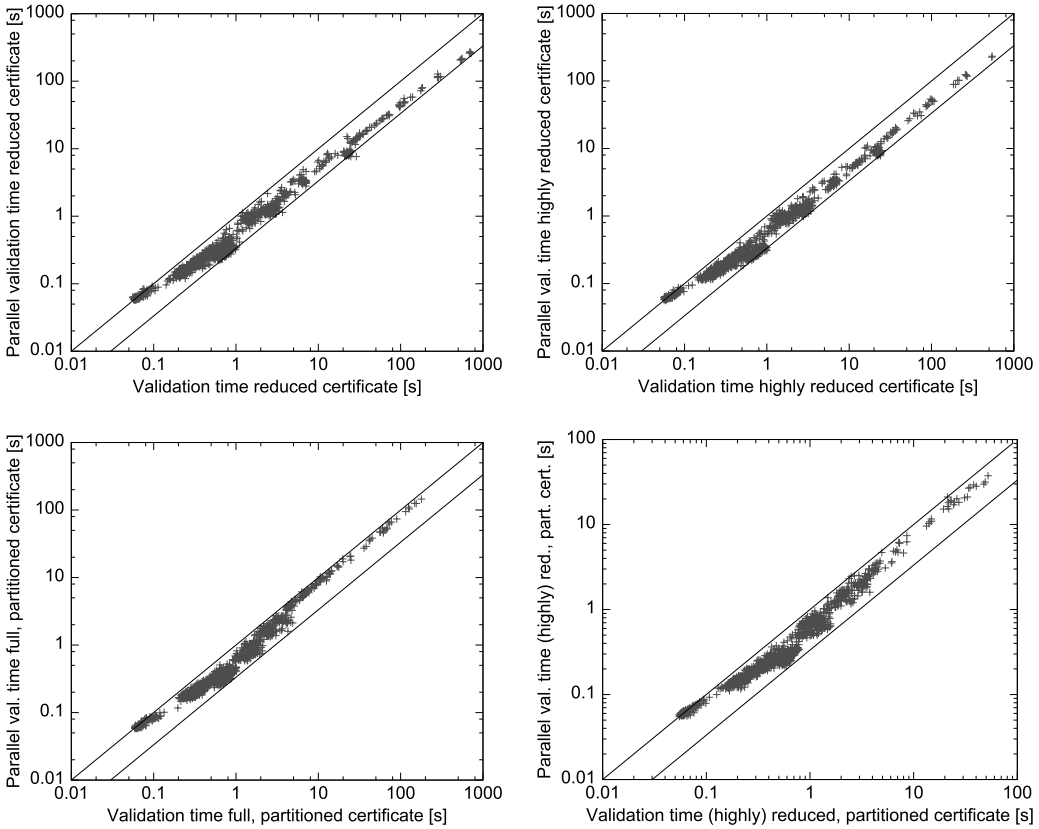

Figure 4.10: Comparing the validation times of sequential validation and parallel validation with three threads for all four optimizations. The top left diagram shows the results for the reduced certificates, the top right for the highly reduced certificates, the bottom left for the full, partitioned certificate, and the bottom right for the combination of the two optimization approaches.

to contain no more than 100 states. Thus, when validation is parallelized, a partition element might be checked faster, but the validation must possibly wait until the next partition element is read. Looking at the details, we observed that for all four optimization approaches parallelization again increased the number of tasks for which validation is faster than verification. For the dataflow analysis between 245 and 445 additional tasks and for model checking between 24 and 92 additional tasks are faster.

Parallelization improves validation, but moderate parallelization is not enough for many tasks to significantly outperform verification. Furthermore, depending on the degree of parallelization different partitions should be used to build partitioned certificates.

4.4.8 RQ 7: Is Our Certification Approach Competitive?

So far, we compared our configurable certification approaches among each other. In this section, we want to find out if our configurable certification approach can compete with existing certification approaches. To this end, we check our CPC approach against the three competitors, precision reuse, backwards strategy, and ARG validation, introduced

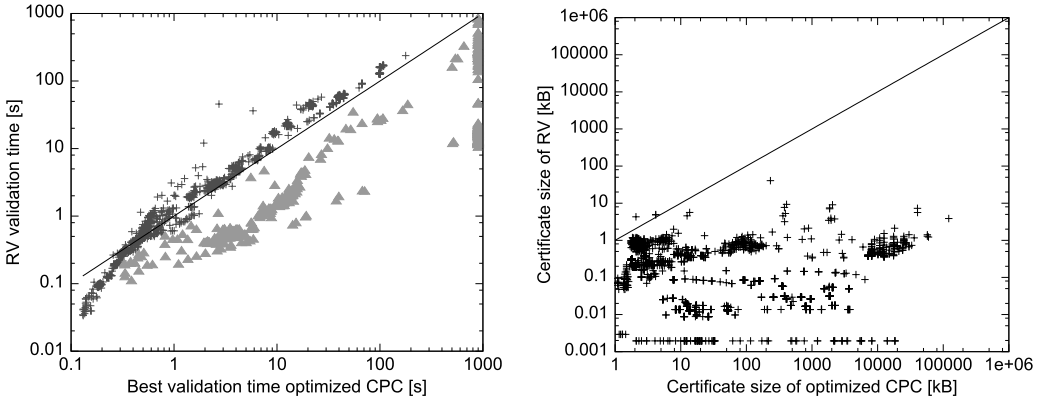

Figure 4.11: Comparing the validation time and the certificate size of the fastest, optimized CPC validation with regression verification.

in the experimental set up part. For each verification task (combination of program and analysis configuration), we compare the performance of all suitable competitors with the performance of the optimized CPC approach with the fastest validation time for that task. As before, we use the three criteria validation time, memory consumption during validation, and certificate size for performance comparison. Still, validation time is the sum of the execution time of the validation algorithm and the time for certificate reading and the memory consumption is the sum of used heap and non-heap memory.

The first competitor we look at is regression verification (RV) [BLN+13]. Remember that regression verification is targeted on analyses that adjust their precision during verification. Most of our analysis configurations do not adjust precisions. In these cases, a comparison with regression verification is identical to a comparison with the analysis itself, thus useless. That is why we restrict our comparison to the three CEGAR model checking analysis octagon $\mathbb{O}$, predicate $\mathbb{P}$, and value analysis $\mathbb{V}$ – the only analyses that adjust precisions. Figure 4.11 shows the comparison of the validation time and the certificate size. The comparison of the memory consumption can be found in the appendix (see Fig. B.6). The diagram on the left of Fig. 4.11 displays the comparison of the validation time in seconds. Triangles represent the results for predicate analysis tasks and the crosses refer to value or octagon analysis tasks. On the right of Fig. 4.11, the comparison of the file sizes in kB is depicted.

Let us first look at the validation times (left diagram). We observe that almost all triangles are below the line. This means that for predicate analysis tasks regression verification is faster than our suggested configurable program certification. The reason is that regression verification uses the standard termination check operator while our approach must use a well-behaving extension. In comparison to the standard operator, the extension is more expensive, e.g., it calls the SMT solver much more often. Looking at the crosses, which represent the results for the other two analyses octagon $\mathbb{O}$ and value analysis $\mathbb{V}$ that do not require a costly well-behaving extension, we observe that regression verification is only faster when validation is fast (simple). If validation takes more than 0.3 s, our CPC approach will already sometimes become better, and with more than 2 s, it will almost always be better. Note that a similar observation can be made for the comparison of the memory consumption (see Fig. B.6 in the appendix). Regression verification beats our approach for predicate analysis and is equally good or worse for the

other two analyses.

Next, we study the last criterion certificate size. Since the structure and the information stored in the certificates is incomparable, regression verification stores a precision and CPC stores abstract states, we decided to compare the file sizes of the certificates. Note that such a comparison is disadvantageous for the regression verification because in contrast to the implemented regression verification we compress our certificates. Still, we observe that on the right diagram of Fig. 4.11, almost all crosses are below the line. Certificates for regression verification are smaller.

Summing up, although our implementation compresses all CPC certificates, the certificates obtained by regression verification are much smaller. Furthermore, for predicate analysis regression verification performs better in terms of time and memory. However, in all cases in which the validation configuration can be derived automatically, i.e., no well-behaving extensions are required, configurable program certification is often faster or uses less memory. Additionally, any CPC validation approach requires only a subset of the analysis' operators, while regression verification uses all of them. Hence, the trusted computing base is larger for regression verification.

Next, we compare our configurable certification with a certification strategy [Ros03], the backwards strategy, typically applied in the certification of dataflow analyses. Originally, such certification approaches stored all abstract states associated with program locations reachable via backward edges. We migrated this idea to arbitrary analyses and store all nodes in the ARG that are reachable via backwards edges. Note that we exclude the predicate analysis from the subsequent comparison because the backwards strategy neither stores precisions nor all states for which a predicate abstraction is computed. Hence, a recomputed predicate abstraction is likely too coarse and lets the backwards strategy fail. Figure 4.12 shows the diagrams for the comparison of validation time and memory consumption, given in seconds and MB, respectively. On the left of Fig. 4.12 we see the results for all dataflow analysis tasks. The results for the remaining results are shown on the right. The comparison of the certificate size[10] can be found in the appendix (see Fig. B.7).

First, let us consider the diagrams on the left of Fig. 4.12, which depict the comparison of the dataflow analysis tasks. Inspecting the upper diagram, which compares the validation times, we notice that the validation times are quite similar. Sometimes, our approach is faster and more often the backwards strategy is faster, but mostly less than 3-times faster. Similarly, the memory consumption of the backwards strategy is most of the time lower. We think that the reasons are as follows. In the implementation, we restrict the termination check to states with equal locations. Like in the dataflow analysis, during validation there is at most one abstract state per program location. In each call, the termination check considers at most one abstract state, which is simple and fast. Hence, it is difficult and sometimes even impossible for our certification approaches to improve the termination check. Furthermore, no matter what optimization approach we use, certificate construction together with ARG construction ensure that the certificates will always contain all abstract states stored by the backwards strategy. Additionally, our certificates also store all states on join locations. Thus, our certificates are typically larger. Since reading is slower than computation, it is faster to compute the merge on join locations than reading the merged state.

Now, we come to the remaining, non-dataflow analyses. These results are depicted on the right of Fig. 4.12. We observe that many of the data points in the upper right figure

[10]Since both approaches mainly store abstract states, we decided to compare the number of stored states, counting states multiple times if stored multiple times.

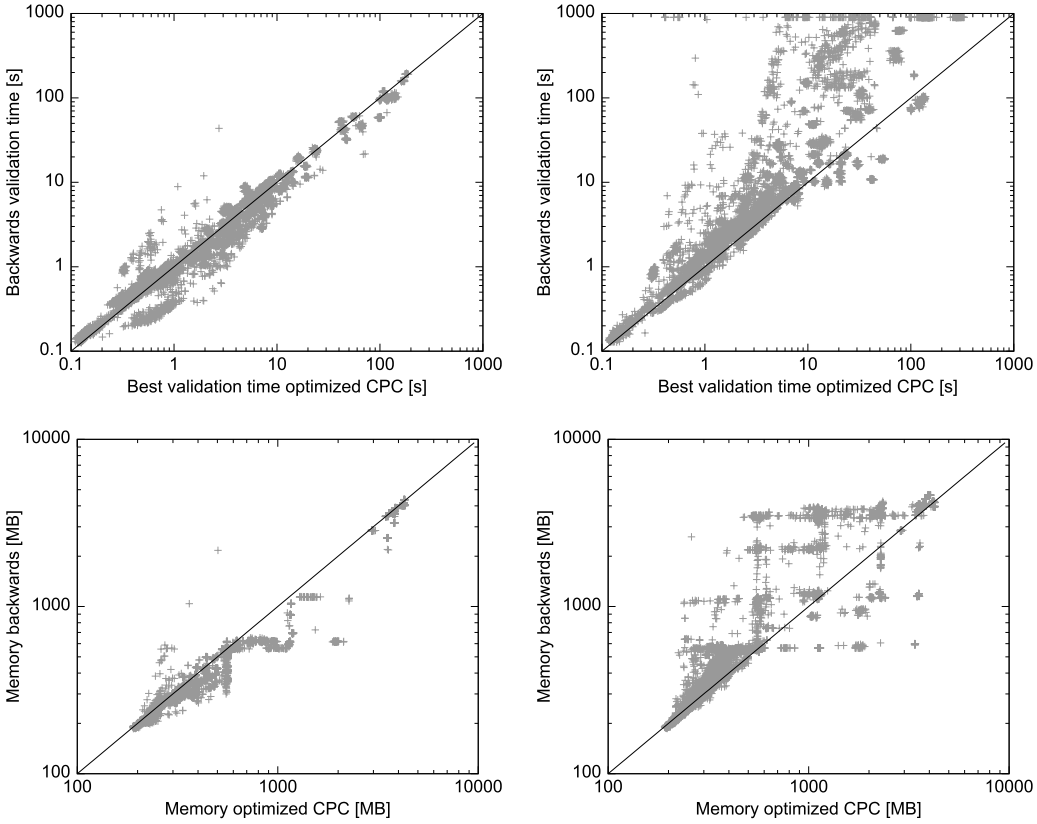

Figure 4.12: Comparing the time and memory consumption of the fastest, optimized CPC validation with the validation of backwards certificates. Comparison is split into the validation of certificates resulting from dataflow analyses (left) and certificates resulting from the remaining analyses (right).

are above the line, i.e., our configurable program certification often performs better than the backwards strategy. In many cases, it performs even much better. For the memory consumption, we can make a similar observation. To our mind, the main reason is that in those more precise analyses multiple abstract states per program location exist. Tens and even several hundreds of states per location are not uncommon. Our approaches restrict the number of states considered by the coverage check. Many of them are not stored or do not occur in the same partition. In contrast, the backwards strategy must always consider all states of a location during the termination check, no matter if they are stored or recomputed.

As already explained, our optimized certificates always store at least all states also stored in the backwards strategy. Thus, it should be obvious that the certificate size of the backwards strategy is smaller or equal than the size of our optimized certificate. In reality, they are always smaller. For more details we refer to Fig. B.7 in the appendix.

All in all, storing only those ARG nodes that are successors of a backward edge in the ARG, as the backwards strategy does, results in smaller certificates. Moreover, the backwards strategy seems to be well-suited for the certification of dataflow analysis results, its original purpose. Often, it is a little faster and uses less memory than our best,

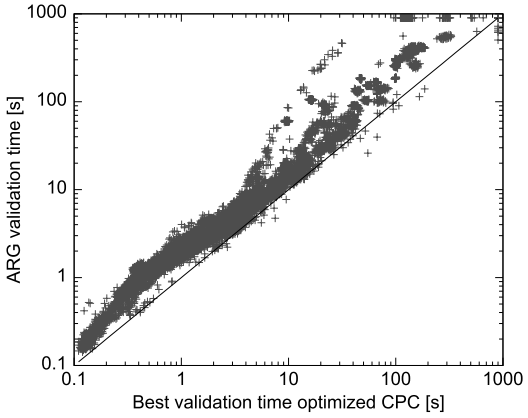

Figure 4.13: Comparison of the fastest, optimized CPC validation time with the validation time for the abstract reachability graph.

optimized CPC approach. In contrast, for more precise analyses than dataflow analyses the backwards strategy is typically outperformed by our optimized CPC approach. Finally, we like to mention that the backwards strategy has the same drawback w.r.t. the trusted computing base as regression verification, the competitor considered before the backwards strategy.

We finish with a comparison of our approach and ARG validation. Following the idea of Henzinger et al. [HNJ+02, HJMS03], ARG validation is similar to checking that the ARG is well-formed. ARG validation is the only competitor that can also construct and validate certificates for all analysis configurations. Figure 4.13 shows the comparison of the validation times in seconds. The omparison of memory consumption and certificate size can be found in the appendix (see Fig. B.8). Due to the structural difference of the ARG and our certificates, we again compare the file sizes of the certificates. However, this time both the ARG and our certificates are compressed.

Looking at Fig. 4.13, we observe that the crosses are mostly above the line, i.e., our optimized certificate validation is typically faster. One reason is that the computation of transfer successors is more costly since it must also bypass those components responsible for ARG construction[11]. Another reason is that ARG reading takes longer than reading one of our certificates, which store only a part of the ARG. Thus, it should be of no surprise that the results of the memory comparison and the certificate size comparison with the ARG size are similar to the time comparison. First, we already mentioned in the description of the experimental set up that for ARG validation, especially reading ARGs, we had to increase the stack size of the JVM. Second, for ARG validation we keep the complete ARG in main memory and not only its nodes. Third, to store the ARG, we do not only need to store its nodes, but also its edges. Hence, for ARG validation more information than in the basic CPC approach must be saved and we already know that the certificate of the basic CPC approach is often bigger than the certificate of the best, optimized CPC approach. In summary, our optimized CPC approach outperforms ARG validation in all categories.

[11] In the software analysis tool CPAchecker, ARG construction is realized with an ARG CPA.

4.4.9 Summary

In practice, the highly reduced certificate is better than the reduced certificate w.r.t. validation performance and certificate size. Similarly, the combination of both approaches is typically better than the full, partitioned certificate. While the highly reduced certificates are smallest, the validation in the combined approach is often faster. Especially, when the highly reduced node set still contains much more abstract states than program locations exist, the combined approach, but also pure partitioning, clearly outperforms highly reduced certificate validation. Although parallelization improves validation, it is not a break-through to achieve significant improvements. Furthermore, parallelization of the reduction approaches is more effective. Nevertheless, our optimization approaches improved the basic approach. This improvement is also observable when comparing the optimization approaches with verification. For example, significant improvements occur more often, but not frequently. At least, we are able to predict the performance of the validation based on the verification. Unfortunately, despite our optimizations certificates are still large. Finally, certificates of two competitors are smaller and sometimes the validation of the competitors is better for specific instances. Generally, CPC competes well with the competitors and seems to be more appropriate when considering the complete set of tasks.

4.5 Discussion

Our optimization approaches are generic, too, but they are tailored to ARGs with more than one node. Applying them to flow-insensitive analyses makes not much sense. Anyway, the trusted computing base remains similar to the basic CPC. Only the validation algorithms, especially the validation algorithm for partitioned certificates, are more complex. Moreover, we also proved soundness and relative completeness. Relative completeness w.r.t. the validation of highly reduced and highly reduced, partitioned certificates is limited to validation (verification) configuration with monotonic transfer relations. The analyses that we use in practice typically provide monotonic transfer relations. Monotonic transfer relations are more or less a theoretical restriction. Finally, automation still requires a well-behaving termination check. From a theoretical point of view, the optimization approaches are a little bit more restrictive.

Next, we discuss our practical experience. We evaluated our optimization approaches on a large set of programs and analysis configurations. Thus, we demonstrated their practical feasibility. Nevertheless, the case of the predicate analysis already revealed that not all optimization variants are always feasible.

Continuing with the certificate generation, we first of all notice that the construction of one of the three partitioned certificates is challenging. The hard part is to select a good configuration for the partition computation. For our experiments, we relied on pre-evaluations to determine such good configurations. Furthermore, the generation of a full, partitioned certificate can become extremely expensive. The computation of a reduced or (highly) reduced, partitioned certificate is acceptable. The generation of highly reduced certificates is even better than the generation of certificates in our basic approach. Nevertheless, we think it is too much to ask from a producer to try out every CPC variant and pick the one that fits the objectives best.

Unfortunately, none of our optimization approach is always best. The highly reduced certificates are the smallest and full, partitioned certificates are the largest. Typically, the validation of the highly reduced certificate is better than that of the reduced certificate.

In contrast, the validation of the full, partitioned certificate is faster for some cases and the validation of a combined certificate is often faster than the validation of the highly reduced certificate. Normally, the combined approach performs better than the validation of the full, partitioned certificate and its certificates are also smaller. The producer does not need to consider the full, partitioned certificate. In practice, the producer can exclude the reduced certificate and the full, partitioned certificate from his considerations. Since we also observed that our optimization approaches mainly exceed the basic approach, the producer can focus on the highly reduced certificate and the combined certificate, most likely the highly reduced, partitioned certificate.

Our optimizations also achieved an improvement of the validation w.r.t. verification. More often, the validation outperforms verification. Nevertheless, significant improvements in order of magnitudes are still uncommon. Similarly, certificates often remain much larger than the program, but our improvements achieved that at least for some tasks the certificates are smaller than the program. We also studied parallelization of all configurable program certification variants to further improve validation. Once more, parallelization is restricted to the reaching definition domain in practice. Furthermore, parallelization of the partitioning approach, which already reads and checks in parallel, is less effective.

Reconsidering the competition with other certification approaches, configurable program certification mostly performed at least equally well. Only the regression verification technique precision reuse is better on the predicate analysis tasks, for which our validation configurations had to adapt the termination check. Moreover, the backwards strategy is slightly better for dataflow analyses, its original application scenario. Additionally, these two competitors also generate smaller certificates.

In summary, for most of the tasks our optimizations improve the basic approach. Despite the optimizations, certificates often remain much larger than the program. Furthermore, for a large set of the tasks the performance improvement over the verification is small, less than an order of magnitude, although we improved the configurable program certification approach. Not all programs (and property combinations) are suitable for certification, in particular configurable program certification. This observation is supported by the fact that configurable program certification often competes well with other certification techniques.

If we had to vote for one of the configurable program certification approaches, we would choose the one considering the highly reduced certificate. Despite its theoretical restrictions, which are less relevant in practice, it seems to provide the best compromise between validation performance, certificate size, and certificate generation. However, if one is mainly interested in validation performance, especially validation time, the combined technique will be the better option.

4.6 Related Work

Our configurable program certification approaches enrich a program with partial verification results to ease, e.g., speed up, a subsequent inspection of program safety. As already mentioned in the introduction, the configurable program certification approaches are the only instances of the abstract protocol from Fig. 1.1 that solely attach additional information to the program and are presented in this thesis. The instances presented in the subsequent chapters either integrate the additional information implicitly into the program or combine the next instance with configurable program certification. Thus, we

decided to already discuss work related to configurable program certification at the end of this chapter.

In the following, we look at approaches that also attach additional information to assure safety or security of software programs. We tried to arrange their presentation according to the producer's verification technique. After the presentation of the related software approaches, we give a short overview on related approaches for configurable hardware. For both kinds, we will see that existing approaches are typically restricted to a particular property or type of analysis.

Theorem Proving *Proof-Carrying Code* (PCC) [NL96, Nec97, NL98c] ensures safety of a program w.r.t. a consumer provided safety policy, namely a verification condition generator, a set of axioms, and a precondition. The verification condition generator is a Floyd style generator that computes a pre- and postcondition for each statement. The precondition ensures that the execution of the statement is safe and the postcondition of a statement is or implies the precondition of the next statement. The producer runs the verification condition generator to get the safety predicate and then proves that the precondition implies the safety predicate. The proof is attached to the program. The consumer reruns the verification condition generator and checks if the attached proof is a valid proof that the precondition implies his generated safety predicate. To deal with loops and functions, the program must be manually annotated, e.g., with loop invariants and pre- and postconditions for functions. Annotations are included in the safety predicate, but need not be trusted. Although it is a general framework, Necula et al. used the framework to mainly assure memory safety and type safety of assembly programs.

Foundational Proof-Carrying Code [AF00, App01] is one approach to remove the large verification condition generator from the trusted computing base. The idea is to encode the machine semantics as partial step semantics in logic and exclude transitions for illegal or unsafe statements. To prove safety, one needs to show that the program never gets stuck, always a successor state exists. If one wants to use inference systems in the proof, one must prove their soundness.

A different approach, which removes the verification condition generator from the trusted computing base, is presentded by Wildmoser et al. [WNKN04, WN05] and Wildmoser [Wil06]. The authors present a generic variant of Necula's PCC approach in which the verification condition generator is parameterized w.r.t. the programming language, the safety policy and the logic. They describe their requirements on the parameters and proved in Isabelle that the verification condition generator is sound when the parameters adhere to the requirements. The generic framework is instantiated to assure type safety and no arithmetic overflow of assembly programs or bytecode.

Another problem of the PCC approach are large proofs. The first solution [NL98a] suggests to use a derivation, called LF_i, of the Edinburgh logic framework. In the logic framework LF_i some proof objects can be implicitly represented by placeholders and are reconstructed during proof checking. LF proofs can be transformed into LF_i proofs by erasure of certain redundant information.

Chaudhuri [Cha12] discuss how to reduce the size of linear logic proofs. The idea is to replace some of the derivation subtrees in the derivation tree by a search bound, called contraction bound. A search bound is used to restrict the search during the recomputation of the deleted subtree.

Another technique to reduce proof sizes is *oracle-based checking* [NR01]. Oracle-based proof checking can be applied when non-deterministic theorem provers are used for the proof search. An oracle string encodes the choices taken by the theorem prover to build

a valid proof. The proof checker uses the oracle string to guide its proof search with the non-deterministic theorem prover, i.e., the oracle string makes the search deterministic and prohibits backtracking.

Interactive Proof-Carrying Code [Tsu00] is based on the idea of interactive proof systems and also tackles the problem of large proofs. The producer still constructs a proof for the desired property. The consumer only receives the program and communicates with the producer to convince himself with overwhelming probability that a proof for the program w.r.t. the property of interest exists. In 2005, Tsukada presents an interactive proof system for memory safety [Tsu05].

Extended Proof-Carrying Code [PD08a, PD08b, PDHL10] is a variation of PCC in which a program that can generate the proof string is attached to the program instead of the proof itself. After the producer verified the program, he uses the proof to generate a program that constructs the proof string. The consumer runs the attached program in a virtual machine, thus it is ensured that the untrusted, proof generating program is safely executed, and then checks the proof regenerated by the program.

Amtoft et al. [ADZ+12] certify conditional information flow of SPARK programs. When their tool checks that the user annotated precondition of a procedure implies the user annotated postcondition, it also generates a Coq proof. The Coq proof shows that the precondition derived from the user annotated postcondition is indeed a valid precondition and that the user defined precondition implies the derived precondition.

Proof-Carrying Authentication [AF99] is a PCC variant for access control. The idea is that a client must provide a proof, a derivation tree in an authentication logic, to the server that his (the client's) access is consistent with the server's access policy. Instead of checking access rights, the server simply checks the proof.

Proof-carrying apps [HNTS16] is a generalization of PCC that ensures that a plug-in never violates API contracts, i.e., for all methods in the plug-in it yields that when they are called in a state satisfying their precondition, then during execution they always guarantee that the precondition of other API method are always fulfilled when these methods are called and at the end of the method the postcondition is true. The producer verifies this behavior and generates a proof artifact which is checked by the consumer. The authors consider different verification techniques for their framework, but so far only the realization with the prover KeY, which produces a certificate in form of a proof script, worked out.

Our approaches do not use theorem provers or mathematical proof objects. In contrast to interactive and extended proof-carrying code [Tsu00, PD08a], our certificates are still some form of proof artifact. Furthermore, configurable program certification is not restricted to a certain property like proof-carrying authentication [AF99] or the approach by Amtoft et al. [ADZ+12]. However, configurable program certification cannot assure security properties like Amtoft et al. [ADZ+12]. During the development of our approaches, we did not focus on the trusted computation base. Our approaches are not foundational like the approaches by Appel et al. [App01] or Wildmoser et al. [WNKN04]. Moreover, only the approach by Wildmoser et al. [WNKN04] is directly configurable. Similarly to Chaudhuri [Cha12], in our optimization approaches we often remove some information from our basic certificate and use a search bound to restrict the number of recomputed abstract states. Finally, we like to mention that our reduction approach follows the idea of Necula [NL98a] and stores only those information which cannot be recomputed.

Type Inference The *typed assembly language* (TAL) [MWCG99] type system ensures that well-typed programs stick to the abstraction of their corresponding source program.

For example, an integer variable is not dereferenced. A compiler is used to transform a correctly typed source program into a well-typed TAL program. Based on the type information and the TAL type system, the consumer type checks the TAL program to validate that it is well-typed.

Certified assembly programming (CAP) [YHS03, YHS04] is an assembly language that allows to annotate assertions to instructions. These assertions represent the preconditions of these instructions and are considered during type analysis. Type analysis based on the rules presented by Yu et al. [YHS03, YHS04] aims at showing that the safety policy is not violated, the program never gets stuck, and when the program executes a instruction with an assertion, this assertions is true before execution of the instruction. Therefore, it is sufficient that the analysis proves that the program is well-typed.

Crary and Weirich [CW00] discuss *resource bound certification*. Their main emphasize is on the certification of computation time. More concretely, they introduce the type system LX_{res} which incorporates a virtual clock in its types. The virtual clock bounds the execution time. Like in TAL, a type checker can be used to check the LX_{res} typing of a program and thus confirm the resource bound.

Vanderwart and Crary [VC05] suggest a typed assembly language for responsiveness. Well-typed assembly programs ensure responsiveness properties, i.e., after at most X statements the program executes a yield statement. They use a compiler to add the yield statements and to provide a proper type annotation, the certificate, which can be used to show that the compiled program is well-typed.

The *mobile resource guarantees framework* [AGH+05] is a framework to certify resource bounds. The producer performs a type analysis of the high level Camelot program. Based on the analysis result, he produces a logic proof for the low level Grail program. The underlying idea is to derive logic assertions from the typing judgments s.t. the assertions describe the semantics of the typing judgment. Finally, the consumer checks the logic proof.

Cassandra [LMS+14] is an app store enhanced with a PCC mechanism to assure information flow policies. Based on an information flow type system, a server proves that the app adheres to the policy. The server sends the type annotations for fields, return values and method parameters, which he derived during the analysis, to the consumer. The consumer uses the type annotations to replay the analysis.

Harren and Necula [HN05] present a generic certification framework based on a parameterized type system. The parameters, which define the safety policy, are the type constructors, subtyping rules, and descriptions how to derive the types of expressions. Given the typing judgment for all basic blocks and the safety policy, the consumer validates the program via type checking.

Hamid et al. [HST+03] present a foundational PCC approach. The producer starts with a type analysis. When this analysis shows that the assembly program is well-formed, the assembly program is compiled into a machine code program in form of an initial machine state. Additionally, the typing judgments are used to construct a proof for the initial machine state, which states that for any machine state reachable from this state a corresponding well-typed assembly program exists.

Wu et al. [WAS03] present a solution how to use a proof scheme, e.g., a type system, in a foundational certification approach. They propose that the producer sends the proof scheme together with a soundness proof. Then, the producer can use the proof scheme to prove program safety. Afterwards, he can send hints like typing annotations or type derivations. When the consumer successfully validated the proof scheme's soundness proof, he can use the hints together with the proof scheme to check whether the program

is indeed safe.

We do not consider type systems in our approaches. Furthermore, our approaches are still not foundational like the approaches by Hamid et al. [HST+03] or Wu et al. [WAS03]. In contrast to TAL [MWCG99], resource bound certification [CW00], the approach by Vanderwart and Crary [VC05], the mobile resource guarantees framework [AGH+05], and Cassandra [LMS+14], our approaches do not consider a specific type of property, but a large class of safety properties. However, our approach only considers safety properties and cannot assure information flow properties like Cassandra [LMS+14]. Since certified assembly programming [YHS03] also considers the validity of assertions, it is already flexible in the property proven, but the type system is fixed. Typically, the type systems used by the approaches are fixed. Only, the approaches of Hamid et al. [HST+03] and Wu et al. [WAS03] are configurable like our approaches. Hamid et al. [HST+03] use a parameterized type system and Wu et al. [WAS03] suggest to send the used proof scheme with its soundness proof.

Abstract Interpretation and Dataflow Analyses Some approaches use the fixpoint information of an abstract interpretation to derive certificates, which are more or less mathematical proofs. Most of the certification approaches for abstract interpretation or dataflow analysis store (a part of) the fixpoint computed by the abstract interpreter and the dataflow analysis, respectively.

Seo et al. [SYY03] propose a technique to compute a Hoare logic proof, a derivation tree, from a fixpoint of an abstract interpretation. Abstract states are translated into first order logic formulae and are used as pre-, postconditions, or loop invariants of the program statements. Based on those pre-, postconditions, loop invariants, and the program structure an algorithm builds a proper Hoare logic derivation tree, the certificate. The consumer checks that the derivation tree is valid.

Proof-producing program analysis [Cha06a] also grasps the abstract states as pre- and postconditions. To ensure that the abstract states are indeed valid pre- and postconditions, for each statement (edge) in the program a proof is constructed that shows that the concretization of the abstract state that describes the precondition of the statement implies the weakest precondition computed from the statement and the concretization of the abstract state that is used as postcondition of that statement.

Besson et al. [BCJ14] and Cornilleau [Cor13] use a similar approach to proof-producing program analysis. However, the proof obligations (verification conditions) differ slightly. For each program point p, it is proven that if a concrete state s satisfies the program point's precondition, then for each transition $s \to s'$ the successor state s' will fulfill the postcondition of the program point. Additionally, for any successor program point p' it is shown that the postcondition of p implies the precondition of p'.

Lightweight bytecode verification [Ros03] is a PCC technique for constraint based dataflow analyses. The goal of lightweight bytecode verification is to obtain a solution (fixpoint) to a dataflow analysis problem for a downloaded program without a complete fixpoint computation. The idea is to store all dataflow facts that are associated with program locations that are backward targets, e.g., loop heads. Rose [Ros03] proved that the fixpoint can be recomputed from the constructed certificate in a single pass over the program when the nodes are visited in ascending order.

Klohs et al. [KK05] improve the memory consumption of lightweight bytecode verification. Dataflow facts, recomputed or stored in the certificate, are kept in memory only as long as they are needed for validation. Additionally, the authors suggest to use the exploration order that reduces the maximal number of required dataflow facts that must

be availabe for a validation step most.

In his PhD thesis, Klohs [Klo09] presents an extension of lightweight bytecode verification to interprocedural dataflow analyses. The validation procedure is quite similar, but instead of dataflow values he uses dataflow functions as dataflow facts. Dataflow functions describe how an input dataflow value changes. They are used to describe how the dataflow value for the current program location is derived from the input value of the corresponding procedure.

Amme et al. [AMA07] also suggest a framework for the safe transport of dataflow facts. They suggest to keep dataflow facts at so called annotation points. Contrary to lightweight bytecode verification, an annotation point is not a backward target, but a program location that has an outgoing edge leading to a backward target. Nevertheless, certificate validation remains similar. The consumer must use a single pass over the program in reverse post order to either reconstruct the complete fixpoint or detect that some of the provided dataflow facts are incorrect.

Abstraction-Carrying Code (ACC) [APH05a, APH05b, APH08] is used to ensure that a constraint logic program adheres to a safety policy, which is described by assertions, e.g., pre- and postconditions. The producer applies an abstract interpretation on the constraint logic program, uses the fixpoint (abstraction) and the safety policy to generate a verification condition, and proves the condition. When the condition is true, the program is correct w.r.t. the safety policy, the fixpoint becomes the certificate. The consumer checks whether the certificate is indeed a fixpoint, uses the fixpoint and the safety policy to generate the verification condition, and proves the generated condition. In a later improvement of ACC [AAPH06, AAH12], only a subset of the fixpoint is stored in the certificate. The idea is to keep the entries of the fixpoint that are relevant, i.e., if not stored at least one update to that entry during validation causes a successor to be computed although a successor for that entry has already been computed during validation for a previous version of that entry. To recompute the fixpoint in a single pass, the consumer must use the same exploration order as the producer.

Besson et al. [BJP06] proposes a PCC technique for certified abstract interpretation. A certificate is a partial fixpoint together with a reconstruction strategy, which describes how to reconstruct the complete fixpoint in a single pass. The reconstruction strategy may also describe when an abstract state can be dropped, i.e., is no longer needed for validation. During validation, the consumer uses the reconstruction strategy and a certified abstract interpreter[12] to reconstruct the fixpoint while subsequently checking program safety and that the reconstruction strategy is valid, i.e., all locations are considered, no location is considered twice, a fixpoint is recomputed. Later Besson et al. [BJT07] relax their certification approach. On the one hand, they discuss how to weaken a fixpoint computed by an abstract interpretation s.t. it still ensures the safety property. On the other hand, they no longer require that the fixpoint can be recomputed in a single pass. Thus, a certificate is no longer a partial fixpoint plus a reconstruction strategy, but a partial fixpoint plus a sequence of program locations. The sequence of program locations determines the exploration order for the recomputation of the fixpoint.

A previous version [JW14] of our basic configurable program certification does not support precision adjustment and uses a safety check, an additional CPA and CCV operator. The safety check examines whether none of the unsafe concrete states, described by a set of abstract states, can be reached. Similarly, previous versions of the two optimization approaches [Jak15] are proposed. These previous versions also rely on the safety check, do not support precision adjustment, and are restricted to transfer functions. Moreover, the

[12] This means the soundness of the abstract interpreter is proven, e.g., with the help of a PCC approach.

definition of a valid, partitioned certificate and a (well-formed) ARG deviates from ours. Especially the last deviation is responsible for the different construction of the certificates. The previous versions of the two optimization approaches [Jak15] rely on the indegree of ARG nodes while in this thesis we use the covered node set. Practically, the constructed certificates are rarely different. Remember that in our evaluation we still use the previous technique [Jak15] to construct the certificates.

EviCheck [SA15] is an approach to certify Android app conformance to a security policy, a set of rules on allowed permission usage. The producer performs a backward reachability analysis on the call graph to compute a mapping from methods to permissions s.t. a method is mapped to a permission only if a respective permission usage can be reached from the call of that method. The map is used as certificate. The consumer checks if the map is valid and if the map implies that the app adheres to the security policy.

Goal-directed weakening [SYYH07] is a generic framework to weaken a fixpoint computed by an abstract interpreter, a potential certificate, s.t. the resulting fixpoint can still prove the property of interest. To instantiate the framework, one needs to define an abstract value slicer describing an extractor domain and a backtracer.

Example-guided abstraction simplification [GR10, GR14] is a technique to simplify an abstract model. The underlying idea is to replace the abstract domain with a simpler one that keeps the same abstract semantics of the model, i.e., it does not introduce new spurious counterexamples. The authors introduce the concept of correctness kernels to compute the simpler abstract domain.

In contrast to proof-producing program analysis [Cha06a] and the approaches by Seo et al. [SYY03], Besson et al. [BCJ14], and Cornilleau [Cor13], our configurable program certification does not generate a formal proof from the fixpoint. Like our basic approach, Abstraction-Carrying Code [APH05b] stores the complete fixpoint, but Abstraction-Carrying Code is applied to constraint logic programs instead of imperative programs and considers a different concept for expressing program safety. Similar to our optimization approaches, especially the reduction approach, lightweight bytecode verification [Ros03] and its optimizations [KK05, Klo09], Abstraction-Carrying Code [APH05b, AAPH06] and the approaches by Besson et al. [BJP06, BJT07] also store parts of the fixpoint and validate or reconstruct the fixpoint. However, we are not restricted to analyses that compute one abstract state per location and our validation is order independent. Moreover, we often store a larger part of the fixpoint to get rid of the merge operator during validation and become order independent. Except for Besson et al. [BJP06] who use a certified abstract interpreter, our trusted computing base is smaller. Additionally, certification techniques based on dataflow analyses typically only validate the fixpoint and do not inspect program properties, e.g., safety properties. Currently, our configurable certification approach does not weaken or simplify the abstraction computed during verification.

Model Checking Certification approaches for model checking differ in the type of property they assure, temporal logic or a restriction to pure safety properties, and the type of certificates.

Peled and Zuck [PZ01] consider certification of automaton-based LTL model checking. They present an algorithm that can be used after successful model checking of program P w.r.t. property ϕ to construct a deductive proof showing that P satisfies ϕ. Similarly, Peled et al. [PPZ01] discuss how to generate a deductive proof from the verification of a just discrete system w.r.t. a LTL property.

Kupferman and Vardi [KV04, KV05] use a bounded, odd ranking function to certify that the language of $M \times \neg\phi$ is empty, i.e., the model M satisfies the LTL property ϕ. The consumer checks that the provided function is indeed a bounded, odd ranking function for $M \times \neg\phi$.

Chaki [Cha06b] describes a technique to certify predicate model checking of C programs w.r.t. LTL properties. Their technique is based on a ranking function, which ensures that the language of the product of program and negated property is empty. The certificate consists of a witness, an encoding of the ranking, and the refutation proofs for a set of verification conditions. The verification conditions check that the witness is indeed a proper encoding.

Evidence-based model checking [TC02] describes the model checking problem by a boolean equation system and uses support sets as certificates. A support set explains the value of a variable in a particular solution of the equation system.

Namjoshi presents the idea of a *certifying model checker* [Nam01], which certifies μ-calculus properties. Certification is based on the correspondence between μ-calculus model checking and memoryless winning strategies in the corresponding parity game. The certificate, an invariant and a ranking function, is extracted from the winning strategy. To validate the certificate, three local conditions of the invariant and the ranking function must be checked.

Hofmann et al. [HNR16] also use the correspondence between μ-calculus model checking and winning strategies in the corresponding parity game. They extend their μ-calculus model checker to produce a memoryless winning strategy, the certificate, in the corresponding parity game and discuss how to check that the strategy is indeed a winning strategy.

Henzinger et al. propose *temporal-safety proofs* [HNJ+02] to certify the result of predicate abstract model checking w.r.t. a temporal safety property. After successful predicate abstract model checking, the generated abstract reachability tree is used to construct the certificate. Basically, the set of ARG nodes, locations plus predicate states, is used as program invariant. To show that this set of ARG nodes is a program invariant, the approach relies on a standard safety (invariant) rule [MP95]. However, the ARG structure is used to simplify the LF proofs of the premise. For example, to show preservation of the invariant for each ARG node it is proved that the strongest postcondition along a syntactical edge is covered by the respective ARG successor node or those nodes that cover that ARG successor.

Dräger et al. [DKFW10] describe certification with the model checker SLAB, which is used to prove unreachability of error conditions in infinite state systems. For certification, the final abstraction is transformed into an inductive verification diagram, a directed graph whose nodes are labeled with predicate states and its edges are labeled with sets of transition relations. Validity of such a verification diagram is checked with a standard invariant (safety) rule [MP95].

Conchon et al. [CMZ15] present the certification approach applied by the model checker Cubicle, a model checker proving safety properties (non-reachability of error states) of parameterized systems. The approach uses the complement of the overapproximation of the set of states from which an error state is reachable as inductive invariant. To show that it is indeed an inductive invariant, Cubicle generates proof obligations for initiality, preservation, and safety. The proof obligations are checked by a SMT solver.

Extreme model checking [HJMS03] is a regression verification technique for predicate abstract model checking of temporal safety properties, which can be used for certification as well. After a successful verification the abstract reachability tree (ART), an abstract

representation of the state space, is stored. Henzinger et al. [HJMS03] present an algorithm that checks if an ART conforms to a program. In principal, conformance checking is similar to checking well-formedness of our ARGs. If the algorithm succeeds, the program is safe; otherwise model checking must continue at those points where the conformance checks failed.

Xia and Hook [XH04] present a technique to certify temporal logic properties of C programs. The producer starts with a predicate analysis. Based on the predicate abstraction, especially the predicates, he computes a boolean program, an abstraction of the original program. Afterwards, he compiles the original program s.t. the boolean program remains a valid abstraction for the compiled program and transfers compiled program, boolean program, and the set of predicates to the consumer. The consumer checks with the help of the set of predicates that the obtained boolean program is a valid abstraction of the compiled program.

Search-Carrying Code [TA10] is a PCC alike technique for explicit state model checking. Instead of a proof, it uses a search script as certificate. The search script describes how the state space exploration was performed. A consumer uses the search script to replay the search while being able to detect whether it is incomplete or incorrect. Based on additional subgraph information, the consumer can partition the search script and perform the replay in parallel.

Precision reuse [BLN+13] is originally intended for regression verification, but can also be used in a certification context. The idea is to store the abstraction precision, which specifies the abstraction level of the analysis after verification, and start the reverification with the precision stored from the last verification.

Correctness witnesses [BDDH16] are proposed as a flexible, exchangeable format for correctness results. The idea is that a different verifier can validate the correctness witness. Technically, a correctness witness is an automaton whose states are labeled with invariants encoded as boolean C expressions and its edges are labeled with statements. To validate the correctness witness, a verifier must run it as an observer automaton and check if the provided invariants are indeed true. Since the witness generating verifier decides which of its local invariants it wants to include in the automaton, a different verifier may fail to validate the correctness witness.

In contrast to the approaches by Chaki [Cha06b], Peled et al. [PZ01, PPZ01], Kupfermann et al. [KV04, KV05], and Hofmann et al. [HNR16], certifying model checkers [Nam01], and evidence-based model checking [TC02] our approach cannot deal with arbitrary temporal logic. It is restricted to safety properties. Like temporal safety proofs [HNJ+02], SLAB [DKFW10], and Cubicle [CMZ15], we also use the safety rule [MP95] to check validity of our certificates. Especially, our basic approach directly checks the premise of the safety rule. Our optimizations are a mixture of invariant recomputation and checking. While existing approaches tend to use an encoding based on logic, we use abstract states to encode the invariant. Furthermore, often the abstraction domain is fixed in the existing approaches. Also, extreme model checking [HJMS03], the approach of Xia and Hood [XH04], and Search-Carrying Code [TA10] are restricted to a particular domain. All of these three use a different kind of certificate. With Search-Carrying Code, we share the idea of certificate partitioning. We apply partitioning at the producer side and Search-Carrying Code partitions at the consumer. Precision reuse [BLN+13] and correctness witnesses [BDDH16] are more flexible w.r.t. the abstract domain, although their certificates are totally different from ours. However, precision reuse is less general than our approach because it is tailored to domains whose abstraction level can be set. Moreover, we think that our approaches cannot be simply replaced by the concept of

correctness witnesses. It remains unclear how to represent certain abstract states with boolean C expressions, e.g., we lack the intuition how to encode the (un)initialization of a variable. Additionally, relative completeness is not considered for correctness witnesses.

Rewrite Systems Alba-Castro et al. [ACAE08] suggest a PCC technique to certify safety properties of Java programs, which is based on rewrite logic. The producer uses an abstract rewrite system derived from the concrete rewrite system representing the concrete Java semantics and a consumer provided annotation of variables. The annotation defines for each variable which abstract domain to use. When the producer succeeds to prove the safety property in the abstract rewrite system, he either provides the set of rewrite sequences or the sequences of applied rewrite steps. The consumer checks that all rewrite steps are correct, that no alternatives are forgotten, and that the safety property is never violated. Later, the approach is extended to certify information flow properties [ACAE09, ACAE10]. For this, the concrete rewriting semantics is extended to consider security classes for each object and context. The abstract rewrite system does not rely on consumer annotations, but only tracks the security classes of objects.

Proof-Carrying Hardware The PCC principle is not only applied to software programs, but also to reconfigurable hardware. Two main approaches exists: Proof-Carrying Hardware and Proof-Carrying Hardware intellectual property.

Proof-Carrying Hardware (PCH) [DKP09, DKP10] was proposed by Drzevitzky et al. in 2009. PCH ensures properties of the bitstream used for reconfiguration. The first instance of PCH considers combinatorial circuits and ensures functional equivalence of the implementation, the logic encoded in the bitstream, and the specification. The producer builds the miter from the bitstream and the specification and proves that the miter is unsatisfiable. The consumer gets the bitstream with a resolution proof for the miter, rebuilds the miter, and checks if the resolution proof is valid for the miter. Later approaches are extended to sequential circuits, mainly based on unrolling, and to ensure correctness of memory access monitors [WDP14] or worst-case completion time of hardware modules [WP16].

Proof-Carrying Hardware Intellectual Property (PCHIP) [LJM11, LJM12] is an approach to ensure security properties of a hardware module given in a subset of Verilog, a hardware description language. The approach is based on a formal definition of Verilog and its semantics in Coq. Both, producer and consumer, translate the hardware module into the Coq representation. Then, the producer generates the correctness proof in Coq and the consumer checks that proof. Later, the approach is extended to data secrecy and information flow properties [JM12, JYM13].

The two approaches PCH and PCHIP have nothing in common with our configurable program certification. They do not consider software programs. Furthermore, they use different proof techniques, SAT solvers and theorem provers, and can also be applied to certain non-safety properties.

5 Programs from Proofs

In this chapter, we present an alternative, general framework to enable a fast program validation for the program executor, the consumer. Similar to the previous configurable program certification approaches, the alternative *Programs from Proofs* (PfP) approach also intends to decrease the validation costs of the consumer. To achieve this goal, we keep the underlying principle of Proof-Carrying Code: shift the major validation effort from the consumer into the producer. However, our Programs from Proofs approach does not provide additional information to speed up the consumer validation, but it changes the structure of the program itself.

Like compilers use program transformations based on constant propagation [NNH05, p. 72], available expressions [NNH05, pp. 39ff], and so forth to accelerate the execution of a program, our Programs from Proofs approach brings the program into an efficiently provable form. Looking at our example program `SubMinSumDiv` (Fig. 2.1), it would be much simpler to show our desired property `nonneg` (after initialization $z := 0$, variable z never holds a negative value), if the computations of the sum and the division were separated. In general, the producer should restructure his program s.t. that the consumer can check the restructured program with an easy analysis, namely a fast and only flow-sensitive dataflow analysis.

Due to their imprecision, dataflow analyses often fail to prove a property on the original program. Remember that our sign dataflow analysis failed to show that our example program `SubMinSumDiv` is safe w.r.t. property `nonneg`. In many cases, more precise and, thus, less efficient analyses, which incorporate (some) path-sensitivity – possibly tuned via refinement –, succeed to show the desired property. We utilize this observation in our Programs from Proofs approach. In principle, we let the producer run a more precise, but also less efficient analysis to prove program safety. After a successful verification, the proof, the ARG, records where path-sensitivity is needed, i.e., where the information of different paths must not be integrated. The producer uses this information to restructure the program s.t. it is verifiable with a simple *dataflow analysis*.

Program transformations have already been used for a while to simplify verification. Well-known compiler optimizations like constant propagation [NNH05, p. 72] or dead code elimination [ALSU07, p. 535], [Ken78] are applied in the front ends of verification tools like CPACHECKER [BK11b] or Astrée [CCF+07] to make subsequent verification easier. Compiler optimizations are one example for a heuristic change. After optimization, it cannot be assured that verification of the optimized program is easier.

Many approaches, e.g., [GJK09, BSIG09, SDDA11], aim at loop restructuring. Gulwani et al. [GJK09] use an invariant generator to refine the structure of a multi-path loop. The goal is to make the path interleavings explicit. Balakrishnan et al. [BSIG09] refine the possible execution through a loop, too. First, they heuristically group the syntactical paths that represent a single, syntactical loop iteration. With the help of abstract interpretation, they detect which sequences of the groups up to a bound k do not describe proper loop iterations and exclude those sequences from the control flow. To simplify verification in which loop invariants must be generated, Sharma et al. [SDDA11] compute phase splitter predicates to decompose multi-phase loops into a sequence of single phase-loops.

Leroy [Ler02] transforms correct Java bytecode into a behavioral equivalent form that meets the requirements his bytecode verification algorithm requires for acceptance. To ease worst case execution time analysis, Puschner [Pus02] describes how to transform a program for which the worst case execution time can be analyzed into a single execution path, i.e., a program that is purely sequential and that realizes branches with the help of a single machine instructions that can conditionally set a value of variable to a (variable) value. Hunt et al. [HS06] use the result of a type based, flow-sensitive information flow analysis to derive an equivalent program that can be analyzed with a flow-insensitive type analysis. Static language refinement [BSI+08] applies path-insensitive forward and backward abstract interpretation to remove infeasible paths that do not reach the error state. Especially during reverse synthesis [YKNW08], the Echo framework [SYK05] uses a heuristic approach called verification refactoring [YKNW08, YKW09], to simplify the program and to reduce the complexity of verification. Amongst others, verification refactoring tries to undo code optimizations. Verification refactoring is stopped when the transformed program meets some metrics. Furthermore, it requires manual effort. Often, verification refactoring must be guided by a user who decides which transformation to apply next. If an applied transformation has not been available before, it must be proven that it is semantics-preserving.

None of the existing approaches guarantees all the properties for our Programs from Proofs approach mentioned above. Our Programs from Proofs approach is specifically tailored to the desired property and the simple analysis used by the consumer, but not limited to one specific analysis. The obtained restructured program can always be proven with the simple consumer analysis. By construction, the restructured program is behaviorally equivalent to the original program. Restructuring is fully automatic, may restructure the complete program, and should not be limited to infeasible paths. Next, we describe the details of our Programs from Proofs approach. As before, we start with a general overview.

5.1 Overview of Programs from Proofs

The idea for Programs from Proofs was introduced by Wonisch et al. [WW12]. As a proof of concept, they used predicate abstraction and a control flow analysis to prove protocol

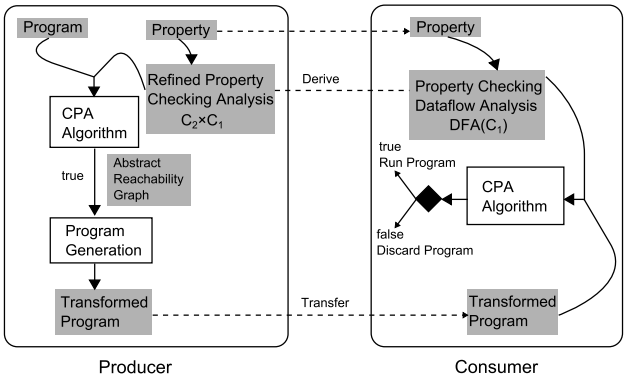

Figure 5.1: Overview of Programs from Proofs

properties. Further Programs from Proofs instances [JW15] use arbitrary (predicated) dataflow analyses to ensure invariants. Recently, we integrated the previous instances into a common setting [JW17]. Thereby, we enlarge the set of considered safety properties and relax restrictions on the supported configurable program analysis instances. In this chapter, we slightly generalize the Programs from Proofs approach [JW17], which integrated the instances in a common setting. Now, the approach also supports precision adjustment. The class of properties will be the same as in the integration [JW17], but we use our common specification of properties, which differs a little from the property automata considered in the integration [JW17].

Figure 5.1 gives an overview of the Programs from Proofs procedure, which is another instantiation of the general principle described in the introduction. First, the producer analyzes whether a program is correct w.r.t. a given property. Essentially, he applies the basic verification principle presented in the background chapter (Sections 2.3 and 2.4). He uses a *refined property checking analysis* to steer the CPA algorithm, which performs the analysis. The refined property checking analysis is a composite analysis enhanced with the property (automaton). The composite analysis consists of two parts: a *property checking analysis* $\mathbb{C}_1$, a simple analysis that will check the property, and a more powerful *enabler analysis* $\mathbb{C}_2$. The idea of the composite analysis is that the enabler analysis should exclude spurious counterexamples found by the simpler property checking analysis. For example, the enabler analysis excludes infeasible paths and separates paths where needed. Note that we assume that the enabler analysis is the factor that makes the producer analysis costly. First, it is more powerful than the property checking analysis. Second, we often use counterexample guided abstraction refinement (CEGAR) [CGJ+00] to determine the required precision of the enabler analysis. After a successful verification (result true), the producer generates a behaviorally equivalent program from the abstract reachability graph, the proof. The clue of the program generation is the following:

$$P \models^{\checkmark}_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} \mathcal{A} \implies P_{\text{trans}} \models^{\checkmark}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})} \mathcal{A} \ .$$

Program generation restructures the original program in such a way that the property checking analysis alone is sufficient to prove the property. Finally, the producer delivers the transformed program to the consumer.

The consumer receives the transformed program from the producer. We assume that the transformed program might be flawed during delivery. To analyze the transformed

program, the consumer only needs the simpler and less expensive property checking analysis of the producer's analysis. Independent of the analysis technique configured by the property checking analysis, the consumer reconfigures the property checking analysis such that it becomes a more efficient but less precise dataflow analysis. Then, the consumer also uses the CPA algorithm to analyze whether the received, transformed program is correct w.r.t. the property inspected by the producer. However, the CPA algorithm is now configured with the dataflow version of the property checking analysis – of course enhanced with the property (automaton). If the verification succeeds, the transformed program can be executed. Otherwise the transformed program is discarded.

The consumer's verification procedure already ensures that the verification will only succeed if the program is correct w.r.t. the analyzed property. Unsafe programs, e.g., flawed programs, are discarded. Hence, for the Programs from Proofs approach we are more interested in relative completeness, i.e., whether the consumer can always verify the transformed program generated by the producer after a successful verification.

Furthermore, note that for the sake of comprehensibility, in our overview we left out the initial abstract state and the initial precision, two additional inputs of the CPA algorithm. The initial abstract states need to fulfill some constraints. For the producer, we require that his initial abstract state is compatible with the Programs from Proofs approach. For example, the initial abstract state uses the initial automaton state and the enabler state does not restrict the property checking state. We discuss compatibility in full detail when all details are known. The initial abstract state of the consumer must be derived from the producer's initial state. In principle, we reuse the property checking state and the automaton state of the producer's initial abstract state, but we need to replace the location information in the property checking state with the initial program location of the transformed program. In contrast, the choice of the initial precision is unrestricted.

We continue with the *trusted computing base* of our Programs from Proofs approach. Remember that the trusted computing base contains all entities that the consumer must trust when he relies on the outcome of his validation. In our Programs from Proofs approach, the consumer must trust the definition of the property checking dataflow analysis enhanced with the property automaton as well as its implementation and the implementation of the CPA algorithm. Of course, the consumer is free to use his own implementations and does not need to rely on the implementations used by the producer. Nevertheless, the consumer derives his analysis configuration, the enhanced property checking dataflow analysis, from the producer's property checking analysis. In many aspects, he must trust the definition of the producer's property checking analysis. Compared to the verification of the producer, which the consumer might do without the Programs from Proofs approach, the consumer does not need to trust the definition of the powerful, often more complex enabler analysis and its implementation. Due to the higher complexity, the implementation of the enabler analysis is likely more error-prone. To further decrease the size of the trusted computing base, the consumer may leave out the ARG construction during his analysis. The ARG is not needed on the consumer side. Hence, he may only execute the CPA algorithm version described in Algorithm 1. Furthermore, the consumer may verify the proper realization of the CPA algorithm implementation. The consumer could also prove the definition of the enhanced property checking dataflow analysis and either directly extracts an implementation for the enhanced property checking dataflow analysis from the proof or shows that his implementation is a correct refinement of the definition.

With the last paragraph, we finished the high level discussion of the PfP approach. However, before we come to the details of the Programs from Proofs approach, we need

```
0: z:=0;
1: if(x<0)
2:   if y<x
3:     z:=-y;
     else
4:     z:=-x;
5:   z:=z+10;
...
```

```
0: z:=0;
1: if(x<0)
2:   if y<x
3:     z:=-y;
4:     z:=z+10;
     else
5:     z:=-x;
6:     z:=z+10;
...
```

Figure 5.2: Beginning of program `SubMinSumDiv` and a sketch of a possible, behaviorally equivalent transformation

to reconsider our property specification. Some property automata are improper for the Programs from Proofs approach. Our property automaton `pos@l5` (left of Fig. 2.2), which specifies that at location l_5 the value of variable z must be greater zero, is an example for such an improper property automaton. The problem is that due to the program transformation the program locations may change. Thus, the property automaton may check different behavior on the original and the transformed program. For illustration, assume that in our example program `SubMinSumDiv` we moved the assignment $z := z+10$; into the branches (see Fig. 5.2). In the transformed program (right of Fig. 5.2), the property automaton `pos@l5` misses to check the behavior of the upper branch ($x < 0$). To inspect the same behavior for the upper branch as in the original program, it must be checked that at location l_4 (line 4) variable z has a value greater zero. Furthermore, in the lower branch ($\neg x < 0$) the value of variable z is inspected to early. Hence, the transformed program does not fulfill the specification. At location l_5 (line 5) of the transformed program, variable z has value zero. To check the same behavior as in the original program, the value of variable z must be inspected at location l_6 (line 6).

We just discussed that the property automaton `pos@l5` is improper because it is affected by the program locations represented in the control state of a concrete state. In general, property automata are unsuitable for the Programs from Proofs approach whenever they contain at least one transition that refers to the control state. More concretely, a transition refers to the control state whenever there exist concrete states $c \in C$ and $c' \in C$ with the same data state $\mathrm{ds}(c) = \mathrm{ds}(c')$ but only c is considered by the transition. Hence, the property automata considered by the Programs from Proofs approach must be independent of the control state. We say they must be *control state unaware*. All transitions that consider a concrete state $c \in C$ in their set C_{sub} must also consider all concrete states $c' \in C$ with the same data state $\mathrm{ds}(c) = \mathrm{ds}(c')$. Our property automaton `nonneg` (right of Fig. 2.2) fulfills this condition. It is an example for a control state unaware property automaton. We will use property automaton `nonneg` throughout this chapter to explain our Programs from Proofs approach. In the following, we formally define when a property automaton belongs to the restricted class of control state unaware property automata.

Definition 5.1 (Control State Unaware Property Automaton)**.** A *property automaton* $\mathcal{A} = (Q, \delta, q_0, q_{\mathrm{err}})$ *is control state unaware* if $\forall (q, op, C_{\mathrm{sub}}, q') \in \delta : C_{\mathrm{sub}} = \{c \in C \mid c' \in C_{\mathrm{sub}} \wedge \mathrm{ds}(c) = \mathrm{ds}(c')\}$.

With the definition of a control state unaware property automaton, we finished the preparatory considerations. In the remainder of this chapter, we now discuss the de-

tails of the Programs from Proofs approach. We maintain the course of the Programs from Proofs procedure and start with the presentation of the producer verification.

5.2 Producer Verification of the Original Program

The overview of the Programs from Proofs procedure states that the producer uses the CPA algorithm to prove program safety. We are already familiar with the CPA algorithm. Thus, we focus on the appropriate configuration of the CPA algorithm for our Programs from Proofs approach. Furthermore, we discuss additional properties of the ARG generated by the producer verification. These additional properties are special to the configuration used by the producer and are required later. Moreover, we explain how to automatically get a suitable precision for the enabler analysis.

We begin with the major part of the configuration, the input CPA. As already mentioned, the CPA used by the producer, a refined property checking analysis, consists of two parts: the property checking analysis and the enabler analysis. Before we come to the details of a refined property checking, we describe the requirements on the property checking analysis. These requirements are much more restrictive than those on the enabler analysis.

5.2.1 Property Checking Analysis

As already mentioned, the main task of a property checking analysis is to inspect program safety on the producer and the consumer side. To examine program safety, the analysis must know the safety specification, which is given in form of a control state unaware property automaton. Throughout this thesis, we use enhanced CPAs to integrate property specifications into the analysis. Hence, a property checking analysis is always a CPA enhanced with a control state unaware property automaton.

However, not every enhanced CPA is a property checking analysis. We require a certain structure of the CPA that is enhanced. To generate a behaviorally equivalent program, the ARG constructed during the producer verification may only represent syntactical program paths. The producer analysis requires a location CPA to consider syntactical program paths only. Syntactical paths in a program, e.g., in the generated program, are always encoded with program locations and control flow edges. Since the consumer should not consider paths that the producer did not, the consumer analysis of the generated program also needs the location CPA. In many cases, a location CPA alone is not appropriate to inspect a given property specification. Especially, if the control state unaware property automaton does not uniquely handle all concrete states, the location CPA will not be sufficient. In this case, the property automaton distinguishes between some data states. Hence, we require an additional CPA $\mathbb{C}$, e.g., a sign CPA $\mathbb{S}$, which can distinguish between the classes of data states that are treated differently in the property specification. Thus, the CPA enhanced in a property checking analysis is a location CPA optionally combined with another CPA.

As already mentioned, the location state plays an important role in the property checking analysis. In the reminder of the chapter, we often need to refer to the location state of an abstract state explored by the property checking analysis. It is very tedious to always distinguish between the structure of the CPA $\mathbb{C}_1$ being enhanced in a property checking analysis (either $\mathbb{L}$ or $\mathbb{L} \times \mathbb{C}$). Similar to the control state of a concrete state, we define a function to access the control state information, the so called *abstract control state*, of the CPA $\mathbb{C}_1$.

Definition 5.2. Let $e_1 \in E_{\mathbb{C}_1}$ be an abstract state. Its *abstract control state* $\text{acs}(e_1)$ is $\text{acs}(e_1) = e_1$ if $\mathbb{C}_1 = \mathbb{L}$ and $\text{acs}(e_1) = l$ if $\mathbb{C}_1 = \mathbb{L} \times \mathbb{C}$ and $e_1 = (l, \cdot)$.

After the digression on an abstract control state, we resume the discussion of the requirements on a property checking analysis. The structural restriction alone is not yet sufficient to generate behaviorally equivalent programs or to enable the verification of the generated program. We need further restrictions on the analysis' operators, especially the transfer relation. To retranslate the generated program into deterministic programming languages like C, we must generate deterministic programs. Thereto, we require that the producer's transfer relation is a function. During program generation, we keep the operations (statements) of the original program. To enable the verification of the generated program, we also need a transfer function in the consumer analysis. The transfer relation of the property checking analysis must be a function. For the verification of the generated program, an arbitrary transfer function is not sufficient. Due to different exploration orders, the consumer might explore a less abstract property checking state than the producer. To still ensure that the successors explored by the consumer are considered in the producer's analysis, we require a monotonic transfer function. Moreover, we must be able to transfer the property checking analysis' exploration on the producer side to the consumer. We only restructure the original program, i.e., the program locations of the CFA edges are adapted, but never the operation. Of course, the location state is affected by the structure change. In contrast, the remaining parts of the abstract state should not see the different structure of the original and generated program. Thus, we require that locations mainly influence the location state. Especially, in the transfer function, which is the same for the producer and the consumer, a renaming of program locations must be feasible.

Finally, we come back to the need for syntactical paths. The location CPA is a necessary requisite. However, we also must ensure that every explored abstract state considers only a single program location. To achieve this requirement, we need to take special care on the operators of the property checking analysis. The merge operator and the termination check of the property checking analysis are not relevant. The producer uses a composition of an enabler and a property checking analysis for which those operators are redefined. Hence, we add the requirements on the merge and termination check later when defining the requirements on the refined property checking analysis, the composition of enabler and property checking analysis. Furthermore, the consumer analysis uses his own, possibly different merge and termination check operators, which will consider this requirement. Thus, we only require that the transfer function and the precision adjustment never compute abstract states with location state $\top_{\mathbb{L}}$ or $\bot_{\mathbb{L}}$. For the precision adjustment, we explicitly forbid this behavior. For the transfer function, only the composite CPA $\mathbb{L}\times\mathbb{C}$ could introduce such behavior if it strengthened the location state. Thus, we exclude strengthening of the location state. Now, we formally summarize the previously discussed requirements on a property checking analysis.

Definition 5.3. Let $\mathbb{C}_1^{\mathcal{A}}$ be an enhancement of a CPA $\mathbb{C}_1$ with a control state unaware property automaton $\mathcal{A}$. Then, $\mathbb{C}_1^{\mathcal{A}}$ *is a property checking analysis* if it adheres to the following requirements:

- CPA $\mathbb{C}_1$ is the location CPA $\mathbb{L}$ or a composite CPA $\mathbb{L}\times\mathbb{C}$, combining location CPA $\mathbb{L}$ and another CPA $\mathbb{C}$.
- The transfer relation $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ is a monotonic function.[1]

[1] For the domain we use the product ordering of $\sqsubseteq$ and the flat order on $\mathcal{G}$ and for the codomain $\sqsubseteq$.

- If CPA $\mathbb{C}_1$ is the location CPA, $\mathbb{C}_1 = \mathbb{L}$, the transfer relation ignores the location state to compute the abstract automaton state, $\forall (l_p, op, l_s), (l'_p, op, l'_s) \in \mathcal{G} : ((l_p, q), (l_p, op, l_s), (l_s, q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \implies ((l'_p, q), (l'_p, op, l'_s), (l'_s, q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$.
- If CPA $\mathbb{C}_1$ is a composite CPA $\mathbb{L} \times \mathbb{C}$, then the transfer relation does not strengthen the location state, $\forall ((l_p, e), g, (l_s, e')) \in \rightsquigarrow_{\mathbb{C}_1} \implies (l_p, g, l_s) \in \rightsquigarrow_{\mathbb{L}}$, and ignores the locations of the control flow edge to compute the non-locations elements of a successor, $\forall (l_p, op, l_s), (l'_p, op, l'_s) \in \mathcal{G} : (((l_p, e), q), (l_p, op, l_s), ((l_s, e'), q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \implies (((l'_p, e), q), (l'_p, op, l'_s), ((l'_s, e'), q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$.
- the precision adjustment does not widen the location state, $\forall e \in E_{\mathbb{C}_1^{\mathcal{A}}}, \pi \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}, S \subseteq E_{\mathbb{C}_1^{\mathcal{A}}} : \mathsf{prec}(e, \pi, S) = (e', \pi') \implies \mathrm{acs}(e) = \mathrm{acs}(e')$.

Many of the above restrictions exclude theoretically possible configurations, which are unlikely in practice. For example, a most precise enhancement of the location CPA $\mathbb{L}$ with an arbitrary control state unaware property automaton always meets the above requirements. Furthermore, we think that in case that the property checking analysis enhances a composite CPA $\mathbb{L} \times \mathbb{C}$, the major restrictions in practice are the following. The transfer function $\rightsquigarrow_{\mathbb{C}}$ is monotonic and CPA $\mathbb{C}$ must be control state independent. Given such a CPA $\mathbb{C}$, one can simply compose $\mathbb{L}$ and $\mathbb{C}$ using the product transfer relation[2] plus any sound merge and termination check. To get a property checking analysis out of the composition, one only needs to apply the most precise enhancement. We obtained the property checking analysis for our example, the enhanced sign dataflow analysis, similarly. However, we relaxed the merge operator s.t. it merges states with same locations. Generally, a less property precise transfer function, which remains monotonic and still ignores the location state to compute the non-location states, and a relaxed merge operator can be used. The resulting CPA is still a property checking analysis.

In our Programs from Proofs approach, we assume that for the producer the property checking analysis alone is not sufficient to prove program safety. Remember that our sign dataflow analysis failed to prove safety of program `SubMinSumDiv` w.r.t. property `nonneg` (see Fig. 2.5). To still prove program safety, the producer extends the property checking analysis with an enabler analysis. Next, we describe what the producer must take into account when he extends the property checking analysis.

5.2.2 Refined Property Checking Analysis

In principle, a refined property checking analysis is an extension of a property checking analysis with an appropriate enabler analysis. Remember that the sole purpose of the enabler analysis is to make a verification with the property checking analysis feasible. The enabler analysis only excludes infeasible paths or separates paths whenever needed. However, it never interferes with the property checking analysis. In principle, the property checking analysis is unaware of the enabler analysis.

In theory, almost any CPA can be used as *enabler analysis*. For our Programs from Proofs approach, it is more important that we appropriately extend the property checking analysis with the enabler analysis. Nevertheless, we have one requirement w.r.t. the enabler analysis, which we need to generate deterministic program. Like in the property checking analysis, its transfer relation must be a function. This leads us to the following definition of an enabler analysis.

[2]We think the product transfer relation is a rather natural combination if one analysis considers the control state and the other the data state.

Definition 5.4. A *CPA* $\mathbb{C}_2$ *is an enabler analysis* if its transfer relation is a function.

To workout in practice, the enabler analysis should be more powerful than the property checking analysis. At least, it should be able to separate some concrete states that cannot be separated by the abstract states of the property checking analysis. For our example, we use the predicate analysis $\mathbb{P}$ to extend the sign dataflow analysis. Similar to the predicate analysis in our example, we often use enabler analyses for which we can set the precision. As discussed later, we apply counterexample guided abstraction refinement (CEGAR) [CGJ+00] to automatically detect a precision for those enabler analyses. Preferably, the computed precision should be coarse but precise enough for successful verification.

So far, we know all restrictions on the two components of a refined property checking analysis. Now, we look at their composition. Since the CPA algorithm must be started with an enhanced CPA, technically we cannot simply compose the enabler analysis with the property checking analysis to get the refined property checking analysis. Technically, we first compose the enabler analysis $\mathbb{C}_2$ with the CPA $\mathbb{C}_1$ that is enhanced in the property checking analysis and then enhance the composite CPA $\mathbb{C}_1 \times \mathbb{C}_2$. However, this is more or less a technical issue. For the abstract domain, there is only a structural difference. In case of the transfer relation, only in the standard composition of the enabler and property checking analysis, the enabler analysis could strengthen the automaton state. Since the enabler analysis must not influence the property checking analysis, this kind of strengthening would be forbidden anyway. Again, solely in the standard composition we may consider the automaton state to strengthen the enabler state. In theory, the standard composition is indeed more flexible. However, we believe that in practice the enabler state is strengthened based on the set of concrete states represented by the other component's abstract state. The automaton state does not influence the meaning of an abstract state. Practically, there should not be a difference w.r.t. strengthening. Next, we need to ensure that the property checking analysis' transfer relation determines the property checking state including the automaton state. The transfer relation of an enhanced CPA $\mathbb{C}_1$ uses the transfer relation of the CPA $\mathbb{C}_1$ to determine the component $e_{\mathbb{C}_1}$ of the successor. Thus, it only remains to be shown that in the technically motivated composition the transfer relation could generate the same successor automaton states as the standard composition. We know that every property automaton is deterministic. Hence, if we consider the composition of $\mathbb{C}_2$ and $\mathbb{C}_1$ before the enhancement, the computed automaton state in the most precise enhancement may only be more precise. We conclude that an enhancement exists that replaces the automaton state of the most precise enhancement by a more abstract one depending on the abstract state of component $\mathbb{C}_1$ (the abstract state considered to determine the automaton state in the property checking analysis). Practically, the technical motivated composition does not impose any restrictions on the transfer relation. Now, let us consider the precision adjustment. When the property checking analysis selects the precision adjustment of the most precise enhancement, the precision adjustment is the same in both composition orders. In all other cases, first composing $\mathbb{C}_2$ and $\mathbb{C}_1$ is more general. The set of possible precision adjustments subsumes the set of configurable precision adjustments in the composition of the enabler analysis $\mathbb{C}_2$ and the property checking analysis $\mathbb{C}_1^{\mathcal{A}}$. Since the merge operator can be freely configured in both cases, there is no difference for the merge operator. The termination check in the standard composition can be configured almost arbitrarily, of course we need some restrictions that the behavior of the property automaton is respected. In contrast, we force the termination check in the enhancement of the composite analysis $\mathbb{C}_2 \times \mathbb{C}_1$ to always consider all states that fit to the automaton state. In a successful verification, the

analysis only explores abstract states whose automaton states are states in the property automaton. We do not see a practical need that the termination check operator considers the automaton state to decide coverage.

In the last paragraph, we recognized that the non-standard composition of the enabler and property checking analysis is no restriction. We continue to present what must be taken into account when the enabler analysis and the property checking analysis are combined. The underlying idea of our Programs from Proofs approach is that the property checking analysis alone can prove safety of the generated program. For this, we require that already in the producer analysis the property checking analysis alone checks safety. The enabler analysis only separates program paths or excludes infeasible paths. This means that the transfer relation of the refined property checking analysis uses the property checking analysis to determine the second element of the composite CPA $\mathbb{C}_2 \times \mathbb{C}_1$ and the automaton state. Like the transfer relations of enabler and property checking analyses, the transfer relation of a refined property checking analysis must be a function. Furthermore, also the refined property checking analysis must take care that no new program behavior is introduced during program transformation. Like in the property checking analysis, a refined property checking analysis must not change the location state. In a standard composition, the requirement on the property checking analysis would be sufficient. However, in our technically motivated composition of the enabler and property checking analysis we need the restriction on the property checking analysis' precision adjustment to ensure that any property checking analysis can be enhanced. Additionally, we must repeat the requirement for the refined property checking analysis to really guarantee that no new program behavior will be introduced. Furthermore, to eliminate new program behavior in the generated program, the merge operator must not widen the location state. Since during program generation we never change program statements, the consumer verification cannot detect that during producer verification a transfer successor is covered by a number of abstract states. At last, we require that the termination check only returns true in case an abstract state is covered by a single state. The following definition formally reconsiders the previously discussed restrictions on a refined property checking analysis.

Definition 5.5. Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a composite CPA $\mathbb{C}_2 \times \mathbb{C}_1$ enhanced with a control state unaware property automaton and $\mathbb{C}_1^{\mathcal{A}}$ be a property checking analysis. Then, CPA $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is a refined property checking analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ if

- the transfer relation $\rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is a function that may only strengthen the abstract state of the enabler analysis and determines the automaton state with the property checking analysis only: $\forall((e_2, e_1), q), ((e_2', e_1'), q') \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}, g \in \mathcal{G} : (((e_2, e_1), q), g, ((e_2', e_1'), q')) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} \implies ((e_1, q), g, (e_1', q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$,
- the precision adjustment never changes the location state, $\forall((e_2, e_1), q) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}, \pi \in \Pi, S \subseteq E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} : \mathsf{prec}(((e_2, e_1), q), \pi, S) = (((e_2', e_1'), q'), \pi') \implies \mathrm{acs}(e_1) = \mathrm{acs}(e_1')$,
- the merge operator keeps the location state of its second parameter, $\forall((e_2, e_1), q), ((e_2', e_1'), q') \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} : \mathsf{merge}(((e_2, e_1), q), ((e_2', e_1'), q')) = ((e_2^m, e_1^m), q^m) \implies \mathrm{acs}(e_1^m) = \mathrm{acs}(e_1')$, and
- if the termination check returns true, then an abstract state has already been explored that covers the checked abstract state, $\forall e \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} : \mathsf{stop}_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}(e, S) \implies \exists e' \in S : e \sqsubseteq_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} e'$.

Note that the above restrictions always allow to refine any property checking. For the transfer relation and the precision adjustment, one could always use the product combination. We discussed earlier that this is feasible despite the non-standard composition in which we first compose CPAs $\mathbb{C}_2$ and $\mathbb{C}_1$ and then enhance the composition. Hence, the restrictions on the transfer relation and the precision adjustment only limit the power of the composition. However, the limitation is needed to enable the successful consumer verification on the transformed program. Furthermore, the requirements on the merge operator are standard for all flow-sensitive analyses. Also, termination check operators that check that an abstract state is covered by a more abstract state are often used in practice. Possibly, the producer must explore more abstract states than he originally intended. Nevertheless, for our Programs from Proofs approach it is always acceptable to impose a higher workload on the producer. The only impact of the requirements on the merge and the termination check operator is that the producer probably must configure a more precise analysis.

For our example, we extend the sign dataflow analysis, which alone failed to prove program `SubMinSumDiv` w.r.t. property `nonneg`, with a predicate analysis $\mathbb{P}_{\mathcal{P}}$.[3] To build the refined property checking analysis $(\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S}))^{\mathcal{A}}$ for our example, we use the product combination of the predicate and the sign dataflow analysis for the transfer relation and the precision adjustment. The merge operator joins abstract states whenever the predicate and the location state are the same, $\mathsf{merge}_{(\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S}))^{\mathcal{A}}}(((p,(l,s)),q),((p',(l',s')),q')) = ((p,(l,s \sqcup_{\mathbb{S}} s')),q \sqcup_{\mathcal{Q}} q')$ if $p = p'$ and $l = l'$. In all other cases, the merge operator does not combine information, i.e., $\mathsf{merge}_{(\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S}))^{\mathcal{A}}}(((p,(l,s)),q),((p',(l',s')),q')) = ((p',(l',s')),q')$. Furthermore, the termination check returns true when an abstract state is covered by a more abstract state.

Before we come to the execution of a refined property checking analysis, like e.g. our example $(\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S}))^{\mathcal{A}}$, we want to introduce the concept of *location updated property checking extraction*. We often employ this concept to convert an abstract state of the producer into the context of the consumer. Since the consumer does not know about the enabler state, we remove this information. Note that the enabler state can safely be removed because its information never affects property checking directly. Additionally, we need to consider that due to the program transformation the location changed. A refined property checking analysis can be started with an arbitrary program. It is unaware of the program transformation. Thus, we must explicitly provide the location state in the new program. Given a location state l and an abstract refined property checking analysis state $((e_2,e_1),q)$, the location updated property checking extraction takes the property checking analysis state (e_1,q) included in the refined property checking analysis state and replaces the location information in abstract state e_1 with l.

Definition 5.6. Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a refined property checking analysis. For any location state $l' \in \mathbb{L}$ and any abstract state $e = ((e_2,e_1),q) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ we define the *location updated property checking extraction* $e[l']$ to be $e[l'] := (l',q)$ if $\mathbb{C}_1 = \mathbb{L}$ and $e[l'] := ((l',e_{\mathbb{C}}),q)$ if $\mathbb{C}_1 = \mathbb{L} \times \mathbb{C}$ and $e_1 = (\cdot, e_{\mathbb{C}})$.

With the last definition, we finished the discussion of a refined property checking analysis. Next, we continue with the execution (result) of a refined property checking analysis.

[3] Note that in case of our example we could also prove the property with sign model checking. However, the state space explored by sign model checking is larger (contains more abstract states) for program `SubMinSumDiv` than using the refined property checking analysis, our extension of the sign dataflow analysis, presented next.

5.2.3 Execution of Refined Property Checking Analyses

So far, we explained which enhanced CPAs can be used as refined property checking analyses in our Programs from Proofs approach. To start the verification, we need two additional inputs: an initial abstract state and an initial precision. We already mentioned in the overview, that no restrictions on the initial precision exist. However, the refined property checking analysis must be started in an initial abstract state appropriate for the Programs from Proofs approach.

As always, the verification must start in the initial automaton state to inspect program safety. Moreover, we already mentioned before that if the analysis explores an abstract state with location state $\top_{\mathbb{L}}$, new program behavior will be introduced. Since we want to inspect at least some program behavior, we must not use $\bot_{\mathbb{L}}$, which represents no concrete state. The only location states that remain are concrete locations. Thus, we require that the location state in the initial abstract state is described by a concrete location. Furthermore, remember that the consumer derives his initial abstract state from the producer's initial abstract state. The consumer must be able to express the restrictions of the producer's initial abstract state with a property checking state only. Often, enabler states can be more precise than any property checking state. For some enabler states, no equivalent property checking state exists. Hence, we assume that the property checking analysis state alone determines which program behavior is inspected. The enabler state must not restrict the analyzed program behavior. The top state $\top_{\mathbb{C}_2}$, which represents all concrete states, never restricts the property checking state and is often a sufficiently good choice. All abstract states of a refined property checking analysis that meet the previously discussed requirements are appropriate initial abstract states. We say that these abstract states are *compatible, initial abstract states.*

Definition 5.7. Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a refined property checking analysis considering control state unaware property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and $e_0 = ((e_{\mathbb{C}_2}, e_{\mathbb{C}_1}), q) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ be an abstract state. Abstract state e_0 *is a compatible, initial abstract state* if

- it uses the initial automaton state, $q = q_0$,
- it considers exactly one program location, $\text{acs}(e_{\mathbb{C}_1}) \in \mathcal{L}$, and
- the enabler state does not restrict the property checking state, $[\![e_{\mathbb{C}_2}]\!]_{\mathbb{C}_2} \supseteq [\![e_{\mathbb{C}_1}]\!]_{\mathbb{C}_1}$.

Until now, we have discussed the configuration of all input parameters in the producer verification. Next, we take a look at the verification of our example program `SubMinSumDiv` w.r.t. property `nonneg`. As we already observed, the verification with the simple sign DFA, a property checking analysis, failed (see e.g. Fig. 2.5). To show safety of program `SubMinSumDiv`, we extended the sign DFA with a predicate CPA, the enabler analysis in our example. We detected that the set of predicates $\mathcal{P} = \{y \geq 0\}$ is sufficient to prove program safety. Later in this chapter, we explain how to automatically determine an appropriate precision of the enabler analysis. Since we want to show program safety for all program executions starting in the initial program location, we use $e_0 = ((\top_{\mathbb{P}}, (l_0, \top_{\mathbb{S}})), q_0)$ as compatible initial abstract state.

Figure 5.3 shows the ARG generated by the producer for our example program `SubMinSumDiv` and the previously discussed configuration. Note that in the ARG we represent $\top_{\mathbb{P}}$ by true. We observe that all automaton states in the ARG are q_0 or q_1. The producer succeeds to show safety of program `SubMinSumDiv` w.r.t. property `nonneg`. Furthermore, we observe that in the ARG all three cases, the computation of the minimum, the sum,

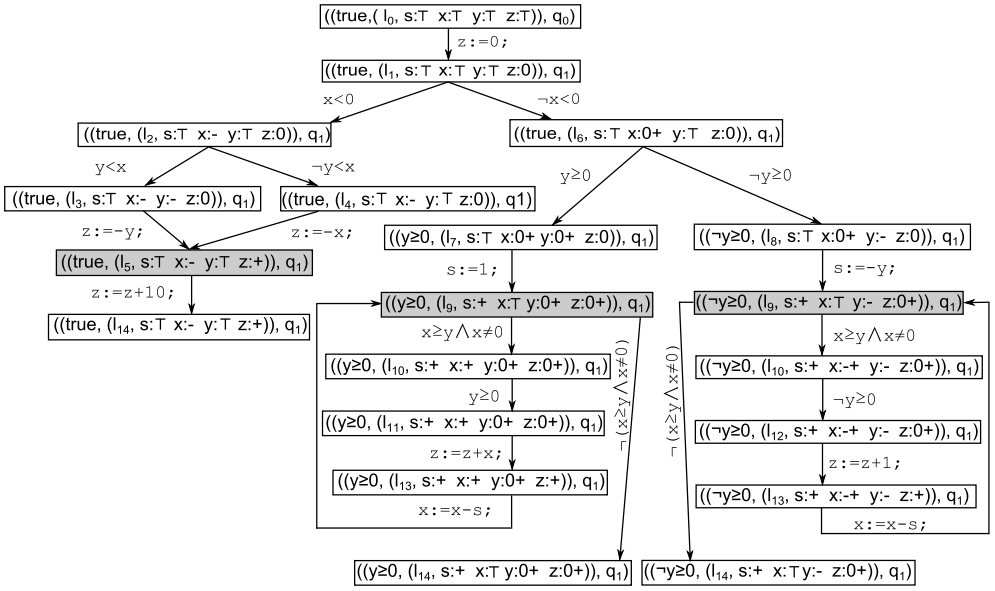

Figure 5.3: Abstract reachability graph constructed during successful verification of program `SubMinSumDiv` with a refined property checking analysis. The refined property checking analysis used is the predicated sign DFA enhanced with property automaton `nonneg`. The considered set of predicates is $\mathcal{P} = \{y \geq 0\}$. The verification started with initial abstract state $e_0 = ((\top_{\mathbb{P}}, (l_0, \top_{\mathbb{S}})), q_0)$ and initial precision $\pi_0 = (\mathcal{P}, \pi_{\text{static}})$ (the only precision available).

and the division, are separated. This separation, especially splitting the computation of the sum and the division, is the key to prove safety. To uncouple the computation of the sum and the division, the while loop is duplicated. Additionally, in each while loop the infeasible branch belonging to the other computation is excluded. Due to these infeasible paths, the property checking analysis alone failed to prove safety. Finally, we want to note that a sign model checking, which never combines abstract states, would also be possible in this special case. Compared to the refined property checking analysis, the sign model checking requires two additional loop unrollings, one for the sum and one for the division computation.

After a successful verification, like the one of our example program `SubMinSumDiv` considered in the previous paragraph, the standard guarantee on the generated ARG is well-formedness. For our Programs from Proofs approach, a well-formed ARG is not sufficient. We require stronger guarantees, which include well-formedness plus some additional requirements. To ensure that the producer generates a behaviorally equivalent program, we also require that all ARG nodes consider concrete locations. Additionally, we also need a sound and deterministic ARG (see Definition 2.12). For the generation of deterministic programs, which we need to translate the generated program into a deterministic programming language like C, the ARG must be deterministic. Moreover, our program generation only restructures the original program, e.g., it unrolls loops, splits paths, moves statements into branches, or removes syntactical but infeasible paths. Since the statements are never modified, neither program execution nor consumer verification

can detect that the producer verification assumed that some parts of a transfer successor are covered by ARG node n while others are covered by a different ARG node n'. Program execution and consumer verification would always assume that all successors allowed by the respective statement are valid. Thus, we need the ARG to meet this assumption. Each transfer successor is completely covered by a single ARG successor, the typical soundness property of an ARG. We unite all these requirements under a common term and call a well-formed ARG that also fulfills the extra requirements *strongly well-formed*.

Definition 5.8. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ and $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ be an abstract state. $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is *strongly well-formed* for e_0 if $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is well-formed for e_0, is sound, deterministic and all nodes consider concrete locations, $\forall((e_2, e_1), q) \in N : \text{acs}(e_1) \in \mathcal{L}$.

It remains to be shown that the producer's verification indeed generates strongly well-formed ARGs during a successful verification. Since the refined property checking analysis is an enhanced CPA, Proposition 2.8 guarantees us the well-formedness property. Furthermore, the transfer relations of refined property checking analyses are functions. Moreover, the termination check operators of refined property checking analyses only return true when the input abstract state e is covered by a single abstract state in the input set S of abstract states. All preconditions for Proposition 2.9 are fulfilled. Hence, Proposition 2.9 ensures soundness and determinism. It remains to be shown that all ARG nodes only consider concrete locations. The following lemma gives us this remaining property for all ARGs constructed during a successful producer verification, which of course starts with a compatible, initial abstract state.

Lemma 5.1. *If Algorithm 2 started with refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$*, compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*, any initial precision* $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*, and program* P *returns* $(true, \cdot, (N, G_{\text{ARG}}, root, N_{\text{cov}}))$*, then* $\forall((\cdot, e_1), \cdot) \in N : \text{acs}(e_1) \in \mathcal{L}$.

Proof. See Appendix pp. 270 f. □

With the previous lemma, we proved that our producer verification also ensures the last property of a strongly well-formed ARG. Now, we simply combine the partial results and state that in the Programs from Proofs approach the ARGs generated during successful producer verification are strongly well-formed.

Proposition 5.2. *If the CPA algorithm (Algorithm 2) started with refined property checking analysis* $(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}$*, compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*, arbitrary initial precision* $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*, and program* P *returns* $(true, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$*, then* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ *is an ARG for* P *and* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *which is strongly well-formed for* e_0.

Proof. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$. From Proposition 2.8, we know that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is an ARG for P and $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ which is well-formed for e_0. From Proposition 2.9, we conclude that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is sound and deterministic. From the previous lemma, we infer that $\forall((e_2, e_1), q) \in N : \text{acs}(e_1) \in \mathcal{L}$. By definition of strongly well-formedness, we conclude that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed for e_0. □

Up to now, we presented all restrictions that we need to ensure that the verification of a correctly transformed program does not fail on the consumer side. For many configurations and programs these restrictions are sufficient. In some cases, we explicitly need to ensure that the consumer verification terminates. The next subsection discusses when termination needs to be considered and which further restriction are needed in this case.

5.2.4 Incorporation of Termination of the Consumer Verification

Termination of the consumer analysis is not always a separate issue. Depending on the structure of the (generated) program and the structure of the refined property checking analysis, the previously discussed restrictions may be enough. In the following, we list the cases for which we know and proved that the restrictions discussed so far are sufficient.

1. The generated program is loop-free.
2. The join-semilattice considered in the abstract domain of the property checking analysis has a finite height.
3. The refined property checking analysis applies model checking without adjusting the property checking analysis part, i.e., the precision adjustment changes at most the enabler state, the analysis never combines abstract states, $\mathsf{merge}_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}(e, e') := e'$, and the termination check always detects coverage by same states.

A refined property checking analysis never merges abstract states with different program locations. Thus, the generated program is always loop-free when the original program is loop-free. Furthermore, many refined property checking analyses already fulfill one of the above requirements and do not need to care for termination. Every refined property checking analysis that extends a property checking analysis that is an enhancement of the location CPA or of a composition of a location CPA and a CPA $\mathbb{C}$ with finite join-semilattice height guarantees the second requirement. Since the lattice considered in the sign CPA $\mathbb{S}$ has a finite height, for our example analysis, the predicated sign dataflow analysis $(\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S}))^{\mathcal{A}}$, we also do not need to take termination into account.

While the three cases from above already cover a large class of refined property checking analyses, they do not fully cover one class of refined property checking analyses that we frequently use in practice. In practice, our refined property checking analyses often use the product transfer function of the enabler and property checking analysis, never adjust precisions[4], merge states with same locations and enabler states only, and stop exploration of an abstract state when it is covered by a more abstract state with same enabler and location state. For this class, we also want to guarantee termination even if the join-semilattice height is infinite. However, we do not show termination for refined property checking analyses of this class only. In contrast, we provide some additional requirements on the refined property checking analysis to guarantee termination. The refined property checking analyses just described automatically fulfill the additional requirements.

The termination proof for the last case utilizes that we know that the producer analysis terminated. Hence, if the consumer analysis can mimic the property checking analysis part of the producer's analysis, termination of the consumer analysis should follow from termination of the producer analysis. To show that the consumer analysis mimics the property checking analysis part of the producer's analysis, we must relate the abstract states considered in the consumer analysis to those states considered in the producer analysis. As we already know, the program locations of the original and generated program may differ. Additionally, program locations of the original program can be duplicated. We cannot just relate program locations. However, since the consumer analysis performs

[4] Assuming that we do not perform lazy refinement, practically, our enabler analyses compute the strongest transfer successor and statically relax them in the precision adjustment. Thus, the relaxation could be easily moved into the transfer relation and no changes in precision adjustment are necessary. Since we cannot distinguish between these two analyses externally, we may think of the implemented analysis as if it was the second one.

a dataflow variant of the property checking analysis, which only computes one abstract state per location, it is sufficient to properly relate the producer's abstract states to the program locations of the consumer's program. Since we do not know when an abstract state should be related to location l and when to a different location l', a producer's abstract state must be mapped to a single location. In the following, we consider an *equivalence relation* $\sim$: $E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} \times E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ on the abstract states to describe the relation between the producer's abstract states and the consumer's program locations. The idea is that each location in the generated CFA is associated with exactly one equivalence class. Furthermore, the equivalence relation itself describes which parts of the producer's abstract states are responsible for the generated locations. Note that not all abstract states of an abstract domain are explored. Non-explored states do not need to be related to a location. For some equivalence classes no associated location might exist. However, if a location l is associated with an equivalence class, then all abstract states in that equivalence class will be related to the location l. Furthermore, due to the requirements on a refined property checking analysis, especially different location states must not be intermixed. Only abstract states with the same location state may be equivalent. For the frequently used class of refined property checking analyses from above, the equivalence relation relates all abstract states with the same enabler and location state. In case of model checking, the equivalence relation becomes the identity relation.

The equivalence relation alone is not yet sufficient. The problem is that the producer analysis may show behavior that the consumer analysis cannot reproduce. For example, the producer combines abstract states of different equivalence classes. This means that the consumer analysis, a dataflow analysis, might combine states considering different locations, which no dataflow analysis ever does. To be able to mimic the producer's behavior, we need to exclude that the producer can do something the consumer analysis cannot do. Therefore, we require that the producer analysis must treat the abstract states of a single equivalence class like the consumer analysis treats locations. For states with same locations, the transfer function of the property checking analysis cannot vary the location information computed along one edge. Additionally, the property checking analysis cannot only use the same location information to sometimes exclude a successor for an edge and sometimes not. This behavior is only possible when all component states of the abstract state are considered. Thus, for a given CFA edge the successors computed by the refined property checking analysis for abstract states of the same equivalence class always belong to the same equivalence class. Moreover, for a given CFA edge the refined property checking analysis either provides no successor for all states within the same equivalence class or successors are only excluded due to the property checking analysis state. Since the consumer analysis never adjusts precisions, the precision adjustment of the refined property checking analysis must keep the property checking state. It may only change the enabler state. Moreover, the resulting abstract state must belong to the same equivalence class as the abstract state that is adjusted. Furthermore, dataflow analyses join abstract states for same locations. The refined property checking must join abstract states of same equivalence classes. Finally, a dataflow analysis stops exploration iff an abstract state is covered by a more abstract state with the same location. Hence, the refined property checking analysis must stop exploration iff an abstract state is covered by a more abstract state of the same equivalence class.

We start the consumer analysis with an initial abstract state that mimics the producer's initial abstract state. Furthermore, the requirements from above ensure that the property checking analysis part of the producer analysis behaves like a dataflow analysis on the generated program (of course the location names do not match). Additionally, the

requirements on the transfer function of the property checking analysis and the refined property checking analysis guarantee us that we can transfer the property checking related successor computation to the consumer analysis. Thus, the consumer analysis should be able to mimic the producer analysis w.r.t. property checking analysis.

To guarantee that a refined property checking analysis ensures termination of the consumer analysis, we only need to find an equivalence relation on the abstract states s.t. the refined property checking analysis in combination with the equivalence relation fulfills the requirements discussed above. In case such an equivalence relation exists for a refined property checking analysis, we say that the analysis is *equivalence relation consistent*. The following definition formally states equivalence relation consistency.

Definition 5.9 (Equivalence Relation Consistency)**.** A refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is *equivalence relation consistent* if there exists an equivalence relation $\sim$: $E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} \times E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ on the abstract states s.t.

- if two abstract states of the same equivalence class both have a successor for CFA edge g, then the successors belong to the same equivalence class, $\forall e, e', e'', e''' \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, g \in \mathcal{G} : e \sim e' \wedge (e, g, e'') \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} \wedge (e', g, e''') \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} \implies e'' \sim e'''$,
- if for some abstract state a transfer successor for CFA edge g exists, then for all other abstract states of the same equivalence class either the property checking analysis excludes any transfer successor or a transfer successor exists for that CFA edge, $\forall e, ((e_2, e_1), q) \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, g \in \mathcal{G} : e \sim ((e_2, e_1), q) \wedge \exists (e, g, \cdot) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} \wedge \exists((e_1, q), g, \cdot) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \implies \exists(((e_2, e_1), q), g, \cdot) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$,
- the precision adjustment neither changes the property checking state, the automaton state, nor the equivalence class, $\forall(((e_2, e_1), q), (((e_2', e_1'), q') \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, \pi, \pi' \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : \mathsf{prec}((((e_2, e_1), q), \pi, S) = ((((e_2', e_1'), q'), \pi') \implies (e_1, q) = (e_1', q') \wedge (((e_2, e_1), q) \sim (((e_2', e_1'), q')$,
- the merge operator joins states iff they are of the same equivalence class and returns the same equivalence class as the second parameter, $\forall e', e'' \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} :$ $\mathsf{merge}(e', e'') \sim e''$ and $e' \sim e''$ implies $\mathsf{merge}(e', e'') = e' \sqcup e''$ and $e' \not\sim e''$ implies $\mathsf{merge}(e', e'') = e''$,
- the termination check operator returns true iff an abstract states is covered by an element of the same equivalence class, $\forall e \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : \mathsf{stop}(e, S) = \exists e' \in S : e \sim e' \wedge e \sqsubseteq e'$.

Reconsider the equivalence relation that relates all abstract states with the same enabler and location state. Using this equivalence relation, we easily see that a refined property checking analysis that uses the product transfer function of the enabler and property checking analysis, never adjusts precisions, merges states with same locations and enabler states only, and stops exploration of an abstract state when it is covered by a more abstract state with the same enabler and location state is equivalence relation consistent.

From the producer's point of view, all refined property checking analyses that are typically relevant in practice guarantee termination of the consumer's analysis. Of course, we need to prove this claim when we look at the consumer analysis. In the remaining cases, the producer could always check after program generation whether the consumer analysis terminates. He solely needs to start the consumer analysis and to abort it whenever it takes longer than his own analysis. Alternatively, he can use his ARG and transform it

into a proof for the generated program. We discuss this alternative in the next chapter. In the following, we continue with the last open question of the producer analysis: how to automatically get a proper abstraction level for the enabler analysis?

5.2.5 Determination of the Enabler Analysis' Abstraction Level

Although it is not a mandatory requirement, we typically use enabler analyses $\mathbb{C}_2$ with an adjustable abstraction level. For example, the set of predicates $\mathcal{P}$ determines the abstraction level of the predicate CPA $\mathbb{P}_\mathcal{P}$ introduced in Section 2.3.1. Enabler analyses with adjustable abstraction level provide the advantage that the producer can start with a very coarse abstraction level. Then, he successively tries out finer abstraction levels until the property can be proven. The producer analysis does not need to become unnecessarily precise and complex. Thus, the constructed ARG becomes only as precise as necessary to prove the desired property with the property checking analysis. However, practically the producer only profits from an adjustable abstraction when a suitable (and coarse) abstraction level can be determined automatically.

Currently, we support two approaches to automatically determine a suitable abstraction level for the enabler analysis: the classical counterexample guided abstraction refinement (CEGAR) scheme [CGJ+00], in which a uniform abstraction model, the ARG, is computed, and lazy refinement [HJMS02], which allows to change the abstraction level during the constructing of the ARG. In the following, we briefly describe how these approaches work.

Uniform Abstraction

Counterexample guided abstraction refinement (CEGAR) [CGJ+00] was proposed by Clarke et al. in 2000 in the context of branching-time-symbolic model checking. The idea behind the CEGAR approach is as follows. If the verification fails due to a too coarse abstraction, a finer abstraction will be computed based on a spurious counterexample and the verification will be restarted with the finer abstraction. Note that in contrast to the predicate CPA from Section 2.3.1, in practice we use the (initial) precision to fix the abstraction. Next, we briefly outline how the classical CEGAR approach automatically determines a suitable abstraction level.

1. Start with the coarsest abstraction level, e.g., for our predicate analysis this would be the empty set of predicates, the representation of the boolean formula true.
2. Run the producer analysis with the current abstraction level.
3. If the producer analysis succeeds, i.e., the current abstraction level was sufficient to prove the program correct, you are done.
4. If the producer analysis fails, try to refine the abstraction, i.e., make it more precise.
   (a) Extract a counterexample and check if it is spurious.
   (b) In case the counterexample is real, the program violates the correctness property, return program unsafe.
   (c) In case the counterexample is spurious, make the abstraction level more precise s.t. the spurious counterexample is excluded. Like for our predicate abstract domain, for some abstract domains an automatic refinement of the abstraction based on interpolation [HJMM04, BL13] can be used.

5. Replace the current abstraction level with the refined abstraction level and go to step 2.

We continue with the second approach which uses a non-uniform abstraction.

Non-Uniform Abstraction

In 2002, Henzinger et al. [HJMS02] introduced lazy refinement for predicate abstraction. Later, this principle is also applied on other abstractions, e.g., in combination with explicit model checking [BL13]. The basic idea of lazy refinement is that in case of a spurious error, those parts of the abstract model, the ARG, are kept that do not relate to the error and only the abstraction for those parts that influence the error is adapted. In the following, we briefly sketch the lazy refinement process.

1. Register the pair of initial abstract state and coarsest abstraction level for exploration.

2. Proceed to explore all states registered for exploration with the abstraction level stored with them until all states registered for exploration are explored or an unsafe state is reached.

3. If the producer analysis succeeds, i.e., all states are explored, you are done.

4. If the producer analysis fails, try to refine the abstraction, i.e., make it more precise.

   (a) Extract a counterexample and check if it is spurious.

   (b) In case the counterexample is real, the program violates the correctness property, return program unsafe.

   (c) In case the counterexample is spurious, make the abstraction level more precise s.t. the spurious counterexample is excluded. Like for our predicate abstract domain, for some abstract domains an automatic refinement of the abstraction based on interpolation [HJMM04, BL13] can be used.

5. Determine the pivot node, the first node from the root node that must become more precise, remove the pivot node and all its descendants from the computed abstraction, the ARG, and unregister them from exploration, register the pivot node's parents with the refined abstraction level for exploration and go to step 2.

Looking at the sketched process, we observe that the ARG produced in such a process is no longer the result of a CPA algorithm execution. However, we are confident that the behavior of the lazy refinement process can be imitated with a proper definition of the producer analysis' precision adjustment operator s.t. it also adheres to the requirements on the precision adjustment operator in a refined property checking analysis. As we will see, the details of the producer's precision adjustment operator are irrelevant. Although in practice it is realized differently, we can still assume that the producer runs the CPA algorithm with a refined property checking analysis.

Next, we want to discuss how we realized the abstraction level refinement in practice.

Our Realization of the Abstraction Level Refinement

The software analysis tool CPAchecker is already capable to execute the producer analysis as long as the precision of the enabler analysis is given. To get a complete prototype implementation for the producer analysis in the PfP approach, we must realize the abstraction level refinement for the enabler analysis in a refined property checking analysis. Our goal for the realization is to easily get a prototype for our evaluation and not a mature abstraction level refinement. It is acceptable that sometimes the refinement fails.

The analysis tool CPAchecker already provides mature CEGAR solutions for predicate abstraction and the value analysis, which also support lazy refinement. However, they are tailored to model checking, typically investigating the reachability of error locations, and are designed to exclude infeasible paths. We like to build upon these existing solutions. First, we noticed that it would be laborious to extend the detection of spurious counterexamples or the interpolation based abstraction refinement. Thus, we decided not to specifically handle counterexamples that are caused due to a merge of two safe results as it is done for example by Fischer et al. [FJM05]. Nevertheless, often not only the operation but also the data states are responsible that the transfer relation computes an unsafe automaton successor. To make this information explicit to the existing refinement process, we apply a little trick. We temporarily include the failed data state check as an assume edge into the program s.t. this assume edge is an outgoing edge of the program location for which an unsafe state was detected. Then, we extend the ARG. We add a successor to the unsafe state that is reachable via an edge labeled by the temporarily added assume edge. The enabler state and the location state of this successor are computed via the transfer relation, the remaining state parts are copied. Thereafter, the extended ARG is forwarded to the standard CEGAR implementation.

In an automatic refinement process, we need to know the data state check for the assumption. Since it is difficult to infer the check from the check applied by the property checking analysis, we use a special concept of the automaton specification language in CPAchecker, the concept of an assumption specification. Hence, we encode these checks as assumptions on all transitions that lead to an error location[5].

Abstraction refinement was the last aspect of the producer's first task, the analysis. We continue with the subsequent task, the generation of a behaviorally equivalent, but more easily verifiable program.

5.3 Program Generation

In the last section, we explained how the producer proves safety of the original program. In this section, we continue with the next step of the producer. First, we explain how to generate an more easily verifiable program from the verification proof, the strongly well-formed ARG constructed during a successful producer verification. Afterwards, we discuss the properties of the generated program.

Remember that in our Programs from Proofs approach the producer's task is to restructure the original program such that the simpler property checking analysis can prove

[5]Note that in CPAchecker the automaton specification always triggers the analysis check. The automaton specification is a mixture of our property automaton and the part of the enhanced CPA computing the automaton state behavior. Furthermore, CPAchecker does not have the concept of $q_\top$ – we always use the error state instead – and transitions to the error location must be encoded explicitly. Ambiguity can be handled by the order of the specification of transitions.

the restructured program on its own. We know that in a refined property checking analysis the property checking analysis alone checks safety. The only tasks of the enabler analysis are to exclude infeasible paths, which the property checking analysis did not detect and (help the property checking analysis) to separate program paths where necessary to prove safety. Additionally, the efforts of the enabler analysis are directly reflected in the structure of the ARG constructed during verification. Based on these insights, we directly construct the generated program from that ARG.

We reuse a simple idea already applied in previous PfP instances [WSW13, JW15, JW17]. The ARG structure defines the structure of the generated program. ARG nodes become program locations, the root node defines the initial program location, and ARG edges are transformed into program edges. To get CFA edges from the ARG edges, we only need to transform the ARG labels. In contrast to CFA edges, which are only labeled with the operation, ARG edges are labeled with a complete CFA edge. Thus, the CFA edge label can be derived from the ARG label by only keeping the operation information and removing information about predecessor and successor. This leads us to the definition of a program from an ARG, our definition of a generated program, which is equivalent to the definition in previous PfP instances [JW15, JW17].

Definition 5.10 (Program from ARG)**.** Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$ be an abstract reachability graph. The *generated program from ARG* $R^P_{\mathbb{C}^\mathcal{A}}$ is control flow automaton $\text{prog}(R^P_{\mathbb{C}^\mathcal{A}}) = (L', G'_{\text{CFA}}, l'_0)$ with $L' = N$, $G'_{\text{CFA}} = \{(l', \mathit{op}, l'') \mid (l', (\cdot, \mathit{op}, \cdot), l'') \in G_{\text{ARG}}\}$ and $l'_0 = \mathit{root}$.

Figure 5.4 shows the program code and the CFA for the program generated from the ARG shown in Fig. 5.3, the ARG built during the producer verification with the predicated sign dataflow analysis $\mathbb{P}_\mathcal{P} \times (\mathbb{L} \times \mathbb{S})^\mathcal{A}$, the refined property checking analysis, of our example program `SubMinSumDiv` w.r.t. property `nonneg`.

As it was intended, the CFA of the generated program has exactly the same structure as the ARG in Fig. 5.3. Compared with the original program `SubMinSumDiv`, we observe that the uppermost if branch remains the same while in the corresponding else branch the cases $y \geq 0$ and $\neg y \geq 0$ are completely split. The while loop has been doubled and the part of the loop required in each case is moved into the respective branch. Note that the case separation is the reason why the simple sign dataflow analysis is able to prove safety of the generated program w.r.t. property automaton `nonneg`.

Before we continue with the properties of the generated program, we finally discuss how to translate the CFA into a programming language representation. In our example, we manually translated the CFA into a human readable form with proper loops. Since the consumer processes the generated program automatically, we do not need a human readable presentation. In practice, we use C programs and, thus, are able to encode loops and branches simply with goto statements. Only one problem remains. In our generated CFA, we have multiple final locations, namely l_{17}, l_{18}, and l_{19}, which we were not able to represent in the program code. In the program code, we only have a single final location l_{17}. This difference between program model and program language representation could become a problem in the verification of the generated program. Assume different automaton states are associated with each final location. In practice, only one final location exists. The different automaton states would be merged into $q_\top$ and the verification of the generated program fails. We suggest two solutions to deal with this practical problem. First, one could use a special operation *nop* which does not change the program state nor occurs in the program. Then, one introduces a new, single final location and adds a CFA edge from every final location to the new one with operation *nop*. In the property

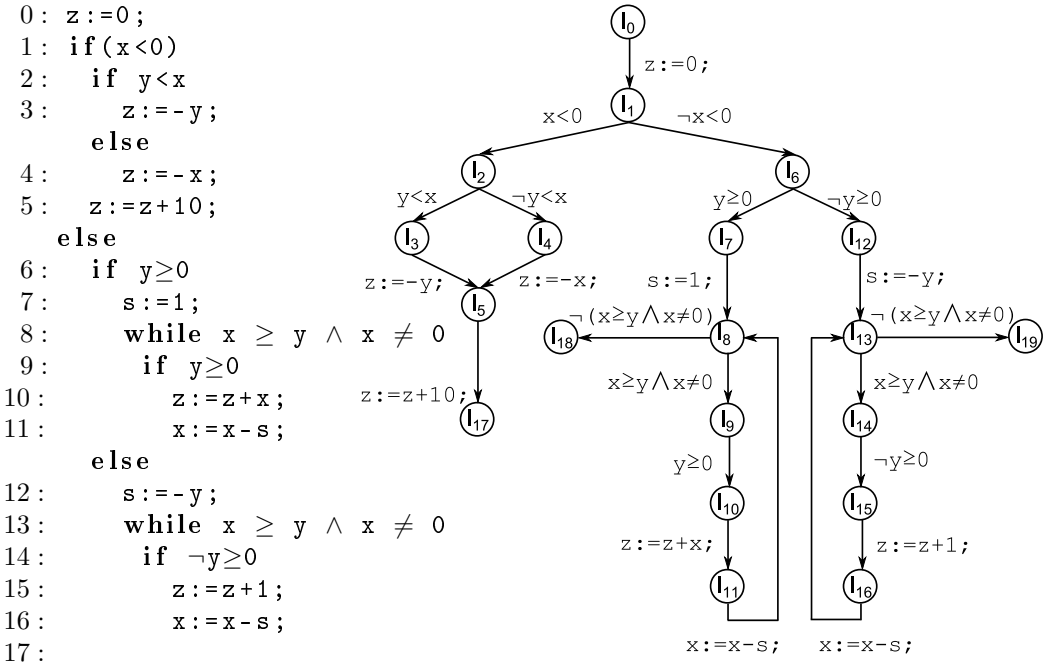

Figure 5.4: Example of a generated program and its CFA. The generated program was constructed from the strongly well-formed ARG in Fig. 5.3 obtained during verification of program `SubMinSumDiv` with predicated dataflow analysis $\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S})$ enhanced with property automaton `nonneg` and initial abstract state $e_0 = ((\top_{\mathbb{P}}, (l_0, \top_{\mathbb{S}})), q_0)$.

automaton, an occurrence of operation *nop* always leads to one specific automaton state that is unequal to the error state. Second, one could use a slightly different consumer analysis that still separates automaton states. This is what we do in practice.

Up to now, we described how to generate the program checked by the consumer. In the following, we discuss important properties of the generated program. These properties facilitate the applicability of our Programs from Proofs approach in scenarios described in the introduction. We start with the most essential property: program and generated program are behaviorally equivalent.

Consider our example, due to the unfolding of the CFA the size of the generated program, e.g., the lines of code, increased. Furthermore, the unfolding of the CFA revealed that some syntactical paths of the program cannot be executed. Namely, we detected that in the else branch of the first if statement depending on the value of y only one path through the loop is possible. Nevertheless, the program operations and the feasible syntactical paths are the same. Thus, the program execution paths remain the same.

Based on this observation, behavioral equivalence is defined by some sort of trace equivalence (see e.g. [vG90]). In general, two systems are trace equivalent when their sets of traces, the sequences of actions that can be observed during system execution, are identical. We continue to discuss what are the traces of a program.

A nearby solution would use the program paths as traces. However, this is a too strict concept of a trace for our setting. On the one hand, locations may be renamed even if the

syntactical structure from the initial location l_0 stays the same. An example for renaming is location l_{12} in Fig. 5.4, which was named l_8 in the original program (see Fig. 2.1). On the other hand, some paths are newly separated syntactically. In our example, the generated program syntactically separates paths with $y \geq 0$ from paths with $\neg y \geq 0$. If paths are separated and do no longer share same locations, one of the location names must become differently, although the path itself does not really change.

For the observable behavior of a program the concrete location names are unimportant. Furthermore, in our Programs from Proofs approach we only consider properties described by control state unaware property automata. That is why we exclude the location information from the traces. Now, a trace is simply a program path without its location information. Syntactically, a trace is a sequence of alternating data states and operations. This idea of a trace leads us to our definition of behavioral equivalence, namely *non-location equivalence*, which we already used in our previous Programs from Proofs framework [JW17]. We define non-location equivalence based on program paths because program paths naturally define the program traces. Two paths p and p' are non-location equivalent if they have the same length and agree on operations and data states. With the concept of trace equivalence in mind, we naturally lift non-location equivalence to sets of program paths.

Definition 5.11 (Non-location Equivalent)**.** Two paths $p := c_0 \overset{g_1}{\to} c_1 \cdots \overset{g_n}{\to} c_n$ and $p' := c'_0 \overset{g'_1}{\to} c'_1 \cdots \overset{g'_m}{\to} c'_m$ in transition system T are *non-location equivalent*, $p =_{\text{nl}} p'$, if $m = n$ and $\forall 0 \leq i \leq n : \text{ds}(c_i) = \text{ds}(c'_i) \wedge (i = 0 \vee \exists op \in Ops : g_i = (\cdot, op, \cdot) \wedge g'_i = (\cdot, op, \cdot))$.

We extend this notation to arbitrary sets S_p, S'_p of paths and write $S_p =_{\text{nl}} S'_p$ if for every path $p \in S_p$ there exists a path $p' \in S'_p$ with $p =_{\text{nl}} p'$ and for every path $p' \in S'_p$ there exists a path $p \in S_p$ with $p' =_{\text{nl}} p$.

Based on the previous definition of equivalence, we next show that the original program and the generated program are behaviorally equivalent. During producer verification, only those program paths are considered that start in an initial state considered by the initial abstract state of the verification. Hence, original and generated program may differ in their behavior in paths not considered by the verification. That is okay because we assume that either the producer and the consumer agreed on the input states, the initial abstract state, of the program or the valid input states, described by the initial abstract state, are part of the producer's offer.

The initial abstract state considered during the producer verification is not suitable to determine those paths in the generated program which are non-location equivalent to the paths the producer explored during his verification. We already demonstrated that program locations in the original and generated program are different. Hence, we need to transform the set of initial states considered by the producer – those states described by the initial abstract state – into a non-location equivalent behavior preserving set. This means, for each initial state we need to keep the data state and must change the control state to the correct program location in the generated program. We know that the generated program directly reflects the paths of the ARG. Moreover, in the (strongly) well-formed ARG constructed by the producer during successful verification, the syntactical paths starting at the root node overapproximate the paths considered during verification. Consequently, the program location representing the root node is the correct choice for the control state.

With these insights, we formulate our behavioral equivalence theorem. This theorem is an adaption of our previous equivalence theorems [JW15, JW17] to the broader class of refined property checking analyses. In the Programs from Proofs approach, the producer

only generates programs after the verification of the original programs with a refined property analysis succeeded. Thus, for program generation a strongly well-formed ARG is used and the root node of that ARG subsumes the initial abstract state used for verification. This leads us to the following theorem, which states that all paths in the original program that start in a concrete state contained in the root node are non-location equivalent with those paths of the generated program that start in the initial program location of the generated program (the ARG's root node) with a data state considered by the root node.

Theorem 5.3 (Behavioral Equivalence)**.** *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *which is strongly well-formed for* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*. Then,* $paths_P([\![root]\!]) =_{\text{nl}} paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ *with* $I = \{c \mid c \in C \wedge \text{cs}(c) = root \wedge \exists c' \in [\![root]\!] : \text{ds}(c') = \text{ds}(c)\}$.

Proof. See Appendix pp. 271 ff. □

Knowing that the original and generated program are behaviorally equivalent, it seems likely that our program generation also preserves program safety. This is indeed true for control state unaware property automata, but not for arbitrary property automata. In the following, we focus on program safety w.r.t. the control state unaware property automaton considered during the producer verification. Later, we come back to the preservation of program safety with respect to arbitrary property automata.

In the following, we will prove that after a successful producer verification, the generated program is safe. First, we show that control state unaware property automata cannot distinguish between non-location equivalent paths. This means, if two program paths are non-location equivalent and one of them is safe w.r.t. control state unaware property automaton $\mathcal{A}$, so will the other one. The following lemma states this property.

Lemma 5.4. *Let* $\mathcal{A}$ *be a control state unaware property automaton,* P*,* P' *be two programs, and* $p \in paths_P(C)$ *and* $p' \in paths_{P'}(C)$*. If* p *is safe w.r.t.* $\mathcal{A}$ *and* $p =_{\text{nl}} p'$*, then* p' *is safe w.r.t.* $\mathcal{A}$*.*

Proof. Since p is safe w.r.t. $\mathcal{A}$, a configuration sequence $(c_0, q_0)\dots(c_n, q_n)$ for p and $\mathcal{A}$ exists s.t. $\forall 0 \leq i \leq n : q_i \neq q_{\text{err}}$ and $p \equiv c_0 \xrightarrow{g_1} \cdots \xrightarrow{g_n} c_n$. From $p =_{\text{nl}} p'$, we get $p' \equiv c'_0 \xrightarrow{g'_1} \cdots \xrightarrow{g'_n} c'_n$ and $\forall 0 \leq i \leq n : \text{ds}(c_i) = \text{ds}(c'_i) \wedge (i = 0 \vee \exists op \in Ops : g_i = (\cdot, op, \cdot) \wedge g'_i = (\cdot, op, \cdot))$. Let $q_0 \xrightarrow{op_0, C^0_{\text{sub}}} q_1 \dots \xrightarrow{op_n, C^n_{\text{sub}}} q_n$ be a run in $\mathcal{A}$ with $\forall 1 \leq i \leq n : c_i \in C^i_{\text{sub}} \wedge g_i = (\cdot, op_i, \cdot)$. Such a run exists because $(c_0, q_0)\dots(c_n, q_n)$ is a configuration sequence for p and $\mathcal{A}$. Show that $\forall 1 \leq i \leq n : c'_i \in C^i_{\text{sub}} \wedge g'_i = (\cdot, op_i, \cdot)$. Select arbitrary $i \in \{1, \dots, n\}$. From $g_i = (\cdot, op_i, \cdot)$ and $g_i = (\cdot, op, \cdot) \Leftrightarrow g'_i = (\cdot, op, \cdot)$, we know that $g'_i = (\cdot, op_i, \cdot)$. From $c'_i \in C$, $\text{ds}(c_i) = \text{ds}(c'_i)$, and $\mathcal{A}$ being control state unaware, we infer that $c'_i \in C^i_{\text{sub}}$. Hence, $(c'_0, q_0), \dots (c'_n, q_n)$ is a configuration sequence for path p' and $\mathcal{A}$. Knowing that $\forall 0 \leq i \leq n : q_i \neq q_{\text{err}}$, path p' is safe w.r.t. $\mathcal{A}$. □

With the previous lemma, we have everything at hand to prove safety of the generated program. We mainly need to combine the previous lemma with the theorem of behavioral equivalence and our knowledge about a (strongly) well-formed ARG, the type of ARG that we use for program generation.

Like we can show behavioral equivalence only for those paths that the producer considered during his verification, we also guarantee program safety only for these paths. We know from behavioral equivalence that in the generated program these are all program paths starting in the initial location *root* with a data state considered by the initial

abstract state of the producer verification. Additionally, we require that the producer indeed proved program safety. The producer must have used a proper initial abstract state, one that considers the initial automaton state. Taking these restrictions into account as well as that the producer generates programs only after a successful verification, which produced a strongly well-formed ARG, we get our proposition of safety for the generated program. This proposition is similar to the safety result in our previous Programs from Proofs framework [JW17].

Proposition 5.5 (Safety). *Let $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ be a control state unaware property automaton and $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph for program P and refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ which is strongly well-formed for $(e, q_0) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$. Then, $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) \models_I \mathcal{A}$ with $I = \{c \mid c \in C \wedge \text{cs}(c) = root \wedge \exists c' \in [\![root]\!] : \text{ds}(c') = \text{ds}(c)\}$.*

Proof. To show $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) \models_I \mathcal{A}$, every path $p' \in paths_{\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ must be safe. Let $p' \in paths_{\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ be arbitrary. From Theorem 5.3, we know that there exists $p \in paths_P([\![root]\!])$ with $p =_{\text{nl}} p'$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is (strongly) well-formed, we get from Lemma 2.7 that a configuration sequence $(c_0, q_0), \ldots, (c_n, q_n)$ exists for path p and $\mathcal{A}$ with $\forall 0 \leq i \leq n : \exists (e, q) \in N : c_i \in [\![e]\!] \wedge q_i \sqsubseteq q$. From $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ being safe ((strongly) well-formed), we infer that $\forall 0 \leq i \leq n : q_i \neq q_{\text{err}}$. Hence, path p is safe w.r.t. $\mathcal{A}$. From Lemma 5.4, we conclude that p' is safe w.r.t. $\mathcal{A}$. It follows that $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) \models_I \mathcal{A}$. □

Like in the former CPC approaches, the previous result can be transferred to program safety. If the initial abstract state (e, q_0) of the producer covers all data states, i.e., $\forall d \in DS : \exists c \in [\![(e, q_0)]\!] : \text{ds}(c) = d$, then the generated program P_{gen} will be safe w.r.t. the property automaton $\mathcal{A}$ considered by the producer verification ($P_{\text{gen}} \models \mathcal{A}$). Note that this includes that if the producer proves that the original program P is safe w.r.t. $\mathcal{A}$[6], $P \models \mathcal{A}$, then the program P_{gen} generated by the producer will also be safe w.r.t. $\mathcal{A}$.

Keeping program safety throughout program generation only enables the consumer to successfully verify the generated program. To guarantee a successful verification, we also need to ensure termination of the CPA algorithm (Algorithm 1). Due to the for loop in line 5 of the CPA algorithm, which iterates over all CFA edges, the generated program must be finite. Furthermore, one of our termination proofs of the consumer verification relies on the fact that the generated program contains only a finite number of program locations. To show the desired properties, we claim that all programs generated from an ARG for a finite program fulfill these two properties.

Proposition 5.6. *Let $R^P_{\mathbb{C}^{\mathcal{A}}}$ be an abstract reachability graph for finite program P. Then, program $\text{prog}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is finite and its set of locations is finite.*

Proof. Let ARG $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$, program $P = (L, G_{\text{CFA}}, l_0)$, and generated program $\text{prog}(R^P_{\mathbb{C}^{\mathcal{A}}}) = (L', G'_{\text{CFA}}, l'_0)$. From definition of the generated program, we know that $L' = N$. Since N is finite (requirement on ARG), we get that L' is finite. From P being finite, we know that $\exists n \in \mathbb{N} : |G_{\text{CFA}}| = n$. By definition $G_{\text{CFA}} \subseteq L \times Ops \times L$. We infer that $|\{op \mid (l_p, op, l_s) \in G_{\text{CFA}}\}| \leq n$, and, thus, finite. From definition of the generated program, we know that $G'_{\text{CFA}} \subseteq L' \times \{op \mid (l_p, op, l_s) \in G_{\text{CFA}}\} \times L'$. We compute that $|G'_{\text{CFA}}| \leq (|L'|^2 \cdot n)$. Since L' is finite, we conclude that G'_{CFA} is finite. Hence, $\text{prog}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is finite. □

[6] The producer may only prove program P safe, if $\{c \in C \mid \text{cs}(c) = l_0\} \subseteq [\![(e, q_0)]\!]$.

The producer verification also uses the CPA algorithm. Thus, the producer verification may only succeed if the original program is finite. We conclude that the generated programs in the PfP approach are finite and have a finite number of program locations.

From a theoretical point of view, we covered all properties of the program generation that we require to apply our Programs from Proofs approach. In practice, we also need to translate the generated program, a CFA, into a program notation. While defining a program's representation, we already recognized that some programming languages do not support nondeterminism. All their programs are represented by deterministic CFAs. To be able to translate the generated program into these programming languages, the generated program must be deterministic. We assume that the producer wants to write the generated program in the same programming language as the original program. If the programming language in which the original program is written does not support nondeterminism, then the CFA of the original program will be deterministic. For this case, we show that the generated program remains deterministic. Note that in our previous Programs from Proofs framework [JW17] we presented a similar result for a subclass of refined property checking analyses.

Proposition 5.7 (Determinism)**.** *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ be an ARG for deterministic program P and refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ s.t. $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed for $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$. Then, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is deterministic.*

Proof. Let $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (L', G'_{\text{CFA}}, l'_0)$ and $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$. Let $(l, op, l'), (l, op, l'') \in G'_{\text{CFA}}$. We need to show that $l' = l''$. The definition of program generation lets us conclude that there exists ARG edges $(((e_2, e_1), q), (l_1, op, l_2), ((e'_2, e'_1), q'))$, $(((e_2, e_1), q), (l_3, op, l_4), ((e''_2, e''_1), q'')) \in G_{\text{ARG}}$ with $l = ((e_2, e_1), q)$, $l' = ((e'_2, e'_1), q')$ and $l'' = ((e''_2, e''_1), q'')$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed, we further infer that $\text{acs}(e_1), \text{acs}(e'_1), \text{acs}(e''_1) \in \mathcal{L}$. From definition of the transfer relation of a refined property checking analysis and $\rightsquigarrow_{\mathbb{L}}$, we get that $\text{acs}(e_1) = l_1 = l_3$, $\text{acs}(e'_1) = l_2$ and $\text{acs}(e''_1) = l_4$. From P being deterministic, we infer that $l_2 = l_4$ and, thus, $(l_1, op, l_2) = (l_3, op, l_4)$. From $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ being deterministic (strongly well-formed), we get that $((e'_2, e'_1), q') = ((e''_2, e''_1), q'')$. Hence, $l' = l''$. □

After we discussed all properties directly related to the applicability of the Programs from Proofs approach, we resume the discussion of program safety w.r.t. arbitrary property automata. We already mentioned that program safety is only preserved for control state unaware property automata. To firm that claim, consider the property automaton `neg@l8` shown in the bottom of Fig. 5.5. This property automaton checks that at location l_8 variable y is negative. Since its transitions treat concrete states with control state l_8 differently compared to states with other control states, it is not control state unaware.

Our program `SubMinSumDiv` (program text shown on the left of Fig. 5.5) is safe w.r.t. property automaton `neg@l8`. For every program path that starts in the initial program location and reaches location l_8 variable y has a negative value (property of else branch). Now, consider the program shown on the right of Fig. 5.5. This program is the generated program, which the producer generated after a successful verification of program `SubMinSumDiv` with the predicated sign DFA analysis $\mathbb{P}_{\mathcal{P}} \times (\mathbb{L} \times \mathbb{S})$ enhanced with property automaton `nonneg` and initial abstract state $((\top_{\mathbb{P}}, (l_0, \top_{\mathbb{S}})), q_0)$. All program paths starting in the initial location of this generated program are non-location equivalent with a program path starting in the initial location of the original program `SubMinSumDiv` (cf. Theorem 5.3). Furthermore, we already stated that the original program is safe w.r.t.

```
 0: z:=0;
 1: if(x<0)
 2:   if y<x
 3:     z:=-y;
      else
 4:     z:=-x;
 5:   z:=z+10;
    else
 6:   if y≥0
 7:     s:=1;
      else
 8:     s:=-y;
 9:     while x ≥ y ∧ x ≠ 0
10:       if y ≥ 0
11:         z:=z+x;
          else
12:         z:=z+1;
13:       x:=x-s;
14:
```

```
 0: z:=0;
 1: if(x<0)
 2:   if y<x
 3:     z:=-y;
      else
 4:     z:=-x;
 5:   z:=z+10;
    else
 6:   if y≥0
 7:     s:=1;
 8:     while x ≥ y ∧ x ≠ 0
 9:       if y≥0
10:         z:=z+x;
11:         x:=x-s;
      else
12:     s:=-y;
13:     while x ≥ y ∧ x ≠ 0
14:       if ¬y≥0
15:         z:=z+1;
16:         x:=x-s;
17:
```

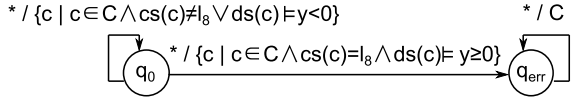

Figure 5.5: Program `SubMinSumDiv`, generated program from Fig. 5.4, and property automaton `neg@l8`

`neg@l8`. However, the generated program is not safe w.r.t. property automaton `neg@l8`. Since variable y is never changed in the generated program, we know from the condition in line 6 that for every program path that starts in the initial location 0 and reaches location 8 variable y is greater zero at location 8.

We showed that program generation does not maintain arbitrary safety properties, at least as long as the class of control state unaware property automata is only a subclass of the class of all property automata. Next, we show that program generation at least keeps all properties expressible by control state unaware property automata. All kinds of properties that we consider in our Programs from Proofs approach remain. Note that this is a necessary condition to transitively apply our Programs from Proofs approach. As before, we show property preservation only for program paths that we considered during a successful verification. This leads us to the following proposition.

Proposition 5.8. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *which is strongly well-formed for* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$. *Furthermore, let* $\mathcal{A}'$ *be a control state unaware property automaton. If* $P \models_{[\![root]\!]} \mathcal{A}'$, *then* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) \models_I \mathcal{A}'$ *with* $I = \{c \mid c \in C \wedge \text{cs}(c) = root \wedge \exists c' \in [\![root]\!] : \text{ds}(c') = \text{ds}(c)\}$.

Proof. We need to show that all paths $p \in paths_{\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ are safe w.r.t. $\mathcal{A}'$. Let $p \in paths_{\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ be an arbitrary path. From Theorem 5.3, we know that there exists $p' \in paths_P([\![root]\!])$ with $p =_{\text{nl}} p'$. Since $P \models_{[\![root]\!]} \mathcal{A}'$, path p' is safe w.r.t. $\mathcal{A}'$. From Lemma 5.4, we infer that p is safe w.r.t. $\mathcal{A}'$. We conclude that $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) \models_I \mathcal{A}'$. □

As before, the previous result can be carried over to program safety if the initial abstract state (e, q_0) of the producer covers all data states, e.g., when the producer verifies program safety.

With the end of the program generation section, we completed the presentation of the producer's task in our Programs from Proofs approach. Next, we continue with the presentation of the consumer part.

5.4 Consumer Verification of the Generated Program

In our Programs from Proofs approach, the consumer verifies the behaviorally equivalent program generated by the producer. The idea behind program generation was to get a structurally simpler program, which can be verified without the enabler analysis, i.e., just with the property checking analysis. In our framework, the consumer uses an even more efficient analysis configuration than the property checking analysis. While the consumer reuses the abstract domain and the transfer function of the property checking analysis, he adapts the other configuration operators to obtain a faster dataflow analysis. The consumer's analysis never adjusts precisions, merges abstract states with same locations, and stops the exploration of an abstract state when it is covered by a more abstract one. These considerations lead us to the following definition of a dataflow analysis derived from a property checking analysis.

Definition 5.12 (Dataflow Analysis of Property Checking Analysis)**.** Let $\mathbb{C}_1^{\mathcal{A}} = (D, \Pi, \rightsquigarrow, \mathsf{prec}, \mathsf{merge}, \mathsf{stop})$ be a property checking analysis. The *dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$ is $\text{DFA}(\mathbb{C}_1^{\mathcal{A}}) = (D, \Pi, \rightsquigarrow, \mathsf{prec}_{\text{DFA}}, \mathsf{merge}_{\text{DFA}}, \mathsf{stop}_{\text{DFA}})$ with static precision adjustment, $\mathsf{prec}_{\text{DFA}}(e, \pi, S) = (e, \pi)$, merge operator $\mathsf{merge}_{\text{DFA}}((e, q), (e', q')) = ((e, q) \sqcup (e', q'))$ if $\text{acs}(e) = \text{acs}(e')$ and $\mathsf{merge}_{\text{DFA}}((e, q), (e', q')) = (e', q')$ otherwise, and termination check $\mathsf{stop}_{\text{DFA}}(e, S) = \exists e' \in S : e \sqsubseteq e'$.

The previous definition changes the configuration of a property checking analysis, a special form of an enhanced CPA, such that it becomes a dataflow analysis. Up to now, it is unclear whether a dataflow analysis of a property checking analysis is still an enhanced CPA, the type of analysis configuration the CPA algorithm requires. To use a dataflow analysis constructed from a property checking analysis within the CPA algorithm, the derived dataflow analysis must be an enhanced CPA. The following proposition states that it is safe to use the derived dataflow analysis configuration.

Proposition 5.9. *Let* $\mathbb{C}_1^{\mathcal{A}}$ *be a property checking analysis and* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of* $\mathbb{C}_1^{\mathcal{A}}$*. Then, a CPA* $\mathbb{C}$ *exists such that* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *is an enhancement of CPA* $\mathbb{C}$ *and control state unaware property automaton* $\mathcal{A}$*.*

Proof. See Appendix p. 273. □

After we know that we can use the dataflow analysis derived from the property checking analysis, we now show that we configured indeed a dataflow analysis. One important property of a dataflow analysis, which we require later, is that it computes only one

abstract state per location. The following lemma claims that this property will be true during the computation of the CPA algorithm if the dataflow analysis is started at a concrete location.

Lemma 5.10. *Let $\mathbb{C}_1^{\mathcal{A}}$ be a property checking analysis, and $P = (L, G_{\text{CFA}}, l_0)$ be a program. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ with $\text{acs}(e) \in L$, initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and P, then after line 1 always $\forall (e', q') \in \mathsf{reached} :$ $\text{acs}(e') \in L \wedge \neg\exists (e'', q'') \in \mathsf{reached} : (e'', q'') \neq (e', q') \wedge \text{acs}(e'') = \text{acs}(e')$.*

Proof. See Appendix pp. 273 f. □

Since the consumer analysis will always start his analysis in the initial program location, we know that the previous lemma applies during computation. The subsequent corollary ensures us that in the final reached set also only one abstract state per location exists.

Corollary 5.11. *Let $\mathbb{C}_1^{\mathcal{A}}$ be a property checking analysis and $P = (L, G_{\text{CFA}}, l_0)$ be a program. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ with $\text{acs}(e) \in L$, initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and P returns $(\cdot, \mathsf{reached}, \cdot)$, then $\forall (e', q') \in \mathsf{reached} : \text{acs}(e') \in L \wedge \neg\exists (e'', q'') \in \mathsf{reached} : (e'', q'') \neq (e', q') \wedge \text{acs}(e'') = \text{acs}(e')$.*

Proof. We know that reached is the reached set in line 29. From Lemma 5.10, we conclude the claim. □

Having finished the discussion of the consumer analysis configuration, we proceed with the two properties each of our approaches must guarantee: soundness and relative completeness. For soundness, we must guarantee that the consumer analysis will only return true, if the generated program is indeed safe. The consumer analysis uses the CPA algorithm[7] to check safety of the generated program. We already proved soundness of the CPA algorithm in Chapter 2 (cf. Theorem 2.3). Hence, we get soundness of the consumer analysis for free.

It remains to show relative completeness. We need to ensure that after the producer successfully verified the original program P, the consumer can successfully verify the generated program P_{gen}, formally $P \models^{\checkmark}_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} \mathcal{A} \implies P_{\text{gen}} \models^{\checkmark}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})} \mathcal{A}$. As before, we consider two aspects: termination of the CPA algorithm and successful verification of the generated program, i.e., the CPA algorithm returns true. Since the termination aspect is more complicated than in the previous approaches, we start with the second part assuming termination.

5.4.1 Successful Consumer Verification on Termination

In this section, we assume that the consumer analysis terminates. Based on this assumption, we want to show that whenever the producer showed safety of the original program, the consumer shows safety of the generated program. More concretely, we are going to show that whenever the CPA algorithm returns true when started with the refined property checking analysis and the original program ($P \models^{\checkmark}_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} \mathcal{A}$), then the CPA algorithm started with the dataflow analysis derived from the property checking analysis and the generated program returns true ($P_{\text{gen}} \models^{\checkmark}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})} \mathcal{A}$).

The CPA algorithm (Algorithm 2) returns true when no state in the reached set considers the error state q_{err} or the automaton top state $q_\top$. To show this, we use the

[7] Since we require no ARG at consumer side, the consumer may use the CPA algorithm without ARG construction (Algorithm 1).

idea behind program generation: the property checking analysis can prove safety on the generated program. From a highlevel point of view, we want the property checking to redo the exploration but now on the generated program. Thus, the abstract states explored by the consumer should be related to the states explored by the producer, the ARG nodes.

We cannot directly compare the ARG nodes with the states explored by the consumer. The states belong to different abstract domains. However, an ARG node only contains an additional enabler state and may consider a different location. If we ignore this information, we can compare ARG nodes and consumer states. Our goal is to compare the two states based on the partial order of the property checking analysis. Having in mind that the states explored by the consumer should be at most as precise as the ARG nodes, we use the location updated property checking extraction to remove the enabler state and to replace the location information by the most general location description $\top_{\mathbb{L}}$ (any location). Now, we say that an abstract state e_{pca} of a property checking analysis is *at least as non-location property checking precise* as an abstract state e_{rpca} of a corresponding refined property checking analysis if e_{pca} is at most as abstract as $e_{\text{rpca}}[\top_{\mathbb{L}}]$. Thereby, $e_{\text{rpca}}[\top_{\mathbb{L}}]$ is the property checking analysis state obtained from e_{rpca} by removing the enabler state and setting the location state to any location $\top_{\mathbb{L}}$. The following definition formally describes the relation and also lifts it to sets of abstract states.

Definition 5.13. Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a refined property checking analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$. An abstract state $e_{\text{pca}} \in E_{\mathbb{C}_1^{\mathcal{A}}}$ is *at least as non-location property checking precise* as $e_{\text{rpca}} \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, $e_{\text{pca}} \sqsubseteq_{\text{npc}} e_{\text{rpca}}$, if $e_{\text{pca}} \sqsubseteq_{\mathbb{C}_1^{\mathcal{A}}} e_{\text{rpca}}[\top_{\mathbb{L}}]$.

We extend this notation to sets $S_{\text{pca}} \subseteq E_{\mathbb{C}_1^{\mathcal{A}}}$, $S_{\text{rpca}} \subseteq E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ of states and write $S_{\text{pca}} \sqsubseteq_{\text{npc}} S_{\text{rpca}}$ if $\forall e \in S_{\text{pca}} : \exists e' \in S_{\text{rpca}} : e \sqsubseteq_{\text{npc}} e'$.

Based on the previous definition, we now show that the states explored by the consumer are at least as non-location property precise as the states explored by the producer, the ARG nodes. Knowing that the producer states neither consider q_{err} nor $q_{\top}$, otherwise the producer did not prove safety, we can infer successful consumer verification.

To prove that the consumer states are at least as non-location property precise as the producer states, we use that the property checking analysis (part) of the producer and the consumer should behave similar. Furthermore, we know that the ARG nodes are used as locations in the generated program and the consumer's dataflow analysis computes one abstract state per location only. With these insights, we prove that a consumer state is at least as property precise as the ARG node that was used to generate the program location considered by that state. More concretely, we compare the abstract state with that location updated property checking extraction of the ARG node that considers the same location as the abstract state. Note that we consider only programs that are generated from strongly well-formed ARGs, e.g., programs generated by the producer.

We start to show that if an abstract state is at least as property precise as the ARG node defining its location state, then for each transfer successor e_s of that state, a node n' in the ARG will exist s.t. e_s is at least as property precise as n'.

Lemma 5.12. *Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ which is strongly well-formed for $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)}$. Furthermore, let $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$ be the generated program, $(e_1, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$, $n \in N$, and $g \in G_{\text{CFA}}$. If $((e_1, q), g, (e_1', q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $\text{acs}(e_1) = n$, and $(e_1, q) \sqsubseteq n[n]$, then there exists $n' \in N$ with $\text{acs}(e_1') = n'$ and $(e_1', q') \sqsubseteq n'[n']$.*

Proof. See Appendix p. 274. □

We use the previous lemma to show that all abstract states in the final reached set are at least as property precise as the ARG nodes that were used to generate the program locations considered by the respective states. Similar to behavioral equivalence or program safety considered in the program generation section, the consumer may not introduce new program behavior. Our claim is only valid when the consumer uses an initial abstract state e'_0 that is at least as precise as the location updated property checking extraction $e_0[l_0]$. This state $e_0[l_0]$ describes the initial state considered by the property checking analysis part of the producer's initial abstract state e_0 but transferred to the changed program structure. Under these restrictions, the subsequent lemma states that after termination reached is at least as property precise as the ARG nodes.

Lemma 5.13. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ *which is strongly well-formed for* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)}$*. Furthermore, let* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) = (L, G_{\text{CFA}}, l_0)$ *the generated program. If Algorithm 2 started with* $\text{DFA}(\mathbb{C}_1^\mathcal{A})$*, initial abstract state* $e'_0 \sqsubseteq e_0[l_0]$*, arbitrary initial precision* $\pi_0 \in \Pi_{\mathbb{C}_1^\mathcal{A}}$*, and* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ *terminates, then it returns* $(\cdot, \mathsf{reached}, \cdot)$ *with* $\forall e \in \mathsf{reached} : \exists n \in N : e \sqsubseteq n[n]$.

Proof. See Appendix pp. 275 f. □

With the previous lemma and $n[n] \sqsubseteq n[\top_\mathbb{L}]$ (cf. definition of location updated property checking extraction), it becomes simple to show that the states explored by the consumer are at least as property precise as the ARG nodes.

Theorem 5.14. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ *which is strongly well-formed for* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)}$ *and* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) = (L, G_{\text{CFA}}, l_0)$*. If Algorithm 2 started with* $\text{DFA}(\mathbb{C}_1^\mathcal{A})$*, initial abstract state* $e'_0 \sqsubseteq e_0[l_0]$*, arbitrary initial precision* $\pi_0 \in \Pi_{\mathbb{C}_1^\mathcal{A}}$*, and* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ *terminates, then it returns* $(\cdot, \mathsf{reached}, \cdot)$ *with* $\mathsf{reached} \sqsubseteq_{\text{npc}} N$.

Proof. From the previous lemma, we know that $\forall e \in \mathsf{reached} : \exists n \in N : e \sqsubseteq n[n]$. Let $e \in \mathsf{reached}$ be arbitrary. We know that $\exists n \in N : e \sqsubseteq n[n]$. Since $n \in \mathcal{L}$, $n \sqsubseteq_\mathbb{L} \top_\mathbb{L}$. From definition of $n[n]$, $n[\top_\mathbb{L}]$, and $\sqsubseteq$, we get $n[n] \sqsubseteq n[\top_\mathbb{L}]$. Since partial order $\sqsubseteq$ is transitive, we know that $e \sqsubseteq n[\top_\mathbb{L}]$. We infer that $e \sqsubseteq_{\text{npc}} n$. We conclude that $\mathsf{reached} \sqsubseteq_{\text{npc}} N$. □

A successful producer verification always uses a refined property checking analysis and generates a strongly well-formed ARG. Furthermore, all ARG nodes in a strongly well-formed ARG neither consider q_{err} nor $q_\top$. Thus, we can easily infer the desired property: after the producer successfully verified the original program, the consumer can successfully verify the generated program if it terminates. Since the CPA algorithm is sound, a compatible, initial abstract state and, thus, the initial abstract state $e_0[l_0]$ of the consumer contains the initial automaton state q_0, we additionally infer that the generated program is safe. The following corollary formulates these claims under the restriction that the consumer does not, as we assume, want to verify new program behavior.

Corollary 5.15. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ *be a refined property checking analysis. If Algorithm 2 started with* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$*, compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$*, initial precision* $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$*, and program* P *returns* $(true, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$*, then if Algorithm 2 started with* $\text{DFA}(\mathbb{C}_1^\mathcal{A})$*, initial abstract state* $e'_0 \sqsubseteq e_0[l_0]$*,* $e'_0 \neq (\cdot, q_\bot)$*, initial precision* $\pi'_0 \in \Pi_{\mathbb{C}_1^\mathcal{A}}$*, and* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) = (L, G_{\text{CFA}}, l_0)$ *terminates, it returns* $(true, \mathsf{reached}, \cdot)$ *and* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) \models_{[\![e'_0]\!]} \mathcal{A}$.

Proof. From Proposition 5.2, we know that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ is an ARG for P and $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ which is strongly well-formed for e_0. From previous theorem, we know that $\mathsf{reached} \sqsubseteq_{\text{npc}} N$. We get that $\forall e \in \mathsf{reached} : \exists e' \in N : e \sqsubseteq_{\text{npc}} e'$. Let $e = (e_1, q)$ and $e' = ((e'_2, e'_1), q')$. From the definition of $\sqsubseteq_{\text{npc}}$, we know that $q \sqsubseteq_{\mathcal{Q}} q'$. From $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ being safe ((strongly) well-formed), we know that $q' \neq q_{\text{err}} \wedge q' \neq q_\top$. Since $\mathcal{Q}$ is a flat lattice and $q \sqsubseteq_{\mathcal{Q}} q'$, we infer that $q \neq q_{\text{err}} \wedge q \neq q_\top$. Hence, Algorithm 2 returns true. From $e_0 = ((\cdot,\cdot), q_0)$ (compatible, initial abstract state) and definition of $e_0[l_0]$, we know that $e_0[l_0] = (\cdot, q_0)$. Since $e'_0 \sqsubseteq e_0[l_0]$ and $e'_0 \neq (\cdot, q_\bot)$, we know that $e'_0 = (\cdot, q_0)$. From Theorem 2.3, we conclude that $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) \models_{[\![e'_0]\!]} \mathcal{A}$. □

As before, the previous result can easily be transferred to program safety. The compatible, initial abstract state e_0 only needs to consider all those concrete states that have a control state equivalent to the initial program location of the original program.

So far, we showed that the consumer analysis will succeed to prove safety of the generated program if his analysis terminates. In the following, we carry on with the examination of termination.

5.4.2 Termination of the Consumer Verification

The consumer analysis uses the CPA algorithm (Algorithm 2) to analyze the generated program. Termination of the CPA algorithm cannot be guaranteed in general. The main problem for the consumer analysis is that no reached set will be computed which is closed under successor computation. In contrast, regularly the set of concrete states represented by the reached set increases. Hence, in a sound analysis like the CPA algorithm new abstract states must be explored. To trigger the exploration, these new states are added to waitlist. The waitlist never becomes empty, the while loop never terminates and, thus, also the CPA algorithm does not terminate.

During the presentation of the producer analysis, we already discussed how the producer could ensure that the consumer analysis terminates. While our discussion is far from complete – it does not cover all theoretical cases in which termination is possible –, we think that we cover the configurations relevant in practice. Remember that in our discussion we identified the following four cases in which a termination of the consumer is possible.

1. The generated program is loop-free.
2. The join-semilattice of the abstract states considered by the property checking analysis, which is used to construct the producer and consumer analysis, has a finite height.
3. The producer analysis applies model checking without adjusting the property checking analysis part, i.e., $\mathsf{prec}(((e_2, e_1), q), \pi, S) = ((\cdot, e_1), q), \cdot)$, $\mathsf{merge}(e, e') := e'$, and $\forall e \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : e \in S \implies \mathsf{stop}(e, S)$.
4. The producer analysis is equivalence relation consistent.

In this section, we show that in each of the four cases termination of the consumer analysis is possible. However, not in all cases termination may be guaranteed for any execution order. We start with some general considerations w.r.t. termination of the consumer analysis. To show termination of the consumer analysis in general, we need to ensure that all loops in the CPA algorithm (Algorithm 2) always terminate. Since the producer

analysis terminated, we know that the original program is finite. Proposition 5.6 ensures us that the generated program analyzed by the consumer is finite. Thus, the for loop in line 5 terminates (finite programs have a finite set of CFA edges). Due to the requirements on the property checking analysis and the construction of the consumer analysis, we know that the consumer analysis uses a transfer function. Hence, the for loop in line 6 always terminates after at most a single iteration. The consumer always starts the analysis of the generated program with an initial abstract state considering the initial program location only. Lemma 5.10 guarantees that the size of the reached set is restricted by the number of program locations. Furthermore, Proposition 5.6 also ensures us that the generated program analyzed by the consumer considers a finite set of program locations. We conclude that the for loop in line 8 always terminates. The for loops in lines 5, 6, and 8 always terminate.

In the following, it remains to be shown that the remaining loop, the while loop in line 3, always terminates. The while loop terminates when the waitlist is empty. Since in each iteration of the while loop at least one element is removed from waitlist and the CPA algorithm inserts at most a single element whenever it changes the waitlist, we only need to show that the number of insertions is restricted. First, we examine termination of the consumer analysis in case the program is loop-free.

Termination for Loop-free Programs In a loop-free program, the number of syntactic paths is bounded. Furthermore, in a dataflow analysis any syntactic program path only contributes once to the analysis solution. Thus, the idea of the termination proof is to show that the number of paths to a program location l restricts the number of times an abstract state considering that program location l is added to waitlist. Since in a loop-free program, the number of program paths is finite, we know from Lemma 5.10 that the consumer analysis only explores abstract states that consider a single program location, and we already recognized that the number of program locations is finite, we conclude that only finitely many times an element is added to waitlist. As discussed before, the termination of the consumer analysis automatically follows for loop-free programs.

First, we show that if the CPA algorithm adds an element to waitlist in the while loop that considers location l_w, then l_w will be reachable from the location l_{init} considered by the initial abstract state (always the initial program location in the consumer analysis), i.e., the analyzed program indeed contains a syntactical path from l_{init} to l_w.

Lemma 5.16. *Let $P = (L, G_{\text{CFA}}, l_0)$ be a program. Furthermore, let $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ and $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ s.t. $\text{acs}(e) \in L$. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$, arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and program P, then for every element $((e', q'), \pi')$ added to* waitlist *during iteration of the while loop a non-empty sequence $g_1 \dots g_n$ exists with $g_1 = (\text{acs}(e), \cdot, \cdot) \wedge \forall 1 \leq i \leq n : g_i \in G_{\text{CFA}} \wedge (i = n \wedge g_n = (\cdot, \cdot, \text{acs}(e')) \vee g_i = (\cdot, \cdot, l') \wedge g_{i+1} = (l', \cdot, \cdot))$.*

Proof. See Appendix pp. 275 f. □

So far, we know that insertions to waitlist are related to program paths. We need to exclude that infinitely many elements per program path are added. In loop-free programs, the number of program paths to a location l can be defined inductively with the help of the program paths to l's predecessors and the number of edges from each predecessors to l. We reuse this definition to show that for each location l we only add one abstract state per program path to l. For this, we need to ensure that during the exploration of an abstract state considering location l' the CPA algorithm adds at most as many abstract states to

waitlist that consider location l as control flow edges $(l', \cdot, l)$ in the analyzed program exist. The following lemma states this claim.

Lemma 5.17. *Let $P = (L, G_{\text{CFA}}, l_0)$ be a program. Furthermore, let $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ and $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ s.t. $\text{acs}(e) \in L$. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state (e, q), arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and program P and $((e', q'), \pi')$ is popped in an iteration of the while loop, then for each $l' \in L$ at most $|\{(\text{acs}(e'), \cdot, l') \in G\}|$ elements $((e'', q''), \pi'')$ with $\text{acs}(e'') = l'$ are added to* waitlist *in the same iteration.*

Proof. See Appendix p. 276. □

We know that insertions to waitlist are related to program paths and in each while loop iteration the number of insertions is restricted by the outgoing control-flow edges of the location considered by the explored state. Now, we have everything at hand to inductively show that the number of paths to a program location l restricts the number of times an abstract state considering that program location l is added to waitlist. As already discussed, the number of program paths is finite and the number of elements added to waitlist is restricted. Thus, the consumer analysis terminates for loop-free programs. The following proposition records the final result of these insights.

Proposition 5.18. *Let $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ and $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ s.t. $\text{acs}(e) \in \mathcal{L}$. If program $P = (L, G_{\text{CFA}}, l_0)$ is finite and does not contain loops, and L is finite, then Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state (e, q), arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and program P terminates.*

Proof. See Appendix pp. 276 f. □

Remember that the generated program is always finite and considers a finite set of program locations. The previous proposition includes termination of the consumer analysis when the generated program is loop-free. Next, we continue to discuss termination for property checking analyses considering a join-semilattice with finite height.

Termination When Analysis' Join-Semilattice has Finite Height We need to show that the consumer analysis will always terminate if it uses an abstract domain whose join-semilattice has a finite height. Remember that a join-semilattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup))$ has a finite height if a $n \in \mathbb{N}$ exists s.t. all chains $e_0 \sqsubset e_1 \sqsubset \cdots \sqsubset e_m$ with $\forall 0 \leq i \leq n : e_i \in E \wedge (i = 0 \vee e_{i-1} \sqsubset e_i)$ are bounded by n $(m \leq n)$ [Muc97, p. 226].

We already discussed that only the termination of the while loop in Algorithm 2 is an issue. To show termination of the while loop in the second case, we use a variant of a well-known principle in termination analysis [Tur49, MP74, CPR11], [KRA09, p. 71]. The underlying idea is that one shows program progress, typically progress of loops, and that the progress that can be made is limited. In the basic form, one needs to show that a quantity (a bound), typically a termination function or ranking function, is always decreased and cannot get below a certain threshold. In contrast, we show that every change of the reached set enlarges the reached set and that there exists an upper bound on the number of times for such an enlargement. Then, we utilize that elements are only added to waitlist when the reached set is changed and each iteration of the while loop removes one element from waitlist.

Before we come to the details of the sketched proof, we must fix the meaning of an enlargement. The following definition formally states when a set of abstract states is

an enlargement of another set of abstract states. Originally, an enlargement of a set of abstract states should describe that the set of covered concrete states is increased. However, this cannot even be guaranteed for dataflow analyses. Theoretically, a more abstract state may represent the same set of concrete states. Thus, we use a different definition, which is related to the procedure of the CPA algorithm. In the style of the CPA algorithm, a set of abstract states is enlarged whenever a state that is not contained in the set is added or a state is exchanged by a more abstract one.

Definition 5.14. Let $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ and $S_1, S_2 \subseteq E_{\mathbb{C}_1^{\mathcal{A}}}$ two sets of abstract states. Set S_2 *is an enlargement of set* S_1, $S_1 \widehat{\sqsubset} S_2$, if $S_1 \sqsubseteq S_2$ and $|S_1| < |S_2| \vee \exists e \in S_1, e' \in S_2 : e \sqsubset e' \wedge S_2 = (S_1 \setminus e) \cup e'$.

To show that every change of the reached set enlarges the reached set, we use the previous definition and compare the reached set directly before a change with the reached set obtained by that change. The following lemma now states the first part of our termination proof: every change of the reached set enlarges the reached set and the number of times the reached set can be enlarged via adding an abstract state is fixed.

Lemma 5.19. *Let* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$. *Let* $\mathsf{reached}_B$ *denote the reached set in Algorithm 2 before a change and* $\mathsf{reached}_A$ *that reached set after a change. If Algorithm 2 is started with* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *arbitrary initial abstract state* $e_0 \in E_{\mathbb{C}_1^{\mathcal{A}}}$ *and precision* $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* $P = (L, G_{\mathrm{CFA}}, l_0)$, *then after initialization of* reached *in line 1 for each change of* reached *it holds that* $\mathsf{reached}_B \widehat{\sqsubset} \mathsf{reached}_A$ *and* $|\mathsf{reached}_B| \leq |\mathsf{reached}_A| \leq |L + 1|$.

Proof. See Appendix pp. 277 f. □

An analysis that uses a join-semilattice with finite height can only finitely many times replace an abstract state by a more abstract state. Together with the previous lemma, we conclude that for consumer analyses based on join-semilattices with finite height the reached set is always enlarged and can only be enlarged finitely many times. Thus, the reached set can only be changed finitely often. We utilize this insight to show that any consumer analysis that considers a join-semilattice with finite height and a finite program with a finite set of locations terminates.

Proposition 5.20. *Let* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$, $P = (L, G_{\mathrm{CFA}}, l_0)$ *a finite program, and* L *being finite. If the join-semilattice* $\mathcal{E}_{\mathbb{C}_1^{\mathcal{A}}}$ *has finite height, Algorithm 2 started with* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *arbitrary initial abstract state* $e_0 \in E_{\mathbb{C}_1^{\mathcal{A}}}$, *arbitrary initial precision* $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* P *terminates.*

Proof. See Appendix pp. 278 f. □

In the beginning of this section, we already recognized that all our generated programs are finite and use a finite set of locations L. Hence, in our Programs from Proofs approach the consumer analysis will always terminate if the join-semilattice used by the property checking analysis has finite height. After we showed termination for the second case, we continue to show termination for the third case, namely, when the producer applies model checking.

Termination When Producer Applies Model Checking So far, the organization of the waitlist, i.e., in which order the elements are removed from waitlist, did not matter to prove termination of the consumer analysis. For the third case, the producer analysis uses model checking, we guarantee termination of the consumer analysis whenever it uses a special organization of the waitlist. Additionally, we also provide an upper bound on the execution time of the consumer analysis. We show that with the special organization of the waitlist the program can be analyzed in a *single pass*, i.e., at most one transfer successor per CFA edge is explored. Assuming that all data structures used in the CPA algorithm can be manipulated in constant time and each execution of a single CPA operation, like e.g. $\rightsquigarrow$, prec, merge, stop, also only takes constant time, the worst case execution time of the consumer analysis is $O(|G| \times |L|)$ and thus polynomial within the input program.

The basis of the following proof is the subsequent observation: all non-enabler parts of abstract states explored by the producer can be generated from the initial abstract state when transitively applying the transfer relation with proper CFA edges. This is possible because the producer analysis never merges and at most adjusts the enabler states. Furthermore, this relation between the initial abstract state and the explored abstract states is recorded in the ARG constructed during the producer analysis. For every node in the ARG, a loop-free path from the root[8] to that node exists in the ARG s.t. for all non-enabler components of the nodes the ith edge in the path matches the ith application of the transfer relation. Due to program generation, we know that a corresponding loop-free path exists in the generated program analyzed by the consumer. From the requirements on the (refined) property checking analysis and the definition of the consumer analysis, we infer that the consumer analysis can replay the producer's property checking analysis behavior on the corresponding loop-free path. Additionally, the consumer analysis cannot change an abstract state once it agrees with the property checking state of the corresponding producer's state (cf. proof of successful consumer verification). In case, we guarantee that for each location we compute the abstract state that agrees with the property checking state of the corresponding producer's state and we explore an abstract state only after the abstract state agrees with the corresponding producer's property checking state, we make sure that the consumer analyzes the generated program in a single pass.

To ensure this intended behavior of the consumer analysis, in line 4 the elements must be popped off from waitlist in a proper order. An abstract state may only be popped off from waitlist when it agrees with the corresponding producer state in the final ARG. The consumer cannot directly recognize when this is the case. However, we mentioned that for each property checking state that must be recomputed a loop-free path from the initial location to the state's location exists that can construct the property checking state. To ensure that an abstract state considering location l is only popped off when it agrees with the corresponding producer state, we need to ensure that the loop-free path that constructs that agreeing state for location l has already been explored by the analysis.

We use the corresponding loop-free paths in the ARG to define an ordering on the consumer's abstract states. The idea of the ordering is as follows. We consider a subgraph of the ARG, which is restricted to the loop-free paths mentioned above. As we will see, this subgraph is a tree and can be obtained when restricting the ARG edges to the edges added at line 24. Since the CPA algorithm always fully explores all successors of an abstract state before it explores the next state, a proper ordering of the ARG nodes (the program locations of the generated program) is sufficient. This ordering must ensure that predecessors in the subgraph have a lower ordinal than their successors. In principal, we require a topological ordering [CLRC07, p. 549] of the subgraph. To get a proper ordering

[8] Since we do not merge, the root node is the initial abstract state.

for the consumer, we finally transfer the ordering of the subgraph to the locations of the generated program. Next, we show the formal definition of a *topological ordering of a restriction of an ARG* to a subset of its edges.

Definition 5.15. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG and $G'_{\text{ARG}} \subseteq G_{\text{ARG}}$ a subset of ARG edges s.t. $n \in N$ is reachable from $root$ via edges from G'_{ARG}. A *topological ordering for* $R^P_{\mathbb{C}^\mathcal{A}}$ *and* G'_{ARG}, if it exists, is a total, injective function $to : N \to \{1, \ldots, |N|\}$ s.t. for every $n, n' \in N$ if a path $root = n_0 \xrightarrow{g_1} \ldots n_{m-1} \xrightarrow{g_m} n_m = n'$ s.t. $\forall 1 \leq i \leq m : g_i \in G'_{\text{ARG}} \wedge g_i = (n_{i-1}, \cdot, n_i)$ and a $j \in \{0, \ldots, m-1\}$ s.t. $n_j = n$ exist, then $to(n) < to(n')$.

We want to use such a topological ordering of an ARG to derive an ordering for the location of the generated program. Since ARG nodes and locations are identical, we directly want to reuse the topological ordering for locations. The following corollary states that any topological ordering is also an ordering for the locations of the generated program.

Corollary 5.21. *Let* $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG,* $G'_{\text{ARG}} \subseteq G_{\text{ARG}}$ *a subset of ARG edges,* $\text{prog}(R^P_{\mathbb{C}^\mathcal{A}}) = (L, G_{\text{CFA}}, l_0)$ *be the generated program, and to be a topological ordering for* $R^P_{\mathbb{C}^\mathcal{A}}$ *and* G'_{ARG}*. Then, to is a total, injective function* $L \to \{1, \ldots, |L|\}$.

Proof. Due to program generation, $N = L$. Thus, $|N| = |L|$ and any total, injective function $to : N \to \{1, \ldots, |N|\}$ is also a total, injective function from L to $\{1, \ldots, |L|\}$. □

We know that when a topological ordering of an ARG exists, we can use this ordering to order the locations of the generated program. It remains to be shown that we can find a topological ordering. Remember that for our proof, we want to use a topological ordering of the subgraph of the producer's ARG that one obtains when restricting the ARG edges to those ARG edges added in line 24 of the CPA algorithm. If this subgraph is indeed a tree, we know that a topological ordering exists. To show that the subgraph is a tree, we show that every ARG node can be reached via ARG edges added in line 24 only and no two ARG edges added in line 24 end in the same ARG node. Additionally, to prove termination we require that for every node in the producer's ARG, a loop-free path from the root to that node exists in the ARG s.t. for all non-enabler components of the nodes on the path the ith edge in the path matches the ith application of the transfer relation and all edges considered by that path are added in line 24 of the CPA algorithm. The following lemma states the third, and, thus, the first aspect.

Lemma 5.22. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ *be a refined property checking analysis s.t. operators* $\mathsf{merge}_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}(e, e') = e'$, $\mathsf{prec}_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}(((e_2, e_1), q), \pi, S) = ((\cdot, e_1), q), \cdot)$, *and furthermore* $\forall e \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}, S \subseteq E_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} : e \in S \implies \mathsf{stop}_{\mathbb{C}_1}(e, \pi, S)$. *If Algorithm 2 started with* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$*, compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$*, a precision* $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$*, and program P returns* $(true, \cdot, R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}})$*,* $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, \cdot)$*, and* $G^{l_{24}}_{\text{ARG}} \subseteq G_{\text{ARG}}$ *is the subset of ARG edges added in line 24 of the CPA algorithm, then for every* $n \in N$ *a sequence* $root, ((e^1_2, e^1_1), q^1), \ldots, ((e^m_2, e^m_1), q^m) = n$ *exists s.t.* $\forall 1 \leq i \leq m : \exists g_i \in G_{\text{CFA}} : (((e^{i-1}_2, e^{i-1}_1), q^{i-1}), g_i, ((e^i_2, e^i_1), q^i)) \in G^{l_{24}}_{\text{ARG}} \wedge ((e^{i-1}_1, q^{i-1}), g_i, (e^i_1, q^i)) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}_1} \wedge \neg\exists 0 \leq j \leq m : i \neq j \wedge ((e^i_2, e^i_1), q^i) = ((e^j_2, e^j_1), q^j)$.

Proof. See Appendix pp. 279 f. □

So far, we know that for every node in the ARG, a loop-free path from the root to that node exists in the ARG s.t. for all non-enabler components of the nodes on the path the ith edge in the path matches the ith application of the transfer relation and all edges of

these path are added in line 24 of the CPA algorithm. It remains to be shown that these paths form a tree rooted in the ARG's root node. Since all those paths already start in the root node, we only need to show that every node has at most one predecessor.

Lemma 5.23. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *be a refined property checking analysis s.t. operators* $\mathsf{merge}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(e,e') = e'$, $\mathsf{prec}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(((e_2,e_1),q),\pi,S) = ((\cdot,e_1),q),\cdot)$, *and furthermore* $\forall e \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : e \in S \implies \mathsf{stop}_{\mathbb{C}_1}(e,\pi,S)$. *If Algorithm 2 started with* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, *compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, *arbitrary precision* $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, *and program* P *returns* $(true,\cdot,R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\mathrm{ARG}},\cdot,\cdot)$, *and* $G^{l_{24}}_{\mathrm{ARG}} \subseteq G_{\mathrm{ARG}}$ *is the subset of ARG edges added in line 24 of the CPA algorithm, then for every* $n \in N$ *it holds true that* $(n',g',n),(n'',g'',n) \in G^{l_{24}}_{\mathrm{ARG}}$ *implies* $(n',g',n) = (n'',g'',n)$.

Proof. Prove by contradiction. Assume that there exists $(n',g',n),(n'',g'',n) \in G^{l_{24}}_{\mathrm{ARG}}$ and $(n',g',n) \neq (n'',g'',n)$. Algorithm 2 adds edges in line 24 only when a transfer successor $(n''',g''',\cdot) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ exists. Furthermore, if it inspects a transfer successor, Algorithm 2 will add at most one edge in line 24. Hence, one of the edges must be added first and n is added to $\mathsf{reached}$ and is never removed (cf. CPA algorithm in combination with property of merge operator). With out loss of generality, assume (n',g',n) was added first. Now, assume later (n'',g'',n) to G_{ARG} is added in line 24. The edge (n'',g'',n) is only added when $\mathsf{stop}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(n,\mathsf{reached}) = false$. Since Algorithm 2 already added n to $\mathsf{reached}$ when it added the first edge and never removed n from $\mathsf{reached}$, we get $\mathsf{stop}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(n,\mathsf{reached}) = true$. Contradiction to the assumption. □

Due to the previous two lemmas, we infer that a tree, a restriction of the ARG to edges $G^{l_{24}}_{\mathrm{ARG}}$ added in line 24, exists that contains a loop-free path for every ARG node from the root node to that ARG node and the ith edge in the path matches the ith application of the transfer relation except for the enabler states. Thus, a topological ordering for the ARG and the ARG edges $G^{l_{24}}_{\mathrm{ARG}}$ exist. Given such an arbitrary topological ordering, and, thus, the definition of an ordering of the generated program's locations, the organization of the $\mathsf{waitlist}$ must ensure that it pops off abstract states with a lower ordinal number first. In case, the CPA algorithm sticks to the previous organization of the $\mathsf{waitlist}$ considering an arbitrary topological ordering of that tree, we say it uses a *tree ordering*.

After we know that the basis of our proof idea is true, we come to the proof of termination within a single pass. The basic idea of the proof is to show that per program location the CPA algorithm explores only one abstract state, i.e., it only pops one abstract state per program location in line 4. Thus, we guarantee termination of the while loop in line 3. Since the transfer function only explores those successors that fit to the program's control flow, all abstract states consider concrete program locations, and we use a transfer function, we explore at most one transfer successor per CFA edge, i.e., we analyze the program in a single pass. Let to denote the ordering of the locations of the generated program. To prove that the CPA algorithm explores only one abstract state per program location, we show that before each iteration i of the while loop the following holds true.

1. The final abstract states for locations with ordinal numbers lower or equal to i have been computed, i.e., $\forall l \in L : to(l) > i \vee \exists (e_1,q) \in \mathsf{reached} : \mathrm{acs}(e_1) = l$ and $\forall (e_1,q) \in \mathsf{reached} : to(\mathrm{acs}(e_1)) \leq i \implies \mathrm{acs}(e_1)[\mathrm{acs}(e_1)] = (e_1,q)$.
2. Only successors for explored abstract states considering locations with ordinal numbers greater or equal to i must still be explored, i.e., $\forall (e_1,q) \in \mathsf{reached} : to(\mathrm{acs}(e_1)) \geq i \Leftrightarrow \exists ((e_1,q),\pi) \in \mathsf{waitlist}$.

3. There exists at most one abstract state per location in the waitlist, i.e., $\forall((e_1, q), \pi)$, $((e_1', q'), \pi') \in \mathsf{waitlist} : ((e_1, q), \pi) = ((e_1', q'), \pi') \vee \mathrm{acs}(e_1) \neq \mathrm{acs}(e_1')$.

The first and second invariant ensure that in iteration i an abstract state considering the location with ordinal number i is explored and that the abstract state agrees with the corresponding producer state. The third invariant states together with the previous insight that no state is explored twice. To prove these invariants, we use an additional helper invariant. It states that all abstract successors of states that are explored, i.e., they are contained in reached but not in waitlist, are covered by an element in waitlist when the location associated with the successor must still be explored. Based on these invariants, we can prove termination of the consumer analysis in a single pass. Now, the following theorem claims termination of the consumer analysis in a single pass when the consumer analysis uses a tree ordering and the producer uses model checking.

Theorem 5.24. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *be a refined property checking analysis s.t. operators* $\mathsf{merge}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(e, e') = e'$, $\forall e \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : e \in S \implies \mathsf{stop}_{\mathbb{C}_1}(e, \pi, S)$, *and* $\mathsf{prec}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(((e_2, e_1), q), \pi, S) = (((\cdot, e_1), q), \cdot)$. *If Algorithm 2 started with refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, *compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, *a precision* $\pi \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, *and program* P *returns* $(true, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *and* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\mathrm{CFA}}, l_0)$, *then Algorithm 2 started with* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, $e_0[l_0]$, *a precision* $\pi' \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *and using a tree ordering for* waitlist *terminates in a single pass.*

Proof. See Appendix pp. 280 ff. □

The previous theorem stated consumer analysis termination in a single pass in case the consumer uses a proper organization of the waitlist and the producer applied model checking. Now, only one case remains in which we want to guarantee termination, namely when the producer uses a equivalence relation consistent analysis.

Termination When Producer Analysis is Equivalence Relation Consistent Remember that the idea of an equivalence relation consistent refined property checking analysis is the following. It groups abstract states into equivalence classes s.t. each equivalence class is a potential program location in the generated program. Furthermore, during analysis abstract states of the same equivalence class are treated similarly to locations in dataflow analyses. With this idea in mind, we basically prove that the consumer analysis mimics the behavior of the producer analysis. From the termination of the producer analysis, we then conclude the termination of the consumer analysis.

To show that the consumer analysis mimics the property checking analysis part of the producer analysis, we need to relate the abstract states explored by the producer and consumer analysis. Since the consumer analysis considers at most one abstract state per program location it is sufficient to use the location information of a consumer's state to identify the corresponding producer state. Due to program generation, a program location in the generated program is an abstract state in the producer's analysis configuration. Since one idea of equivalence relation consistency was to group together producer states responsible for the same program location, we use the equivalence relation $\sim$ and the location information of the consumer state to relate a consumer's abstract state to a producer's abstract state. In each imitation step, we want to uniquely relate the consumer and producer states. First, we show that in each step different producer states are never related to the same location state. We utilize that in an equivalence relation two elements e_1 and e_2 can only be equivalent with e_3 when they are both equivalent. The following

lemma states that at any point in time no two different producer states with the same equivalence class exist simultaneously in the reached set.

Lemma 5.25. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalence relation consistent, refined property checking analysis and $\sim$ an equivalence relation which shows that $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is equivalence relation consistent. If Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, arbitrary precision $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and program P, then after line 1 it always holds true that $\forall e, e' \in \mathsf{reached} : e = e' \vee e \not\sim e'$.*

Proof. See Appendix pp. 286 f. □

We ensured that we can map producer states to program locations of the generated program. From the previous lemma, we now easily conclude the opposite direction. The following corollary states that different nodes in the final ARG generated by the producer – hence different program location of the generated program – are not equivalent.

Corollary 5.26. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalent consistent, refined property checking analysis and $\sim$ an equivalence relation which shows that $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is equivalence relation consistent. If Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, a precision $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and program P returns $(\cdot, \cdot, R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$ and $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$, then $\forall n, n' \in N : n = n' \vee n \not\sim n$.*

Proof. We know that if Algorithm 2 reaches line 29, then $N = \mathsf{reached}$. From Lemma 5.25 and $N = \mathsf{reached}$, it follows that $\forall n, n' \in N : n = n' \vee n \not\sim n$. □

Due to the previous corollary, different program locations cannot be mapped to the same producer state. Now, we know that we can uniquely map program locations and producer states.

Another aspect, which we require when we want to mimic the producer's property checking analysis behavior, is that if we relate a producer to a consumer state, we can also relate their respective transfer successors. Remember that we use the equivalence relation $\sim$ to relate producer and consumer states. Additionally, a producer state will only be related to a consumer state if the consumer's location information is equivalent with the producer state. In the consumer's successor, the location information is determined by the end point of the CFA edge considered during the consumer's transfer successor computation. Furthermore, we know that the location information of the transfer predecessor is determined by the start point of that CFA edge. Due to program generation, the CFA edge corresponds to an ARG edge. So far, we know that the producer state ps is equivalent to the predecessor p of the ARG edge (the start point of the CFA edge) and ps' transfer successor is equivalent to the transfer successor of p. Since the transfer successor of p may be different from the end point of the ARG edge (the location information of the consumer's transfer successor), we need to show that the transfer successor is equivalent to the end point of the ARG edge. The following lemma claims this equivalence.

Lemma 5.27. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalent consistent, refined property checking analysis and $\sim$ an equivalence relation which shows that $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is equivalence relation consistent. If Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, a precision $\pi \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and program P returns $(\cdot, \cdot, (N, G_{\text{ARG}}, root, N_{\text{cov}}))$, then $\forall (n, g, n') \in G_{\text{ARG}} : \forall (n, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} : e' \sim n'$.*

Proof. See Appendix pp. 287 f. □

Due to the previous lemma, we now know that if we relate a producer to a consumer state, we can also relate their respective transfer successors.

A relation of producer and consumer states is not enough. We additionally require that the consumer state $e_c = (e_1, q_1)$ is non-location equivalent with the property checking analysis part of the producer state e_p, i.e., $e_c = e_p[\text{acs}(e_1)]$. Furthermore, we need to ensure that if the producer explores a transfer successor, a corresponding edge in the generated program will exist. To guarantee the last property, it is sufficient that the producer's property checking analysis state is at most as abstract as the property checking analysis state considered by the ARG that is responsible for the location information considered in the consumer state. When all these conditions are met by producer state e_p and consumer state e_c, we say that *state e_p is mimicked by state e_c*. In the following, we formally define this imitation relation.

Definition 5.16. Let $\sim$ be an equivalence relation which shows that refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is equivalence relation consistent, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ be an ARG for $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ and program P, and $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$ the program generated from that ARG. Then, an abstract state $e_p = ((e^p_2, e^p_1), q^p) \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ *is mimicked by* $e_c = (e_1, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$, denoted by $e_p =_\sim e_c$, if $\text{acs}(e_1) = ((e^l_2, e^l_1), q^l) \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$[9], $e_p \sim \text{acs}(e_1)$, $(e^p_1, q^p) \sqsubseteq (e^l_1, q^l)$, and $e_p[\text{acs}(e_1)] = e_c$.

Based on the previous definition, we now prove that the consumer analysis can mimic the producer analysis. In principle, we inductively show that before each iteration of the while loop in line 3 of the CPA algorithm, the reached set of the producer is mimicked by the reached set of the consumer and an organization of the consumer's waitlist exists s.t. it is an imitation of the producer's waitlist. Prior to the sketched termination proof, we want to show the following aspect, which is helpful to prove termination. If a consumer state mimics a producer state, then the consumer can mimic all successors explored for that producer state and no successors of the consumer state exist that cannot be mimicked by a producer successor. Hence, the consumer can imitate the producer's successor exploration. The subsequent lemma claims that the desired property is indeed true.

Lemma 5.28. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalence relation consistent, refined property checking analysis, $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be a dataflow analysis for property checking analysis $\mathbb{C}_1^{\mathcal{A}}$, $e_p = ((e^p_2, e^p_1), q^p) \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, and $e_c = (e^c_1, q^c) \in E_{\mathbb{C}_1^{\mathcal{A}}}$. Furthermore, assume that Algorithm 2 started with $(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, arbitrary precision $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, and program $P = (L^p, G^p_{\text{CFA}}, l^p_0)$ returns $(\mathit{true}, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$. Now, let ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$ and $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$ be the generated program. If $\text{acs}(e^c_1) \in L$, $\text{acs}(e^p_1) \neq \bot_{\mathbb{L}}$, and $e_p =_\sim e_c$, then there exists a bijective function $bt : \{(e_p, g, e'_p) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} \mid g \in G^p_{\text{CFA}}\} \to \{(e_c, g', e'_c) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \mid g' \in G_{\text{CFA}}\}$ with $bt((e_p, g, e'_p)) = (e_c, g', e'_c) \implies e'_p =_\sim e'_c$.*

Proof. See Appendix pp. 288 f. □

The previous lemma ensures that the consumer can mimic the transfer successor exploration. To mimic the complete producer analysis, the consumer's initial abstract state must mimic the producer's initial abstract state. Furthermore, the consumer must mimic the producer's precision adjustment, merge, and termination check behavior. If the consumer can mimic also these operations, we can inductively show that before each iteration

[9] Due to program construction this is always the case if $\text{acs}(e_1) \in L$

of the while in line 3 of the CPA algorithm, the reached set of the producer is mimicked by the reached set of the consumer and an organization of the consumer's waitlist exists s.t. it is an imitation of the producer's waitlist. In this case, we can conclude termination of the consumer analysis if it mimics the producer analysis because we already know that the producer analysis terminates. We claim that such an imitation is possible and we will use it in the proof of the following theorem, which states termination for our fourth and last case. The next theorem says that an exploration order for the control flow edges and an appropriate management of the waitlist exist such that the consumer analysis terminates when the producer analysis uses an equivalence relation consistent, refined property checking analysis.

Theorem 5.29. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *be an equivalence relation consistent, refined property checking analysis. Assume Algorithm 2 started with* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$*, compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$*, any precision* $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$*, and program* P *returns* $(true, \cdot, R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$*. Now, let* $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$*. Then, exploration orders of* G_{CFA} *for each while loop iteration of Algorithm 2 and a management of* waitlist *exist s.t. Algorithm 2 started with* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$*,* $e_0[l_0]$*, any precision* $\pi_0' \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$*, and* $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$ *terminates.*

Proof. See Appendix pp. 289 ff. □

The previous theorem completed our considerations w.r.t. termination of the consumer analysis. We proved termination of the consumer analysis for all four cases, which we identified to be typically relevant in practice. Furthermore, we already discussed how to deal with the remaining cases, either the producer tries out whether the consumer analysis terminates on the generated program or he transfers his proof to the generated program. Moreover, termination was the last aspect of the consumer analysis we had to discuss. Thus, we are done with the presentation of the consumer part of the Programs from Proofs approach. In the following section, we want to study if and when the possibility to verify arbitrary program properties is kept along program transformation.

5.5 Reverification of the Generated Program

So far, we showed that in the PfP setting producer verifiability of a property on the original program implies consumer verifiability of the same property on the generated program. However, the safety property considered in the Programs from Proofs instance may not be the only one the consumer is interested in. To convince the consumer that the generated program also fulfills the other properties of interest, the consumer must be able to prove them with an appropriate analysis configuration. Of course, the producer could combine all properties of interest into a single property automaton and could try to find an analysis configuration that shows safety w.r.t. the combined property. On the one hand, the producer reverifies properties he already proved. For example, a different consumer was already interested in that property or the producer iteratively applies the PfP approach to ensure all properties of interest. On the other hand, the producer might combine all analyses that he would use to prove one of the properties of interest. In this case, the resulting combined analysis is a lot more complex, probably slower. Also, it might be less reliable. Defining and implementing the composition is an additional source for errors. Thus, verifying the properties one by one could be more beneficial. Hence, the program generation on the producer side should not only enable the verification of the

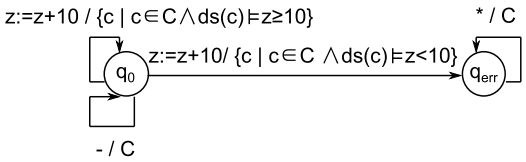

Figure 5.6: Control state unaware property automaton for property: after execution of statement `z:=z+10;` variable z has a value greater than or equal to ten

same property with a simpler analysis. Moreover, it should also preserve verifiability of all other properties of interest. At best, we would be able to prove a property $\mathcal{A}$ on the generated program with the same analysis with which we proved that property $\mathcal{A}$ on the original program.

We already showed that only properties that can be specified with a control state unaware property automaton are guaranteed to remain valid on the generated program (see Proposition 5.8). In principle, a control state unaware property might be successfully proven for the generated program if it had been successfully proven for the original program. However, that properties remain valid is only the necessary condition. Still, the restructuring of the original program during program generation could prohibit verification of the generated program. The analysis of the generated program could now fail for any ordering of waitlist, e.g., it may no longer terminate. For example, consider our program `SubMinSumDiv` and its transformation (both shown in Fig. 5.5). We can use an interval analysis [CC77], an analysis that approximates the possible values of a variable by an interval, to prove that program `SubMinSumDiv` fulfills the control state unaware property shown in Fig. 5.6, which states that after execution of statement `z:=z+10;` variable z has a value greater than or equal to ten. For the original program `SubMinSumDiv`, the interval analysis terminates when the paths in the while loop are explored in an interleaved way or the else branch of the while loop is explored after the if branch. In contrast, the interval analysis never terminates on the generated program because in the generated program the two execution possibilities of the while loop are explicitly split. Assuming unbounded integers and an interval analysis that combines abstract states solely with the join operator, the interval analysis never finishes the exploration of the while loop in line 13. In each iteration, the right bound of the interval for variable z is increased by one.

Identifying all cases in which we can assure that verifiability is transferred along program generation is difficult. Similar to our example, we assume that in general not only the analysis is important but also how the program is restructured. Thus, in this thesis we only consider one class of analyses, those analyses the consumer already uses in the Programs from Proofs approach. Having in mind that one goal of the Programs from Proofs approach is to enable program verification with a simple analysis, we think that the consumer preferably wants to apply such simple analyses. Since the interval analysis, which belongs to the simple analyses the consumer may use, failed to terminate on the generated program, we also exclude the termination aspect. In the previous section, we proved that arbitrary dataflow analyses terminate when they are applied on loop-free programs or if they have a finite lattice height. In these cases, termination is guaranteed. For the remaining cases, we think that termination depends on the restructuring of the original program. Nevertheless, we believe that already for the original program termination is not assured for any exploration order.[10] The consumer could already fail to prove

[10] We did not prove this claim.

the original program. Thus, it should be okay that he fails on the generated program. The following proposition now states maintenance of the verifiability for dataflow variants of property checking analyses in case the program generation uses a strongly well-formed ARG, as the producer does in our Programs from Proofs approach. Note that we assume that the transfer of the verifiability will only be guaranteed if the analysis whose behavior should be transferred and the producer's refined property checking analysis start in the same program location. We think that this is naturally the case since a consumer wants to guarantee properties for all program executions he executes and typically program executions will start in the initial program location.

Proposition 5.30. *Let* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1')^{\mathcal{A}'}}$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2\times\mathbb{C}_1')^{\mathcal{A}'}$, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1')^{\mathcal{A}'}}$ *is strongly well-formed for* $((e_2,e_1),q)\in E_{(\mathbb{C}_2\times\mathbb{C}_1')^{\mathcal{A}'}}$, $\mathrm{acs}(e_1)\in\mathcal{L}$, *and* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1')^{\mathcal{A}'}}) = (L', G'_{\mathrm{CFA}}, l'_0)$. *If Algorithm 2 started with* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *initial abstract state* $e_0=(e,q)\in E_{\mathbb{C}_1^{\mathcal{A}}}$ *with* $\mathrm{acs}(e)=\mathrm{acs}(e_1)$, *arbitrary initial precision* $\pi_0\in\Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* P *returns* $(true,\cdot,R^P_{\mathbb{C}_1^{\mathcal{A}}})$, *then if Algorithm 2 started with* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *initial abstract state* e'_0 *s.t.* $e'_0=(l'_0,q)$ *if* $e\in E_{\mathbb{L}}$ *and* $e'_0=((l'_0,c),q))$ *if* $e_0=(\cdot,c)$, *arbitrary initial precision* $\pi'_0\in\Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1')^{\mathcal{A}'}})$ *terminates, it returns* $(true,\cdot,\cdot)$.

Proof. See Appendix pp. 294 f. □

The previous proposition tells us that all properties that can be described by a control state unaware property automaton and that can be proven with a dataflow analysis on the original program can be proven with the same dataflow analysis on the generated program assuming termination of the CPA algorithm. Thus, the Programs from Proofs approach is transitive and the producer may iteratively apply the Programs from Proofs approach to assure various properties. With this insight, we finish our theoretical discussion of the Programs from Proofs approach and continue to practically evaluate our Programs from Proofs framework.

5.6 Evaluation

So far, we proved that the consumer can verify the generated program with a simpler analysis, which is often also coarser than the producer analysis. Although the consumer uses a simpler and often coarser analysis than the producer, we cannot theoretically prove that the consumer validation is more efficient. The problem is that the generated program can be larger. Hence, one goal of our evaluation is to examine if the verification is indeed simpler in terms of verification time and memory consumption. Second, we want to study the generated programs, especially their sizes in comparison to the original programs. Our third goal is to compare our PfP approach with existing, alternative approaches, which aim at a fast consumer validation. Typically, these are approaches applying the Proof-Carrying Code principle. One such alternative is the general configurable program certification framework presented in the previous two chapters.

First, we introduce our evaluation set-up, i.e., the considered programs and properties, the analysis configurations, the alternative PCC approaches with which we compare our PfP approach, and the set up of the execution environment. Thereafter, we discuss the results of our evaluation.

5.6.1 Evaluation Setting

In the following paragraphs, we introduce the verification tasks, the competitors for the consumer validation, and provide details on the infrastructure we used for our evaluation. We start with the introduction of the verification tasks. A verification task consists of an analysis configuration, a program, and a correctness property. Note that our verification tasks never only differ in the correctness property. Thus, we use the acronym of the refined property checking analysis and the program name to refer to a verification task of the producer or the corresponding validation tasks of the consumer. Further note that we reuse and extend the verification tasks from previous PfP presentations [WSW13, JW15, JW17]. Next to new programs, we also consider new analysis configurations. For example, we consider property checking analyses with non-static precision adjustment and property checking analyses whose precision is in between dataflow analyses and model checking.

Configurations In theory, our PfP approach is a general framework. For example, it can be instantiated with various refined property checking analyses, each described by an analysis configuration. To build a refined property checking analysis, we could select from three enabler analysis $\mathbb{C}_2$:

- a predicate analysis $\mathbb{P}$ with adjustable block encoding [BKW10], which uses the SMT solver MathSAT5 [CGSS13] version 5.3.10, depending on the analysis type of the property checking analysis the predicate analysis either abstracts at functions and loop heads or at every join node in the CFA,
- an octagon analysis $\mathbb{O}$ based on the Apron library [JM09a],
- and explicit-state model checking $\mathbb{V}$ [BL13] with refinement selection [BLW15b] and path prefix slicing [BLW15c], which tracks variable values similar to constant propagation [NNH05, p. 72].

For the construction of the property checking analysis $\mathbb{C}_1^{\mathcal{A}}$, we selected one of the following abstract domains:

- an interval domain $\mathbb{I}$ [CC77], which tracks per location and variable an interval of possible values of that variable,
- the location abstract domain $\mathbb{L}$ introduced in the background chapter,
- the combination $\mathbb{L}\times\mathbb{S}$ of sign and location abstract domain introduced in Chapter 2 – in the following we use $\mathbb{S}$ to refer to this domain –,
- a product combination $\mathbb{SI}$ of the sign DFA domain $\mathbb{S}$ and the interval $\mathbb{I}$ abstract domain,
- an abstract domain $\mathbb{U}$ which tracks the uninitialized variables per location, and
- a value abstract domain $\mathbb{V}$, the same domain that is used in the third enabler analysis. We write $\widetilde{\mathbb{V}}$ if we combine the value abstract domain with precision adjustment.

For each abstract domain, the transfer relation is fixed. However, we need to configure the precision adjustment operator, the merge operator, and the termination check. All property checking analyses except for those that are based on the abstract domain $\widetilde{\mathbb{V}}$ use a static precision adjustment, which does not change the abstract state. Property checking

analyses based on $\widetilde{\mathbb{V}}$ should only track variable values for a predefined set of variables. Hence, their precision adjustment replaces concrete variable values of non-tracked variables by any value.

To configure the analysis type, we need to define the merge operator and the termination check. All analysis use the most common termination check $\mathsf{stop}(e, S) := \exists e' \in S : e \sqsubseteq e'$. To configure dataflow analyses, we configured the merge operator to join abstract states when the location element and the property automaton state is the same. For all abstract domains except for the location domain $\mathbb{L}$, we built one dataflow analysis. We exclude the location domain because it considers location elements only. A merge operator for location states that joins location states when they are the same is identical to a merge operator for location states that never merges abstract states. Never merging abstract states is the standard configuration of the merge operator to get a model checking analysis. Again, we built one model checking analysis for almost each of the abstract domains. We left out the interval abstract domain $\mathbb{I}$ and the combined abstract domain $\mathbb{SI}$ because their behaviors are similar to the value abstract domain. Furthermore, we constructed one additional property checking analysis for the combined abstract domain. This analysis merges abstract states when the location, the sign state, and the property automaton state are the same. We yield an analysis whose precision lies in between dataflow analysis and model checking. In total, we got 12 different property checking analyses.

To cover a broad spectrum of configurations, we tried to combine all property checking analyses with all enabler analyses. We left out the combination of the value analysis enabler with property checking analyses considering the interval $\mathbb{I}$, the combined $\mathbb{SI}$, or the value abstract domain $\mathbb{V}, \widetilde{\mathbb{V}}$. In these cases, the property checking analysis' abstract domain is the same or more precise A combination with the value analysis enabler is of no value. Thus, we got 29 different refined property checking analysis configurations in total. All refined property checking analyses use the product combination of the precision adjustment and the transfer relation. Note that we do not consider improvements of the transfer relation based on dynamic multi-edges or multi-edges because either the program generation or the property automaton evaluation cannot be performed properly. Like the property checking analyses, the refined property checking analyses use the most common termination check. Furthermore, the merge operator never combines abstract states with different enabler states and otherwise produces an abstract state with the same enabler state as the two input states and a property checking analysis state that is computed by the merge operator of the property checking analysis. The predicate enabler $\mathbb{P}$ abstracts at functions and loop heads only if the property checking analysis applies model checking and otherwise it abstracts at every join node in the CFA. For the enabler analyses $\mathbb{O}$ and $\mathbb{V}$, we used the best refinement strategy of the following, promising heuristics: NONE, LENGHT_MIN, LENGTH_MAX, DOMAIN_MIN, and DOMAIN_MIN+LENGTH_MIN (the default configuration). Moreover, we always used a CEGAR [CGJ+00] approach to determine the precision for the enabler analysis $\mathbb{C}_2$. We tried out both types of refinements, either starting from scratch after refinement or applying lazy refinement [HJMS02].

Programs and Properties To examine the general applicability of our approach, we also need to examine the PfP approach with different types of programs and safety properties. We selected our programs from three different verification benchmarks [KHCL07, SI13, Bey16], other research papers [DLS02, BSI+08, WSW13, JW15, JW17], some self-written programs, and our example program `SubMinSumDiv`. The following listing gives an overview of the set of programs.

Buffer overflow benchmark [KHCL07] `interproc`, `NetBSD`, `nosprintf`, `sendmail`, `SpamAssassin`

NECLA static-small [SI13] `inf1`, `inf4`, `inf6`, `inf8`

Software verification competition benchmark [Bey16] `cdaudio`, `diskperf`, `invertstring`, `kbfiltr1`, `kbfiltr2`, `kundu`, `lockfree3.0`, `lockfree3.1`, `memslave1`, `memslave2`, `pipeline`, `pipeline2`, `relax`, `s3srvr`, `testlocks5`, `testlocks5d`, `testlocks6`, `testlocks7`, `testlocks8`, `testlocks12`, `testlocks12d`, `tokenring02`, `tokenring03`, `tokenring04`, `transmitter01`, `transmitter02`

Research papers `condsum` [JW15], `ESP`, `ESPa`, `ESPb`, `ESPc` [DLS02], `facnegsum` [JW17], `locks` [WSW13], `SLR`, `SLRb` [BSI+08]

Self-written programs `addIteration`, `div`, `fibonacci`, `fraction`, `harmonicMean`, `invertarray`, `invertsorted`, `liststatistics`, `palindrom`, `PfP`, `PfPb`, `PfPc`, `powerapprox` `propertyInFlag`

For the programs from the buffer overflow benchmark [KHCL07], we checked that no bufferoverflow occurs, a typical invariant property. When the property checking analysis considered the abstract domain $\mathbb{U}$, which tracks uninitialized variables, we proved that no uninitialized variables are used. We verified that no null pointers are dereferenced in the programs `lockfree3.0`, `addIteration`, and `lockfree3.1`. In case the property checking analysis only tracks locations ($\mathbb{L}$), our verification examines whether the programs fulfill pure protocol properties, i.e., they respect a certain ordering on some of the program statements. We reuse protocols considered by Wonisch et al. [WSW13] and us [JW17] in previous papers. For the programs we wrote ourselves, we used standard properties like no null pointer dereference, no division by zero, no buffer overflow plus some properties that are specifically tailored to the structure of the program. In the remaining cases, we mainly looked at (subaspects of the) properties provided by the programs, i.e., properties encoded by error labels or assertions. Since labels may be different for the original and the generated program, we typically translated the encoded properties into a property automaton that checks the property instead of checking only the reachability of the error location. To properly capture the encoded property, the translated property automaton often monitors certain program operations and the value of some program variables. Sometimes, we also had to instrument the program to enable checking of the property.

Validation Competitors To study if our Programs from Proofs approach is also a practical alternative for an efficient consumer validation, our goal is to compare the PfP approach with different PCC techniques. We use our basic configurable program certification approach, denoted by CPC, as the baseline. Furthermore, we select the combination of reduction and partitioning (CPC^{RP}), the optimized configurable certification approach whose validation was typically fastest, as one instance for a technique that stores a subset of the reachable, abstract states. For certificates constructed for a refined property checking analysis using the predicate analysis as enabler analysis, we used the reduced node set during certificate construction. In all other cases, the certificate is constructed from the highly reduced node set. As partitioning strategy we tried the `BEST_FIRST` strategy. Certificates are easy to construct with this strategy and certificate validation performs similar to the validation of certificates constructed by other strategies available in CPAchecker (see Henrik Bröcher's bachelor thesis [Brö16]). Only when we failed

to construct a certificate with this strategy, e.g., due to a timeout, we went on with a random partition. To get partitions of size more than one even when the ARG is small, we restricted the size of each set of partition nodes to ten. Note that we think that one instance for a technique storing a subset of the reachable abstract state is sufficient. Thus, we choose the one considered in our previous comparison of PCC approaches that especially performs best when the producer analysis, like our refined property checking analyses, is more precise than a dataflow analysis. Hence, we excluded the approach storing only abstract states that are reachable via backward edges in the ARG. Additionally, we consider one approach (ARG) that stores the abstract reachability graph and checks that the ARG is constructed properly, i.e., it fulfills similar properties like a well-formed ARG. This approach works similarly to the approaches of Jhala et al. and Henzinger et al. [HNJ+02, HJMS03], but its implementation is a lot more general and allows to certify the analysis of arbitrary CPAs. Since the precision of all enabler analyses is determined via counterexample guided abstraction refinement, we also compare with a technique for regression verification (RV) [BLN+13] which stores the determined enabler precision. In total, we compare our PfP approach with four different PCC techniques.

Execution Set Up As before, we run the experiments with the benchmarking evaluation framework BenchExec [BLW15a]. Every verification and validation task is executed on a machine with an Intel Xeon E5-2650 v2 CPU at 2.6 GHz and with 135 GB of RAM. Furthermore, each task must be completed within 15 minutes of CPU time and must use no more than 15 GB of RAM. For the PCC approach that checks the abstract reachability tree, we additionally had to increase the stack size of the Java virtual machine from 1024 kB to 51200 kB. Once again, we performed our experiments with the CPAchecker revision 23042 available in the runtime_verification branch[11] and Java HotSpot(TM) 64-Bit Server VM 1.8.0_101.

The generated programs and the certificates were constructed once before the evaluation. For each of the 127 verification tasks, we required the enabler analysis to successfully verify the original program. Furthermore, the consumer uses the more efficient variant of the CPA algorithm, which does not compute an abstract reachability graph (cf. Algorithm 1), while the producer must use the standard algorithm to perform the refinement of the enabler analysis. Similarly, we disabled the loop structure detection for the generated programs, which is only needed for the refined property checking analysis and is difficult to detect on the generated program.

In the following, we examine the average of 10 experimental executions. The results for all 127 tasks can be found in Tab. B.3 in the appendix. Note that in the appendix for each of the 127 tasks we either include the results based on a refined property checking analysis with lazy refinement or one without. We present the version in which the consumer analysis of the generated program performs better. When the consumer analysis of the program generated from the refined property checking analysis with lazy refinement is faster, we marked the program name with an asterisk. Analyses using the octagon enabler often timed out when using lazy refinement. In contrast, for the predicate enabler the variant with lazy refinement seems to provide the better results.

We start with the question whether the consumer verification of the generated program is more efficient than the producer verification.

[11] `https://svn.sosy-lab.org/software/cpachecker/branches/runtime_verification/`

Table 5.1: Extract of the comparison of the producer and consumer verification showing all tasks in which the producer's verification was better in time or memory

$\mathbb{C}_P$	program	#r	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{OU}$	pipeline2	11	8.81	260.19	0.03	577.60	3971.20	6.88
$\mathbb{VU}$	cdaudio	1	2.75	2.89	0.95	365.60	344.80	0.94
$\mathbb{VL}$	tokenring04	22	14.15	43.05	0.33	1056.00	537.30	0.51
$\mathbb{PS}$	transmitter02	6	8.63	5.47	1.58	494.70	553.10	1.12
$\mathbb{OS}$	powerapprox	1	9.86	10.73	0.92	1036.40	996.80	0.96
$\mathbb{OU}$	pipeline2	11	8.51	258.16	0.03	579.30	3903.00	6.74
$\mathbb{VU}$	cdaudio	1	3.00	3.63	0.83	388.80	380.00	0.98

5.6.2 RQ 1: Does the Consumer Verification Outperform the Producer Verification?

In this section, we want to study whether the consumer verification of the generated program is easier than the producer verification. In our context, easier means the consumer verification is faster and requires less memory.

Looking at the results for the complete set of verification tasks, which can be found in Tab. B.5 in the appendix, we observe that the consumer verification is indeed often easier. The consumer can verify the complete set of tasks in 647.13 s requiring 38,043.1 MB while the producer requires more than twice as long, 1,401.52 s, and 30% more memory, namely 49,526.3 MB. Only in 6% of the verification tasks, the consumer verification was slower or used more memory. Table 5.1 shows these 7 verification tasks.

Next to the acronym for the refined property checking analysis and the program name, Tab. 5.1 presents the number of refinements performed by the refined property checking analysis, the verification time V_P of the producer, the verification time V_C of the consumer, the speed-up of the consumer verification, the memory used by the producer and consumer analysis, and their relation. The verification time of the producer consists of the time spent in the CPA algorithm plus the time required for refinement. Moreover, the verification time of the consumer is the time of the CPA algorithm plus the additional parsing costs for the consumer, the difference of the parsing costs of the generated and the original program. All times are given in seconds and memory consumption, used heap plus used non-heap, is represented in MB.

Looking at Tab. 5.1, we observe that only in one case, the task considering program `transmitter02`, the memory usage of the consumer is higher, although the analysis is faster. We think that the reason is the high number of merges, more than 18,000 merges. In contrast, the refined property checking analysis uses model checking and never merges. Another reason could be the larger size of the generated program. Also in the two tasks for program `pipeline2` the slow down and the higher memory usage is caused by the high number of merges, more than 993,000. Note that in two of the three cases, we observed a high number of merges, although the producer uses a model checking analysis and we showed that the consumer can use a tree ordering to verify its program in a single pass. On the one hand, a tree ordering does not exclude merging. Although an exploration ordering without merges exists, a tree ordering does not ensure that this exploration order is used. On the other hand, a correct tree ordering is difficult to compute. When we only consider the generated program, we have the structure of the ARG but no longer its abstract states.

Thus, we never compute a tree ordering, but rely on heuristics for the exploration order.

In the remaining four verification tasks depicted in Tab. 5.1, only a slow down of the consumer's verification time exists. This slow down is caused by the additional parsing costs, especially the construction of the CFA. In these cases, the consumer execution of the CPA algorithm is always more than 1 s faster than the producer's execution and the additional time for parsing the program without CFA construction is always smaller than 0.9 s. Thus, constructing the CFA from a goto program seems to be more complex. We have already observed this phenomena for loop detection, which we disabled for the consumer verification.

The presented results remain valid when we compare the consumer verification with the best verification times and memory usages for the producer verification, i.e., the more efficient refined property checking analysis of the two variants with and without lazy refinement. However, in 28 cases the producer verification becomes significantly faster. Thus, the sum of all producer times is decreased from 1,401.52 s to 1,277.8 s.

The results are also stable, when looking at the total time for verification. However, the difference between producer and consumer verification can become significantly smaller. Furthermore, we observe a relation between the fraction of the verification time on the total time and the difference between the total time of producer and consumer verification.

So far, we recognized that the consumer verification is often more efficient. The sum overall tasks is twice as fast. Now, we want to study if and when we can achieve significant improvements of the consumer verification. Since we observed that a significant memory usage decrease is typically coupled with an important speed-up of the verification time, we focus on verification times only.

Table 5.2 shows all verification tasks in which the consumer verification achieved a significant speed-up, i.e., a speed-up of at least six. The structure of the table is the same as for the previous table. Furthermore, note that due to nondeterministic refinement in some cases the refinements varied among the 10 verification runs. In these cases, the number of refinements is a floating point number rather than a natural one.

Inspecting Tab. 5.2, we observe that in approximately 35% of the verification tasks, 44 of 127, the consumer verification experiences high speed-ups. Considering the refined property checking analysis configuration, 65% of the configuration, i.e., 19 of 29 refined property checking analysis configurations, achieved high speed-ups. We notice that high speed-ups are achieved for all property checking analyses except for the model checking analysis applying a value analysis $\mathbb{V}$ with precision adjustment. Moreover, high speed-ups are possible with all three enabler analysis. This supports the general applicability of our Programs from Proofs framework.

Nevertheless, not all refined property checking analysis configurations behave the same. When looking at the enabler analysis, we see that whenever a combination of enabler and property checking analysis is missing, typically the combination with the octagon enabler analysis is missing. The predicate analysis is the enabler for which often the highest speed-ups and more often significant speed-ups are achieved. We think that the reasons are the difference between the number of refinements and how the refinement is performed. For example, we see that the predicate analysis required more refinements and the octagon analysis performs only very few refinements. From our point of view, the difference in refinements is also related to the power of the enabler analysis.

Next to the enabler analysis, also the number of refinements seems to play an important role for high speed-ups. Often, a larger number of refinements is performed when large speed-ups are achieved. However, this is not always the case. A high speed is achieved for the verification task considering model checking analysis $\mathbb{OV}$ and program `testlocks12d`,

Table 5.2: Extract of the comparison of the producer and consumer verification showing all tasks in which the consumer's verification was at least 6 times faster than the producer's verification

$\mathbb{C}_P$	program	#r	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{PI}$	nosprintf*	3	0.59	0.07	8.47	243.90	210.20	0.86
	interproc*	2	0.32	0.05	6.07	241.00	210.00	0.87
	NetBSD*	4	0.68	0.08	8.39	245.20	214.90	0.88
	PfPb*	2	0.18	0.02	7.91	235.10	205.90	0.88
	PfPc*	2	0.22	0.03	6.97	229.70	202.60	0.88
$\mathbb{OI}$	SpamAssassin	1	0.93	0.16	6.02	244.00	214.30	0.88
$\mathbb{PS}$	PfPb*	2	0.17	0.03	6.37	232.90	196.20	0.84
	fraction	4	0.52	0.07	7.26	242.30	201.40	0.83
	lockfree3.0*	14	1.97	0.12	16.70	288.40	202.30	0.70
$\mathbb{VS}$	inf6*	4	0.28	0.03	8.23	230.50	201.50	0.87
	kundu*	10	3.00	0.36	8.41	346.00	214.90	0.62
	memslave1*	22	7.52	0.91	8.31	564.00	273.20	0.48
	PfPc	2	0.26	0.04	6.74	222.70	199.70	0.90
$\mathbb{PSI}$	invertsorted*	21	9.92	0.67	14.76	565.70	233.80	0.41
	div	20	5.37	0.30	17.72	441.60	215.20	0.49
	fibonacci*	3.7	0.50	0.06	7.81	242.90	205.40	0.85
$\mathbb{PU}$	pipeline2*	230	227.81	0.23	985.32	1792.80	211.50	0.12
$\mathbb{VU}$	pipeline2*	13	2.93	0.32	9.13	367.70	221.50	0.60
$\mathbb{PV}$	testlocks5*	30.2	2.64	0.30	8.93	304.60	214.60	0.70
	testlocks5d*	8.2	0.90	0.12	7.33	257.50	215.50	0.84
	testlocks12	18	329.79	8.23	40.05	1713.20	591.80	0.35
	testlocks12d	12	5.71	0.27	21.37	540.90	219.00	0.40
$\mathbb{OV}$	memslave1	7.1	4.39	0.35	12.51	368.00	226.80	0.62
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	35	17.63	0.47	37.20	780.60	233.70	0.30
	memslave2	35	18.31	0.63	28.87	766.00	241.20	0.31
	testlocks6*	161	25.70	0.54	47.50	771.10	229.80	0.30
$\mathbb{PSI}$	fibonacci*	3	0.45	0.06	7.42	243.20	206.40	0.85
	palindrom*	18	6.02	0.48	12.65	475.70	221.20	0.46
	invertarray*	18	17.90	0.32	55.38	523.60	217.90	0.42
$\mathbb{PL}$	locks*	2	0.15	0.01	10.85	232.80	199.70	0.86
	tokenring03	59	198.41	2.84	69.79	1733.20	299.20	0.17
	memslave1*	271	155.63	4.16	37.43	1793.00	368.80	0.21
$\mathbb{OL}$	s3srvr	1.4	1.06	0.15	7.07	260.00	237.10	0.91
$\mathbb{VL}$	tokenring02	16	5.99	0.22	27.74	549.50	213.50	0.39
$\mathbb{PS}$	kundu*	20	15.15	0.60	25.06	594.00	228.50	0.38
	transmitter01*	5	1.66	0.18	9.31	278.70	206.10	0.74
$\mathbb{VS}$	inf6*	4	0.31	0.03	9.24	228.50	204.10	0.89
	kundu*	10	6.61	0.14	47.92	481.10	202.60	0.42
	transmitter02*	5	5.07	0.21	23.98	438.10	206.20	0.47
$\mathbb{PU}$	pipeline*	15	6.25	0.22	28.65	474.90	215.70	0.45
$\mathbb{VU}$	s3srvr*	6	3.81	0.61	6.23	385.60	231.90	0.60
	pipeline2*	14	3.05	0.31	9.82	370.90	220.20	0.59
$\mathbb{OV}$	testlocks5*	5	2.73	0.35	7.90	330.90	224.00	0.68
	testlocks12d*	1	135.46	0.26	520.62	1771.80	217.80	0.12

although only one refinement is required. Moreover, a high number of refinements is not a guarantee for a high speed-up. For example, the dataflow analysis $\mathbb{PU}$ on program `cdaudio` needs 32 refinements, but the achieved speed-up is below four.

All in all, the number of refinements, influenced by the power of the enabler analysis, is a factor for high speed-ups. Next to this factor, other factors must exist. The difference of the program sizes of the original and generated program might be a another factor. We study this hypothesis in the next section.

5.6.3 RQ2: How Much Larger Are The Generated Programs?

Often, we must separate control flow paths and unroll loops to obtain a more easily verifiable program. Thus, we expect that the generated program is larger. In this section, we want to study how much larger programs can become. We consider two metrics, the number of program locations and the program's file size in bytes.

Comparing the original and the generated program sizes, we observe that in 85 of 127 cases the generated program has a higher number of program locations and in 77 of 127 cases the file size of the generated program is larger. The number of program location can become 377 times larger and the file size is increased up to 406 times. Note that despite the large increase in program size, we think that the program generation is practically feasible. The times for program generation, we measured once during program generation, were always below 40% of the producer's analysis time and took on average around 4% of the producer's analysis time.

Surprisingly, the generated program sometimes also becomes smaller. In 40 cases the number of program locations in the generated program and in 50 cases the file size of the generated program is smaller. At best, the size of the generated program becomes about 30% of the original program size. The reason for the decrease is that the refined property checking analyses detect some of the infeasible, syntactical program paths. These paths are excluded from the ARG and, thus, from the generated program. Looking at the complete set of verification tasks, in total the generated programs contain about 5.8 times more program locations and their files are 4.13 times larger. Finally, we recognize that the trend, increase or decrease, is often the same in both metrics. In approximately 85% (107 of 127) of the cases the trend is the same. For the detailed numbers, we refer to Table B.6 in the appendix.

Next, we shortly want to discuss whether we obtain smaller programs when we selected the refinement strategy for which the producer verification is faster. Indeed, the faster analysis often generates smaller programs. In 118 of 127 cases the number of program locations and in 108 cases the file size is smaller. For the remaining cases, the faster producer verification variant generates at most 21% larger programs. For our examples this means not more than 101 additional location and 1856 extra bytes. On average the increase is below 0.5%, i.e., less than 1 location and 16 bytes. Hence, the verification time is a good criterion to generate the smaller program variant.

In the previous section, we found out that the number of refinements needed by the refined property checking analysis to verify the program influences the consumer performance. Now, we want to examine the influence of the program size increase. Since we added the difference in the parsing time of the generated and original program to the consumer verification time, we think that a large increase in program size is an obstacle for high performance improvements on the consumer side. To study the relation between a large performance gain for the producer and the increase of the program size, Table 5.3 shows the program sizes for all verification tasks also considered in Tab. 5.2, namely

those in which the speed-up for the consumer verification was at least 6. We use $\#\text{loc}_P$ and $\#\text{loc}_C$ to refer to the number of program location in the original and the generated program and $\#\text{bytes}_P$ and $\#\text{bytes}_C$ to refer to the respective file sizes in bytes.

Looking at Tab. 5.3, we see that in 42.5% (17 of 40) of the depicted cases, the generated program requires less program locations. In 67.5% (27 of 40) of the cases the number of program locations is increased by a factor of at most 1.5. Often, a high speed-up is achieved when the generated program is of similar size. The complete set of verification tasks supports our observation that a small increase in program size is an indicator for a faster consumer verification. In 86 of our verification tasks, the number of program locations is increased by at most two and in 84 of these 86 verification tasks, the consumer experiences a speed-up of more than two. In 74 of the 86 tasks the consumer even achieves a speed-up above three.

However, a decrease or small increase in program size is not a necessary condition. Consider the verification task for program `testlocks12` in Tab. 5.3. For this verification task, the generated program is increased by a factor of more than 140. We believe that the program increase is compensated by a higher number of refinements. Generally, we noticed that for all verification tasks depicted in Tab. 5.3 in which the number of program locations is increased by a factor of at least 4 the refined property checking analysis required at least 18 refinements.

Furthermore, we recognize that for large speed-ups beyond 20 often the program size of the generated program is increased by at most 2 or lots of refinements are needed. For the best speed-up, we have both, a decrease in program size and a high number of refinements. Moreover, if we look at verification tasks in which the producer analysis required more than five refinements and the generated program is increased by at most two, then we will observe that the consumer verification is at least 5 times faster than the producer verification. Thus, we think that the combination of the number of refinements during producer verification and the relation of the program sizes is a good indicator to estimate the performance of the consumer. We continue to examine how our PfP approach competes with the four alternative PCC approaches.

5.6.4 RQ3: Does the PfP Approach Compete With PCC Approaches?

After we found out that often the consumer benefits from the program transformation, we now examine if the Programs from Proofs framework is an alternative to existing frameworks To study this question, we compare the performance of the Programs from Proofs approach with the performance of the four competitors introduced in the evaluation setting. In detail, we compare the validation times and the memory usage. As before, the validation time of the consumer in the PfP approach is the time for the CPA algorithm plus the additional parsing costs. Since all competitors are Proof-Carrying Code approaches, their validation time consists of the total time required for certificate checking and certificate reading. In all cases, the memory usage is the required heap and non-heap memory.

First, let us compare the validation times. The complete comparison can be found in Tab. B.7 in the appendix. Table 5.4 shows all verification tasks in which the validation time of at least one competitor was better, i.e., faster. Next to the acronym of the refined property checking analysis and the program name, the validation time V_{PfP} in the PfP approach followed by the validation times of the PCC approaches are given. All validation times are in seconds. The last column shows the speed-up of the PfP approach with regard

Table 5.3: Evaluating the impact of the relation of the original and generated program size on verification tasks with consumer speed-up of more than 6.

$\mathbb{C}_P$	program	$\#\text{loc}_P$	$\#\text{loc}_C$	$\frac{\#\text{loc}_C}{\#\text{loc}_P}$	bytes_P	bytes_C	$\frac{\text{bytes}_C}{\text{bytes}_P}$
	nosprintf*	70	71	1.01	2108	1644	0.78
	interproc*	53	49	0.92	1541	953	0.62
$\mathbb{PI}$	NetBSD*	68	102	1.50	2087	2014	0.97
	PfPb*	16	14	0.88	150	139	0.93
	PfPc*	23	24	1.04	223	245	1.10
$\mathbb{OI}$	SpamAssassin	130	120	0.92	3121	2247	0.72
	PfPb*	16	14	0.88	150	139	0.93
$\mathbb{PS}$	fraction	44	62	1.41	1205	1372	1.14
	lockfree3.0*	119	67	0.56	2069	1637	0.79
	inf6*	41	24	0.59	360	244	0.68
	kundu*	402	302	0.75	7841	6046	0.77
$\mathbb{VS}$	memslave1*	1080	967	0.90	28163	52921	1.88
	PfPc	23	25	1.09	223	257	1.15
	invertsorted*	34	430	12.65	369	8601	23.31
$\mathbb{PSI}$	div	24	161	6.71	313	2803	8.96
	fibonacci*	29.7	47	1.58	301	689	2.29
$\mathbb{PU}$	pipeline2*	637	421	0.66	13473	9509	0.71
$\mathbb{VU}$	pipeline2*	637	677	1.06	13473	16070	1.19
	testlocks5*	81	227	2.80	1762	3374	1.91
$\mathbb{PV}$	testlocks5d*	73	78	1.07	1530	1019	0.67
	testlocks12	172	24646	143.29	3922	381105	97.17
	testlocks12d	150	246	1.64	3202	2964	0.93
$\mathbb{OV}$	memslave1	1080	432	0.40	28163	7891	0.28
	memslave1	1080	580	0.54	28163	18043	0.64
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave2	1087	585	0.54	28252	18186	0.64
	testlocks6*	103	606	5.88	2386	10644	4.46
	fibonacci*	29	47	1.62	301	689	2.29
$\mathbb{PSI}$	palindrom*	26	306	11.77	244	5447	22.32
	invertarray*	26	243	9.35	219	3630	16.58
	locks*	23	23	1.00	280	294	1.05
$\mathbb{PL}$	tokenring03	498	7195	14.45	8917	137888	15.46
	memslave1*	1080	12252	11.34	28163	285883	10.15
$\mathbb{OL}$	s3srvr	571.6	821	1.44	43877	19715	0.45
$\mathbb{VL}$	tokenring02	406	736	1.81	7254	14199	1.96
$\mathbb{PS}$	kundu*	402	626	1.56	7841	15487	1.98
	transmitter01*	322	205	0.64	5702	3793	0.67
	inf6*	41	23	0.56	360	225	0.63
$\mathbb{VS}$	kundu*	402	137	0.34	7841	5343	0.68
	transmitter02*	420	235	0.56	7398	4314	0.58
$\mathbb{PU}$	pipeline*	619	521	0.84	13000	12369	0.95
$\mathbb{VU}$	s3srvr*	518	1204	2.32	43877	28425	0.65
	pipeline2*	637	645	1.01	13473	15318	1.14
$\mathbb{OV}$	testlocks5*	81	292	3.60	1762	3930	2.23
	testlocks12d*	150	232	1.55	3202	2769	0.86

Table 5.4: Comparison of the consumer validation times in PfP and PCC approaches for all verification tasks in which at least one PCC approach has a smaller consumer validation time than the PfP approach. Validation times are given in seconds.

$\mathbb{C}_P$	program	V_{PfP}	V_{RV}	V_{ARG}	V_{CPC}	$V_{\text{CPC}^{\text{RP}}}$	$\frac{\min V_{\text{PCC}}}{V_{\text{PfP}}}$
$\mathbb{OS}$	testlocks7	1.15	1.26	1.46	1.32	0.98	0.85
	inf4*	0.32	0.28	0.57	0.53	0.27	0.84
$\mathbb{VS}$	memslave1*	0.91	0.99	1.42	1.33	0.77	0.85
$\mathbb{OSI}$	palindrom	0.48	0.49	0.73	0.78	0.42	0.87
$\mathbb{OU}$	pipeline2	260.19	7.13	8.81	7.88	6.76	0.03
$\mathbb{VU}$	cdaudio	2.89	2.38	3.73	3.24	2.63	0.82
$\mathbb{OSI}$	invertsorted	0.81	0.73	1.17	0.96	F	0.90
$\mathbb{VL}$	tokenring04	43.05	4.40	4.59	5.65	3.12	0.07
$\mathbb{PS}$	transmitter02	5.47	2.22	2.83	3.27	2.88	0.41
$\mathbb{OS}$	powerapprox	10.73	9.14	4.56	4.77	F	0.42
$\mathbb{PU}$	diskperf	3.64	3.14	5.83	5.02	5.12	0.86
$\mathbb{OU}$	pipeline	1.18	1.14	2.27	2.13	1.74	0.97
	pipeline2	258.16	5.63	8.65	8.17	6.61	0.02
$\mathbb{VU}$	cdaudio	3.63	3.31	3.96	3.47	2.77	0.76
$\mathbb{O}\widetilde{\mathbb{V}}$	relax*	0.47	0.40	0.94	0.74	0.53	0.84

to the best, the fastest, PCC approach. Note that some PCC approaches sometimes failed to validate the certificate either due to a bug in the implementation of the CPA or because validation took more than 15 minutes. These cases are marked with a `F` (failure) or `TO` (timeout) in the tables.

Looking at Tab. 5.4, we observe that the PfP approach is rarely worse than the PCC approaches. Only in about 12% (15 of 127) of the verification tasks, the consumer validation in the PfP approach is worse. In about half of the cases, 8 of the 15 cases, the difference is rather small, namely at most 0.4 s. In the remaining 7 of the 15 cases, the producer verification already outperformed the PfP consumer verification. The 7 cases are also the reason why the PfP approach is outperformed by almost all PCC approaches when looking at the sum of all validation times. However, if we exclude all tasks in which the PfP consumer performed worse than the producer, the sum of the PfP consumer's validation time will be the smallest with less than 70 s. The closest PCC approach is our optimized variant, which already requires around 180 s, more than twice as long. From the point of validation time, our PfP approach is definitely an alternative. Note that this remains true even when we consider the validation times for both certificate variants, certificates from analyses with or without lazy refinement, and always select the faster validation time for each verification task. The trend, validation time is faster or lower than the PfP approach, is the same. However, the speed-up of the PfP approach decreases.

Considering the total times, we recognize that the total time is dominated by the set up and parsing time. The differences between the times become smaller, but the trend slow down or speed-up remains. Moreover, the trend is relatively stable even when we select the best total time for each PCC approach instead of the time required for validation of the certificate constructed from the faster refined property checking analysis. Only in 10 additional cases, the PfP approach is outperformed by the best PCC approaches.

Table 5.5: Comparison of the consumer's memory consumption in PfP and PCC approaches for all verification tasks in which at least one PCC approach has a lower memory usage than the PfP approach. Memory consumption, used heap plus used non-heap, is displayed in MB.

$\mathbb{C}_P$	program	M_{PfP}	M_{RV}	M_{ARG}	M_{CPC}	$M_{\text{CPC}^{\text{RP}}}$	$\frac{M_{\text{PfP}}}{\min M_{\text{PCC}}}$
$\mathbb{OS}$	testlocks7	265.0	281.9	241.9	234.3	248.9	1.13
$\mathbb{VS}$	memslave1*	273.2	268.0	266.2	264.7	260.8	1.05
$\mathbb{OU}$	pipeline2	3971.2	552.8	622.9	575.2	557.8	7.18
$\mathbb{OSI}$	invertsorted	237.7	241.2	234.7	228.6	F	1.04
$\mathbb{VL}$	tokenring04	537.3	545.5	546.4	572.1	426.1	1.26
$\mathbb{PS}$	transmitter02	553.1	304.2	314.7	300.3	293.8	1.88
$\mathbb{OS}$	powerapprox	996.8	1077.9	360.0	552.6	F	2.77
$\mathbb{OU}$	pipeline2	3903.0	536.9	616.0	570.2	553.6	7.27
$\mathbb{VU}$	cdaudio	380.0	377.0	420.2	394.6	361.2	1.05

Furthermore, we observe that sometimes the validation time of the PfP approach is significantly better, i.e., at a total difference of at least 1 s it is more than twice as fast, than the PCC approach based on regression verification [BLN+13] – the approach that stores the precision. In case, the producer does not apply lazy refinement – this are all verification tasks in which the program name is not marked by an asterix –, the validation time of the PCC approach based on regression verification reflects the time required for the last iteration of the refined property checking analysis. In eight cases, we observed that the PfP approach performed better than this PCC approach and the refined property checking analysis did not use lazy refinement. Since all three enabler analyses occurred in these 8 cases, this supports our beliefs that the enabler analysis is more complex than the property checking analysis. In case, the refined property checking analysis uses lazy refinement and we observe such a significant difference, this is at least an indicator for an enabler analysis that is more complex than the property checking analysis.

Next, we want to examine if the PfP approach is also an alternative when considering memory usage. Again, the complete comparison can be found in Tab. B.8 in the appendix. Table 5.5 shows all verification tasks in which the memory consumption of at least one competitor was better, i.e., lower. Next to the acronym of the refined property checking analysis and the program name, the memory M_{PfP} used in the PfP approach followed by the memory usage of the PCC approaches are presented. Memory usage is always given in MB. The last column shows the improvement of the PfP approach with regard to the best, the lowest, PCC approach. Note that some PCC approaches sometimes failed to validate the certificate either due to a bug in the implementation of the CPA or because validation took more than 15 minutes. These cases are marked with a `F` (failure) or `TO` (timeout) in the tables.

We observe that only in 9 of 127 tasks our PfP approach uses more memory than the best PCC approach. For all nine cases, the validation costs of the PfP approach are already higher and typically the memory usage of the faster PCC approach is lower. Furthermore in 6 of the 9 tasks, the producer performed better than the PfP consumer. Like for the validation times, we notice that these 6 tasks caused that the sum of the memory usage overall verification tasks is the highest for the consumer of the PfP approach. Excluding the tasks in which the PfP consumer is worse than the producer, we again get the lowest

sum for the PfP consumer. Note that these results remain stable, although the numbers might change, when selecting for each task and each PCC approach the PCC certificate whose validation required less memory.

Finally, we want to briefly compare the storage overhead. For the PfP approach this is the difference between the file size of the generated and the original program. Of course, the storage overhead of the PfP approach is better when the generated program is smaller. However, we observed that the storage overhead is often, namely in 62 of 127 cases, worse than the regression verification based PCC approach, which reuses the precision [BLN+13]. For our verification tasks, that PCC approach required only 0.2 MB additional disk storage, in contrast to the PfP approach which required 3.6 MB. In extreme, the PfP approach required 10,000 times more disk storage for a verification task. Compared to the remaining PCC approaches, the PfP approach performs better. It is worse in only 27 cases and at most 60 times worse. In total, it is even better than the remaining PCC approaches, which require between 9.6 and 14.9 MB additional disk storage.

All in all, our PfP framework is a real alternative to achieve an efficient validation on consumer side.

5.6.5 Summary

Our experiments demonstrated that the consumer verification is typically easier, i.e., more efficient, than the refined property checking analysis and can compete with existing PCC approaches. Compared to the producer verification, in more than one third of our verification tasks we achieved a significant improvement in verification time. Memory usage can be decreased significantly, too. However, the reduction is coupled with a faster verification and is less often significant. Often, the program size is increased moderately. Sometimes, it is even decreased, but also tasks with an enormous program size increase exist.

We observed that the number of refinements, the enabler analysis, and the increase in program size play an important role to get a significantly more efficient consumer verification. Simultaneously, high speed-ups are achieved for various refined property checking analyses.

Summing up, our experiments support our claim that the PfP approach is a general alternative to get an efficient validation for the consumer.

5.7 Discussion

Like our configurable program certification framework, Programs from Proofs is a general framework, which enables a fast consumer validation. However, its requirements on the producer's analysis configuration are more strict. For example, it requires location information, the transfer relation must be a (monotonic) function, and a subpart of the analysis must be able to check the property. Hence, it does not support such a broad class of analyses like the CPC approach. Furthermore, the Programs from Proofs approach can only be used to ensure control state unaware property automata specifications. Syntactically, these specifications are a real subset of the possible property automata. From a semantics point of view, it is currently an open problem whether the language of property automata and the language of control state unaware property automata cover the same class of safety properties, e.g., every property automaton can be transformed into a control state unaware property automaton s.t. the set of programs considered to be safe by

the respective automaton is the same. In contrast to the CPC approach, the consumer's configuration can always be derived automatically. The PfP approach is fully automatic.

Due to program generation, the Programs from Proofs approach must deal with a problem that does not exist in the CPC setting: the behaviorally equivalent original and generated program are not necessarily safe w.r.t. the same property automata. Program safety can differ for property automata that are not control state unaware. Since program generation is under control of the producer, who offers the program, the producer can decide if he wants to sell a program that lacks a certain property. Furthermore, we assume that the producer applies the Programs from Proofs technique when he generally wants to support a simple, dataflow based verification of his programs's properties and these properties are expressed by control state unaware property automata. Thus, for the offered program all properties of interest should be provable with a simple dataflow analysis. We showed that if a simple dataflow analysis can prove a control state unaware property on the original program, the same analysis, when terminating, is able to prove the same property on the generated program. Simultaneously, this means that our Programs from Proofs approach is transitive.

The fact that the consumer validation fails to terminate after a successful producer verification is also a new aspect. We believe that the termination problem is rather a theoretical than a practical problem. To our mind, we proved that the consumer analysis terminates for all practically relevant analyses. Furthermore, we discuss in the next chapter how to eliminate the termination problem. Additionally, we think that the incorporation of widening and precision adjustment into the consumer analysis could solve the termination problem – at least in some cases. Note that in case of a proper incorporation, a proof similar to the termination proof in case of an equivalence relation consistent refined property checking analysis can be used to show that the consumer analysis succeeds. Nevertheless, the theoretical termination problem is the reason why not all PfP instances can be guaranteed to be relatively complete. Following the above argumentation, all practically relevant instances are indeed relatively complete.

Despite the drawbacks, a successful application of the Programs from Proofs approach guarantees a sound, simple and fast dataflow analysis based validation for the consumer. At best, only a single pass over the program is needed. For a refined property checking analysis based on model checking, we theoretically proved this limit. Also, our evaluation with a variety of configurations, properties, and programs – however no industrial size programs –, supports the idea of a fast validation. Often, the PfP validation was (slightly) better than the PCC competitors. The PfP approach is a theoretical and practical alternativ

5.8 Related Work

The Programs from Proofs framework is based on the idea of a work shift from the consumer to the producer. The same idea has already been applied by many different Proof-Carrying Code approaches. Proof-Carrying Code approaches have already been extensively discussed in the previous chapter and will not be considered here.

However, Proof-Carrying Code approaches are not the only approaches in literature that apply the idea of a work shift. First of all, previous instances [WSW13, JW15, JW17] of the Programs from Proofs framework can be found.

Wonisch et al. [WSW13] present a single instance to verify protocol properties, a subclass of the properties expressable in the property automaton. The producer uses a

predicate enabler analysis plus a location analysis for the property checking analysis and configures the refined property checking analysis to model checking. Note that Wonish et al. [WSW13] use a slightly different notion of an abstract reachability graph and, thus, the program generation differs slightly. Furthermore, the consumer does not use the CPA framework, but its own algorithm. Hence, the consumer analysis terminates in a single pass independently of the exploration order, which we additionally require.

Jakobs et al. [JW15] describe a set of instances, which uses a predicated dataflow analysis, a combination of predicate analysis with an arbitrary dataflow analysis, for the refined property checking analysis. All instances prove invariants. The program generation is the same as described in this thesis and also the theoretical results are comparable. In principle, the instances are a subset of the instances we cover with our framework. Basically, the subset contains all instances of our framework that consider invariants, use a predicate enabler analysis, do not adjust precisions, join if location and predicate state are the same, and stop if the state is covered by a more abstract one.

A previous, general Programs from Proofs framework [JW17] is quite similar to our current framework and achieves similar theoretical results. The major differences are that it does not support precision adjustment and restricts the property checking analysis' transfer relation to the most precise transfer relation.

Hunt and Sands [HS06] sketch a framework similar to the Programs from Proofs idea, but for a totally different class of analyses and properties. Their goal is to simplify type based verification of information flow security. They start with a flow-sensitive security type analysis. Then, they transform the program based on the flow-sensitive typing. Basically, the generated program uses a set of variables for each variable in the program, namely one per security type, and replaces each variable by the variable associated with the currently valid security type. Program generation is realized as an extension to the existing type inference rules. The resulting program is typeable with a more efficient flow-insensitive analysis and keeps the input-output behavior of the original program.

We are not aware of further frameworks with the same idea. Thus, we continue to discuss work related to some aspects of the Programs from Proofs framework. The Programs from Proofs framework is built on various components: an enhancement of the property checking analysis s.t. it becomes more precise and a program generation that excludes infeasible paths, is tailored to the property of interest, and relies on the proof, the ARG, constructed by the producer. In the following, we discuss related work for all those components. We conclude with a short glimpse at the role of the program structure when one wants to assure program properties or detect violations.

5.8.1 Making Analyses More Precise

We apply our Programs from Proofs approach, when a simple property checking analysis fails to prove a program's property. In this case, our Programs from Proofs approach starts to construct a more precise refined property checking analysis from the simple property checking analysis. To get a more precise analysis, we combine an enabler analysis with the property checking analysis, configure how and when to combine abstract states, and automatically compute a proper abstraction level for the enabler analysis. Next, we discuss techniques that are related to those techniques that we apply in the refined property checking analysis to get a more precise variant of the analysis the consumer performs.

Specific Analysis Combinations In this section, we study concrete combinations of analyses mainly focusing on combinations that want to make one analysis more precise. Two widespread trends for concrete combinations exist in literature: the addition of predicate abstraction and the combination of numerical and heap analysis. We start with those combinations adding predicate abstraction, continue with heap analysis extensions, and finally look at combinations that utilize the execution history.

A *predicated array dataflow analysis* [MHM98] combines predicates with array dataflow values. The dataflow values are valid when the associated predicates evaluate to true. A predicate embedding operation is used to get a more precise dataflow value and predicate extraction is needed to derive a predicate from the dataflow value.

A *predicated lattice* [FJM05] is a combination of predicate analysis and an arbitrary dataflow analysis. The idea is that a set of predicates separates the state space and, thus, refines the dataflow analysis. Hence, an abstract state is a map from a set of predicates to dataflow facts. During a merge, dataflow facts with the same predicate are joined. Furthermore, the set of predicates is automatically computed with a CEGAR scheme.

Gurfinkel and Chaki present the interface NumPredDom [GC10], which allows to define combinations of predicate and numerical abstract domains. Next to the standard operations on an abstract domain like join, meet, widen, abstract postcondition, etc., further methods like the projection to predicate or numerical abstract state and a reduce operator must be implemented for the interface NumPredDom.

Winter et al. present a *predicative backward dataflow analysis* [WZH+13] which combines path predicates with dataflow facts to analyze an instrumented program. In their setting dataflow facts are the assertions in the instrumented program and are encoded by predicates, too. Furthermore, the transfer relation is based on the weakest precondition. To scale the analysis, conjunctions of path predicates are simplified or abstracted as unknown.

Mihaila and Simon [MS14] combine a predicate with a numerical abstract domain. Predicates temporarily capture information lost during a join of two numerical states and are later used to reestablish lost information, i.e., make the numerical state more precise. Typically after assume statements, a reestablishment of lost information is tried out.

Blazy et al. [BBY14, BBY16] propose a predicated analysis, which is based on a *predicated domain*. A predicated domain is a combination of a predicate and another abstract domain. Abstract states consist of two parts: a mapping from predicates to abstract values and a context describing which predicates are valid. If the context determines that a predicate in the mapping is true, the corresponding abstract value will be valid, too.

Transition relation strengthening [JIG+06] is a technique to compute stronger postconditions in predicate abstraction. The idea is to use a subset of the invariants, the fixpoint, computed by a previous static analysis. More concretely, the conjunction of predecessor state and the corresponding invariant is built. Then, this conjunction is used to compute a stronger postcondition.

Beyer et al. [BHTZ10] combine an explicit heap analysis with shape abstraction. For all pointers that are tracked, the analysis starts with the explicit heap. After a finite number of steps the analysis utilizes the explicit heap information to determine the initial shape class for that pointer. From now on, shape abstractions are used for that pointer.

Magill et al. [MBCC07] combine separation logic with arithmetic abstraction to prove memory safety. If a property cannot be proven with separation logic, an arithmetic program will be derived from the abstract transition system of the separation logic analysis. The derived program is structurally equivalent to the abstract transition system, but program statements are modified s.t. they do not reference to the heap. Then, an arithmetic

analysis is run on the derived program and the resulting invariants are combined with the separation logic invariants to get a more precise abstract transition system.

Ferrara et al. [FFJ12, Fer14, Fer16] discuss the combination of heap and numerical analysis. The numerical analysis tracks information about variables and heap identifiers while the heap analysis is responsible for the abstract representation of the heap state. If the heap analysis changes the heap identifiers, it must inform the numerical analysis. The numerical analysis uses the information how heap identifiers changed to adapt its state accordingly.

Ferrara et al. [FMN15] also present a different combination of heap and value analysis. They propose an abstract interpretation for a heap analysis that is parametrized by a value analysis. The heap is presented by a graph with incorporated value information. Based on edge-local identifiers, graph edges are annotated with value information about source or target field values and their relation.

Handjieva and Tzolovski [HT98] propose a partitioned analysis of program paths, i.e., the analysis separately analyzes paths with different (abstract) control flow. The idea is to combine an abstraction for sets of control flow paths with an arbitrary abstract domain. Sets of control flow parts are described by a regular expression on branch node or loop head outcomes. A test approximation parameter can be used to control the regular expressions – and thus the partitioning – created during analysis.

Trace partitioning [MR05, RM07] is a framework to partition execution traces based on their history of memory or control flow states. The partitioning may even be adapted during analysis. An abstract state represents one element, a set of traces, in a particular partition. The partitioning abstract domain [RM07] defines how to combine a static trace abstraction, which at least separate traces based on their last control flow state, with an arbitrary abstract domain. In principle, the partitioning abstract domain separates abstract states with different trace abstractions.

So far, we have not considered heap analysis instances for our Programs from Proofs framework. Although, we have not used the concept of transition relation strengthening [JIG+06], our framework allows a similar refinement of the predicate successor state. In principle, one may use a strengthening operator to refine the predicate successor state based on the property checking successor state.

Many of our refined property checking analyses used during evaluation are combinations of a predicate enabler analysis and a property checking analysis. Furthermore, earlier instance of the PfP framework [WSW13, JW15] were restricted to a predicate enabler analysis. However, it is essentially for our framework that the predicate analysis cannot refine the remaining abstract values as suggested for predicated array dataflow analysis [MHM98] or by Mihaila and Simon [MS14]. In contrast to Winter et al. [WZH+13], our property checking states are not encoded by predicates. Additionally, we do not consider mappings from predicates to abstract states as done in predicated lattices [FJM05] and predicated domains [BBY14, BBY16], but we explicitly separate states with different predicate states. This separation is the key to generate a more easily verifiable program.

Finally, some of our refined property checking analysis instances apply some kind of trace partitioning. They separate abstract states with different enabler states, i.e., we separate certain memory valuations, but we never apply a partitioning as described by Handjieva and Tzolovski [HT98].

Frameworks for Combination of Analyses So far, we looked at specific analysis combinations, which aim at refining a particular analysis. We continue to examine general frameworks for combinations whose purpose is to define more precise analyses.

Cousot and Cousot [CC79] propose two techniques, *reduced product* and *reduced cardinal product*, for a combination of two analyses that is more precise than the product combination and the cardinal power[12], respectively. Both techniques remove duplicates from the combined domain, i.e., for all abstract states that abstract the same set of concrete states only one representative is kept. All operators are adapted to use the reduced version of the combined domain.

Another proposed combination is the *open product* [CLCVH94, CCH00] for the analysis of logic programs, which refines the transfer relation. The idea is to use test queries on both abstract states to determine which transfer relation to use for each component. Test queries may also be used to refine transfer successors.

Lerner et al. [LGC02] use the concept of an *integrated analysis* to combine two analyses. The transfer relation of an integrated analysis either computes an abstract successor or a replacement graph. A replacement graph describes a behaviorally equivalent replacement of the analyzed program part and often corresponds to the transformation a compiler optimizer would do. If a replacement graph is returned, the transfer relation must be applied again, but now on the replacement graph instead of the program part analyzed previously. In a combined analysis, all components must reapply the transfer relation whenever one component returns a replacement graph. Thus, the analysis information of a single component is propagated to other components via the replacement graph.

Gulwani and Tiware introduce the concept of a *logical product* [GT06] of lattices to combine two abstract interpretations. Both domains must be given in form of a logical lattice over a theory. The theory defines the abstract states, namely all logical conjunctions of atomic facts in the theory. The logical product of two logical lattices combines the two theories s.t. the abstract states in the composed domain are all conjunctions of sets of atomic facts belonging to one of the theories and it defines a proper partial order.

The Astrée static analyzer [CCF+07] builds a hierarchy of abstract domains using unary and product functors to combine domains. The hierarchy defines which information from other domains a domain can use to refine its own results. Each analysis provides extract and refine operators to provide information to other domains and improve its results based on information obtained from other domains. Information between domains are exchanged in an abstract message domain.

A *composite program analysis* [BHT07] allows to flexibly compose two configurable program analyses. The abstract domain is the product domain, but the transfer relation may strengthen each abstract successor element based on the other abstract successor element. Additionally, merge and termination check can be configured, e.g., sharpened. Instances discussed by the authors cover e.g. the product combination, a predicated lattice instance, and a combination of a predicate and a shape analysis in which the transfer relation strengthens the shape analysis state.

A *composite CPA+* [BKW10] composes two configurable program analyses with precision adjustment in a flexible way. Like in the composite program analysis, the abstract domain is the product domain, the transfer relation may use strengthening, and also merge and termination check operator can be configured. Furthermore, the precision adjustment operator can be configured, too. Instances discussed by the authors use the precision adjustment operator to switch from an explicit to a symbolic analysis of values or the heap after a certain threshold is reached.

McCloskey et al. [MRS10] present a combination of basic domains that share facts with the help of user defined predicates. A predicate is associated with one domain, but

[12] For example, the predicated lattice is one form of a cardinal power.

predicates may refer to predicates of other domains. Additionally, shared information is considered by the operations of each component domain.

Typically, the discussed frameworks are more flexible than the composition of enabler and property checking analysis in the Programs from Proofs framework. They allow that all components of the abstract states may be refined based on the information provided by other analyses [CLCVH94, CCH00, CCF+07, BHT07, BKW10, MRS10] or do not use a product abstract domain [CC79, GT06]. For our Programs from Proofs approach, it is important that a product abstract domain is used and that the enabler state cannot refine the property checking state. Finally, we would like to mention that a refined property checking analysis belongs to a strict subclass of a composite CPA+.

Techniques for a More Precise Combination of States In case of abstract domains with infinite height, the most precise combination of abstract states, the join, cannot guarantee termination of the analysis. A less strict operator, typically a widening operator [CC77], for the combination of abstract states must be used. Often, widening operators are too relaxed and result in a large set of false alarms. All approaches discussed next try to define a combination of abstract states that is more strict than a standard widening operator.

Bagnara et al. [BHRZ05] present a theoretical framework to derive a more precise widening operator from an existing one based on a limited growth ordering of the abstract states. They consider a framework instantiation for the domain of convex polyhedra.

Lookahead widening [GR06a] is a technique to improve the analysis result. The idea is to widen different loop phases separately. To achieve this goal, the analysis considers a pair of abstract states – both belonging to the same domain. The first, the main value, determines the control flow to consider in the current loop phase and is never widened during a loop phase. The second, the pilot value, tries to establish a stable state for the current loop phase. When the pilot value gets stable, it replaces the main value and the next loop phase is analyzed.

Simon and King describe *widening with landmarks*. The idea is that a landmark describes a behavior that is not possible so far and an estimate how many iterations are needed to enable that behavior. Based on these landmarks, widening is bound to an overapproximation of the join, which enables that behavior with the fewest number of iterations.

Laviron and Logozzo suggest *hints* [LL09], a theoretical concept for a refinement of an operator like join or widen. They present a set of examples for such refinements based on syntactical or semantic program information.

Localized widening [AS13] improves widening at loop heads. Instead of joining the abstract state entering a loop with the abstract state obtained from the analysis of the loop body and then apply widening, widening is restricted to the abstract state resulting from the analysis of the loop body and thereafter the information entering the loop is joined.

Apinis et al. [ASV13, ASS+16] introduce a *combined widening and narrowing* operator, which allows to interleave widening and narrowing. Widening is applied if the abstract state that should be integrated with the existing state is larger. Similarly, narrowing is used when that state is smaller.

Gulavani and Rajamani [GR06b] propose to compute precise joins whenever necessary to prove a property. They use a *hint set* that saves in which iteration a precise join instead of widening must be used. The hint set is determined with a CEGAR scheme. Initially, the hint set is empty. Whenever a counterexample is found by the analysis, it is tried to

find an iteration in which widening is responsible for the counterexample. When such an iteration is found, the iteration is added to the hint set, otherwise the analysis failed.

Gupta et al. present *extrapolation with care sets* [WYGI07], a generalized widening operator that does not guarantee termination of the analysis and tries to avoid to generate abstract states of a particular form. The care set describes which states should be avoided. More concretely, extrapolation with care sets guarantees that the result of the extrapolation does not contain any state described by the care set if its input states do not contain such a state. Furthermore, Gupta et al. discuss how to use counterexamples to iteratively compute a proper care set for a polyhedra analysis.

Gulavani et al. introduce *interpolated widening* [GCNR08] to limit the upper bound computed by the widening operator. Interpolated widening ensures that if both input states are more precise than an interpolant considered by the interpolated widening operator, its result will not be larger than that interpolant. The set of interpolants considered by the interpolated widening results from the analysis of spurious counterexamples found so far during analysis.

Not all presented improvements for widening are applicable in our framework. Combined widening and narrowing [ASV13, ASS+16] is not possible because we require that the result of a merge is always as precise as the already explored state (second argument). Furthermore, we cannot realize localized widening [AS13] because we typically cannot distinguish along which edge an abstract state is computed. Moreover, we excluded widening in the PfP consumer analysis. The combination of abstract states is always a join. In contrast, the merge behavior of the refined property checking analysis can be adjusted to some form of widening. However, we think that widening of the property checking state does not make much sense because the consumer analysis uses a variant of the property checking analysis with precise joins. Additionally, we excluded widening for some program and refined property checking analysis combinations in case we want to ensure termination of the consumer analysis. Furthermore, we do not always want to combine abstract states. In practice, we do not use widening. We either join or do not combine two abstract states. Hence, we selectively merge. Techniques for selective merge are discussed next.

Selective Merge In this paragraph, we consider approaches in which the analysis only sometimes merge states with same locations.

Property simulation [DLS02] is used to verify temporal safety properties encoded by a finite state machine. An abstract state consists of an execution state and a property state, a state of the finite state machine. In property simulation mode, the analysis merges abstract states if the property state is the same, i.e., only the execution states must be merged. In a first realization of property simulation [DLS02], constant propagation is used for the execution state. Later [HYD05], a symbolic execution state is used. To further improve property simulation, Das et al. [DDY06] suggest to extend the abstract states with additional predicates and only merge if the predicate evaluation and the property state are the same. Predicates are detected with a heuristic CEGAR scheme. Another adaption of property simulation, called value flow simulation [DADY04], uses value flow facts, may and must value alias sets, instead of a property state.

Gupta et al. [SISG06] propose the concept of an *elaboration*, a structural unfolding of the control flow graph, and suggest to compute the fixpoint for the elaboration. Instead of constructing the elaboration, they use a merge heuristic during fixpoint computation to decide whether an abstract successor should be merged or replicated.

Next to interpolated widening, Gulavani et al. [GCNR08, GCNR10] also suggest to do not join the results of different branches of an if statement when a join has been responsible

for a spurious counterexample.

The technique proposed by Handjieva and Tzolovski [HT98], which does not join paths with different (abstract) control flow, and trace partitioning [MR05, RM07], which does not combine abstract states with different trace abstractions, can also be seen as one form of a selective merge.

In our PfP framework, we can configure when and how to merge in the refined property checking analysis. While we can configure the merge operator s.t. it always combines abstract states for same location or it never combines abstract states, we only considered the latter case in our experiments. More often, we use a selective merge, which merges in case the enabler state, the location state, and the property automaton state are identical. For some configurations, we also required that parts of the property checking state are the same. Thus, we often use a combination of trace partitioning [RM07] and property simulation [DLS02] for our selective merging.

Refining the Abstraction Level Now, we consider automatic approaches to refine the abstraction level of an analysis.

Counterexample-guided abstraction refinement (CEGAR) [CGJ+00, CGJ+03] was proposed for automatic, iterative abstraction refinement of labeled Kripke structures in the context of temporal ACTL* verification. The process starts with an initial abstraction. When verification fails, it is first checked whether the counterexample can be replayed in the original Kripke structure. If the counterexample cannot be replayed, i.e., it is spurious, the predecessor of the spurious transition will be considered and a refined abstraction will be built, which separates those states that can reach that predecessor from the remaining states.

Different variants and optimizations of the basic CEGAR scheme exists. *Lazy abstraction* [HJMS02] computes a non-unique abstraction. If an error is detected during construction of the abstract state space and it is not real, a pivot node, a node for which the concretization of the counterexample fails, will be searched. The subtree of the abstract state space beneath the pivot node is deleted and the abstraction from the pivot node and its descendants is refined s.t. the spurious counterexample is excluded. Gupta and Clarke [GC05] generate a set of *broken traces* for a counterexample. A broken trace is a sequence of pairs of concrete states s.t. the states in each pair belong to the same abstraction state and the second element of pair i can reach the first element of pair $i+1$. When all of the generated broken traces break, i.e., a pair exists in which the two states are not identical, the broken traces are added to the set of all generated broken traces and a new abstraction is built based on that set. Beyer et al. [BHMR07] suggest to use a *path program*, a restriction of the original program to locations and transitions occurring in the counterexample, instead of the counterexample, a single path. Furthermore, the refinement procedure should consider an invariant of the path program, the so called path invariant. Gupta et al. [GPR10] propose a *non-monotonic refinement* scheme. The idea is to consider all spurious counterexamples detected so far and to compute any abstraction that excludes all those spurious counterexample. Beyer et al. [BLW15c] utilize *sliced prefixes* of infeasible counterexamples. Sliced prefixes ignore some assume statements in the counterexample, i.e. some assume statements are replaced by true, but only sliced prefix are considered that remain infeasible. Each such sliced prefix is used to get a refinement candidate and the analysis then chooses the best candidate.

Some refinement techniques are directly tailored to abstract interpretation alike analyses. Cousot et al. [CGR07] describe an iterative refinement of an abstract domain based on the results of a forward least fixpoint and a backward greatest fixpoint. Manevich et

al. [MFH+07] present a theoretical setting for a localized, refinement of abstract interpretations. An abstraction associates one abstract domain with each program location and a refinement is another mapping from locations to abstract domains s.t. for each location the new abstract domain is as least as precise as the previous and the abstract semantics excludes the counterexample. Zhang et al. [ZNY13] try to find the coarsest abstract transfer relation in a class of transfer relations that can be used within a dataflow analysis to prove the desired property. The techniques iteratively restricts the available transfer relations. In each iteration it runs the dataflow analysis with the coarsest transfer relation available. If the analysis fails, the counterexample will be analyzed to exclude all transfer relations that produce the same counterexample.

The approaches presented so far are mainly general. In our Programs from Proofs frameworks we want to use existing refinements for concrete enabler analysis. In the following, we discuss some refinement techniques for concrete abstract domains.

Henzinger et al. [HJMM04] discusses an approach to automatically and locally refine *predicate abstraction* in a CEGAR style. The idea is to encode a counterexample as a trace formula, which conjoins the constraint of each statement on the counterexample. When the formula is infeasible, the idea is to split the trace formula at each location in the counterexample and use craig interpolation to determine the additional predicates for that location. Leroux et al. [LRS16] discuss *interpolation abstraction*, a technique to guide the computation of interpolants. Chao et al. [WYGI07] heuristically refine an abstraction based on variable hiding and predicate abstraction. The heuristic may suggest to track additional variables or to add new predicates.

Beyer and Löwe [BL13] introduce an interpolation based CEGAR scheme for an *explicit value analysis*. The abstraction level is determined by the set of variables that are precisely tracked.

Loginov et al. [LRS05] use inductive learning to refine *shape abstractions* with new instrumentation relations. At an information loss point, they increase the set of examples of the learner by a temporary structure built by the abstract transformer before creating the canonical abstraction.

Beyer et al. [BHT06] discuss lazy refinement for a combination of *predicate and shape abstraction*. The refinement procedure refines the predicate abstraction in a standard way when the trace formula is infeasible. If the trace formula is infeasible, an extended trace formula will be constructed, which also includes shape information. Based on the extended trace formula it is tried to refine the shape abstraction. Refinement of the shape abstraction first tries to add more pointers and thereafter refines shape classes.

We have not developed our own refinement technique for enabler analyses. In our Programs from Proofs approach, we want to use existing techniques to refine our enabler analysis. So far, we utilize predicate abstraction refinement similar to the approach of Henzinger et al. [HJMM04] but in combination with adjustable block encoding [BKW10]. Furthermore, we used the CEGAR scheme for an explicit value analysis [BL13] in combination with sliced prefixes [BLW15c] and a variant of that scheme to refine the octagon domain.

5.8.2 Program Transformation

Program transformations are used in various contexts. For instance, Dershowitz and Manna [DM77] use program transformations to derive a program from an existing one. They suggest to define the transformation based on an analogy between the specification of the existing program and the specification of the program, which should be derived.

Slicing (see e.g. [Tip95]) extracts a subprogram or a set of program statements that influence a slicing criterion, e.g., a set of variables at a certain location. Typical applications for slicing are debugging, program analysis, software maintenance, and testing. For example, Harman et al. [HFH+99] use amorphous slicing to simplify a test program for a hypothesis on program behavior. Jaffar et al. [JMNS12, JM14] present a technique that improves backward slicing. First, they construct a symbolic execution tree annotated with variable dependency information. Whenever the symbolic execution analysis backtracks, their approach computes dependency information, determines witnesses for the dependency information, and extracts interpolants for infeasible paths. Furthermore, the analysis stops the exploration of a node n whenever a node n' considering the same program location exists, n excludes at least all infeasible paths of n' and at n at least one witness for each dependency information is valid. Originally [JMNS12], they used the dependency information from that symbolic execution tree to slice the original program. Later [JM14], they directly transform that annotated symbolic execution tree into a path-sensitively sliced control flow graph, the program slice.

During *refactoring* [MT04] programs are transformed into programs with same external behavior, e.g., same input output semantics, but with improved internal or external quality attributes like maintainability, extensibility, or complexity.

Compiler optimizations (see e.g. [Sch73]) try to transform a program into a behaviorally equivalent but more efficiently executable program. Kirner and Haas [KH14] propose a technique to reuse existing test sets of the original program for the optimized program while still obtaining the same coverage metric results. Basically, they use coverage profiles to describe which compiler optimizations preserve the coverage metric of a test set during compilation and restrict the compiler optimizations to those that keep the metrics. We continue with program transformations that are beneficial for program validation and analysis.

Program Transformations Removing Infeasible Paths In this paragraph, we consider program transformation approaches that remove infeasible paths from a program.

Control refinement [GJK09] is a semantics and bound preserving transformation of multi-phase loops. First, the loop body is flattened into a set of loop paths s.t. all paths only contain assumes, assignments, or loops, but no choices. With the help of invariants, it is found out how the loop paths may be combined.

Balakrishnan et al. [BSIG09] propose a refinement of loops based on a partitioning of the loop into *fragments*. They start with a presentation of the loop in which all fragments can be interleaved arbitrarily. Then, they use abstract interpretation to determine and exclude infeasible sequences of the fragments.

Brauer et al. [BKKN10] also refines the control structure of a loop. For each syntactical path through the loop body, they compute a pre- and a postcondition transformer. Then, they subsequently build the new loop structure as follows. Start with a node for the initial condition. If the intersection of the loop path's precondition and a node's condition is not empty, i.e., the loop path is feasible, compute the postcondition of the loop path for that intersection, add a node for that postcondition if none exists and add an edge between the two nodes.

Sharma et al. [SDDA11] present an approach to transform a multi-phase loop into a behaviorally equivalent sequence of single phase loops. The approach is based on *phase splitter predicates*. The phase splitter predicate is used to transform a loop into two loops s.t. the loop condition of the first is conjoined with the negated phase splitter predicate and the condition of the second loop is conjoined with the phase splitter predicate. Thereby,

the phase splitter predicate ensures that the semantics are presereved and that one if statement exists whose branch evaluation is fixed in each of the two loops but the branch evaluation differs in both loops. This if statement can be simplified after splitting the loop. Candidates for phase splitter predicates are the weakest preconditions of each branch condition.

An early approach, which removes infeasible paths from programs, is dead code elimination [Ken78]. Dead code elimination is applied during the compiler optimization phase to remove unreachable code.

Bodík et al. [BGS97] improve a definition-use analysis based on detected, infeasible paths. First, branch correlations are used to detect infeasible paths, but instead of transforming the program the infeasible paths are marked in the control flow graph. The subsequent definition-use analysis uses the information in the marked control flow graph to ignore detected infeasible paths.

Static language refinement [BSI+08] uses forward and backward abstract interpretation to detect and exclude infeasible paths. In their first approach, they use backward abstract interpretation to compute for each node in the control flow graph an overapproximation of the set of states that can reach that node from the initial location. Based on these sets, they use SAT techniques to identify sets of nodes that cannot be on a feasible path together because they require conflicting sets of initial states. All paths that consider such a complete set of nodes is excluded from the original program. In their second approach, they integrate a simplified version into a forward analysis procedure. Iteratively, it performs a two step algorithm. First, it computes a forward and backward fixpoint of the current program. Second, it detects all nodes for which the forward and backward fixpoint result contradicts and removes all nodes that are control dependent on that node. The procedure stops when the property is proven or no further nodes can be deleted.

Heizmann et al. [HHP09] start with an intersection of the control flow automaton and the complement of the property automaton. Whenever a spurious counterexample is found, they construct a canonical interpolant automaton, which is derived from the sequence of interpolants on the counterexample. Then, they intersect the interpolant automaton with the automaton under verification to exclude infeasible paths including the spurious counterexample. This intersection is considered for verification in the next step.

Fehnker et al. [FHS09, FHS10] apply a similar approach. They start with an interval automaton, an abstraction of the program. Whenever a counterexample is found, they set up an interval equation system for the counterexample to detect whether it is feasible. In case the counterexample is infeasible, they build an observer that excludes some infeasible paths including the spurious counterexample and that is combined with the current abstraction.

Also Junker et al. [JHFK12] stick to this principle. They start with an annotated control flow graph. Whenever a counterexample is found, they compute the weakest precondition of the counterexample to detect whether it is feasible. For infeasible counterexamples, they also build an observer to exclude all paths with the same infeasible subpath as the counterexample and combine the observer with the current abstraction.

Our program generation in the PfP approach may also remove infeasible paths when such paths are detected during program verification, i.e., an infeasible path does not occur in the abstract reachability graph. Furthermore, in the PfP approach removal of infeasible paths is not restricted to loops. Removing infeasible paths can be seen as one heuristic approach to simplify program verification, the main aim of our program transformation. However, removal of some infeasible paths is simply a byproduct of our transformation

for simplification. In the following, we study approaches that directly apply program transformation to simplify program validation.

Program Transformations Simplifying Validation In this paragraph, we mainly cover approaches that transform a program to facilitate or ease program validation.

Cook et al. [CKV11] encode a verification task, namely if a program P satisfies a temporal logic formula, into a program P_e. If the program P_e encoding the search always returns true, program P will satisfy the temporal logic formula. Cook et al. suggest to use a program analysis to check that program P_e never returns false.

Fioravanti et al. [FPP05, DFPP13a, DFPP13b, AFPP14, DFNP14] encode verification problems as constraint logic programs. Then, they iteratively use transformation rules like folding, unfolding, goal replacement definition, etc. to transform the constraint logic program into another equisatisfiable constraint logic program until the verification result can be determined. A first instance [FPP05] encodes the validity of a CTL formula on a Kripke structure. Later instances [DFPP13a, DFPP13b, AFPP14, DFNP14] focus on safety verification of imperative programs. To this end, the imperative program, its semantics, and the reachability of an error property is encoded. The first transformations remove the interpreter part. Thereafter, the initial and error conditions are propagated. Iterated specialization [AFPP14] suggests an alternating propagation of initial and error conditions. To further improve the verification, iterated specialization can be combined with interpolation [DFNP14].

Testability transformation [HHH+04, MBH05, HHF05, LF11] aims at simplifying test set generation. The idea is to transform the program and possibly the test criterion, e.g. statement coverage, s.t. a test set that fulfills the test criterion on the transformed program will also fulfill the test criterion on the original program. However, the transformed program need not be semantics preserving. Proposed instances replace boolean variables (flags) in conditions by integer variable expressions [HHH+04, LF11], transform unstructured programs into semantics preserving, structured programs without exit statements [HHF05], or remove nested if statements [MBH05].

Korel et al. [KHC+05] use a variant of testability transformation. They use a data dependence analysis to identify all statements that are considered by an automatic test tool during test data search for a target statement. Only those identified statements are added to the transformed program. Then, the transformed program is used to guide the test case search for the original program.

Jiang and Su [JS08] present an approach similar to testability transformation. They use statistical data on error paths which were observed during the execution of a system. This data is used to detect which branches are likely taken in a failure case. Based on this information, they remove unlikely branches to get a failure, but not semantics preserving program for which testing and analysis is easier.

Ball et al. [BMMR01] propose a method to transform a C program into a boolean program that is structurally equivalent to the C program and represents a predicate abstraction of the C program. The boolean program is then checked with Bebob [BR01].

Off-card code transformation [Ler02] of Java byte code transforms Java bytecode into equivalent Java bytecode which fulfills the assumptions of Leroy's efficient bytecode verifier. The transformation of valid standard Java byte code becomes necessary whenever the efficient bytecode verifier should be used to validate the bytecode.

Verification refactoring [YKNW08, YKW09] is used in the Echo framework [SYK05], more concretely, during reverse synthesis [YKNW08], to facilitate or simplify subsequent verification. During verification refactoring, a set of transformations, which provably pre-

serve the semantics, is applied to reduce the program complexity. The transformations are selected from a given set by a user or heuristics and often undo optimizations. Complexity metrics are used to detect whether the program complexity is reduced sufficiently or further transformations must be applied. However, a specific complexity value does not automatically guarantee easy verification.

Further approaches use program transformation to simplify worst case execution time analyses. Negi et al. [NRM04] sketch two ideas for such code transformations, which e.g. try to remove infeasible paths. Puschner [Pus02] transforms a program for which the worst case execution time can be computed into a semantics preserving single path program. Single path programs have only a single execution path and no data dependent branches. Thus, their worst case execution time is simpler to compute. The transformation converts loops into counting loops and for if statements both branches are executed and afterwards conditional move statements are used to establish the correct state. Chen et al. [CML13] transform a regular expression of a program's paths into a set of regular expressions without choice and * operator. The resulting set of regular expressions describes the same set of executable program paths as the original one. However, some infeasible paths may be excluded by the resulting regular expression.

Travkin and Wehrheim [WT15] transform parallely composed programs to use verification techniques assuming sequential consistency for the verification of the original program under total store order semantics. They separately transform each sequential component of the composition. More concretely, they compute a store buffer graph, an abstraction of the store buffer behavior for that component, and generate the new component from the store buffer graph.

While some of the presented transformations are not behavior preserving, our program generated from the original program is provably trace equivalent to the original program. In contrast to verification refactoring [YKNW08, YKW09], the transformation is fully automatic. Furthermore, we provably achieve a simpler verification, a property that approaches like verification refactoring [YKNW08, YKW09] and testability transformation [HHH+04] do not guarantee. Many of the approaches rely on fixed transformation rules, but in our PfP framework a previous verification determines how the program is restructured, i.e., our transformation highly depends on the program property and the analysis configuration.

Program Transformations Increasing the Analysis Precision In this section, we consider program transformations that are applied to get a better analysis result and are not discussed in one of the previous two sections.

Holley and Rosen [HR81] present two approaches that use a set of qualifiers to improve dataflow analyses. *Data flow tracing* transforms a control flow graph into one that explicitly contains the qualified paths. The new nodes are pairs of control location and qualifier. An edge is introduced when both parts agree on the transition. *Context tupling* is similar to Fischer's predicated lattices and uses maps from qualifiers to data flow facts as abstract states.

Ammons and Larus [AL98] profile a program to build a *hot path graph* to achieve better optimizations of paths often executed in a program. The idea of the hot path graph is to separate hot program paths, which are often executed. Then, they analyze the hot path graph and combine nodes that refer to the same program location and either both nodes are not hot or no data flow fact of a hot node is reduced.

Thakdur and Govindarajan [TG08] transform a program to get a program on which a given dataflow analysis is more precise. Their approach is built on the concept of

destructive merges, nodes with a precision loss due to a merge. The idea is to split such nodes and the following descendants as long as the analysis may gain precision from the split.

Similarly, *property-oriented expansion* [Ste96] is a technique to get better optimized programs. The idea is to expand the program model and duplicate a location node when two different dataflow facts must be merged. Thus, the expanded program model corresponds to an abstract transition system obtained when model checking the original program model with the dataflow analysis domain. Finally, the optimization is applied to the expanded program model.

Control node splitting [MMM12] is an approach to get a behaviorally equivalent transformer automaton, a model of the program (behavior), from an existing transformer automaton. The approach uses a partition of the set of all values to split a node into a set of nodes, one for each set of values in the partition. Partitions of values are determined heuristically.

We definitely do not consider profiling data during our transformation. Furthermore, we do not explicitly determine if we lose precision during merge, but we rely on the refinement to incorporate enough information into the enabler analysis state to separate two property states whenever a combination would result in a information loss significant for property verification. Depending on our configuration, especially the configuration of the merge operator, one might argue that the refined property checking analysis does some kind of qualified dataflow analysis [HR81] and control node splitting [MMM12]. For example, the pair of enabler and property automaton state correspond to a qualifier or define a partition of the set of concrete states.

5.8.3 Proofs and Program Extraction

The *proofs as programs* paradigm (see e.g. [BC85]) relies on Howard's observation [How80] of the correspondence between mathematical proofs and programs, namely the similarities between the rules of inference in mathematical proofs and the rules for term construction in the lambda calculus. Many theorem provers build on the proofs as programs paradigm to offer a program extraction mechanism. Paulin-Mohring and Werner [PMW93] define an extraction of ML programs from Coq proofs. A revised program extraction [Let03] in Coq allows to extract programs in ML, Hasekell, or OCaml. Berghofer and Nipkow [BN02] introduce a functional (ML) program extraction for a subset of Isabelle/HOL. Later, Berghofer [Ber03] describes the general framework for program extraction in Isabelle and Constable and Moczydłowski [CM06] presented an alternative extraction for constructive Isabelle/HOL. Benl et al. [BBS+98] discuss program extraction in the interactive theorem prover MINLOG. Basin [Bas91] applies the proofs as programs principle to hardware circuits and synthesize circuits from constructive proofs in Nurpl.

In *code-carrying theories* [JSS07] the program and its correctness proofs are formulated in a theorem prover. Thereafter, the real program is extracted from the definitions in the theorem prover. *Code-carrying theory* [VM08], an alternative for Proof-Carrying Code, makes use of this principle. The consumer receives the definitions and proofs and extracts the executable program itself.

Lensink et al. [LSvE12] present a technique to extract Java code annotated with JML from a PVS specification.

All discussed program extraction methods get a mathematical description of the program, either a proof or a definition. In contrast, the Programs from Proofs approach extracts its program from an abstract reachability graph, a model of the abstract state

space of a program.

5.8.4 Program Structure and Program Properties

The program structure plays an important role when one wants to assure program properties or detect violations. For languages like SPARK [Bar12], which forbid the use of pointers and exceptions and only allow side-effect free functions program, verification is much simpler. Often, the program structure influences the state space of a program. This is for example observed by Groote et al. [GKO12], which present specification guidelines to build models with small state spaces. Furthermore, program properties are often related to certain programming *patterns*.

Bug patterns [FNU03, HP04, ZZ07, HK13] describe code patterns that do not adhere to common coding practice and are likely an error. In the literature, one can find descriptions of concurrency bug patterns [FNU03, HK13], bug patterns for Java [HP04] or for aspect oriented programming like AspectJ [ZZ07]. Tools like FindBugs [HP04] and COBET [HK13] use these bug patterns to efficiently detect likely errors.

Patterns are not only used to detect errors. Code patterns can be used to detect infeasible branches [DZB14]. Tools like Checkstyle [Bf15] use patterns to detect whether a program violates a given *coding style*. Coding styles are used to improve readability and understandability of code, an important aspect for the maintainability property of a software. Moreover, *design patterns* [JGVH95] describe approved solutions for well-known programming problems and help to develop software solutions with better quality properties – not necessarily functional properties. In contrast, *antipatterns* [BMMIM98] describe programming solutions that are known to lead to problematic programs with poor quality.

Our Programs from Proofs approach does not use patterns. However, it restructures a program s.t. verification becomes simpler.

6 Integration of PfP and CPC

In the previous chapters, we looked at two alternative, configurable frameworks. Both aim to lower the validation costs of the consumer. To further improve the consumer validation, we want to combine the two frameworks and, thus, benefit from the best characteristics of the two alternative frameworks. Amongst others, we want to profit from the guaranteed success, including termination, of the consumer validation in configurable program certification. Simultaneously, we would like to use an even simpler, less complex configuration for the consumer like in the Programs from Proofs approach.

Our idea is to let the producer perform the refined property checking analysis known from the Programs from Proofs approach. Like in the Programs from Proofs approach, the consumer still receives and validates the program generated from the producer's proof. However, the consumer does not verify the generated program. Instead, the consumer applies the validation approach known from the configurable program certification approaches. He validates the generated program with the help of a certificate, which is additionally constructed by the producer.

Many parts of the producer and consumer tasks in the sketched combined approach can be adopted from the previous Programs from Proofs and configurable program certification approach. Only, the certificate generation must be redesigned. As we will see, the naïve combination in which the producer verifies the generated program, too, and uses existing certificate construction techniques is inappropriate. Hence, given a proof, an ARG, for the original program, the producer must construct a certificate for the generated program.

Existing approaches that are tailored to certificate generation for generated programs cannot directly be reused for the certificate generation. Some consider different types of proofs, e.g. Hoare logic proofs [MN07, Hur09]. In other cases, one needs to specify a translation function [Riv03, BGKR06]. Translation validation [PSS98] focuses on translation correctness instead of a safety property. In another class of approaches [NL98b, RD99, WSF02, GKD+07], a certificate is constructed during program generation without taking a proof of the original program into account. In our context, this means that the generated program must be verified. We see later that this is not the best option.

Before we come to the details of the combination, especially focusing on the certificate construction, and the evaluation of our combination, we first give detailed reasons for a combination of our two alternative frameworks.

6.1 Motivation

Several reasons exist why one might profit from combining the Programs from Proofs approach with the configurable program certification approach. One major reason is that for some configurations the consumer verification in the Programs from Proofs approach may no longer terminate. With the combination, we want to overcome the termination problem of the consumer. The idea is that the certificate provides enough information s.t. in the consumer validation at most a single abstract state per program location must be explored. The consumer validation of the generated program terminates.

Another aspect tackles the performance of the consumer validation. Recapturing our validation algorithms, the consumer does not need to merge nor adjust precisions. Note that in some of our Programs from Proofs examples, the consumer performs a high number of merges (more than 1000). More importantly, certificate validation explores each element in the certificate once. It does not need to reexplore abstract states after a merge nor to bother about states deleted by a merge anyway. Remember that we observed that certificate validation especially benefits from these factors, i.e., is faster, when the ratio of the number of explored successors to the number of ARG nodes is high. Some of the consumer tasks in our Programs from Proofs evaluation achieved high ratios. Another reason depends on the precision adjustment of the property checking analysis. In case, the property checking analysis widens abstract states during precision adjustment, we instantly benefit from certificate validation. Since in our setting a dataflow analysis never widens in the precision adjustment, it may require multiple merges to compute the widened state or only explores a more precise state, which we think is more expensive to explore. In contrast, the certificate may contain the widened state. Certificate validation does not need to merge to compute the widened state nor to explore a more precise state.

Additionally, the parallelization of all our certificate validation algorithms is simple and straightforward. A parallelization of the CPA algorithm, the verification, is not so obvious. If we use the combination, the consumer validation can easily be adapted and profit from the current hardware trend of a growing number of processing units.

At last, the trusted computing base of the consumer is decreased even further. Especially, the consumer no longer needs to rely on the merge, and thus the join operator of the abstract domain. Now that we know why a combination of our approaches might be beneficial, we continue to describe how to combine the approaches.

6.2 The Naïve Combination

The first idea one typically would come up with is to sequentially combine the Programs from Proofs approach with the configurable program certification approach. Figure 6.1 shows the process of this combination. The first part of the producer analysis is identical with the producer analysis in the Programs from Proofs approach. Instead of stopping after program generation, the producer continues with the consumer analysis of the Programs from Proofs approach. Notice that the producer part beginning at the verification of the generated program is typical for a producer in the configurable program certification approach. After the verification of the generated program finished successfully, the

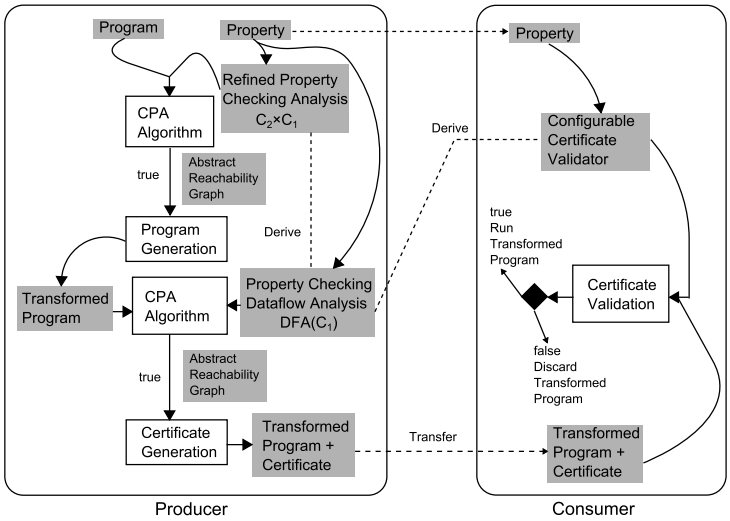

Figure 6.1: Naïve combination of the Programs from Proofs and the configurable program certification approach

producer applies one of the certificate construction techniques presented for configurable program certification. In contrast to the producer, the consumer follows the standard certificate validation procedure known from the configurable program certification approach. The Programs from Proofs approach and, thus, the combination of the approaches are completely transparent to the consumer.

The described combination of the approaches is simple to realize. One simply needs to reuse the existing approaches and their implementations in a proper order. Soundness and relative completeness of the combination directly follows from soundness and relative completeness of the configurable program certification approach.

However, the combination has some major drawbacks. When the refined property checking analysis widens the property checking part of an abstract state, the widening result is not transferred to the consumer. More importantly, the producer can only construct a certificate if the verification of the original and the generated program succeeds. Remember that the verification of the generated program may not terminate, although the verification of the original was successful. The consumer no longer directly observes the termination problem of the Programs from Proofs approach, but the presented combination also does not overcome that problem. Additionally, the approach is inefficient for the producer. To construct the certificate, the producer has to redo the verification on the generated program. In principle, he does the verification twice. Thus, we think that such a combination is to naïve. We continue to discuss more sophisticated combinations, which overcome the mentioned drawbacks of the naïve combination.

6.3 Certificates for the Generated Program from the Producer's Proof

In this section, we propose a combination of the Programs from Proofs and the configurable program certification approach that is more sophisticated than a pure sequential

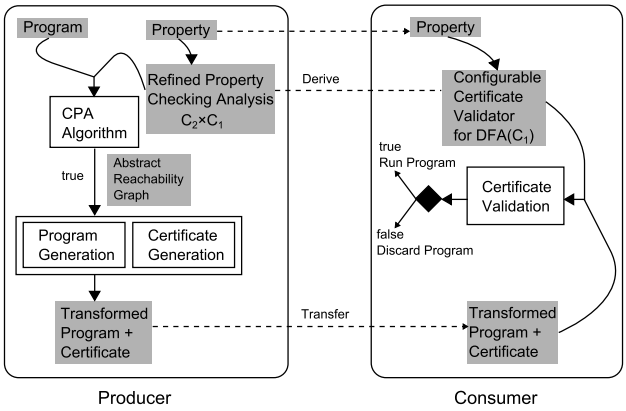

Figure 6.2: Proposed combination of the Programs from Proofs and the configurable program certification approach

composition, the naïve combination. Our goal is to combine the approaches in such a way that the additional effort for the producer is low. The producer analysis should remain close to the producer analysis in the Programs from Proofs approach. At most, we want to add an extra certificate generation task. Moreover, like in the naïve combination the consumer validation should follow the standard validation procedure of the configurable program certification approach.

Figure 6.2 depicts the general overview of our solution for such a combination. As required, the producer's tasks are mainly identical with the producer's tasks in the Programs from Proofs approach, except for one step. After successful verification of the original program with a refined property checking analysis, the producer does not only use his proof, an ARG, to generate a program which can be verified with the simpler property checking analysis alone. Additionally, the producer uses the constructed ARG to build a certificate for the generated program. Note that this certificate is constructed w.r.t. the abstract domain considered by the property checking analysis – the analysis that is sufficient to verify the generated program. Furthermore, the consumer's process is rather identical to the consumer's tasks in the configurable program certification approach. The only difference is the derivation of the consumer's validation configuration. The producer's verification configuration, a refined property checking analysis, is tailored to the verification of the original program and not to the verification of the generated program, which the consumer inspects. Hence, the consumer should not derive his validation configuration from the complete producer analysis. In contrast, the consumer should restrict his derivation to those parts of the refined property checking that are relevant for the verification of the generated program, i.e., the dataflow analysis variant of the property checking analysis part. Hence, the consumer derives his validation configuration from the dataflow analysis variant of the property checking analysis part. Due to the special form of the termination check operator of that dataflow analysis, we know that the termination check is well-behaving. The validation configuration of the consumer can be derived fully automatically.

Critically inspecting the overview, we observe that once more we left out the initial abstract state and the initial precision from our overview. To complete the input of the producer analysis, we require an initial abstract state and an initial precision. Like in the Programs from Proofs approach, the initial abstract state must be compatible with the

refined property checking analysis. The consumer requires only one additional input, an initial abstract state. We use the same initial abstract state for the consumer as in the Programs from Proofs approach.

Next, we reflect the trusted computing base of our combination. Recap that the trusted computing base incorporates all elements that the consumer must rely on if he trusts the outcome of his validation. Looking at the consumer's validation, we recognize that the consumer must trust two elements: his validation configuration, the configurable certificate validator from the dataflow analysis variant of the property checking analysis, and the certificate validation algorithm. For both elements, the trusted computing base contains the definition and the respective implementation used by the consumer. Compared to the producer, we think that the certificate validation algorithms are similarly complex than the CPA algorithm. Moreover, the validation configuration is much simpler than the producer's configuration. The validation configuration gets rid of the often more complex enabler analysis $\mathbb{C}_2$ as well as the merge and precision adjustment operator of the property checking analysis. Also, the termination check operator becomes quite standard. Compared to the Programs from Proofs approach, the merge and precision adjustment operator of the property checking analysis are excluded from the trusted computing base. We also observe that the trusted computing base is smaller than in the configurable program certification approaches, since we exclude parts of producer's abstract domain.

After we discussed the high level aspects of our combination, we come to its realization. Recapturing the overview of the proposed combination, we remember that many steps of the producer are already known from the Programs from Proofs approach (cf. Chapter 5). Only the certificate generation is new. Furthermore, the consumer validation remains the same as in configurable program certification (cf. Chapters 3 and 4). In the following, we discuss the only unknown task in the proposed combination, the construction of a certificate for the generated program on the basis of the producer's ARG.

We present two alternative approaches how to construct a certificate for the generated program from the ARG built by the producer for the original program. Both alternatives can construct any certificate type proposed in the configurable program certification. The first approach converts the producer ARG into an ARG for the generated program and then applies one of the certificate construction techniques known from configurable program certification. The second approach transforms a certificate constructed from the producer's ARG for the original program into a certificate for the generated program. While the first approach is more memory consuming – it additionally nearly copies the producer's ARG –, we will show that for some certificate types the size of its generated certificates could be smaller than the size of the certificates from the second approach. Furthermore, we prove relative completeness for both combination variants. Note that we do not need to prove soundness. The consumer uses the validation algorithms from the configurable program certification for which we already proved soundness in case the initial abstract state considers the initial automaton state. Due to the requirements on the initial abstract state of the consumer, we know that the automaton state considered by the consumer's initial abstract state is the initial automaton state. Hence, our combination is sound by default.

Before we come to the details of the two approaches, we first describe a common transformation operation. Often, (some of) the producer's ARG nodes must be transferred to abstract states of the property checking analysis, which consider location of the generated program. In the Programs from Proofs, we already transfer a single abstract state of the refined property checking analysis into a single abstract state of the property checking analysis. For example, we transfer the producer's initial abstract state into a

proper initial abstract state for the consumer. To prove successful consumer verification, we regularly transfer abstract states of the refined property checking analysis into abstract states of the property checking analysis. For all transformations, we applied the definition of location updated property checking extraction (Definition 5.6). Next, we reuse location updated property extraction to transform (sets of) sets of ARG nodes. Like in some of the handwritten proofs for the PfP approach, we do not want to update the location with an arbitrary location, but with the location associated with the ARG node that we transform. Following the notation in the Programs from Proofs approach, for any ARG node n we use n to refer to the abstract state as well as to the location associated with the ARG node. The meaning should be clear from the context. When we use the location updated property checking extraction, the abstract state is always notated in front followed by the location surrounded in square brackets. To transform a set of ARG nodes, one principally applies the location updated property checking extraction for each ARG node in the set using the location associated with the ARG node. For the transformation of sets of sets of ARG nodes the transformation of a set of ARG node is used on each set of ARG nodes.

Definition 6.1. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}$. For a subset $N_{\text{sub}} \subseteq N$, we define $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] := \{n[n] \mid n \in N_{\text{sub}}\}$. We extend this notation to sets $\mathcal{N} \in 2^{2^N}$ of subsets of ARG nodes and define $\mathcal{N}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] := \{N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] \mid N_{\text{sub}} \in \mathcal{N}\}$.

In the following, we typically use the transformation of a set of sets of ARG nodes to transform a partition of a subset of ARG nodes. Such a partition of a subset of ARG nodes is required to construct one of our proposed partitioned certificates. When we transform (sets of) sets of ARG nodes in general, we might translate two different elements to the same element. Thus, some of the properties of a partition might not be fulfilled after transformation. The following lemma ensures that for our special transformation a transformation of a partition results in a partition.

Lemma 6.1. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for refined property checking analysis* $(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}$. *Let* $N_{\text{sub}} \subseteq N$ *be a subset of nodes and* $\text{partition}(N_{\text{sub}})$ *be a partition of* N_{sub}. *Then, the transformation* $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ *of partition* $\text{partition}(N_{\text{sub}})$ *is a partition of* $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$.

Proof. See Appendix pp. 295 f. □

With the definition of the transformation of partitions and sets of ARG nodes, we have everything at hand to describe our two combinations. We proceed with the first approach, which transforms the ARG and then constructs the certificate from the transformed ARG.

6.3.1 Certificates from the Transformed Provider's Proof

In this section, we describe the first of our two proposed combinations. Remember that the first combination transforms the ARG constructed by the producer analysis into a proper ARG for the generated program. Thereafter, it uses the standard techniques of our configurable program certification to construct a certificate for the generated program.

The following definition describes how to transform the ARG. The idea is simple. We use the transformation of a subset of ARG nodes as defined above for the set of ARG and covering nodes. With the help of the location updated property extraction, we transform the root node in the same way as a single node in the set of ARG nodes. Finally, for

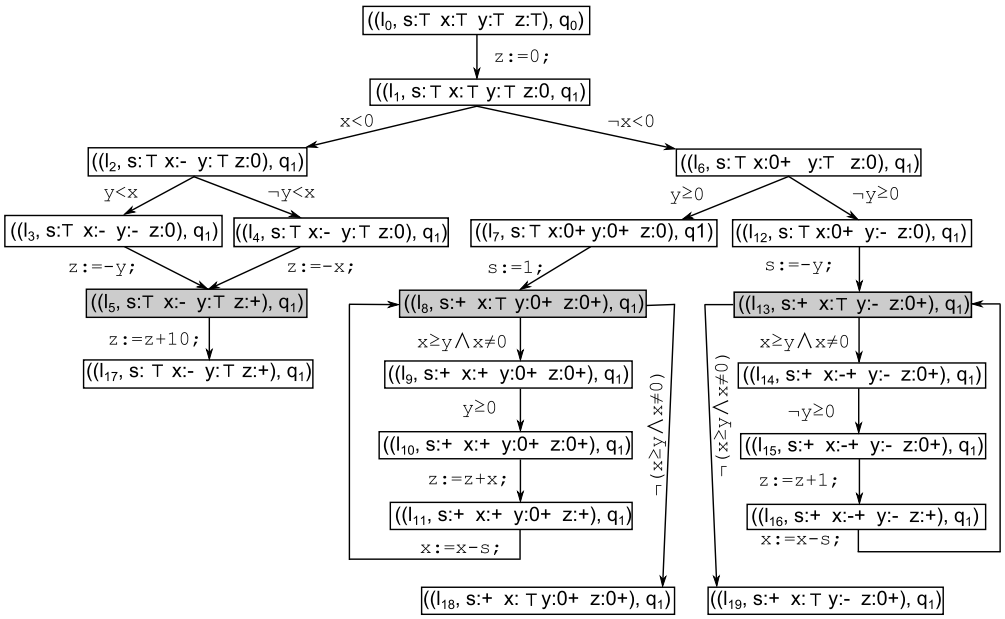

Figure 6.3: Transformation of the ARG from Fig. 5.3 which was constructed during successful verification of our example refined property checking analysis on program `SubMinSumDiv`

every ARG edge in the original ARG g we add an ARG edge g' from the transformed predecessor to the transformed successor of edge g. Additionally, we label the edge g' by the CFA edge in the generated program, which originates form the ARG edge g.

Definition 6.2. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ and program P. The *transformation of ARG* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ *for* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *and generated program* $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$ with $N' = N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$, $G'_{\text{ARG}} = \{(p[p], (p, op, s), s[s]) \mid (p, (\cdot, op, \cdot), s) \in G_{ARG}\}$, $root' = root[root]$ and $N'_{\text{cov}} = N_{\text{cov}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$.

Figure 6.3 shows the transformation of the ARG constructed for our example producer analysis in the Programs from Proofs approach. Note that we directly use the location names associated with the abstract state. More concretely, we use the location names of the program that is shown in Fig. 5.4 and is generated from the ARG in Fig. 5.3. Furthermore, as before we label edges only by program instructions and not by the complete CFA edge. Comparing the transformed ARG with the original one (see Fig. 5.3), we observe that the structure is the same. Only the enabler state, the predicate state, is removed and the location names are adapted.

So far, we only described how to transform an ARG. To apply the configurable program certification program approach, we need to ensure that the transformed ARG is a well-formed ARG for the generated program and the dataflow analysis variant $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ of the property checking analysis $\mathbb{C}_1^{\mathcal{A}}$. First, we show that after transformation we get an ARG for the generated program and $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$.

Lemma 6.2. *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ be an abstract reachability graph for program P and refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. Then, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ is an abstract reachability graph for program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$.*

Proof. See Appendix p. 296. □

Due to the previous lemma, we know that we can transform any ARG for the original program and a refined property checking analysis into an ARG for the generated program. Next, we show that when we apply the transformation in our combination of the PfP and the CPC approach, then the transformed ARG will be well-formed. We know that we transform the ARG only when the verification of the original program with the refined property checking analysis was successful. From the Programs from Proofs approach, we remember that the ARG constructed during verification is strongly well-formed (cf. Proposition 5.2). Based on this inside, we show that if the initial ARG, the ARG constructed during verification, is strongly well-formed, then the transformed ARG will fulfill the required property for the configurable program approach, i.e., the transformed ARG is well-formed.

Proposition 6.3. *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. If $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed for $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, then $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ is an ARG for $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ which is well-formed for $e_0[root]$.*

Proof. See Appendix pp. 296 ff. □

In our CPC approach, well-formed ARGs are the basis to construct valid certificates. Certificates that are generated from well-formed ARGs by a CPC technique are accepted by the validation algorithm when a suitable configurable certificate validator with a well-behaving coverage check is used. Due to the previous proposition, we know that in our first combination each of the configurable program certification approaches constructs such valid certificates. Furthermore, we know that we must derive the consumer's configurable certificate validator from the dataflow analysis $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$. Our definition of a dataflow analysis' termination check operator and Corollary 3.4 let us conclude that the termination check is also a well-behaving coverage check. If the consumer uses this coverage check in his validation configuration, the producer will act as depicted in the overview (Fig. 6.2), and the consumer receives the generated program and the certificate constructed by the producer, relative completeness of our certification approaches will guarantee us that the consumer will accept the generated program. This is stated by the following theorem.

Theorem 6.4. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a refined property checking analysis. Furthermore, let $\mathbb{V}^{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})})$ be the configurable certificate validator for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and coverage check $\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}$. If Algorithm 2 started with CPA $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, initial precision $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, and program P returns $(true, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$, then*

- *the validation algorithm for certificates (Algorithm 3, p. 54) started with configurable certificate validator $\mathbb{V}^{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})})$, initial abstract state $e_0[root]$, certificate $\text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$, and program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ returns true.*

- *the validation algorithm for reduced certificates (Algorithm 4, p. 82) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, initial abstract state* $e_0[root]$*, reduced certificate* $\mathrm{cert}_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$ *or highly reduced certificate* $\mathrm{cert}_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$*, and program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*
- *the validation algorithm for partitioned certificates (Algorithm 5, p. 106) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, the abstract state* $e_0[root]$*, a partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}}), R^P_{\mathbb{C}^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$*, which is a full, partitioned, reduced, partitioned, or highly reduced, partitioned certificate, and program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*

Proof. From definition of $\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}$ and Corollary 3.4, we know that $\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}$ is a well-behaving coverage check. Hence, $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$ is a CCV. From $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ being a refined property checking analysis and definition of $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, we infer that $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ is monotonic. From Proposition 5.2, we infer that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$is an ARG for program P and $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ that is strongly well-formed for e_0. From Proposition 6.3, we conclude that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ is an ARG for program $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ that is well-formed for $e_0[root]$. Since Algorithm 2 terminates, when started with P, we know that P is finite. From Proposition 5.6 and P being finite, we infer that program $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is finite. Now, $e_0[root] \sqsubseteq e_0[root]$ (reflexivity of partial order $\sqsubseteq$) and Theorems 3.10, 4.11, and 4.28 let us conclude the claim of this theorem. □

We already discussed that our combinations are sound. The previous theorem ensured us that our first combination is relative complete and that the consumer configuration can be derived fully automatically. Our first combination guarantees the three properties automation, soundness, and relative completeness all our approaches must provide. No further theoretical aspects need to be discussed. Next, we continue with our second combination, which first constructs the certificate and then transforms it.

6.3.2 Transformation of Original Program Certificates

In this section, we present the details of our second, proposed combination. The idea of the second combination is to apply our standard techniques for certificate construction first, i.e., generate a certificate from the ARG that was built during the producer's verification. Then, this approach transforms that certificate into a proper certificate for the generated program. Next, we describe how to transform a certificate built from the producer's ARG.

The transformation of a certificate constructed for the original program by one of our configurable program certification approaches is straightforward. No matter which kind of certificate we transform, we always keep the certificate's structure and the size of the original certificate when stored. Hence, we only need to transform the set(s) of ARG nodes stored in the certificate for the original program. To transform these sets, we again use the transformation of a set of ARG nodes, which is based on location updated property checking extraction. The following definition formally describes the transformation of certificates. Since the transformation of the three types of partitioned certificates is technically the same, we decided to generally define the transformation of an arbitrary partitioned certificate.

Definition 6.3. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$.

- The *transformation of a certificate* $\text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = N$ from ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and generated program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is

$$\text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] := N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \ .$$

- The *transformation of a reduced certificate* $\text{cert}_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (N_{\text{R}}, n)$ from ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and generated program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is

$$\text{cert}_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] := (N_{\text{R}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], n) \ .$$

- The *transformation of a highly reduced certificate* $\text{cert}_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (N_{\text{hR}}, n')$ from ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and generated program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is

$$\text{cert}_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] := (N_{\text{hR}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], n') \ .$$

- The *transformation of partitioned certificate* $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (parts, n)$ from ARG and $\text{partition}(N_{\text{sub}})$ of a subset of ARG nodes $N_{\text{sub}} \subseteq N$ for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and generated program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is

$$\begin{array}{rl} & \text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ := & (\{(pn[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], bn[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \mid (pn, bn) \in parts\}, n) \ . \end{array}$$

In the following, we provide an example of a certificate transformation. Below, we show the (highly) reduced certificate[1] from the ARG in Fig. 5.3, which was generated by our example producer analysis in the Programs from Proofs approach. Beneath the (highly) reduced certificate we depict its transformation. Again, we directly use the location names associated with the ARG nodes, namely those of the generated program shown in Fig. 5.4. As intended, the transformation only removes the predicate state, the enabler state, and adjusts the program locations. The number of abstract states remains the same. Furthermore, in both certificates the number of abstract states that can be recomputed during validation is limited to 20.

$$\left(\left\{ \begin{array}{l} ((\mathit{true}, (l_0, s: \top\ x: \top\ y: \top\ z: \top)), q_0), \\ ((\mathit{true}, (l_5, s: \top\ x: -\ y: \top\ z: +)), q_1), \\ ((\neg y \geq 0, (l_9, s: +\ x: \top\ y: -\ z: 0+)), q_1), \\ ((y \geq 0, (l_9, s: +\ x: \top\ y: 0+\ \ z: 0+)), q_1) \end{array} \right\} \ , \ 20 \right)$$

$$\left(\left\{ \begin{array}{l} ((l_0, s: \top\ x: \top\ y: \top\ z: \top), q_0), \\ ((l_5, s: \top\ x: -\ y: \top\ z: +), q_1), \\ ((l_{13}, s: +\ x: \top\ y: -\ z: 0+), q_1), \\ ((l_8, s: +\ x: \top\ y: 0+\ \ z: 0+), q_1) \end{array} \right\} \ , \ 20 \right)$$

Up to now, we described how to transform certificates. Next, we show that the transformed certificates are proper witnesses, which can be used by the consumer to successfully check

[1] The highly reduced and the reduced node set are the same for the ARG considered during certificate construction.

correctness of the generated program. From the results of the first combination, we already know that the certificates generated in the first combination are such proper witnesses. If we are able to show that we could have constructed the transformed certificate with the first combination, our transformed certificates will be proper witnesses.

As we will prove, the transformed certificate, the transformed highly reduced certificate, the transformed full, partitioned certificate, and the transformed highly reduced, partitioned certificate are identical with their counterparts generated in the first combination. However, this is not necessarily true for the transformation of reduced or reduced, partitioned certificates. The problem is that the reduced node set of the original ARG can become larger than the reduced node set of the transformed ARG. Due to a precision adjustment or a merge, which only affects the enabler state, an ARG successor in the original ARG may not be a transfer successor of the refined property checking analysis, solely because its enabler state is not a successor of the enabler analysis' transfer relation. The other parts of the state agree with those of the transfer successor. Thus, the corresponding ARG node in the transformed ARG is a transfer successor of the property checking analysis. For example, consider an ARG that is identical with the ARG shown in Fig. 5.3 except for the ARG node $((\neg y \geq 0, (l_{14}, s : + \; x : \top \; y : - \; z : +0)), q_1)$ which is replaced by $((true, (l_{14}, s : + \; x : \top \; y : - \; z : 0+)), q_1)$. An analysis might have constructed such an ARG because the predicate analysis adjusts its precision for location l_{14} when y is negative. In this case, the reduced node set from the ARG contains all abstract states in the reduced certificate from above plus $((true, (l_{14}, s : + \; x : \top \; y : - \; z : 0+)), q_1)$. In contrast, the reduced node set of the transformed ARG is the same as the set of abstract states in the transformed certificate from the previous example. Hence, for the transformation of certificates based on the reduced node set we need to separately show that we get proper witnesses.

First, let us look at identity for all certificates whose construction does not use the reduced node set. The basic that stores the set of ARG nodes. Due to the definition of certificate transformation and the transformation of the ARG, the transformed basic certificate and the basic certificate from the transformed ARG are identical.

All other certificates store the size of the original certificate. Since during transformation we do not change the stored size, we first prove that the stored sizes are identical. A transformed certificate stores the size of the set of ARG nodes in the original ARG. A certificate from the transformed ARG stores the size of the set of ARG nodes in the transformed ARG. The following lemma states that these sizes, the size of the set of ARG nodes in the original and the transformed ARG are identical.

Lemma 6.5. *Let* $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *and* $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$. *Then,* $|N| = |N'|$.

Proof. By definition, $N' = N[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ and N' contains at most $|N|$ elements. We need to show that $\neg \exists n, n' \in N : n \neq n' \wedge n[n] = n'[n']$. Let $n[n] = (e_1, q)$ and $n'[n'] = (e'_1, q')$. If $n[n] = n'[n']$, then $e_1 = e'_1$ and $q = q'$. From $e_1 = e'_1$ and definition of $n[n] = n'[n']$, we get $\text{acs}(e_1) = \text{acs}(e'_1)$ and, thus, $\text{acs}(e_1) = n = \text{acs}(e'_1) = n'$. We conclude that $|N'| = |N|$. □

Due to the previous lemma, we know that the stored sizes are identical. To show that the highly reduced certificates are identical, it remains to be shown that the highly reduced node sets are identical. Furthermore, we also need that identity when looking at the identity of the highly reduced, partitioned certificate. The following lemma claims the

desired identity of the transformed highly reduced node set from the original ARG and the highly reduced node set of the transformed ARG.

Lemma 6.6. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}$. *Then,* $N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})])$.

Proof. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$. By definition we get

$$\begin{aligned}
& N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] \\
= \;& (\{root\} \cup N_{\text{cov}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] \\
= \;& \{root[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]\} \cup N_{\text{cov}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] \\
= \;& \{root'\} \cup N'_{\text{cov}} \\
= \;& N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]) \;\; .
\end{aligned}$$

□

Due to the previous lemmas, the definition of a highly reduced certificate and its transformation, we can now prove identity of the highly reduced certificates. Next, we consider identity for the full, partitioned and the highly reduced, partitioned certificate. For this, it remains to be shown that their sets of partition elements are identical.

Basically, for each partition element an identical partition element must exist. First, let us ignore the boundary nodes of a partition element. Then, identity of partition elements is restricted to identity of partition nodes and constructing a partitioned certificate from the transformed ARG s.t. the partition elements are identical is simple. We already showed that the sets of ARG nodes and the highly reduced node sets are identical. Hence, the set of all partition nodes is the same. Recapturing our construction of a partitioned certificate, we only require a partition that is identical to the set of all sets of partition nodes in the transformed certificate. The partition that one obtains when transforming the partition used to construct the certificate from the original ARG fulfills this requirement.

In reality, partition elements are only identical when the set of partition nodes and the set of boundary nodes are identical. It is left over to be shown that the transformation of the set of boundary nodes is proper, i.e., it matches the construction of the set of boundary nodes in the partitioned certificate construction from the transformed ARG. Our partitioning approach builds the set of boundary nodes from a vertex contraction of an ARG to a subset of ARG nodes (all nodes occurring in the partition), and a subset (a set of partition nodes) of the former subset . Thus, we show that transforming a set of boundary nodes obtained from a vertex contraction of the original ARG to a subset N_{sub} of its ARG nodes and a subset N'_{sub} of N_{sub} is identical to the result of a transformation of the ARG and the subsets plus a subsequent construction of the set of boundary nodes. The subsequent construction builds the set of boundary nodes from a vertex contraction of the transformed ARG to the transformation of N_{sub} and the subset of subset of ARG nodes resulting from the transformation of N'_{sub}. The following lemma claims this identity.

Lemma 6.7. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}$. *Furthermore, let* $N'_{\text{sub}} \subseteq N_{\text{sub}} \subseteq N$. *Then,* $\text{bound}(N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]))$ $= \text{bound}(N'_{\text{sub}}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}))[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$.

Proof. See Appendix pp. 298 f. □

As explained before, a proper transformation of the set of boundary nodes was the missing piece to show that the transformation of a partition element yields a proper partition element and, thus, the identity of the transformed certificate and the certificate from the transformed ARG in case of the full, partitioned and the highly reduced, partitioned certificate. Before we consider the identity of the transformed certificate and the certificate from transformed ARG for these two certificate types, we generally want to prove that when we transform a partitioned certificate from a partition and the producer ARG, we get the partitioned certificate from the transformed partition and the transformed ARG. This correspondence between a transformed partitioned certificate and a partitioned certificate from the transformed ARG does not only allow us to prove identity in case of the full, partitioned and the highly reduced, partitioned certificate, but also lets us easily prove relative completeness in case our second combination transforms a reduced, partitioned certificate. The following lemma asserts this correspondence.

Lemma 6.8. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *and* $\mathrm{partition}(N_{\mathrm{sub}}) = \{p_1, \ldots, p_n\}$ *be a partition of* $N_{\mathrm{sub}} \subseteq N$. *Then,* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}}), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$.

Proof. See Appendix pp. 299 f. □

With the last lemma, we have everything at hand to show that the transformation of a certificate, a highly reduced certificate, a full, partitioned certificate, or a highly reduced, partitioned certificate results in a certificate that is identical with a certificate, a highly reduced certificate, a full, partitioned certificate, or a highly reduced, partitioned certificate constructed from the transformed ARG. In case of the partitioned certificates, the transformed certificates are only identical when the partitions used to construct the partitioned certificates are related, i.e., to construct the partitioned certificate from the transformed ARG also the transformed partition must be used. The following proposition states the identity between the certificates.

Proposition 6.9. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for a refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. *Then,*

- $\mathrm{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \mathrm{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$,
- $\mathrm{cert}_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \mathrm{cert}_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$,
- *if* $\mathrm{partition}(N[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) = \mathrm{partition}(N)[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$,

$$\begin{array}{rl} & \mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & \mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \end{array} \text{, and}$$

- *if* $\begin{array}{l} \mathrm{partition}(N_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])) \\ = \mathrm{partition}(N_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \end{array}$

$$\begin{array}{rl} & \mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & \mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \end{array} .$$

Proof. See Appendix pp. 300 f. □

Based on the previous proposition, we can easily show relative completeness of the second approach when it transforms a certificate, a highly reduced certificate, a full, partitioned certificate, or a highly reduced, partitioned certificate. Remember that we already showed for the first combination that the consumer validation succeeds whenever the consumer considers the generated program and a certificate constructed from the transformed ARG by a process conformant producer. Since a transformed certificate, a transformed highly reduced certificate, a transformed full, partitioned certificate, and a transformed highly reduced, partitioned certificate are identical with their counterparts generated from the transformed ARG, the consumer cannot distinguish between the first and second approach. His validation automatically succeeds.

It remains to show relative completeness when the reduced node set is considered during certificate generation. Remember that while proving relative completeness of our configurable program certification approaches, we already considered that during certificate construction a larger subset of the ARG nodes than the reduced node set is used for certificate construction. Relative completeness should be of no problem, if these transformed certificates are identical with a certificate that is constructed from the transformed ARG and that considers at least all nodes of the reduced node set. Due to the definition of a reduced certificate and our previous result that a transformed partitioned certificate from a partition and the producer ARG is identical with the partitioned certificate from the transformed partition and the transformed ARG, we only need to show that the reduced node set of the transformed ARG is a subset of the transformed reduced node set and the transformed reduced node set is a subset of the set of ARG nodes of the transformed ARG. Note that we require the first relation to apply our relative completeness results from our configurable program approaches and the second requirement to ensure that such a certificate can be constructed from the transformed ARG. The following lemma establishes the required subset relations.

Lemma 6.10. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$*. If* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *is strongly well-formed for* $e_0 \in E(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$*, then* $N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \subseteq N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \subseteq N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$.

Proof. See Appendix p. 301. □

With the last lemma, we have all pieces together to prove that our second combination also comes with the relative completeness property. The following theorem claims that if the producer sticks to his process shown in Fig. 6.2, the consumer validation will succeed for the generated program, any certificate constructed by that process conformant producer, and the validation configuration derived from the dataflow analysis variant of the property checking analysis when reusing the termination check as coverage check. Since the described consumer validation is the one that we use in our proposed combination of the PfP and the CPC approach, the subsequent theorem automatically states relative completeness.

Theorem 6.11. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *be a refined property checking analysis. Furthermore, let* $\mathbb{V}^{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})})$ *be the configurable certificate validator for* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *and coverage check* $\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}$*. If Algorithm 2 started with CPA* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$*, compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*, initial precision* $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$*, and program* P *returns* $(true, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *and* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$*, then*

- *the validation algorithm for certificates (Algorithm 3, p. 54) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, initial abstract state* $e_0[root]$*, certificate* $\mathrm{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$*, and program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*
- *the validation algorithm for reduced certificates (Algorithm 4, p. 82) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, initial abstract state* $e_0[root]$*,* $\mathrm{cert}_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ *or* $\mathrm{cert}_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$*, as well as program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*
- *the validation algorithm for partitioned certificates (Algorithm 5, p. 106) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, initial abstract state* $e_0[root]$*, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}}), R^P_{\mathbb{C}^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ *which is constructed from a full, partitioned, reduced, partitioned or highly reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}}), R^P_{\mathbb{C}^{\mathcal{A}}})$*, and program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*

Proof. See Appendix pp. 302 f. □

From the previous theorem, we conclude that our second combination is relative complete and that a consumer can derive his validation configuration fully automatically. Moreover, we already discussed that our combinations are sound. Also, our second combination provides the important properties automation, soundness, and relative completeness all our approaches must have. From a theoretical point of view, we know that both proposed combinations of the Programs from Proofs and the configurable program certification approach work out. Next, we examine the practical impact of the combination of PfP and CPC.

6.4 Evaluation

In the evaluation, we want to study the value of the combination of the Programs from Proofs approach with the configurable program certification in cases in which we do not need the combination to solve the termination problem of the Programs from Proofs approach. Especially, we want to know if and when the combination improves the consumer's validation of the generated program.

To examine this question, we apply our configurable program certification approach to all of the 127 consumer verification tasks examined during the evaluation of the Programs from Proofs approach. Since the transfer relations of all property checking analyses used in the PfP evaluation are monotonic, we use the certification variant whose validation typically performed best w.r.t. execution time. Thus, we construct highly reduced, partitioned certificates, which were constructed once before evaluation, and read and check partition elements in parallel. For construction of these certificates, we use the partitioning strategy `BEST_FIRST`, which allows an efficient certificate generation and certificate validation performs similarly to the other partitioning strategies [Brö16]. Furthermore, we restricted the size of each set of partition nodes to 10. Hence, we are able to partition the certificate even when the generated programs are small.

Note that the proposed combination variants are not implemented in CPAchecker. In practice, the CFA constructed from the generated C program might be different from the CFA one would obtain from the ARG, especially because optimizations are performed

during CFA construction. Furthermore, we are lacking insight in how the CFA nodes are labeled. However, we need to know the node label names to properly transform the proof. Therefore, we used the naïve combination to construct the certificates. To still simulate the proof transformation, during certificate construction we run the consumer analysis with the same precision adjustment that the producer used for the property checking analysis part.

For evaluation, we use the same execution set up as in the evaluation of the Programs from Proofs approach. Again, we run the experiments with the benchmark evaluation framework BenchExec [BLW15a] to restrict each task to 15 GB of RAM and two cores of an Intel Xeon E5-2650 v2 CPU at 2.6 GHz. The time limit was set to 15 minutes of CPU time. All experiments were performed with the CPAchecker version available in the `runtime_verification` branch[2] in revision 23042 and Java HotSpot(TM) 64-Bit Server VM 1.8.0_101. We repeated all experiments 10 times and study the average of these 10 runs. The results for all 127 tasks can be found in the appendix in Tab. B.9.

In the following, we investigate the gain of the proposed combination on our examples. Note that we do not present the total times because the observations would be similar. The time added to the time for actual validation is the same for verification and certificate validation of the generated program. Thus, only the improvement might be smaller. We start with the question if the consumer validation can be improved when it is already worse than the producer.

6.4.1 RQ 1: Does Certification Help When the Consumer is Worse Than the Producer?

To study if the combination may help to overcome the problem that in the PfP approach some consumer verifications were slower than the corresponding producer verification, we provide the results of our combination for all 7 tasks in which the consumer performed worse. Like in the PfP evaluation, we use the acronym of the refined property checking analysis and the program name to identify the verification task. Next to the verification times, V_P and $V_{\mathbb{C}}$, and memory usage, M_P and $M_{\mathbb{C}}$, known from the PfP evaluation, Tab. 6.1 also provides the time for certificate validation $V_{\mathbb{V}}$, the time for certificate checking and reading plus the additional parsing costs for the generated program, the memory used by the certificate validation, the relation of the times and memory usages for verification and certificate validation on the generated program, and the number of merges $\#\sqcup$ performed during verification of the generated program. We continue to display times in seconds and memory usage in MB.

Looking at Tab. 6.1, we observe that if the bad consumer performance was mainly caused by the additional parsing costs – these are all cases in which the number of merges $\#\sqcup$ is below 200 – the performance of the consumer cannot be improved significantly if at all. For program `transmitter02` and `powerapprox`, the consumer's performance becomes much better and now beats the producer analysis in time and memory. The reason is that certificate validation circumvents the merging costs, which are an important performance factor in these cases. We would expect a similar improvement for the two tasks related to the `pipeline2` programs. Unfortunately, we failed to construct the certificates for these tasks. A construction within the time limit of 15 minutes failed. Even after we set the time limit far beyond 15 minutes, certificate construction failed, now due to memory problems. The problem is that we fail to verify the program when we also want to construct the ARG, which we need for certificate construction.

[2] `https://svn.sosy-lab.org/software/cpachecker/branches/runtime_verification/`

Table 6.1: Comparison of the producer verification of the original program with verification and highly reduced, partitioned certificate validation on the generated program. Next to the validation times and the memory consumption, for the verification of the generated program also the number of merges is provided. All times are given in seconds and memory consumption, used heap plus used non-heap, is represented in MB.

$\mathbb{C}_P$	program	#⊔	V_P	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	M_P	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
$\mathbb{OU}$	pipeline2	993k	Proof Construction Failed							
$\mathbb{VU}$	cdaudio	190	2.75	2.89	2.90	1.00	365.6	344.8	351.9	1.02
$\mathbb{VL}$	tokenring04	0	14.15	43.05	42.27	1.02	1056.0	537.3	547.5	1.02
$\mathbb{PS}$	transmitter02	18465	8.63	5.47	1.36	4.02	494.7	553.1	253.7	0.46
$\mathbb{OS}$	powerapprox	1022	9.86	10.73	7.39	1.45	996.8	1036.4	553.7	0.56
$\mathbb{OU}$	pipeline2	993k	Proof Construction Failed							
$\mathbb{VU}$	cdaudio	188	3.00	3.63	3.69	0.98	388.8	380.0	381.7	1.00

For the special case in which consumer verification in the PfP approach already performed worse than producer verification, the combination of PfP and CPC sometimes improves the consumer validation such that it becomes better than the producer verification. Next, we want to know in general if the combination of the PfP and CPC approach likely improves the validation of the consumer.

6.4.2 RQ 2: Does Certification Improve the Consumer's Performance?

During the examination of the previous research question, we already observed that the consumer does not always benefit from the combination of the PfP and CPC approach. This observation remains valid when looking at all evaluation tasks. The consumer profits from the combination only in about 15% (20 of 127) of the cases.

The following Tab. 6.2 shows the results for those programs for which an improvement was achieved. As before, we use the acronym of the refined property checking analysis and the program name to identify the verification task. The next two columns present the number of merges #⊔ performed and the ratio of transfer successors computed during verification to the size of the set of ARG nodes. For dataflow analyses, which the PfP consumer performs, we used this ratio to predict the speed-up of certificate validation. Thereafter, the times for the consumer verification and certificate validation of the generated program as well as the speed-up of the certificate validation are shown. The last three columns display memory usage of the consumer in the PfP and the combined approach as well as their relation. Again, times are given in seconds and memory usage is given in MB.

First of all, we observe that the consumer may benefit from the combination independently of the analysis type of the refined property checking analysis or the used enabler analysis. Compared to previous improvements, e.g., in the PfP approach, the improvement is less significant. This is okay because an improvement on the consumer side becomes a multiplicative factor in the improvement w.r.t. the producer. Once again, the memory consumption is mostly improved when the validation time is improved, too. The first task is the only exception for this rule. However, memory consumption is improved less often. Moreover, larger improvements seem to be linked to a high number of merges. Despite

Table 6.2: Extract of the comparison of verification and highly reduced, partitioned certificate validation on the generated program, which contains all tasks for which the time or the memory usage of the combination is better. Next to the validation times and the memory consumption, for the verification also the number of merges and the ratio of computed transfer successors to the size of the set of ARG nodes are provided. All times are given in seconds and memory consumption, used heap plus used non-heap, is represented in MB.

$\mathbb{C}_P$	program	$\#\sqcup$	$\frac{\#\text{suc}}{\lvert N\rvert}$	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
$\mathbb{OS}$	testlocks7	134	0.36	1.15	1.32	0.87	265.0	259.4	0.98
	testlocks8	0	0.19	2.07	1.99	1.04	283.1	297	1.05
$\mathbb{VS}$	memslave1*	1744	0.51	0.91	0.58	1.56	273.2	241.1	0.88
	memslave2	2261	0.19	1.26	0.78	1.62	296.1	243.8	0.82
$\mathbb{PV}$	kbfiltr2*	32	0.42	0.77	0.70	1.10	244.2	260.1	1.07
	testlocks12	1	0.26	8.23	8.16	1.01	591.8	557.8	0.94
$\mathbb{OV}$	memslave1	19	0.20	0.35	0.32	1.09	226.8	238.6	1.05
	memslave2	255	0.16	0.59	0.29	2.03	229.6	233.2	1.02
	kbfiltr1	13	0.36	0.56	0.54	1.03	234.3	244.4	1.04
	kbfiltr2	32	0.40	0.96	0.92	1.04	251.6	266.3	1.06
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	114	0.09	0.47	0.34	1.39	233.7	241.7	1.03
	memslave2	256	0.22	0.63	0.34	1.88	241.2	243.5	1.01
	kbfiltr2*	34	0.43	0.84	0.71	1.19	250.6	257.9	1.03
$\mathbb{O}\widetilde{\mathbb{V}}$	kbfiltr1	14	0.33	0.55	0.52	1.05	236.9	247.4	1.04
$\mathbb{OSI}$	invertsorted	40	0.50	0.81	0.80	1.02	237.7	242.8	1.02
$\mathbb{PL}$	s3srvr	0	0.50	3.09	3.06	1.01	340.1	351.6	1.03
$\mathbb{VL}$	tokenring04	0	0.10	43.05	42.27	1.02	537.3	547.5	1.02
$\mathbb{PS}$	transmitter02	18465	2.37	5.47	1.36	4.02	553.1	253.7	0.46
$\mathbb{OS}$	transmitter02	120	0.25	0.31	0.30	1.03	214.4	220.6	1.03
	powerapprox	1022	0.51	10.73	7.39	1.45	996.8	553.7	0.56

of a high number of merges, in two cases, the `testlocks7` and the second occurrence of the `transmitter02` program, we do not gain a high speed-up. Considering the ratio of computed transfer successors to the number of ARG nodes, we would not have expected a high speed-up for these cases. Furthermore, the verification time is small and we assume that start reading from disk, which takes a fixed amount of time, might be another reason for the decrease and the low speed-up, respectively. From the opposite point of view, the `memslave` examples let us conclude that small verification times do not automatically prohibit good improvements.

We observed that the consumer rarely benefits from the combination. At last, we want to study the indicators for the performance improvement.

6.4.3 RQ 3: When does Certification Improve the Consumer's Performance?

To study the indicators for the performance improvement, we first need some candidate indicators. We obtain our first candidate indicator from the observation that the precision adjustment in a property checking analysis part of the producer analysis may widen ab-

Table 6.3: Extract of the comparison of verification and highly reduced, partitioned certificate validation on the generated program, which contains all tasks for which the PfP producer analysis uses a non-static precision adjustment in the property checking analysis part. Next to the validation times and the memory consumption, for the verification also the number of merges and and the ratio of computed transfer successors to the size of the set of ARG nodes are provided. All times are given in seconds and memory consumption, used heap plus used non-heap, is represented in MB.

$\mathbb{C}_P$	program	$\#\sqcup$	$\frac{\#\text{suc}}{\lvert N\rvert}$	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	114	0.09	0.47	0.34	1.39	233.7	241.7	1.03
	memslave2	256	0.15	0.63	0.34	1.88	241.2	243.5	1.01
	kbfiltr2*	34	0.43	0.84	0.71	1.19	250.6	257.9	1.03
	testlocks6*	64	0.22	0.54	0.60	0.91	229.8	247.5	1.08
$\mathbb{O}\widetilde{\mathbb{V}}$	kbfiltr1	14	0.33	0.55	0.52	1.05	236.9	247.4	1.04
	testlocks5*	0	0.22	0.33	0.42	0.78	221.1	238.4	1.08
	testlocks5d*	0	0.15	0.12	0.21	0.55	214.1	227.5	1.06
$\mathbb{P}\widetilde{\mathbb{V}}$	testlocks5*	0	0.36	0.65	0.67	0.97	230.8	243.9	1.06
	interproc*	0	0.27	0.11	0.22	0.47	213.2	229.8	1.08
	nosprintf*	0	0.33	0.15	0.28	0.51	216.3	231.0	1.07
$\mathbb{O}\widetilde{\mathbb{V}}$	testlocks5d*	0	0.21	0.11	0.21	0.52	217.5	224.4	1.03
	relax*	3	0.37	0.47	0.55	0.87	230.4	241.4	1.05
	nosprintf	0	0.34	0.14	0.26	0.54	216.8	226.3	1.04

stract states while the consumer analysis uses a static precision adjustment operator. For some locations, the abstract states in the transformed proof may be more abstract than any abstract state computed by the consumer analysis for the same location. Since we assume that transfer successor computation of more abstract state is simpler and faster, certificate validation might be faster. Furthermore, the consumer analysis may need several iterations to obtain the same abstract state by merging. Hence, our first hypothesis is that the use of a non-static precision adjustment in the property checking analysis may be an indicator for performance improvements.

To get further indicators, we recapture the CPA algorithm (Algorithm 1) and the certificate validation algorithm (Algorithm 5). The main differences are that the CPA algorithm always considers all abstract states in the termination check and that the CPA algorithm merges abstract states. Next to the costs for merging, for some program locations transfer successors must be computed multiple times. Since in CPAchecker the termination check typically considers only abstract states with same locations, the first difference becomes unimportant. Our observations lead us to our second hypothesis, the merge behavior is one indicator for performance improvements. Next, we continue to check our hypotheses.

To attest our first hypothesis, we investigate the performance of the combination on all tasks in which the producer used a non-static precision adjustment in the property checking analysis. Table 6.3 shows all verification tasks in which the precision adjustment operator is different for the property checking analysis and its DFA version. The meaning of the displayed columns agrees with those in the previous table.

Looking at Tab. 6.3, we observe that only the validation time is improved. Furthermore, we see that a difference in the precision adjustment alone is not sufficient for an

improvement. Additionally, some merge operations must be performed during the verification of the generated program. Whenever the number of merges is below 100, we only sometimes observe an improvement by the combination. Above this threshold we always observed an improvement. The dependence on the merge behavior leads us to the investigation of our second hypothesis: the merge behavior is an indicator for improvement.

To attest our second hypothesis, we investigate the performance of the combination on all tasks with a sufficiently large number of merges. Table 6.4 (p. 242) shows our results for all verification tasks for which the PfP consumer performs at least 30 merges. The presented columns are the same as in the previous table.

Looking at Tab. 6.4, we see that the number of merges is a bad indicator. The reason is that a high number does not automatically imply large merging costs. Although more than 100 merges are performed, for many tasks checking that no uninitialized variables are used the merging costs are less than 10% of the verification costs. Furthermore, the number of merges does not capture how many abstract states must be reexplored due to a merge. Thus, we think it is better to use the predictor known from the evaluation of the configurable program certification approach and look at the ratio of the number of computed abstract successors to the number of ARG nodes. In Tab. 6.4, we highlighted all tasks in bold face for which the ratio is greater or equal than 0.14. In these cases, we observe that often the certificate validation is at least 40% faster than the verification of the generated program. Furthermore, note that if the PfP producer uses a non-static precision adjustment in the property checking analysis, a smaller ratio may be sufficient for an improvement. For some tasks related to the refined property checking analysis $\mathbb{P}\widetilde{\mathbb{V}}$, the consumer analysis benefits from the combination although the ratio is 0.09.

6.4.4 Summary and Final Remarks

Summing up, from a performance point of view the combination is only beneficial when merging and reexploration caused by merging is a significant factor for the verification of the generated program. In these cases, the combination even improves the consumer validation in such a way that it becomes better than the producer analysis of the original program. However, we rarely observe significant ratios of the number of explored abstract states to the number of ARG nodes, the predictor for the costs caused by merging. One reason may be that in program generation syntactical paths are separated to enable the verification with the simpler analysis.

Another drawback of the combination of the approaches is that we always need additional disk storage for the certificate. Furthermore, we observed that in 18 cases certificate validation with the generated program is worse than the producer verification of the original program. In three cases, the verification of the generated programs is already worse. In one case, model checking of `diskperf` with domain $\mathbb{PU}$, the speed-up of the consumer was only 8%. In all remaining cases, the verification time of the original program was already below 0.2 s and we assume that the set up time for reading from disk caused the slowdown.

Although we do not present the numbers, we briefly want to discuss if selecting the best results for certification of both generated programs, the one with and without lazy refinement, instead of those belonging to the generated program makes any difference. For the memory usage, we observe that the best values performs slightly better in 34 cases. The relation of memory usage of certificate validation and generated program verification can be improved by at most 0.05 percentage points. The validation times can be improved in 13 cases. Validation times can become at most 0.3 s faster, but the sum of

all improvements is bounded by 1 s. Hence, we think looking at the best numbers is not relevant.

6.5 Discussion

Compared to the previous two frameworks, the combination is more storage intensive. The consumer receives an often bigger generated program or needs to store an additional certificate. This is a price one has to pay to decrease the trusted computing base and to exclude the termination problem of the Programs from Proofs approach.

Unfortunately, the combination lacks practical applicability. So far, we only realized and evaluated the naïve combination. However, we think that it is not generally impossible to implement the two proposed, more sophisticated combinations. One only needs to further investigate on the CFA construction process to get a deeper insight how the CFA of the generated program will look like.

Also the efficiency of the combination is disappointing, we rarely observed a performance improvement. While we tried to cover a large variety of configurations and properties and, thus, showed generality of our approach, our set of 127 verification tasks is comparatively small. To confirm the performance observations, the evaluation should be repeated on a larger set of benchmarks including larger and more practically relevant programs. Furthermore, we can imagine that our combination benefits from additional parallelization in the consumer analysis, e.g., inspecting a partition element in parallel or checking multiple partition elements in parallel. Note that such a parallelization is simple to realize. However, in this thesis we focused on the feasibility of a combination and the theoretical properties of such a combination. We did not consider further improvements of the combination. The effect of an additional consumer analysis parallelization should definitely be evaluated in future.

From a theoretical point of view, both proposed combinations are satisfactory. Both approaches are well-founded, i.e., they are sound and relatively complete. Furthermore, the consumer analysis can be performed fully automatic. Our integration of the PfP and CPC approach guarantees the automation property stated in the introduction.

Looking at the two alternatives to generate the certificate in the combination, we showed that the constructed certificates are the same when a certificate, a highly reduced certificate, a full, partitioned certificate, or a highly reduced, partitioned certificate for the generated program should be created. For these certificates, the certificate construction depends on the structure of the ARG only. Since the original and the transformed ARG have the same structure, the certificate construction is principally the same. We do not see any reason to apply the first combination, the certificate generation from the transformed producer's proof, which is more memory consumptive due to the ARG transformation, when the producer wants to construct a certificate, a highly reduced certificate, a full, partitioned certificate or a highly reduced, partitioned certificate for the generated program. When the producer would like to built a reduced certificate or a reduced, partitioned certificate, the choice of the combination technique depends on the producer's preferences, less memory consumptive generation or a smaller certificate, and the producer's verification configuration. Remember that for some of our examples the reduced and the highly reduced node set are the same. The equivalence is often caused by the properties of the verification configuration, e.g., no widening in the precision adjustment, an abstract state is merged with at most one explored abstract state, and a merge always guarantees that the termination check returns true. In these cases, the second combination, the transfor-

mation of certificates, should be preferred. In all other cases, the producer's preferences dictate which combination to use.

6.6 Related Work

The combination of the Programs from Proofs and the configurable program certification approach reused many components of these two approaches. The only new component is the certificate construction for the generated program. Since related work for the reused tasks has already been studied in the previous chapters, we focus on related work that deals with certificate construction for generated programs. As done in the naïve combination, one opportunity to produce a certificate for the generated program is to apply an existing approach for program certification on the generated program. Such approaches for program certification have already been considered in Section 4.6. In the following, we only discuss approaches that produce certificates for the generated program while being aware of the program generation.

Proof Producing Generators We start to study approaches in which the entity that produces the generated program helps to or even entirely constructs the proof.

A *certifying compiler* [NL98b, RD99, PHG05, GKD+07] compiles a source program into a target program and additionally constructs a certificate witnessing certain properties of the target program. The first certifying compiler, the Touchstone compiler [NL98b], certifies properties related to type and memory safety. Later, also I/O equivalence of the source and target program [RD99, PHG05] and semantic equivalence of the output traces of the source and target program [GKD+07] are considered.

Compilers are not the only generators that construct certificates witnessing correctness properties during program generation. Fischer et al. [WSF02] present a certification extension of a template based source code synthesizer. In their approach, the template used for code synthesis must carry template annotations related to the property of interest. With the help of the template annotations, the synthesizer adds the concrete annotations, preconditions, postconditions, and loop invarinats to the generated program. Afterwards, a theorem prover utilizes the annotations of the generated program to prove the correctness of the program w.r.t. the property of interest. Myreen et al. [MO14] describe how to translate a program written in higher order logic into a ML program, simultaneously constructing a proof that the transformation is semantics preserving.

In the described approaches, the generator that generates the program also constructs a proof based on the knowledge obtained during generation. In contrast to these approaches, we start in a different situation. We do not start from scratch to get a proof for the generated program, but we already have a proof for the program which we transform and which we use to construct the proof for the generated program. Furthermore, certificate construction must not proceed simultaneously with the program generation.

Translation Validation In 1998, Pnueli, Siegel, and Singerman introduced the concept of *translation validation* [PSS98]. The idea of translation validation is that after translation, an external analyzer gets the original and translated program and constructs a proof that the translation was correct. Different variants of proof and correct translation exist in the literature, some are discussed below. Originally [PSS98], the translation validator inductively proved that the translated program is a refinement of the original one. A similar induction principle is used by Ryabtsev and Strichmann [RS09] to prove correctness of

a Simulink to C translation. Necula [Nec00] describes how an optimizing compiler ensures after each optimization step that the original and the optimized code are equivalent in the sense that the sequence of call and return statements are the same. In the approach by van Engelen et al. [vEWY04], the translation validator proves that the order of function calls and the memory output remains the same. In contrast to typical translation validation approaches, not the complete program, but only those parts that are affected by the transformation are considered in the proof. Myreen et al. [MSG09] present a translation validator proving I/O equivalence. Their translation validator uses proof producing decompilation to extract the function of the translated program and then uses a theorem prover to show equivalence of the original function with the extracted function.

In contrast to translation validation, we do not want to certify a correct transformation but we want to modify an existing proof, certificate s.t. it can be applied on the generated program. Furthermore, the consumer may be unaware of the program generation step at the producer. The generated program is the good traded by the producer. Theoretically, we already showed that the generated program is semantically equivalent. Thus, the producer should ensure that the implemented generation is correct. Although it is currently not integrated into the program generation task, the producer might use the concept of translation validation to ensure that the implemented program generation worked as intended.

Proof Transformation In this paragraph, we examine approaches that adapt existing proofs into a valid proof for the generated program.

The simplest form of adaption is to reuse the existing proof for the original program for the transformed program. Barthe et al. [BRS06] show that for their fixed compilation of their imperative language into their assembly language, the proof obligations obtained by their weakest precondition semantics for the imperative program, the source program, and the compiled assembly program are syntactically equivalent. Proof obligations are preserved along the compilation, a simple translation without any optimizations. Thus, the proof, the certificate, for the source program can be reused for the compiled program. Similar results are obtained for their compilation of a Java fragment into Java bytecode [BGP08]. Saabas et al. [MUSU06] show that in their compilation from language WHILE to SGOTO and SGOTO to WHILE the derivable Hoare logic triples are preserved.

Rival [Riv03, Riv04] proposes a technique called *invariant translation* to transfer the result of an abstract interpretation from a source program to its compiled program. The translation is based on two mappings, a mapping between some of the original and the compiled program locations and a mapping between some variables of the original and the compiled program. The idea is that the mappings relate the original program and the compiled program s.t. after a correct compilation the observational semantics defined by the mappings is the same. To translate an invariant, a mapping of program locations to abstract states, a projection function restricts the invariant to locations and variables occurring in the mapping. Then, an invariant translation function translates the restricted invariant into the restricted abstract domain considered by the compiled program.

Certificate translation [BGKR06, BGKR09, BK08, BK11a] is an approach to transform a certificate for a program along a single program optimization to get a certificate for the optimized program. A certificate consists of a set of proofs for the proof obligations obtained from the specification, the program annotation, and the program semantics. The underlying idea of certificate translation is to integrate the information used for optimization into the certificate. For example, a certifying analyzer is used to compute the

dataflow facts that are needed for the optimization and to create a certificate for the correctness of the dataflow facts. These dataflow facts are integrated into the annotations of the optimized program where needed. Then, the certificate translator builds a certificate from the original certificate, the certificate for the dataflow facts, and possibly a justification for the optimization changes. A justification shows that the annotations of the original program in combination with the dataflow facts imply the annotation of the optimized program. Generally, Barthe et al. show the existence of certificate translators for standard compiler optimizations and present example instances. In early stages [BGKR06, BGKR09], Barthe et al. studied certificate translation in the context of register transfer language programs and proof obligations computed via the weakest precondition calculus for the register transfer language. In subsequent work [BK08, BK11a], Barthe et al. generalized certificate translation. Optimizations of arbitrary programs that can be represented by control flow graphs are considered. Furthermore, program annotations are relaxed to abstract states and proof obligations are derived with the help of abstract interpreters matching the annotation.

A *proof-transforming compiler* [MN07, NMM08, NCM14] translates a source code proof into a bytecode proof. The proof-transforming compiler transforms a proof, e.g., a derivation tree, given in a special Hoare logic. The transformation is based on transformation functions for statements and expressions. Nordio et al. [NCM14] additionally require transformation functions for methods and classes.

Saabas et al. suggest a technique called *proof optimization* [JOS+09] to transform a Hoare logic proof on the original program into a Hoare logic proof for the optimized program. In proof optimization, the dataflow analysis result, which is described by a type derivation and which was used to optimize the program, is utilized to guide the proof transformation. Saabas et al. [JOS+09] demonstrate the concept of proof optimization for a partial redundancy elimination optimization.

Hurlin [Hur09] describes an approach that rewrites a separation logic proof, a Hoare triple based derivation tree, for a sequential program into a valid derivation tree for a parallel program. Based on the inference rules, the basic idea underlying the rewriting is to transform sequential executions that consider different parts of the heap to run in parallel. Thus, rewriting does not only adapt the proof, but simultaneously generates the parallel version of the original program.

Our proof construction uses the same underlying idea as all the approaches presented in this paragraph: transform an existing proof for the original proof into a proof for the generated program. However, the presented approaches consider transformation scenarios in which the program is translated into a low-level language or the program is optimized. We use program transformation to simplify program validation. Furthermore, in our combination the consumer validation is based on a simpler abstract domain than the producer's proof. We cannot reuse the producer's proof. In comparison with proof-transforming compilers [MN07, NMM08, NCM14], proof optimization [JOS+09], and the approach by Hurlin [Hur09] one major difference is that we use proofs, certificates, based on abstract states instead of Hoare logic. Similar to us, Rival [Riv03, Riv04] translates invariants obtained from abstract interpretation. Though, the approach requires a mapping of the original and generated program, an abstract domain for the generated program, and an invariant translation function. At the cost of flexibility, in our approach the abstract domain of the producer's proof dictates the abstract domain for the certificate of the generated program. Moreover, the transformation procedure is automatically defined by the producer's proof. Similar to invariant translation [Riv04], the certificate translation approach [BGKR06] only provides example instances of certificate translators and focuses

on the existence of certificate translators. Additionally, certificate translation focuses on proofs written in a proof algebra. Finally, none of the approaches transform abstract reachability graphs.

Table 6.4: Extract of the comparison of verification and highly reduced, partitioned certificate validation on the generated program which contains all tasks for which the PfP consumer verification merged at least 30-times. Next to the validation times and the memory consumption, for the verification also the number of merges and and the ratio of computed transfer successors to the size of the set of ARG nodes are provided. All times are given in seconds and memory consumption, used heap plus used non-heap, is represented in MB.

$\mathbb{C}_P$	program	$\#\sqcup$	$T_\sqcup$	$V_\mathbb{C}$	$V_\mathbb{V}$	$\frac{V_\mathbb{C}}{V_\mathbb{V}}$	$M_\mathbb{C}$	$M_\mathbb{V}$	$\frac{M_\mathbb{V}}{M_\mathbb{C}}$
$\mathbb{PI}$	**NetBSD***	64	0.02	0.08	0.15	0.53	214.9	225.1	1.05
$\mathbb{OI}$	sendmail	71	0.04	0.24	0.32	0.76	218.1	230.7	1.06
	invertstring	42	0.02	0.20	0.31	0.63	215.7	227.7	1.06
$\mathbb{PS}$	harmonicMean*	50	0.03	0.14	0.18	0.76	207.1	217	1.05
	lockfree3.0*	49	0.02	0.12	0.17	0.67	202.3	214.2	1.06
$\mathbb{OS}$	**testlocks7**	134	0.26	1.15	1.32	0.87	265	259.4	0.98
$\mathbb{VS}$	kundu*	60	0.08	0.36	0.41	0.87	214.9	229.3	1.07
	memslave1*	1744	0.31	0.91	0.58	1.56	273.2	241.1	0.88
	memslave2	2261	0.43	1.26	0.78	1.62	296.1	243.8	0.82
	lockfree3.1*	38	0.02	0.12	0.17	0.69	204	210.3	1.03
$\mathbb{PSI}$	invertsorted*	41	0.07	0.67	0.71	0.95	233.8	238.1	1.02
$\mathbb{PU}$	cdaudio	149	0.04	2.70	2.79	0.97	345.7	354.2	1.02
	pipeline2*	96	0.02	0.23	0.32	0.73	211.5	226	1.07
$\mathbb{OU}$	pipeline	320	0.05	1.15	1.32	0.87	256.1	264.7	1.03
	pipeline2	993k	91.63	Proof Construction Failed					
$\mathbb{VU}$	cdaudio	190	0.06	2.89	2.90	1.00	344.8	351.9	1.02
	pipeline2*	163	0.03	0.32	0.42	0.76	221.5	231.7	1.05
$\mathbb{PV}$	kbfiltr2*	32	0.05	0.77	0.70	1.10	244.2	260.1	1.07
$\mathbb{OV}$	**memslave2**	255	0.14	0.59	0.29	2.03	229.6	233.2	1.02
	testlocks6	63	0.03	0.36	0.45	0.80	226.4	239.4	1.06
	kbfiltr2	32	0.04	0.96	0.92	1.04	251.6	266.3	1.06
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	114	0.09	0.47	0.34	1.39	233.7	241.7	1.03
	memslave2	256	0.16	0.63	0.34	1.88	241.2	243.5	1.01
	kbfiltr2*	34	0.05	0.84	0.71	1.19	250.6	257.9	1.03
	testlocks6*	64	0.04	0.54	0.60	0.91	229.8	247.5	1.08
$\mathbb{OSI}$	invertsorted	40	0.08	0.81	0.80	1.02	237.7	242.8	1.02
$\mathbb{PS}$	kundu*	85	0.09	0.60	0.76	0.80	228.5	241.3	1.06
	transmitter01*	106	0.08	0.18	0.21	0.86	206.1	218.3	1.06
	transmitter02	18465	2.85	5.47	1.36	4.02	553.1	253.7	0.46
$\mathbb{OS}$	transmitter01	88	0.08	0.18	0.23	0.78	205.2	216.2	1.05
	transmitter02	120	0.14	0.31	0.30	1.03	214.4	220.6	1.03
	powerapprox	1022	3.48	10.73	7.39	1.45	996.8	553.7	0.56
$\mathbb{VS}$	transmitter02*	114	0.10	0.21	0.34	0.63	206.2	218.6	1.06
$\mathbb{PU}$	cdaudio	203	0.06	4.69	5.07	0.93	449.8	456.8	1.02
	diskperf	53	0.05	3.64	4.00	0.91	359.1	372.4	1.04
$\mathbb{OU}$	pipeline	461	0.07	1.18	1.32	0.89	260.2	260.3	1.00
	pipeline2	993k	95.84	Proof Construction Failed					
$\mathbb{VU}$	cdaudio	188	0.07	3.63	3.69	0.98	380	381.7	1.00
	pipeline2*	155	0.03	0.31	0.39	0.79	220.2	231.4	1.05

7 Conclusion

In this thesis, we presented and evaluated various solutions to disclose the quality of software programs in global on-the-fly markets. All solutions follow the two-party protocol described in the introduction (see Fig. 1.1). The goal of this protocol is an efficient consumer validation that a purchased program adheres to a certain correctness property. Thereto, the software producer enriches that program with additional information obtained during his verification. To guarantee a broad applicability, the correctness property and the verification can be configured.

We proposed solutions from two lines of research, configurable program certification and Programs from Proofs, and discussed their integration. Always, the producer starts with an enhanced configurable program analysis. The enhanced configurable program analysis uses the concept of a configurable program analysis [BHT07, BHT08] to set the analysis abstraction and its type. Additionally, an enhanced configurable program analysis integrates the inspection of a safety property, which is specified by a property automaton, into the analysis. When the analysis succeeds to verify the configured property, it returns a proof, an abstract reachability graph of a particular shape. Depending on the analysis configuration, the ARG is well-formed or strongly well-formed. Next, the producer uses the ARG to enrich the program with additional information. This is different for the two lines of research.

The configurable program certification approaches construct a certificate. In the basic form, the certificate contains the ARG nodes. The certificate of the first optimization, the reduction approach, mainly stores a subset of the ARG nodes. This subset contains all ARG nodes that cannot be easily recomputed. To get a smaller trusted computing base and to avoid a fixed exploration order during validation, which is especially important for parallel validation, we decided to store more nodes than many related approaches. Our second optimization partitions the certificate, either the basic or the reduced variant, into a set of elements that can be checked independently. Thus, certificate reading and the certificate validation algorithm can be executed in parallel. Given a certificate and the program, the consumer derives his validation configuration from the producer's analysis configuration and then checks that the certificate represents or is extendable to a safe overapproximation of the program's reachable state space.

The Programs from Proofs approach integrates the information, which simplifies the consumer validation, into the program itself. It restructures the program s.t. the restruc-

tured program can be verified with a simple property checking dataflow analysis instead of a refined property checking analysis. Basically, the ARG becomes the restructured program. Thus, the approach deletes infeasible paths detected by the producer analysis or separates syntactical paths that are separated in the ARG. To validate the restructured program, the consumer also executes an enhanced configurable program analysis, but he only uses the property checking analysis part of the producer analysis and reconfigures it to a dataflow analysis.

The proposed combination of the two approaches builds a certificate for the restructured program. Like in the Programs from Proofs approach, the producer starts with a refined property checking analysis and then computes the restructured program. Additionally, one of the certificates known from configurable program certification is constructed for the restructured program. The certificate can be validated with a validator derived from the consumer analysis in the PfP approach, the property checking analysis part of the producer analysis, which is reconfigured to perform a dataflow analysis. To derive the certificate, either the ARG is transformed into an ARG for the restructured program and standard certificate construction is used or a certificate is constructed for the original program that is translated into a certificate for the restructured program. Given the restructured program and the certificate, the consumer performs the standard certificate validation proposed by the respective CPC instance.

To ensure the applicability of these solutions, we examined the five properties put on them as realizations of the two-party protocol depicted in Fig. 1.1. We always proved soundness and relative completeness. Additionally, we identified the constraints that ensure that our solutions run automatically. Practically relevant configurations meet these constraints. Efficiency is studied experimentally. All approaches are implemented in the software analysis tool CPAchecker [BK11b]. For the certificate validation algorithms, we even implemented a sequential and a parallel version. We evaluated all our approaches with different analyses, programs, and properties. Our experiments revealed that not only in theory, but also in practice our approaches are generally applicable. For all of the approaches, we observed that for some tasks the consumer validation is efficient, much better than the producer analysis w.r.t. execution time and memory usage. Furthermore, efficiency is not restricted to particular properties or analyses. For practical usability, we also analyzed when the approaches are efficient. Next, we compare the strengths and weaknesses of our approaches.

7.1 Discussion

First, let us look at the trusted computing base of the approaches. In the CPC approaches, the consumer must trust the validation configuration, typically the abstract domain, the transfer relation, and the termination check operator of the verification configuration, plus the respective validation algorithm. In contrast, the consumer validation in the PfP approach relies only on the CPA algorithm and the property checking analysis. The combined approach neither requires a merge and precision adjustment operator nor the enabler analysis. Thus, the combination of the PfP and the CPC approach has the smallest trusted computing base. However, it is difficult to compare the trusted computing base of the PfP approach with the one of the CPC approach. We think that the complexity of the CPA algorithm and the validation algorithms are similar. It should not matter much which algorithm is in the trusted computing base. The main problem are the configurations. The validation configuration can get rid of the merge and the precision

adjustment operator. The analysis configuration of the consumer in the PfP approach does not require the enabler analysis. Although it is difficult to compare these elements, we believe that the enabler analysis is more complex. The trusted computing base of the PfP approach seems to be smaller than the one of the CPC approach.

Next, we compare the approaches based on our five requirements. All approaches are proven sound. In contrast to the CPC approach and the combination, the PfP approach is not always relatively complete. For some producer analysis configurations, we fail to prove termination of the consumer analysis. These configurations are uncommon in practice. Also, the CPC approach has a theoretical limitation. While the PfP approach and the combination always run fully automatic, the CPC validation requires manual intervention when the termination check is not well-behaving. However, practically relevant termination checks are well-behaving. Once again, the combination performs best with respect to automation, soundness, and relative completeness. The CPC and the PfP approach suffer from different weaknesses, which are less important in practice.

Now, we discuss the generality of the approaches. Some of the CPC approaches require monotonic transfer relations and all of them require well-behaving coverage checks to ensure relative completeness. One can always choose a variant that does not rely on monotonic transfer relations. Additionally, in theory one could always select the well-behaving coverage check $\mathsf{cover}(e, S) := [\![e]\!] \subseteq \bigcup_{e' \in S} [\![e']\!]$. However, its implementation might be complicated. Theoretically, the CPC approach works with every configuration. In contrast, the PfP approach, and thus the combination, is not applicable to such a broad class of analyses. First, the producer analysis must be at least flow-sensitive – remember we required the location CPA. Furthermore, the transfer relation must be a function and the producer analysis requires a refined property checking analysis, a restrictive form of a combined CPA. This combined CPA links an enabler analysis with a property checking analysis s.t. the enabler analysis does not directly influence the property checking analysis and only the property checking analysis inspects the property. Moreover, the PfP approach and the combination can only be used to assure properties expressible in control state unaware property automata.

The last requirement is efficiency. First, we like to mention that validation in both approaches often performs equally well or better than competing validation approaches. Furthermore, the best CPC approach and the PfP approach outperform the verification for many tasks. Considering the best CPC approach, we observe that the CPC approach achieves higher speed-ups and more severely decreases the memory usage. The CPC approach also achieves significant speed-ups for more tasks, but relating the number of tasks with high speed-ups to the number of tasks the PfP approach is better. Of course, such a comparison is not really meaningful. Since we used a much larger set of tasks in the CPC evaluation and we picked our PfP tasks manually, we could have selected our PfP tasks luckily. However, we also compared the PfP and the CPC approach on the PfP tasks and often the PfP validation is slightly faster. This supports the impression that the PfP approach is more efficient than the CPC approach. In cases, in which the combination can improve the PfP approach considerably, the combination can be better than the CPC approach, but need not be.

Besides efficiency, we also studied the storage overhead of our approaches. The CPC approach additionally needs to store a certificate and in the PfP approach the program can become significantly larger. Restructured programs are not always larger. Furthermore, during evaluation of the PfP approach we observed that the CPC certificates are often larger than the increase of the restructured program. The storage overhead of the CPC

approach seems to be larger than the one of the PfP approach. Since the combination stores the restructured program plus a certificate, its storage overhead is the largest.

Finally, we would like to remark that certificate generation, particularly in the partitioning approach, can be more challenging than the construction of the restructured program. On the one hand, the construction of a (full,) partitioned certificate is sometimes really time consuming. On the other hand, the configuration of a proper partition is already difficult due to the large degree of opportunities.

Based on these insights, we suggest to use the CPC approach whenever the PfP approach is not applicable. Since at first glance the PfP approach seems to be slightly more efficient, has less storage overhead, and likely a smaller trusted computing base, we would prefer the PfP approach over the CPC approach. Nevertheless, when the consumer validation in the PfP approach does not terminate or is inefficient, i.e., the number of transfer successors computed during validation is much larger than the number of program locations, we suggest to use the combination of the approaches. We think that whenever the producer would like to use the PfP approach, it can be expected that he tries out the consumer validation, too. If the validation reveals, that it will not terminate or takes too long, the producer should switch to the combined approach.

7.2 Future Work

For all our approaches, we have one improvement in mind, which aims at an extension of the producer analysis. Sometimes, a single analysis is not sufficient to prove that a program fulfills a certain property. In such a case, multiple analyses can be used, each considering a part of the program. Techniques like conditional model checking [BHKW12] can be used to realize such a cooperation of analyses. When all analyses are successful, each analysis produced an incomplete, but safe ARG. Furthermore, the set of ARGs constructed by all analyses form a valid proof. To support verification with multiple analyses in our approaches, we suggest to transform the set of ARGs into a single ARG. We assume that the root node of all ARGs considers the same set of locations. The idea for the combination is to use some kind of product construction. The combination starts with the combination of the root nodes. Given a combined state, we determine the ARG successors in the combined ARG. For each CFA edge appearing on a label of an ARG edge which leaves one of the ARG nodes considered in the combined node, we check if transfer successors for this edge exist for all states considered in the combined state. If no successor exists for at least one of these states, no ARG edge for this CFA edge will be introduced in the combined ARG. Otherwise, we compute for each combined state the ARG successors reachable via that CFA edge. If no ARG successors exists for a state, i.e., the corresponding partial verification does not consider that part of the program state space, we will use the top state as only successor. Then, the cartesian product of these successors become those ARG successors that are reachable by that CFA edge from the combined state. We are confident that we can use the combined ARG to construct certificates that can be validated with a configurable certificate validator derived from the product combination of the analyses. Furthermore, we think that the combined ARG can be used in the PfP approach or the combination of the PfP and the CPC approach when the analyses consider the same property checking analyses. Nevertheless, termination of the PfP consumer verification may still be an issue. However, we are not sure whether our idea to support verification by multiple analyses also works if not only the enabler analysis but also the property checking analysis is different. Moreover, we still do not

know which safety properties are covered by the PfP approach. This question leads us to future work w.r.t. our property specification.

Property Specification First of all, we are interested in the expressiveness of our property specification language, the property automata. We would like to know if property automata cover all safety properties or if property automata can only describe a subclass of the safety properties. In the latter case, we want to investigate which class(es) of safety properties can and cannot be expressed by a property automaton.

On purpose, we restricted property automata to deterministic transition relations. Our intuition was that the property automata do not become more expressive with non-deterministic transition relations. To prove this intuition, we suggest to use a powerset construction similar to the one that is used to prove the equivalence of deterministic and nondeterministic finite automata [RS59]. However, we must take the concrete states into account. First, the transitions must be split s.t. after splitting the resulting property automaton is equivalent to the original property automaton, but for any two transitions $(q, op_1, C_1, q_1), (q, op_2, C_2, q_2)$ either $op_1 \neq op_2$, $C_1 = C_2$, or $C_1 \cap C_2 = \emptyset$. Afterwards, standard powerset construction can be used.

For the Programs from Proofs approach, we need control state unaware property automata. Syntactically, control state unaware property automata are a subclass of the property automata. However, we are unsure if this is also a semantic restriction, i.e., the PfP approach can assure fewer properties. At best, we could transform an arbitrary property automaton $\mathcal{A}$ into a control state unaware property automaton $\mathcal{A}'$ s.t. the two property automata agree on program safety, i.e., $\forall I \subseteq C, P : P \models_I \mathcal{A} \iff P \models_I \mathcal{A}'$. For the Programs from Proofs approach, it would be sufficient to show a weaker property. The two property automata, the property automaton $\mathcal{A}$ and the transformed, control state unaware property automaton $\mathcal{A}'$, must only agree on the safety of the original program P, i.e., $\forall I \subseteq C : P \models_I \mathcal{A} \iff P \models_I \mathcal{A}'$. We could imagine that for deterministic programs it is possible to find a transformation that ensures at least $P \models \mathcal{A} \iff P \models \mathcal{A}'$. Instead of referring to the corresponding location directly, our idea is to encode the paths from the initial location to the corresponding location in the control state unaware property automaton. Typically, the producer and the consumer are interested in program safety w.r.t. the states considering the initial program location. From a practical point of view, a transformation that ensures at least $P \models \mathcal{A} \iff P \models \mathcal{A}'$ often eliminates the restriction to control state unaware property automata. Still, there are open research questions w.r.t. the proposed approaches. Next, we consider open questions for the CPC approach.

Configurable Program Certification We have two ideas for future improvements of the configurable program certification approaches.

To reduce the memory consumption, we can imagine to follow the idea of Klohs et al. [Klo09] and keep abstract states in memory as long as they are required for certificate checking. Property checking must be performed per abstract state and must be moved to the loop that explores the transfer successors of an abstract state. A simple solution for the validation of reduced certificates, which does not need any bookkeeping, forgets all explored states that are not part of the reduced certificate, after the state's successors are explored. However, it must use a counter for the explored states instead of using the reached set size. For the other validation algorithms, we may require some bookkeeping, which tells the respective algorithm when to forget an abstract state.

Software programs are often updated, errors are corrected or new features are integrated. So far, each time the complete certification process must be performed after each

update. Following the idea of Albert et al. [AAP06], we do not want to perform the complete CPC approach, especially not the complete consumer validation. Instead, we would like to send to the consumer the information how to update the program and how to adapt the certificate and then let the consumer revalidate those parts that are affected by the update or the changed certificate. The next paragraph presents open research questions w.r.t. the Programs from Proofs approach.

Programs from Proofs Program restructuring is the factor of success for the PfP approach, but restructuring also invalidates certain properties of the program. We know that only properties expressible by control state unaware property automata remain valid after program restructuring. Moreover, validity is not sufficient. To convince someone of the validity of a property, it must be provable. We already showed that provability, except for termination, is kept when the property can be proven with a dataflow analysis. However, many different analysis types may be used to prove a property. Thus, we should investigate under which conditions which other analyses can prove a property on the restructured program, when they already successfully proved the property on the original program. Similarly, the next paragraph proposes to extend the integration of the PfP and the CPC approach to show the validity of a property on the generated program with the help of a translated certificate.

Integration of PfP and CPC To further integrate the two approaches, we are specifically interested in certificate translation. The question is whether it is possible to transform a valid certificate for the original program into a valid certificate for the generated program. This has several advantages. First, we do not need to re-execute the producer part of the CPC approach after the application of the PfP approach. The PfP and the CPC approach can be used together to ensure different properties without considering the order of their application. Second, also our proposed integration of the PfP and the CPC approach becomes transitive.

Similar to program provability, we think that certificate translation may be infeasible if the property is not expressed by a control state unaware property automaton. If the abstract states do not consider concrete locations, we can imagine to reuse the existing certificates. Given that abstract states store concrete locations, a first idea for the translation of the basic certificate copies for each location $l' = (l, \dots)$ in the generated program all abstract states e referring to location l and replaces the location information in e by l'. We are not sure if a similar transformation can be applied to the other types of certificates. However, we believe that the idea can be refined to transform at least a well-formed ARG into a well-formed ARG for the generated program. This ARG can be used to construct the other certificates.

7.3 Resume

We proposed three instances of the abstract protocol (cf. Fig. 1.1), a generic solution to disclose the quality of a software program in global on-the-fly markets. All three of them are competitive with state of the art and more or less meet the five requirements put on all instances of the abstract protocol. Moreover, the choice for one of the three instances mainly depends on the property of interest and the producer verification. Future research directions provide us an even deeper understanding of the frameworks' limits and, more importantly, enlarge the application scenario of the frameworks presented in this thesis.

A Proofs

A.1 Outstanding Proofs for Chapter 2 Programs and Their Verification

Lemma 2.2. *If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns $(true, \dots)$, then for every path $p \in paths_P([\![e_0]\!])$ there exists a configuration sequence $(c_0, q_0) \dots, (c_n, q_n)$ for p and $\mathcal{A}$ s.t. $\forall 0 \leq j \leq n : q_j \neq q_{\text{err}}$.*

Proof. Let $\mathsf{reached}'$ be the reached set considered in line 29. We show by induction over the length of paths $p \in paths_P([\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}})$ that a configuration sequence $(c_0, q_0) \dots, (c_n, q_n)$ for p and $\mathcal{A}$ exists s.t. $\forall 0 \leq j \leq n : \exists e_j \in \mathsf{reached}' : c_j \in [\![e_j]\!]_{\mathbb{C}^{\mathcal{A}}} \wedge e_j = (\cdot, q_j) \wedge q_j \neq q_{\text{err}}$.

Basis Let $c_0 \in paths_P([\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}})$ be a path of length 0. From the definition of paths, we know that $c_0 \in [\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}}$. By definition (c_0, q_0) is a configuration sequence for path c_0 and $\mathcal{A}$. Since Algorithm 2 added e_0 to $\mathsf{reached}$ in line 1, it replaces elements e'' in $\mathsf{reached}$ only in line 12 and only by more abstract states e_{new}, $e'' \sqsubseteq_{\mathbb{C}^{\mathcal{A}}} e_{\text{new}}$ (definition of e_{new} and Eq. 2.5), and partial order $\sqsubseteq_{\mathbb{C}^{\mathcal{A}}}$ is transitive, there exists $e'_0 = (e', q') \in \mathsf{reached}'$ with $e_0 \sqsubseteq_{\mathbb{C}^{\mathcal{A}}} e'_0$. By requirements on $\sqsubseteq_{\mathbb{C}^{\mathcal{A}}}$ and $c_0 \in [\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}}$, we get $c_0 \in [\![e'_0]\!]_{\mathbb{C}^{\mathcal{A}}}$. By definition of $\sqsubseteq_{\mathbb{C}^{\mathcal{A}}}$, we can conclude that $q_0 \sqsubseteq q'$. Since $\mathcal{Q}$ is a flat lattice, either $q' = q_0$ or $q' = q_\top$. We know that $q' = q_0 \neq q_{\text{err}}$ because Algorithm 2 returned $(true, \cdot, \cdot)$.

Step Let $p := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \xrightarrow{g_i} c_i \in paths_P([\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}})$ be a path of length i. By definition of paths, $p' := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \in paths_P([\![e_0]\!]_{\mathbb{C}^{\mathcal{A}}})$ and p' has length $i - 1$. From induction, we know that a configuration sequence $(c_0, q_0) \dots, (c_{i-1}, q_{i-1})$ for p' and $\mathcal{A}$ exists s.t. $\forall 0 \leq j \leq i - 1 : \exists e_j \in \mathsf{reached}' : c_j \in [\![e_j]\!]_{\mathbb{C}^{\mathcal{A}}} \wedge e_j = (\cdot, q_j) \wedge q_j \neq q_{\text{err}}$. Algorithm 2 always adds a pair of abstract state e and a precision π to $\mathsf{waitlist}$ if it adds e to $\mathsf{reached}$, it removes an element $(e', \cdot)$ from $\mathsf{waitlist}$ only in line 4 or if it removes e' from $\mathsf{reached}$. Since $e_{i-1} \in \mathsf{reached}'$, we know that there exists

at least one pair (e_{i-1}, π_{i-1}) which was removed from waitlist in line 4. Hence, Algorithm 2 computed e_{i-1}'s abstract successors in line 6. From $c_{i-1} \in [\![e_{i-1}]\!]_{\mathbb{C}^\mathcal{A}}$ and $c_{i-1} \xrightarrow{g_i} c_i$, we can conclude that an abstract transition$(e_{i-1}, g_i, e_s) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ exists with $c_i \in [\![e_s]\!]_{\mathbb{C}^\mathcal{A}}$ (overapproximation of transfer relation Eq. 2.2). From definition of $\rightsquigarrow_{\mathbb{C}^\mathcal{A}}$, especially transfer relation of most precise enhancement and enhancement, and $q_{i-1} \in Q$ (follows from configuration sequence for p'), we can conclude that $e_s = (\cdot, q_s)$ and either $(q_{i-1}, op, C_{\text{sub}}, q_s) \in \delta$ with $c_i \in C_{\text{sub}}$ and $g_i = (\cdot, op_i, \cdot)$, or $q_s = q_\top$. Either Algorithm 2 added e_s into reached or the termination check returned true in line 18 while checking e_s. In the first case, let $e_r = e_s$. In the second case, we know from soundness of the termination check, that there exists $e_c \in$ reached with $c_i \in [\![e_c]\!]_{\mathbb{C}^\mathcal{A}}$. By the definition of most precise enhancement and enhancement, we know that $e_c = (\cdot, q_c)$ and $q_s \sqsubseteq q_c$. Since $\mathcal{Q}$ is a flat lattice, either $q_c = q_s$ or $q_c = q_\top$. In the second case, now let $e_r = e_c$. Since e_r was an element of reached, Algorithm 2 replaces elements e'' in reached only in line 12 and only by more abstract states e_{new}, $e'' \sqsubseteq_{\mathbb{C}^\mathcal{A}} e_{\text{new}}$ (definition of e_{new} and Eq. 2.5), and partial order $\sqsubseteq_{\mathbb{C}^\mathcal{A}}$ is transitive, there exists $e'_r = (e*, q*) \in$ reached$'$ with $e_r \sqsubseteq_{\mathbb{C}^\mathcal{A}} e'_r$. By requirements on $\sqsubseteq_{\mathbb{C}^\mathcal{A}}$ and $c_i \in [\![e_r]\!]_{\mathbb{C}^\mathcal{A}}$, we get $c_i \in [\![e'_r]\!]_{\mathbb{C}^\mathcal{A}}$. By definition of $\sqsubseteq_{\mathbb{C}^\mathcal{A}}$ and transitivity of $\sqsubseteq_{\mathbb{C}^\mathcal{A}}$, we can conclude that $q_s \sqsubseteq q*$. Since $\mathcal{Q}$ is a flat lattice, either $q_s = q*$ or $q* = q_\top$. We know that $q* \neq q_\top$ and $q* \neq q_{\text{err}}$ because Algorithm 2 returned $(true, \cdot, \cdot)$. From this, we conclude that $q_s \in Q$ and $q_s \neq q_{\text{err}}$. We finally get from the definition of a configuration sequence that $(c_0, q_0) \ldots, (c_{i-1}, q_{i-1})(c_i, q_s)$ is a configuration sequence for p and $\mathcal{A}$. The induction hypothesis follows.

□

Lemma 2.5. *If Algorithm 2 started with CPA $\mathbb{C}^\mathcal{A}$ enhanced with property automaton $\mathcal{A}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^\mathcal{A}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^\mathcal{A}}$ returns $(\cdot, \cdot, R^P_{\mathbb{C}^\mathcal{A}})$, then $R^P_{\mathbb{C}^\mathcal{A}}$ is an abstract reachability graph for P and $\mathbb{C}^\mathcal{A}$.*

Proof. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$.

Algorithm 2 only adds abstract states of the input domain $D_{\mathbb{C}^\mathcal{A}}$ to reached. It follows that reached $\subseteq E_{\mathbb{C}^\mathcal{A}}$. Since Algorithm 2 adds at most one element to reached when it changes reached and it terminates, it can change reached only finitely many times. Hence, reached is finite. In line 29, $N' =$ reached. Thus, $N' \subseteq E_{\mathbb{C}^\mathcal{A}}$ and N' is finite.

In line 2, root becomes e_0 and in line 1 reached $:= \{e_0\}$. Obviously, $e_0 \in$ reached. If Algorithm 2 removes root from reached which is only possible in line 12 ($root = e''$), then in line 15 it updates root by $e_{\text{new}} \in$ reached which was added to reached in line 12 ($e'' \neq e_{\text{new}}$). We can conclude from line 29 that $root' = root$, $N' =$ reached and from $root \in$ reached we get $root' \in N'$.

Algorithm 2 starts with the empty set of edges which obviously fulfills $G_{\text{ARG}} = \emptyset \subseteq$ reached $\times\, G_{\text{CFA}} \times$ reached. Removal of edges from G_{ARG} never violates the property $G_{\text{ARG}} \subseteq$ reached $\times\, G_{\text{CFA}} \times$ reached, but the removal of states from reached may do. Algorithm 2 only removes elements e'' from reached in line 12. We show that after line 13 it holds that $G_{\text{ARG}} \subseteq$ reached$\times G_{\text{CFA}} \times$reached if it was true before. If Algorithm 2 removes e'' and the property was true before, all edges $(e, g, e') \in G_{\text{ARG}}$ with $e = e'' \vee e' = e''$ violate the property. Before Algorithm 2 removes all these edges in line 13, it adds edges to G_{ARG}. It adds edges (e_p, g, e_{new}) if (e_p, g, e'') is contained in the set of edges. Since e_{new} was added to reached in line 12 ($e_{\text{new}} \neq e''$), (e_p, g, e'') was contained before line 12 and the property is assumed to be true before the execution of line 12, we know that $g \in G_{\text{CFA}}$ and $e_p \in$ reached or $e_p = e''$. If (e_p, g, e_{new}) is added, this edge either will not

violate the property or it will be removed again in the same line. Thus, after line 13 the property is reestablished if it was true before line 12. In lines 24 and 28, Algorithm 2 only adds an edge (e, g, e') if $e \in \mathsf{reached}$ (condition of if), $e' \in \mathsf{reached}$ (added in line 19 and definition of coveringSet, respectively), and $g \in G_{\text{CFA}}$ (loop body of for in line 5). Thus, if $G_{\text{ARG}} \subseteq \mathsf{reached} \times G_{\text{CFA}} \times \mathsf{reached}$ holds before line 24 or 28, it holds after their execution. The set of ARG edges adheres to the condition $G_{\text{ARG}} \subseteq \mathsf{reached} \times G_{\text{CFA}} \times \mathsf{reached}$ in the beginning, if this property holds and it is violated by an execution step (only possible in line 12), the next execution step reestablishes the property, we can conclude that the property is always true outside the loop body of the most inner for loop. Since in line 29 $N' = \mathsf{reached}$, $G'_{\text{ARG}} = G_{\text{ARG}}$, and Algorithm 2 never violates the condition $G_{\text{ARG}} \subseteq \mathsf{reached} \times G_{\text{CFA}} \times \mathsf{reached}$ outside the loop body of the most inner for loop, we can conclude that $G'_{\text{ARG}} \subseteq N' \times G_{\text{CFA}} \times N'$.

In line 2, N_{cov} becomes the empty set which is obviously a subset of any reached set. If Algorithm 2 removes a node e'' from reached which is only possible in line 12, then in line 17 it removes the same node from N_{cov}. If it adds node e_{new} in line 17 or node e_{prec} in line 23, we know that e_{new} was added to reached in line 12 and e_{prec} in line 19. If Algorithm 2 adds the set coveringSet to N_{cov}, we know from the definition of coveringSet in line 26 that only elements from reached are added. We can conclude that in line 29 $N_{\text{cov}} \subseteq \mathsf{reached}$. Furthermore, we know that in line 29 $N'_{\text{cov}} = N_{\text{cov}}$ and $N' = \mathsf{reached}$. Hence, $N'_{\text{cov}} \subseteq N'$.

From the definition, we conclude that $R^P_{\mathbb{C}^\mathcal{A}}$ is an ARG for P and $\mathbb{C}^\mathcal{A}$. □

Lemma 2.6. *If Algorithm 2 started with CPA $\mathbb{C}^\mathcal{A}$ enhanced with property automaton $\mathcal{A}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^\mathcal{A}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^\mathcal{A}}$ returns $(\cdot, \cdot, R^P_{\mathbb{C}^\mathcal{A}})$, then $R^P_{\mathbb{C}^\mathcal{A}}$ is an ARG for P and $\mathbb{C}^\mathcal{A}$ which is rooted, complete, well-covered, and well-constructed.*

Proof. Let $R^P_{\mathbb{C}^\mathcal{A}} = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$. From Lemma 2.5, we know that $R^P_{\mathbb{C}^\mathcal{A}}$ is an ARG for P and $\mathbb{C}^\mathcal{A}$.

Rootedness In line 2, root becomes e_0. In this case, $e_0 \sqsubseteq root$ (reflexivity of partial order $\sqsubseteq$). Thereafter, Algorithm 2 replaces root in line 15 by e_{new} if $root = e''$. We know that e_{new} is the result of a merge with second parameter e''. Hence, from Eq:. 2.5 it follows that $root = e'' \sqsubseteq e_{\text{new}}$. From $e_0 \sqsubseteq root$ and $root \sqsubseteq e_{\text{new}}$ we get $e_0 \sqsubseteq e_{\text{new}}$ (transitivity of partial order $\sqsubseteq$). We can conclude from line 29 that $e_0 \sqsubseteq root'$.

Completeness Show that before each iteration of the while loop $\forall n \in \mathsf{reached}, g \in G_{\text{CFA}} : \exists (n, \cdot) \in \mathsf{waitlist} \vee (n, g, e) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}} \implies \exists n' \in E_{\mathbb{C}^\mathcal{A}} : e \sqsubseteq n' \wedge ((n, g, n') \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S))$.

Before the first execution of the while loop, Algorithm 2 adds e_0 to reached and (e_0, π_0) to waitlist the claim holds.

Consider arbitrary while loop iteration i. For every element e, which Algorithm 2 adds to reached in lines 4-28, it adds an element $(e, \cdot)$ to waitlist. Furthermore, in lines 5-28 an element $(e, \cdot)$ is removed from waitlist only if e is removed from reached. For states added to reached or removed from waitlist after line 5 nothing must be shown. Additionally, deleting a node from reached does not affect completeness.

In iteration i, (e, π) is popped from waitlist. If $(e, \cdot)$ is (re)added to waitlist or e is removed from reached in the same iteration nothing needs to be shown. Assume no element $(e, \cdot)$ will be added to waitlist and e is not removed. For each

CFA edge $g \in G_{\text{CFA}}$, Algorithm 2 explores each $(e, g, e') \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$. Due to $e \in \mathsf{reached}$ and $\neg\exists(e, \cdot) \in \mathsf{waitlist}$ (assumption), it either adds (e, g, e_{prec}) to G_{ARG} ($\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathit{false}$) or for all $e_r \in \mathsf{coveringSet}$ it adds edges (e, g, e_r) to G_{ARG}. We get by definition of e_{prec} in line 7 and from prec that $e' \sqsubseteq e_{\text{prec}}$. In the first case, there exists e_{prec} with $e' \sqsubseteq e_{\text{prec}}$ and $(e, g, e_{\text{prec}}) \in G_{\text{ARG}}$. In the second case, we know that $\mathsf{coveringSet} \subseteq N_{\text{cov}}$ (due to line 27), and $\mathsf{stop}(e_{\text{prec}}, \mathsf{coveringSet})$. It follows that $\mathsf{coveringSet} \subseteq \{n'' \in N_{\text{cov}} \mid (e, g, n'') \in G_{\text{ARG}}\}$ and, thus, $\mathsf{coveringSet} \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e, g, n'') \in G_{\text{ARG}}\}$. Hence, there exist e_{prec} with $e' \sqsubseteq e_{\text{prec}}$ and $S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(e_{\text{prec}}, S)$.

After execution of iteration i the property would hold, if Algorithm 2 never removed edges from G_{ARG} nor deleted nodes from N_{cov}, since it only removes $(e, \cdot)$ from $\mathsf{waitlist}$ in lines 5-28 if it removes e from $\mathsf{reached}$. It remains to be shown that the removal of edges from G_{ARG} nor the removal of nodes from N_{cov} to the set G_{ARG} of edges, never violates the property, i.e., if $n \in \mathsf{reached}, g \in G_{\text{CFA}}$, $(n, g, e) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ and there exists $n' \in E_{\mathbb{C}^\mathcal{A}} : e \sqsubseteq n' \wedge ((n, g, n') \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S))$, then after the removal of edges from G_{ARG} or nodes from N_{cov} this is still true or there exists $(n, \cdot) \in \mathsf{waitlist}$. Algorithm 2 removes edges in line 13 and 20. In line 13, Algorithm 2 removes edges $(e'', \cdot, \cdot)$, $(e_{\text{new}}, \cdot, \cdot)$, and (e_p, g, e''). Since it removed e'' from $\mathsf{reached}$ ($e'' \neq e_{\text{new}}$), removing edges $(e'', \cdot, \cdot)$ does not violate the property. Since $e'' \neq e_{\text{new}}$, in line 11 it adds $(e_{\text{new}}, \cdot)$ to $\mathsf{waitlist}$. Removing edges $(e_{\text{new}}, \cdot, \cdot)$ does not violate the property. Consider edges (e_p, g, e'') with $e_p \neq e''$ and $e_p \neq e_{\text{new}}$ (the remaining cases). Let $(e_p, g, \widehat{e}) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ be arbitrary and there exists $n' \in E_{\mathbb{C}^\mathcal{A}} : \widehat{e} \sqsubseteq n' \wedge ((e_p, g, n') \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S))$ If $\exists(e_p, \cdot) \in \mathsf{waitlist}$ nothing must be shown. Assume $\neg\exists(e_p, \cdot) \in \mathsf{waitlist}$. If $(e_p, g, n') \in G_{\text{ARG}}$, we only need to consider the case that $(e_p, g, n') = (e_p, g, e'')$ (removal of edge). Algorithm 2 adds an edge (e_p, g, e_{new}) to the set of ARG edges which is not removed ($e_{\text{new}} \neq e''$, $e'' \neq e_p \neq e_{\text{new}}$). Due to the definition of e_{new} in line 9, we get $e'' \sqsubseteq e_{\text{new}}$ (property of merge). Since partial order $\sqsubseteq$ is transitive, we get $\widehat{e} \sqsubseteq e_{\text{new}}$. The property is not violated. If $\exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S)$, we only need to consider the case that $e'' \in \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\}$ due to an edge $(e_p, g, n'') = (e_p, g, e'')$ with $n'' = e'' \in N_{\text{cov}}$ (removal of edge). Algorithm 2 adds an edge (e_p, g, e_{new}) to the set of ARG edges which is not removed ($e_{\text{new}} \neq e''$, $e'' \neq e_p \neq e_{\text{new}}$). Due to the definition of e_{new} in line 9, we get $e'' \sqsubseteq e_{\text{new}}$ (property of merge). Furthermore, in line 17 Algorithm 2 adds e_{new} to N_{cov}. We infer that $S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} \sqsubseteq (\{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} \setminus \{e''\}) \cup \{e_{\text{new}}\}$. We have at most one element (e_p, g, e'') which is replaced by (e_p, g, e_{new}). After line 17, the property is reestablished. In line 20, Algorithm 2 removes edges $(e_{\text{prec}}, \cdot, \cdot)$. Due to line 19, we know that $(e_{\text{prec}}, \pi_{\text{prec}}) \in \mathsf{waitlist}$. Removing these edges does not violate the property.

Algorithm 2 only removes nodes from N_{cov} in line 17. Removing a node e'' from N_{cov} may only violate the property if an edge $(\cdot, \cdot, e'')$ exists. Due to line 13, we know that no such edge exists.

Thus, after iteration i and before iteration $i + 1$ the hypothesis holds.

At line 29, $N = \mathsf{reached}$ and $\mathsf{waitlist} = \emptyset$. We conclude that $R^P_{\mathbb{C}^\mathcal{A}}$ is complete.

Well-Coveredness Show that before each iteration of the while loop a total, injective function $\text{cov} : S'_{\text{TCNC}} \to \mathsf{reached} \setminus N_{\text{cov}}$ exists with $\forall(n, g, e) \in S'_{\text{TCNC}} : e \sqsubseteq$

$\mathrm{cov}((n,g,e)) \wedge (n,g,\mathrm{cov}((n,g,e))) \in G_{\mathrm{ARG}}$ and $S'_{\mathrm{TCNC}} := \{(n,g,e) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \mid n \in \mathsf{reached} \setminus \{e \mid (e,\pi) \in \mathsf{waitlist}\} \wedge g \in G_{\mathrm{CFA}} \wedge \neg\exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : e \sqsubseteq n' \wedge ((n,g,n') \in G_{\mathrm{ARG}} \wedge n' \in N_{\mathrm{cov}} \vee \exists S \sqsubseteq \{n'' \in N_{\mathrm{cov}} \mid (n,g,n'') \in G_{\mathrm{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n',S)\}$.

Before the first execution of the while loop, Algorithm 2 adds e_0 to reached and (e_0,π_0) to waitlist the claim holds.

Consider arbitrary while loop iteration i. For every element e, which Algorithm 2 adds to reached in lines 4-28, it adds an element $(e,\cdot)$ to waitlist. Furthermore, in lines 5-28 an element $(e,\cdot)$ is removed from waitlist only if e is removed from reached. For states added to reached or removed from waitlist after line 5 nothing further must be shown.

First show that adding an element $\widehat{e}$ to N_{cov} does not violate the property. We need to show that if $\exists (n,g,e) \in S'_{\mathrm{TCNC}} : \mathrm{cov}((n,g,e)) = \widehat{e}$ before execution of the respective line, then $(n,g,e) \notin S''_{\mathrm{TCNC}}$ (S''_{TCNC} denotes the respective set after execution of the respective line). We know that $(n,g,\mathrm{cov}((n,g,e))) = (n,g,\widehat{e}) \in G_{\mathrm{ARG}}$ and $e \sqsubseteq \mathrm{cov}((n,g,e)) = \widehat{e}$. After adding $\widehat{e}$ to N_{cov}, we get that $(n,g,\widehat{e}) \in G_{\mathrm{ARG}}$, $e \sqsubseteq \widehat{e}$, and $\widehat{e} \in N_{\mathrm{cov}}$. Hence, $(n,g,e) \notin S''_{\mathrm{TCNC}}$. Thus, the function $\mathrm{cov}' : S''_{\mathrm{TCNC}} \to \mathsf{reached} \setminus N_{\mathrm{cov}}$ with $\mathrm{cov}'((n',g',e')) = \mathrm{cov}((n',g',e'))$ fulfills the desired properties (no new elements can be added, because reached was not changed).

In iteration i, (e,π) is popped from waitlist. If $(e,\cdot)$ is added to waitlist or removed from reached in the same iteration nothing needs to be shown. Assume no element $(e,\cdot)$ will be added to waitlist and e is not removed from reached. For each CFA edge $g \in G_{\mathrm{CFA}}$, Algorithm 2 explores each $(e,g,e') \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$. Due to $e \in \mathsf{reached}$ and $\neg\exists(e,\cdot) \in \mathsf{waitlist}$ (assumption), it either adds (e,g,e_{prec}) to G_{ARG} ($\mathsf{stop}(e_{\mathrm{prec}},\mathsf{reached}) = \mathit{false}$) or for all $e_r \in \mathsf{coveringSet}$ it adds edges (e,g,e_r) to G_{ARG}. We get by definition of e_{prec} in line 7 and from prec that $e' \sqsubseteq e_{\mathrm{prec}}$. First, consider $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(e_{\mathrm{prec}},\mathsf{reached}) = \mathit{false}$. If e_{prec} is added to N_{cov}, we know that $(e,g,e') \notin S'_{\mathrm{TCNC}}$. Since adding a state to N_{cov} does not violate the property, we know that the desired property is fulfilled. If e_{prec} is not an element of N_{cov}, we know that $e_{\mathrm{prec}} \notin \mathsf{reached}$ before execution of line 19. Hence, not exists $\mathrm{cov}((\widehat{e},\widehat{g},\widehat{e'})) = e_{\mathrm{prec}}$. Thus, if $(e,g,e') \in S'_{\mathrm{TCNC}}$, we can safely map (e,g,e') to e_{prec}. Second, consider $\mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(e_{\mathrm{prec}},\mathsf{reached}) = \mathit{true}$. We know that $\mathsf{coveringSet} \subseteq N_{\mathrm{cov}}$ (due to line 27), and $\mathsf{stop}(e_{\mathrm{prec}},\mathsf{coveringSet})$. It follows that $\mathsf{coveringSet} \subseteq \{n'' \in N_{\mathrm{cov}} \mid (e,g,n'') \in G_{\mathrm{ARG}}\}$ and, thus, $\mathsf{coveringSet} \sqsubseteq \{n'' \in N_{\mathrm{cov}} \mid (e,g,n'') \in G_{\mathrm{ARG}}\}$. We conclude that $(e,g,e') \notin S'_{\mathrm{TCNC}}$. Since adding a state to N_{cov} does not violate the property, we know that the desired property is fulfilled.

After execution of iteration i the property would hold, if no edges from G_{ARG} and no elements from N_{cov} or reached are removed. It remains to be shown that neither removal of edges from the set G_{ARG} nor the removal of elements from N_{cov} or reached, violate the property. Algorithm 2 removes edges in line 13 and 20. In line 13, Algorithm 2 removes edges $(e'',\cdot,\cdot)$, $(e_{\mathrm{new}},\cdot,\cdot)$, and (e_p,g,e''). Since it removed e'' from reached ($e'' \neq e_{\mathrm{new}}$), removing edges $(e'',\cdot,\cdot)$ does not violate the property. Since $e'' \neq e_{\mathrm{new}}$, in line 11 it adds $(e_{\mathrm{new}},\cdot)$ to waitlist. Removing edges $(e_{\mathrm{new}},\cdot,\cdot)$ does not violate the property. Consider edges (e_p,g,e'') with $e_p \neq e''$ and $e_p \neq e_{\mathrm{new}}$ (the remaining cases). Let $(e_p,g,\widehat{e}) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ be arbitrary. If $\exists(e_p,\cdot) \in \mathsf{waitlist}$ or $e_p \notin \mathsf{reached}$ nothing must be shown. Assume $\neg\exists(e_p,\cdot) \in \mathsf{waitlist}$ and $e_p \in \mathsf{reached}$. Consider two cases. First, show that if $(e_p,g,\widehat{e}) \notin S'_{\mathrm{TCNC}}$, then this property remains. From $(e_p,g,\widehat{e}) \notin S'_{\mathrm{TCNC}}$, we conclude that $\exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : \widehat{e} \sqsubseteq$

$n' \wedge ((e_p, g, n') \in G_{\text{ARG}} \wedge n' \in N_{\text{cov}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S))$ or $e_p = e$ and $(e, g, \widehat{e})$ will be or is currently explored. We only need to consider the cases in which an edge is the reason why $(e_p, g, \widehat{e}) \notin S'_{\text{TCNC}}$. If $(e_p, g, n') \in G_{\text{ARG}} \wedge n' \in N_{\text{cov}}$, we only need to consider the case that $(e_p, g, n') = (e_p, g, e'')$ (removal of edge). An edge (e_p, g, e_{new}) is added to the set of ARG edges which is not removed ($e_{\text{new}} \neq e''$, $e'' \neq e_p \neq e_{\text{new}}$). Due to the definition of e_{new} in line 9, we get $e'' \sqsubseteq e_{\text{new}}$ (property of merge). Since partial order $\sqsubseteq$ is transitive, we get $\widehat{e} \sqsubseteq e_{\text{new}}$. Furthermore, e_{new} is added to N_{cov} in line 17 ($e'' = n' \in N_{\text{cov}}$). It still holds that $(e_p, g, \widehat{e}) \notin S'_{\text{TCNC}}$. The property is not violated. If $\exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S)$, we only need to consider the case that $e'' \in \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\}$ (removal of edge). Aan edge (e_p, g, e_{new}) is added to the set of ARG edges which is not removed ($e_{\text{new}} \neq e''$, $e'' \neq e_p \neq e_{\text{new}}$). Due to the definition of e_{new} in line 9, we get $e'' \sqsubseteq e_{\text{new}}$ (property of merge). Furthermore, in line 17 e_{new} is added to N_{cov}. We infer that $S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} \sqsubseteq (\{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} \setminus \{e''\}) \cup \{e_{\text{new}}\}$. Since (e_p, g, e'') is replaced by (e_p, g, e_{new}) and e_{new} is added to N_{cov} if e'' was contained in N_{cov}, it still holds that $(e_p, g, \widehat{e}) \notin S'_{\text{TCNC}}$. The property is not violated. Second, consider $(e_p, g, \widehat{e}) \in S'_{\text{TCNC}}$. If $(e_p, g, \widehat{e}) \notin S''_{\text{TCNC}}$ or $\text{cov}((e_p, g, \widehat{e})) \neq e''$, after execution of line 13 nothing must be shown. From $(e_p, g, \widehat{e}) \in S'_{\text{TCNC}}$ and $\text{cov}((e_p, g, \widehat{e})) = e''$, we conclude that $\widehat{e} \sqsubseteq e''$ and $(e_p, g, e'') \in G_{\text{ARG}}$. After execution of line 13, we have $(e_p, g, e_{\text{new}}) \in G_{\text{ARG}}$ ($e'' \neq e_p \neq e_{\text{new}}$) and $\widehat{e} \sqsubseteq e_{\text{new}}$ (transitivity of $\sqsubseteq$, definition of e_{new} in line 9, and property of merge). If there exists $\text{cov}(\cdot) = e_{\text{new}}$, we know that e_{new} was already contained in reached before line 12. Hence, in line 17 e_{new} is added to N_{cov} and after line 17 $(e_p, g, \widehat{e}) \notin S''_{\text{TCNC}}$. Nothing further must be shown. If not exists $\text{cov}(\cdot) = e_{\text{new}}$, we can safely map $(e_p, g, \widehat{e})$ to e_{new}. Since at most one mapping $\text{cov}(\cdot) = e''$ exists, after line 17 the property is reestablished.

In line 20, Algorithm 2 removes edges $(e_{\text{prec}}, \cdot, \cdot)$. Since in line 19, it adds $(e_{\text{prec}}, \pi_{\text{prec}})$ to waitlist, we know that not exists $(e_{\text{prec}}, \cdot, \cdot) \in S''_{\text{TCNC}}$, S''_{TCNC} denotes the set after removal of edges. It is safe to remove the edges $(e_{\text{prec}}, \cdot, \cdot)$. The property still holds.

Algorithm 2 only removes an element from N_{cov} in line 17. Since it removes an element e'' from N_{cov} only if it removes all edges $(\cdot, \cdot, e'')$, we already proved that the property still holds.

Algorithm 2 only removes an element from reached in line 12. Since it removes an element e'' from reached only if it removes all edges $(\cdot, \cdot, e'')$, we already proved that we no more elements are added to S'_{TCNC}. Assume $\text{cov}((n, g', n')) = e''$ exists. We know that $(n, g', e'') \in G_{\text{ARG}}$ and $n' \sqsubseteq e''$. Since (n, g, n') only remains in S'_{TCNC} if $e'' \neq n \neq e_{\text{new}}$, we need to consider only this case. An edge (n, g', e_{new}) is added to the set of ARG edges which is not removed ($e_{\text{new}} \neq e''$, $e'' \neq n \neq e_{\text{new}}$). Due to the definition of e_{new} in line 9, we get $e'' \sqsubseteq e_{\text{new}}$ (property of merge). Since partial order $\sqsubseteq$ is transitive, we get $n' \sqsubseteq e_{\text{new}}$. Since e_{new} is added to N_{cov} if e_{new} was contained in reached before execution of line 12, we infer that (n, g', n') is no longer an element from S'_{TCNC} or not exists $\text{cov}(\cdot) = e_{\text{new}}$. The property is not violated.

Thus, after iteration i and before iteration $i + 1$ the hypothesis holds.

At line 29, $N =$ reached and waitlist $= \emptyset$, we conclude that $R^P_{\mathbb{C}^{\mathcal{A}}}$ is well-covered.

Well-Constructedness Algorithm 2 starts with the empty set of edges which obviously fulfills the well-constructedness property. In line 13 it adds edges (e_p, g, e_{new}) for which we know that (e_p, g, e'') is already an ARG edge. Thus, if the ARG edges fulfill

the well-constructed properties before execution of line 13, there exists $(e_p, g, \cdot) \in \rightsquigarrow$. In this case, adding edges (e_p, g, e_{new}) does not violate the well-constructedness property. If an edge $(e, g, \cdot)$ is added in line 24 or line 28, we know that $g \in G_{\text{CFA}}$ (line 5) exists and $(e, g, e') \in \rightsquigarrow$ (loop body of for each in line 6). The set of ARG edges adheres to the well-constructedness property if it did before the insertion. Since the set of ARG edges adheres to the well-constructedness property in the beginning, this property is never violated by an insertion if it was fulfilled before, deletion of ARG edges does not violate well-constructedness, and in line 29 $G_{\text{ARG}} = G'_{\text{ARG}}$, we can conclude that $R^P_{\mathbb{C}^\mathcal{A}}$ is well-constructed.

□

Lemma 2.7. *Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an abstract reachability graph for program P and enhancement $\mathbb{C}^\mathcal{A}$ of CPA $\mathbb{C}$ with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$ s.t. $R^P_{\mathbb{C}^\mathcal{A}}$ is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$. Then, for all paths $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P([\![root]\!])$ there exists a configuration sequence $(c_0, q_0), \ldots, (c_n, q_n)$ for p and $\mathcal{A}$ with $\forall 0 \leq i \leq n : \exists (e, q) \in N : c_i \in [\![e]\!] \wedge q_i \sqsubseteq q$.*

Proof. We show by induction over the path length that for every path $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P([\![root]\!])$, a configuration sequence $(c_0, q_0), \ldots, (c_n, q_n)$ for p and $\mathcal{A}$ with $\forall 0 \leq i \leq n : \exists (e, q) \in N : c_i \in [\![e]\!] \wedge q_i \sqsubseteq q$ exists,

Basis Let $c_0 \in paths_P([\![root]\!])$ be a path of length 0. By definition of paths $c_0 \in [\![root]\!]$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is an well-formed ARG, $root = (e_r, q_r) \in N$ and it is rooted, i.e., $e_0 \sqsubseteq root = (e_r, q_r)$. From $e_0 = (e, q_0)$ and definition of $\sqsubseteq$, we conclude that $q_0 \sqsubseteq q_r$. By definition (c_0, q_0) is a configuration sequence for path c_0 and $\mathcal{A}$. The induction hypothesis holds.

Step Let $p := c_0 \xrightarrow{g_1} c_1 \ldots c_{i-1} \xrightarrow{g_i} c_i \in paths_P([\![root]\!])$ be a path of length i. By definition of paths, $p' := c_0 \xrightarrow{g_1} c_1 \ldots c_{i-1} \in paths_P([\![root]\!])$ is a path of length $i-1$. From induction, we know that a configuration sequence $(c_0, q_0) \ldots, (c_{i-1}, q_{i-1})$ for p' and $\mathcal{A}$ exists s.t. $\forall 0 \leq j \leq i-1 : \exists (e_j, q'_j) \in N : c_j \in [\![(e_j, q'_j)]\!] \wedge q_j \sqsubseteq q'_j$. Let $g_i = (l_i, op_i, l'_i)$. Since δ is complete, there exists at least one transition $(q_{i-1}, op_i, C^i_{\text{sub}}, q_i) \in \delta$ with $c_i \in C^i_{\text{sub}}$. By definition $(c_0, q_0) \ldots, (c_{i-1}, q_{i-1})(c_i, q_i)$ is a configuration sequence for p and $\mathcal{A}$. We need to show that $(e_i, q'_i) \in N$ exists with $c_i \in [\![(e_i, q'_i)]\!]$ and $q_i \sqsubseteq q'_i$. From $c_{i-1} \in [\![(e_{i-1}, q'_{i-1})]\!]$ and $c_{i-1} \xrightarrow{g_i} c_i$, we can conclude that an abstract transition $((e_{i-1}, q'_{i-1}), g_i, (e_s, q_s)) \in \rightsquigarrow$ exists with $c_i \in [\![(e_s, q_s)]\!]$ (overapproximation of transfer relation Eq. 2.2). From definition of $\rightsquigarrow$, especially transfer relation of most precise enhancement and enhancement, $q'_{i-1} = q_{i-1} \in Q \vee q'_{i-1} = q_\top$ (follows from configuration sequence for p', $q_{i-1} \sqsubseteq q'_{i-1}$), $(q_{i-1}, op_i, C^i_{\text{sub}}, q_i) \in \delta$ with $c_i \in C^i_{\text{sub}}$, and due to determinism of property automaton not exists $(q_{i-1}, op_i, C^i_{\text{sub}}, \widehat{q}) \in \delta$ with $c_i \in C^i_{\text{sub}}$ and $q_i \neq \widehat{q}$, we can conclude that $e_s = (\cdot, q_s)$ with either $q_s = q_i$ or $q_s = q_\top$. Hence, $q_i \sqsubseteq q_s$. From completeness of $R^P_{\mathbb{C}^\mathcal{A}}$ (part of well-formedness), we know there exist $((e_{i-1}, q'_{i-1}), g, (e_i, q'_i)) \in G_{\text{ARG}}, (e_i, q'_i) \in N$, and $(e_s, q_s) \sqsubseteq (e_i, q'_i)$ or there exists $(e*, q*) \in E_{\mathbb{C}^\mathcal{A}}$ with $(e_s, q_s) \sqsubseteq (e*, q*)$ and $S \sqsubseteq \{n'' \in N_{\text{cov}} \mid ((e_{i-1}, q'_{i-1}), g, n'') \in G_{\text{ARG}}\} \subseteq N$ with $\mathsf{stop}_{\mathbb{C}^\mathcal{A}}((e*, q*), S)$. In the first case, we conclude that $c_i \in [\![(e_i, q'_i)]\!]$ (meaning of $\sqsubseteq$) and $q_i \sqsubseteq q'_i$ (transitivity of partial order $\sqsubseteq$). The induction hypothesis follows for the first case. In the second case, we get from the definition of the enhancement of the termination check that $\mathsf{stop}_{\mathbb{C}}(e*, S') = true$ with $S' = \{e \mid (e, q) \in S \wedge q* \sqsubseteq q\}$. By meaning of $[\![\cdot]\!]$, soundness of termination

check, meaning of $\sqsubseteq$, $(e_s, q_s) \sqsubseteq (e*, q*)$ and $c_i \in [\![(e_s, q_s)]\!]$, a state $(e'_s, q'_s) \in S$ exists with $c_i \in [\![(e'_s, q'_s)]\!]$ and $q_s \sqsubseteq q* \sqsubseteq q'_s$. Since $S \sqsubseteq \{n'' \in N_{\text{cov}} \mid ((e_{i-1}, q'_{i-1}), g, n'') \in G_{\text{ARG}}\} \subseteq N$, it follows from definition that $(e_i, q'_i) \in N$ exists with $(e'_s, q'_s) \sqsubseteq (e_i, q'_i)$. By meaning of $\sqsubseteq$, it follows that $c_i \in [\![(e_i, q'_i)]\!]$. By transitivity of partial order $\sqsubseteq$, $q_i \sqsubseteq q_s$, $q_s \sqsubseteq q* \sqsubseteq q'_s$, $q'_s \sqsubseteq q'_i$ (definition of $\sqsubseteq$ and $(e'_s, q'_s) \sqsubseteq (e_i, q'_i)$), we get $q_i \sqsubseteq q'_i$. The induction hypothesis follows.

□

Proposition 2.9. *Let CPA $\mathbb{C}^{\mathcal{A}}$ be an enhancement of CPA $\mathbb{C}$ with property automaton $\mathcal{A}$ s.t. $\forall e \in E_{\mathbb{C}^{\mathcal{A}}}, S \subseteq E_{\mathbb{C}^{\mathcal{A}}} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(e, S) \implies \exists e' \in S : e \sqsubseteq_{\mathbb{C}^{\mathcal{A}}} e'$ and $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ is a function. If Algorithm 2 started with CPA $\mathbb{C}^{\mathcal{A}}$, program P, initial abstract state $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$, and arbitrary precision $\pi_0 \in \Pi_{\mathbb{C}^{\mathcal{A}}}$ returns $(\cdot, \cdot, R^P_{\mathbb{C}^{\mathcal{A}}})$, then $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$ which is sound and deterministic.*

Proof. Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$. From Lemma 2.5, we know that $R^P_{\mathbb{C}^{\mathcal{A}}}$ is an ARG for P and $\mathbb{C}^{\mathcal{A}}$.

Soundness Algorithm 2 starts with the empty set of ARG edges which is obviously sound. In line 13, it adds edges (e_p, g, e_{new}) for which we know that (e_p, g, e'') is already an ARG edge and if G_{ARG} is sound before execution of line 13, we also know that for all $(e_p, g, e') \in \rightsquigarrow$ it holds that $e' \sqsubseteq e''$. Since $e'' \sqsubseteq e_{\text{new}}$ (definition of e_{new} and Eq. 2.5), and partial order $\sqsubseteq$ is transitive, we get $e' \sqsubseteq e_{\text{new}}$. Adding an edge in line 13 does not violate the soundness property. If Algorithm 2 adds an edge (e, g, e_{prec}) in line 24, we know there exists $(e, g, e') \in \rightsquigarrow$ (loop body of for in line 6) and $e' \sqsubseteq e_{\text{prec}}$ (definition of e_{prec} in line 7 and property of precision adjustment). Since $\rightsquigarrow$ is a function, the set of ARG edges is sound after execution of line 24 if it was sound before its execution. If Algorithm 2 adds an edge (e, g, e_r) in line 28, we know that $\mathsf{coveringSet} = \{e_r\}$ and $e_{\text{prec}} \sqsubseteq e_r$ (definition of $\mathsf{coveringSet}$, sets with less elements are smaller, and a single element must exist which is at least as abstract as e_{prec} (definition of termination check)). Since we know there exists $(e, g, e') \in \rightsquigarrow$ (loop body of for in line 6), $e' \sqsubseteq e_{\text{prec}}$ (definition of e_{prec} in line 7 and property of precision adjustment), and partial order $\sqsubseteq$ is transitive, we get $e' \sqsubseteq e_r$. Since $\rightsquigarrow$ is a function, the set of ARG edges is sound after execution of line 28 if it was sound before its execution. Since the set G_{ARG} of ARG edges is sound at the beginning, removing edges never violates soundness, adding edges cannot violate the property, and in line 29 $G_{\text{ARG}} = G'_{\text{ARG}}$, the ARG $R^P_{\mathbb{C}^{\mathcal{A}}}$ is sound.

Determinism Algorithm 2 starts with the empty set of ARG edges which is definitely deterministic. In line 13, it adds edge (e_p, g, e_{new}) only if an edge (e_p, g, e'') exists and it deletes (e_p, g, e''). Thus, if the set of edges was deterministic before execution of line 13 it is deterministic after the execution of line 13. In lines 24 and 28, Algorithm 2 adds only a single edge ($\mathsf{coveringSet} = \{e_r\}$, same reason as for soundness). Since the transfer relation $\rightsquigarrow$ is a function, at most one edge $(e, g, \cdot)$ is added in either line 24 or 28 during each execution of for in line 5. Hence, the set of ARG edges remains deterministic after each execution of for in line 5, if no ARG edges $(e, \cdot, \cdot)$ exists before execution of that for loop. We need to show that if $(e, \cdot)$ is removed from $\mathsf{waitlist}$ in line 4, no such edges exist. We show before the execution of line 4 for each element $(e, \cdot) \in \mathsf{waitlist}$ no edge $(e, \cdot, \cdot)$ exists in the set of ARG edges. Before the first execution, we know that $G_{\text{ARG}} = \emptyset$. If $(e, \cdot)$ is added to $\mathsf{waitlist}$ in line 11 or line 19, all edges $(e, \cdot, \cdot)$ are removed from G_{ARG}. If before

execution of line 11 or 19 for each element $(e, \cdot) \in \mathsf{waitlist}$ no edge $(e, \cdot, \cdot)$ exists in the set of ARG edges, then after the execution of line 13 and 20, respectively, for each element $(e, \cdot) \in \mathsf{waitlist}$ no edge $(e, \cdot, \cdot)$ exists in the set of ARG edges. We know that $\forall (e_w, \cdot) \in \mathsf{waitlist} : e_w \in \mathsf{reached}$ and before execution of line 19 it yields that $\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathit{false}$. Since $G_{\text{ARG}} \subseteq \mathsf{reached} \times G_{\text{CFA}} \times \mathsf{reached}$, at line 19 no edge $(e_{\text{prec}}, \cdot, \cdot)$ exists. In lines 24 and 28, edges $(e, \cdot, \cdot)$ are added only if $\neg \exists (e, \cdot) \in \mathsf{waitlist}$. This does not violate the property. We conclude that if before execution i of line 4 for each element $(e, \cdot) \in \mathsf{waitlist}$ no edge $(e, \cdot, \cdot)$ exists in the set of ARG edges, then before execution $i+1$ of line 4 for each element $(e, \cdot) \in \mathsf{waitlist}$ no edge $(e, \cdot, \cdot)$ exists in the set of ARG edges. Hence, before execution of line 4 for each element $(e, \cdot) \in \mathsf{waitlist}$ no edge $(e, \cdot, \cdot)$ exists in the set of ARG edges. Since the set of ARG edges is deterministic in the beginning, this property is never violated, and in line 29 $G_{\text{ARG}} = G'_{\text{ARG}}$, the ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is deterministic.

□

A.2 Outstanding Proofs for Chapter 3 Configurable Program Certification

Lemma 3.6 (Configuration Sequence Coverage)**.** *If Algorithm 3 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ returns true, then certificate $\mathcal{C}_{\mathbb{C}^\mathcal{A}}$ covers at least one configuration sequence per path.*

Proof. We show by induction over the path length that for each path $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in \mathit{paths}_P(\llbracket e_0 \rrbracket)$ a configuration sequence $(c_0, q_0), \ldots, (c_n, q_n)$ for p and $\mathcal{A}$ exists with $\forall 0 \leq i \leq n : \exists (e, q) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}} : c_i \in \llbracket e \rrbracket \wedge q_i \sqsubseteq q$.

Basis Let $c_0 \in \mathit{paths}_P(\llbracket e_0 \rrbracket)$ be a path of length 0. By definition of paths, $c_0 \in \llbracket e_0 \rrbracket$. Since Algorithm 3 returns true, $\mathsf{cover}(e_0, \mathcal{C}_{\mathbb{C}^\mathcal{A}}) = \mathit{true}$. By definition of coverage check cover, we know that there exists $(e, q) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}}$ with $c_0 \in \llbracket (e, q) \rrbracket$ and $q_0 \sqsubseteq q$. By definition, (c_0, q_0) is a configuration sequence for path c_0 and $\mathcal{A}$. The induction hypothesis holds.

Step Let $p := c_0 \xrightarrow{g_1} c_1 \ldots c_{i-1} \xrightarrow{g_i} c_i \in \mathit{paths}_P(\llbracket e_0 \rrbracket)$ be a path of length i. By definition of paths, $p' := c_0 \xrightarrow{g_1} c_1 \ldots c_{i-1} \in \mathit{paths}_P(\llbracket e_0 \rrbracket)$ is a path of length $i-1$. From induction, we know that a configuration sequence $(c_0, q_0) \ldots, (c_{i-1}, q_{i-1})$ for p' and $\mathcal{A}$ exists s.t. $\forall 0 \leq j \leq i-1 : \exists (e_j, q'_j) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}} : c_j \in \llbracket (e_j, q'_j) \rrbracket \wedge q_j \sqsubseteq q'_j$. Let $g_i = (l_i, op_i, l'_i)$. From $c_{i-1} \in \llbracket (e_{i-1}, q'_{i-1}) \rrbracket$ and $c_{i-1} \xrightarrow{g_i} c_i$, we conclude that an $\text{entry}((e_{i-1}, q'_{i-1}), g_i, (e_s, q_s)) \in \rightsquigarrow$ exists with $c_i \in \llbracket (e_s, q_s) \rrbracket$ and $q_s = q_\top$ or $q_s \in Q \wedge (q'_{i-1}, op_i, C^i_{\text{sub}}, q_s) \in \delta \wedge c_i \in C^i_{\text{sub}}$ (requirement on transfer relation of CCV). If $q_s = q_\top$, set q_i to any $q \in Q$ with $(q_{i-1}, op, C^i_{\text{sub}}, q_s) \in \delta \wedge c_i \in C^i_{\text{sub}}$. Due to completion of δ, at least one exists. If $q_s \in Q$, set $q_i = q_s$. We know from $(q'_{i-1}, op_i, C^i_{\text{sub}}, q_s) \in \delta$, $q_{i-1} \sqsubseteq q'_{i-1}$ and $\mathcal{Q}$ being a flat lattice that $q_{i-1} = q'_{i-1}$. For both cases, we conclude that $(c_0, q_0) \ldots, (c_{i-1}, q_{i-1})(c_i, q_i)$ is a configuration sequence for p and $\mathcal{A}$ and $q_i \sqsubseteq q_s$. From $(e_{i-1}, q'_{i-1}) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}}, g_i \in G_{\text{CFA}}$ (definition of paths) and Algorithm 3 returning true, we know that $\mathsf{cover}((e_s, q_s), \mathcal{C}_{\mathbb{C}^\mathcal{A}})$ is checked in line 6 and $\mathsf{cover}((e_s, q_s), \mathcal{C}_{\mathbb{C}^\mathcal{A}}) = \mathit{true}$. From the requirements on a coverage check

and $c_i \in [\![(e_s, q_s)]\!]$, it follows that there exists $(e_i, q_i') \in \mathcal{C}_{\mathbb{C}^\mathcal{A}}$ with $c_i \in [\![(e_i, q_i')]\!]$ and $q_s \sqsubseteq q_i'$. From transitivity of $\sqsubseteq$, we get $q_i \sqsubseteq q_i'$. The induction hypothesis follows.

□

Theorem 3.10 (Relative Completeness)**.** *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^\mathcal{A}}$ be an ARG for finite program P and enhancement $\mathbb{C}^\mathcal{A}$ of CPA $\mathbb{C}$, and $R^P_{\mathbb{C}^\mathcal{A}}$ be well-formed for $e \in E_{\mathbb{C}^\mathcal{A}}$. Algorithm 3 started with CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and certificate $\mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}})$ returns true.*

Proof. Let ARG $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ and program $P = (L, G_{\mathrm{CFA}}, l_0)$. By definition $\mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}}) = N$ and N is finite. Since certificate and program are finite, we infer that Algorithm 3 terminates (Lemma 3.9). Prove by contradiction that Algorithm 3 returns true. Assume Algorithm 3 returns false. Algorithm 3 may return false in line 2, 7, or 8.

Assume Algorithm 3 returns false in line 2. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is well-formed, $e \sqsubseteq root \in N$. By transitivity of partial order $\sqsubseteq$, we get $e_0 \sqsubseteq root$. If Algorithm 3 returns false in line 2, $\mathsf{cover}(e_0, \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}})) = \mathsf{cover}(e_0, N) = false$. Since cover is well-behaving (consistent with partial order) and $e_0 \sqsubseteq root \in N$, $\mathsf{cover}(e_0, N) = true$. Contradiction to assumption, Algorithm 3 does not return false in line 2.

Next, assume Algorithm 3 returns false in line 7. If Algorithm 3 returns false in line 7, an abstract state $e \in \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}}) = N$ exists s.t. $(e, g, e') \in \rightsquigarrow, g \in G_{\mathrm{CFA}}$ and $\mathsf{cover}(e', \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}})) = \mathsf{cover}(e', N) = false$. From $R^P_{\mathbb{C}^\mathcal{A}}$ being well-formed, we know that $e'' \in N = \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}})$ exists with $e' \sqsubseteq e''$, or $e''' \in E_{\mathbb{C}^\mathcal{A}}$ with $e' \sqsubseteq e'''$ and $S \sqsubseteq \{n'' \in N_{\mathrm{cov}} \mid (e, g, n'') \in G_{\mathrm{ARG}}\}$ with $\mathsf{stop}(e''', S) = true$ exist. In the first case, we know that $\mathsf{cover}(e', N) = \mathsf{cover}(e', \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}})) = true$ (cover consistent with partial order). In the second case, we know that $\mathsf{cover}(e''', S) = true$ ($\mathsf{stop}(e''', S) \implies \mathsf{cover}(e''', S)$, definition of CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$). Since $e' \sqsubseteq e'''$, $S \sqsubseteq \{n'' \in N_{\mathrm{cov}} \mid (e, g, n'') \in G_{\mathrm{ARG}}\} \sqsubseteq N$ (definition of $\sqsubseteq$ and $\{n'' \in N_{\mathrm{cov}} \mid (e, g, n'') \in G_{\mathrm{ARG}}\} \subseteq N_{\mathrm{cov}} \subseteq N$), and cover is monotonic (well-behaving), we conclude that $\mathsf{cover}(e', N) = \mathsf{cover}(e', \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}})) = true$. Contradiction to assumption in both cases, Algorithm 3 does not return false in line 7.

Finally, assume Algorithm 3 returns false in line 8. If Algorithm 3 returns false in line 8, then an abstract state $(e, q) \in \mathrm{cert}(R^P_{\mathbb{C}^\mathcal{A}}) = N$ exists with $q = q_\top \vee q = q_{\mathrm{err}}$. We know that no such state exists in N ($R^P_{\mathbb{C}^\mathcal{A}}$ well-formed, thus safe). Contradiction to assumption, Algorithm 3 does not return false in line 8. Hence, Algorithm 3 returns true. □

A.3 Outstanding Proofs for Chapter 4 Optimization of CPC

A.3.1 Outstanding Proofs for Reduced Certificates

Lemma 4.3. *If Algorithm 4 started with CCV $\mathbb{V}^{D_{\mathbb{C}^\mathcal{A}}}$ for abstract domain $D_{\mathbb{C}^\mathcal{A}}$ enhanced with property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\mathrm{err}})$, program P, initial abstract state $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$, and reduced certificate $\mathcal{RC}_{\mathbb{C}^\mathcal{A}}$ returns true, then the reached set at the state of termination of Algorithm 4 covers at least one configuration sequence per path.*

Proof. Let $\mathsf{reached}'$ denote the reached set at the state of termination of Algorithm 4. We show by induction over the length of paths $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P([\![e_0]\!])$ that for each path p a configuration sequence $(c_0, q_0), \ldots, (c_n, q_n)$ for p and $\mathcal{A}$ exists with $\forall 0 \leq i \leq n : \exists (e, q) \in \mathsf{reached}' : c_i \in [\![e]\!] \wedge q_i \sqsubseteq q$.

Basis Let $c_0 \in paths_P([\![e_0]\!])$ be a path of length 0. By definition of paths $c_0 \in [\![e_0]\!]$. Since Algorithm 4 returns true, $\mathsf{cover}(e_0, \mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}) = \mathit{true}$. By definition of coverage check cover, we know that $(e, q) \in \mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ exists with $c_0 \in [\![(e, q)]\!]$ and $q_0 \sqsubseteq q$. Since Algorithm 4 only adds states to reached and it added $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ to reached in line 3, we get $(e, q) \in \mathsf{reached}'$. By definition (c_0, q_0) is a configuration sequence for path c_0 and $\mathcal{A}$. The induction hypothesis holds.

Step Let $p := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \xrightarrow{g_i} c_i \in paths_P([\![e_0]\!])$ be a path of length i. By definition, $p' := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \in paths_P([\![e_0]\!])$ is a path of length $i-1$. From induction, we know that a configuration sequence $(c_0, q_0) \dots, (c_{i-1}, q_{i-1})$ for p' and $\mathcal{A}$ exists s.t. $\forall 0 \leq j \leq i-1 : \exists (e_j, q'_j) \in \mathsf{reached}' : c_j \in [\![(e_j, q'_j)]\!] \wedge q_j \sqsubseteq q'_j$. Let $g_i = (l_i, op_i, l'_i)$. From $c_{i-1} \in [\![(e_{i-1}, q'_{i-1})]\!]$ and $c_{i-1} \xrightarrow{g_i} c_i$, we can conclude that there exists an abstract transition $((e_{i-1}, q'_{i-1}), g_i, (e_s, q_s)) \in \rightsquigarrow$ with $c_i \in [\![(e_s, q_s)]\!]$ and $q_s = q_\top$ or $q_s \in Q \wedge (q'_{i-1}, op_i, C^i_{\text{sub}}, q_s) \in \delta \wedge c_i \in C^i_{\text{sub}}$ (requirement on transfer relation of CCV). If $q_s = q_\top$, set q_i to any $q \in Q$ with $(q_{i-1}, op, C^i_{\text{sub}}, q_s) \in \delta \wedge c_i \in C^i_{\text{sub}}$. Due to completion of δ, at least one exists. If $q_s \in Q$, set $q_i = q_s$. We know from $(q'_{i-1}, op_i, C^i_{\text{sub}}, q_s) \in \delta$, $q_{i-1} \sqsubseteq q'_{i-1}$, and $\mathcal{Q}$ being a flat lattice that $q_{i-1} = q'_{i-1}$. For both cases, we conclude that $(c_0, q_0) \dots, (c_{i-1}, q_{i-1})(c_i, q_i)$ is a configuration sequence for p and $\mathcal{A}$ and $q_i \sqsubseteq q_s$. From Algorithm 4 returning true, we know that $\mathsf{waitlist} = \emptyset$. Since every state in reached was also added to waitlist and $\mathsf{waitlist} = \emptyset$, every state in reached was popped at least once from waitlist in line 5. From $(e_{i-1}, q'_{i-1}) \in \mathsf{reached}'$, $g_i \in G_{\text{CFA}}$ (definition of paths), we know that $((e_{i-1}, q'_{i-1}), g_i, (e_s, q_s)) \in \rightsquigarrow$ was explored in line 7, and either $\mathsf{cover}((e_s, q_s), \mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}) = \mathit{true}$ or (e_s, q_s) was added to reached. In the first case, from the requirements on a coverage check and $c_i \in [\![(e_s, q_s)]\!]$, it follows that an abstract state $(e_i, q'_i) \in \mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ exists with $c_i \in [\![(e_i, q'_i)]\!]$ and $q_s \sqsubseteq q'_i$. From transitivity of $\sqsubseteq$, we get $q_i \sqsubseteq q'_i$. Since Algorithm 4 only adds states to reached and it added $\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}$ to reached in line 3, we get $(e_i, q'_i) \in \mathsf{reached}'$. The induction hypothesis follows for the first case. In the second case, (e_s, q_s) is added to reached and thus $(e_i, q'_i) = (e_s, q_s) \in \mathsf{reached}'$. The induction hypothesis follows.

□

Lemma 4.6 (Termination)**.** *Let $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^{\mathcal{A}}$ and coverage check* cover*, and let program $P = (L, G_{\text{CFA}}, l_0)$ be finite. Then, Algorithm 4 started with $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, P, initial abstract state $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$, and finite reduced certificate $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\text{sub}}_{\mathbb{C}^{\mathcal{A}}}, n)$ terminates.*

Proof. Algorithm 4 terminates if the number of program edges is finite, for each pair $(e, g) \in E_{\mathbb{C}^{\mathcal{A}}} \times G_{\text{CFA}}$ only finitely many elements $(e, g, e') \in \rightsquigarrow$ exists, and the while loop terminates. The number of program edges are finite (finite program). Since the transfer relation is a transfer relation of a CPA (definition of $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$), it follows from Eq. 2.3 that $\forall (e, g) \in \mathcal{C}_{\mathbb{C}^{\mathcal{A}}} \times G_{\text{CFA}} : \exists n \in \mathbb{N} : |\{(e, g, e') \in \rightsquigarrow\}| \leq n$. Algorithm 4 only adds states to waitlist which are also added to reached. Due to the condition of the while loop and Algorithm 4 never removing states from reached, Algorithm 4 can only add finitely many different states to reached and thus to waitlist or terminates. Since Algorithm 4 pops an element from waitlist in each while loop iteration and Algorithm 4 terminates if waitlist becomes empty, Algorithm 4 terminates if an abstract state is added at most once to waitlist. Since it never removes a state from reached and $\mathsf{waitlist} \subseteq \mathsf{reached}$, it only adds

states to waitlist which have not been added to reached and thus to waitlist. Algorithm 4 terminates. □

Lemma 4.7. *Let $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^{\mathcal{A}}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ be an ARG for finite program P and enhancement $\mathbb{C}^{\mathcal{A}}$ of CPA $\mathbb{C}$, and $R^P_{\mathbb{C}^{\mathcal{A}}}$ be well-formed for $e \in E_{\mathbb{C}^{\mathcal{A}}}$. If Algorithm 4 starts with CCV $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and reduced certificate $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}, |N|)$ s.t. $N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq N$, then at line 10* reached $\subseteq N$.

Proof. Let $P = (L, G_{\mathrm{CFA}}, l_0)$. Prove by induction over the changes of reached that reached $\subseteq N$.

Basis The set reached is initialized in line 3 with $\mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq N$. The induction hypothesis follows.

Step After the initialization states e' are only added to reached in line 9. Since Algorithm 4 only adds states to waitlist which are also added to reached and it never removes states from reached, we know that in line 9 it holds that $\exists e \in \mathsf{reached} \subseteq N$, $g \in G_{\mathrm{CFA}}$, $(e, g, e') \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ (definition of $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$) and $\neg\mathsf{cover}(e', \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}})$. From $R^P_{\mathbb{C}^{\mathcal{A}}}$ being complete (well-formed), we deduce that $\exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : e' \sqsubseteq n' \wedge ((e, g, n') \in G_{\mathrm{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\mathrm{cov}} \mid (e, g, n'') \in G_{\mathrm{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S))$. In the first case, we know that if $e' \neq n'$, then $n' \in N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}$. Since cover is consistent with partial order (well-behaving), we know that if $e' \neq n'$, then $\mathsf{cover}(e', \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}) = true$. We know that in line 9 $e' = n'$. From $(e, g, e') \in G_{\mathrm{ARG}}$, we conclude that $e' \in N$. In the second case, the fact that cover is monotonic (well-behaving), $\mathsf{stop}(e, S) \implies \mathsf{cover}(e, S)$ (definition of $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$), and $\{n'' \in N_{\mathrm{cov}} \mid (e, g, n'') \in G_{\mathrm{ARG}}\} \sqsubseteq N_{\mathrm{cov}} \sqsubseteq N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \sqsubseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}$ (definition of $N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ and $\sqsubseteq$), gives us $\mathsf{cover}(e', \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}) = true$. We conclude that in line 9 $e' \in N$. The induction hypothesis follows.

□

Lemma 4.8. *Let $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^{\mathcal{A}}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ be an ARG for finite program P and enhancement $\mathbb{C}^{\mathcal{A}}$ of CPA $\mathbb{C}$ and $R^P_{\mathbb{C}^{\mathcal{A}}}$ be well-formed for $e \in E_{\mathbb{C}^{\mathcal{A}}}$. Algorithm 4 started with CCV $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and reduced certificate $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}} = (\mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}, |N|)$ s.t. $N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq N$ returns true.*

Proof. From the definition of an ARG, we know that $\mathcal{RC}_{\mathbb{C}^{\mathcal{A}}}$ is finite ($\mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}} \subseteq N$ and N finite). From Lemma 4.6, we deduce that Algorithm 4 terminates. Prove by contradiction that Algorithm 4 returns true. Assume that Algorithm 4 returns false. Algorithm 4 may return false in line 2 or 10.

Assume that Algorithm 4 returns false in line 2. Since $R^P_{\mathbb{C}^{\mathcal{A}}}$ is well-formed $e \sqsubseteq root \in N$. Due to transitivity of partial order $\sqsubseteq$, we get $e_0 \sqsubseteq root$. From definition of $N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ and $N_{\mathrm{R}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}$, we know that $root \in \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}$. If Algorithm 4 returns false in line 2, $\mathsf{cover}(e_0, \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}) = false$. Since cover is well-behaving (consistent with partial order) and $e_0 \sqsubseteq root \in \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}$, $\mathsf{cover}(e_0, \mathcal{C}^{\mathrm{sub}}_{\mathbb{C}^{\mathcal{A}}}) = true$. Contradiction to assumption, Algorithm 4 does not return false in line 2.

Assume that Algorithm 4 returns false in line 10. If Algorithm 4 returns false in line 10, $|\mathsf{reached}| > |N|$ or $\exists(\cdot, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\mathrm{err}}$. The previous lemma gives

us that at line 10 $\mathsf{reached} \subseteq N$. We infer that $|\mathsf{reached}| \leq |N|$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is safe (well-formed), $\neg\exists(\cdot, q) \in N : q = q_\top \vee q = q_{\text{err}}$. Hence, $\neg\exists(\cdot, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$. Contradiction to assumption, Algorithm 4 does not return false in line 10.

Thus, Algorithm 4 returns true. □

Lemma 4.9. *Let* $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover}) = (D_{\mathbb{C}^\mathcal{A}}, \rightsquigarrow, \mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^\mathcal{A}$ *and coverage check* cover *which is well-behaving,* $\rightsquigarrow$ *be monotonic,* $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for finite program* P *and enhancement* $\mathbb{C}^\mathcal{A}$ *of CPA* $\mathbb{C}$*, and* $R^P_{\mathbb{C}^\mathcal{A}}$ *be well-formed for* $e \in E_{\mathbb{C}^\mathcal{A}}$*. If Algorithm 4 starts with CCV* $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$*, program* P*, initial abstract state* $e_0 \sqsubseteq e$*, and certificate* $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$*, then at line 10* $\mathsf{reached} \sqsubseteq N$ *and* $|\mathsf{reached}| \leq |N|$.

Proof. Let $P = (L, G_{\text{CFA}}, l_0)$. Let $S_{\text{TCNC}} := \{(n, g, e) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}} |\ n \in N, g \in G_{\text{CFA}} \wedge \neg\exists n' \in E_{\mathbb{C}^\mathcal{A}} : e \sqsubseteq n' \wedge ((n, g, n') \in G_{\text{ARG}} \wedge n' \in N_{\text{cov}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S)\}$ be the set of abstract successor computations of abstract successors which are covered by non-covering nodes. Let $\text{cov} : S_{\text{TCNC}} \to N \setminus N_{\text{cov}}$ be a total, injective function with $\forall(n, g, e) \in S_{\text{TCNC}} : e \sqsubseteq \text{cov}(e) \wedge (n, g, \text{cov}(e)) \in G_{\text{ARG}}$. (such a function exists because $R^P_{\mathbb{C}^\mathcal{A}}$ is well-covered (well-formed)). For any $e, e' \in E_{\mathbb{C}^\mathcal{A}}$ s.t. $e \sqsubseteq e'$ and any $g \in \mathcal{G}$ let $f_{e,e',g} : \{(e, g, \cdot) \in \rightsquigarrow\} \to \{(e', g, \cdot) \in \rightsquigarrow\}$ be a total, injective function with $\forall(e, g, e_s) \in \rightsquigarrow : f((e, g, e_s)) = (e', g, e'_s) \implies e_s \sqsubseteq e'_s$ (such functions exists because $\rightsquigarrow$ is monotonic). Prove by induction over the changes of $\mathsf{reached}$ that a total, injective function $h : \mathsf{reached} \to N$ exists with $\forall e \in \mathsf{reached} : e \sqsubseteq h(e)$, $\forall e \in \mathsf{reached} : (e \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies h(e) = e) \wedge (h(e) \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies h(e) = e)$, $\forall e' \in \mathsf{reached} : h(e') \notin N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies \exists e \in \mathsf{reached}, g \in G_{\text{CFA}} : ((e, g, e') \in \rightsquigarrow \wedge f_{e,h(e),g}((e, g, e')) \in S_{\text{TCNC}} \wedge \text{cov}(f_{e,h(e),g}((e, g, e'))) = h(e'))$.

Basis The set $\mathsf{reached}$ is initialized in line 3 with $N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}})$ (definition of highly reduced certificate). Show that the identity function $id : \mathsf{reached} \to \mathsf{reached}, id(e) = e$, is a function with the desired properties. From the definition of $N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}})$, we know that $N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N$. With $\mathsf{reached} = N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}})$, we get $id : \mathsf{reached} \to N$. The identity function is definitely total and injective, and additionally fulfills that $\forall e \in \mathsf{reached} : (e \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies h(e) = e) \wedge (h(e) \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies h(e) = e)$. Since partial order $\sqsubseteq$ is reflexive, we have $\forall e \in \mathsf{reached} : e \sqsubseteq id(e)$. Since the range of id is $N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}})$, the induction hypothesis follows.

Step After the initialization, states e' are only added to $\mathsf{reached}$ in line 9. From the induction hypothesis, we conclude that before execution of line 9 a total, injective function $h : \mathsf{reached} \to N$ exists with $\forall e \in \mathsf{reached} : e \sqsubseteq h(e)$, $\forall e \in \mathsf{reached} : (e \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies h(e) = e) \wedge (h(e) \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies h(e) = e)$, $\forall e' \in \mathsf{reached} : h(e') \notin N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) \implies \exists e \in \mathsf{reached}, g \in G_{\text{CFA}} : ((e, g, e') \in \rightsquigarrow \wedge f_{e,h(e),g}((e, g, e')) \in S_{\text{TCNC}} \wedge \text{cov}(f_{e,h(e),g}((e, g, e'))) = h(e'))$. Since Algorithm 4 only adds states to $\mathsf{waitlist}$ which are also added to $\mathsf{reached}$ and it never removes states from $\mathsf{reached}$, we know that in line 9 it holds that $\exists e \in \mathsf{reached}$, $g \in G_{\text{CFA}}$, $(e, g, e') \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ (definition of $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$) and $\neg\mathsf{cover}(e', N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}))$ and $e' \notin \mathsf{reached}$. Since $e \in \mathsf{reached}$, $e \sqsubseteq h(e)$. With $g \in G_{\text{CFA}} \subseteq \mathcal{G}$, we know that we fixed a function $f_{e,h(e),g}$. From the definition of that function we get $f_{e,h(e),g}((e, g, e')) = (h(e), g, e'_s)$ with $(h(e), g, e'_s) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ and $e' \sqsubseteq e'_s$. From cover being consistent with partial order (well-behaving) and $\mathcal{C}^{\text{sub}}_{\mathbb{C}^\mathcal{A}} = N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}})$ (definition of highly reduced certificate), we know that $\neg\exists e'' \in N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}}) : e' \sqsubseteq e''$. Due to transitivity of $\sqsubseteq$ and definition of $N_{\text{hr}}(R^P_{\mathbb{C}^\mathcal{A}})$, we get that $\neg\exists e'' \in N_{\text{cov}} : e'_s \sqsubseteq e''$. From cover being monotonic

(well-behaving), $\mathsf{stop}(e,S) \implies \mathsf{cover}(e,S)$ (definition of $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$), $e' \sqsubseteq e'_s$, $N_{\text{cov}} \sqsubseteq N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, and $\neg\mathsf{cover}(e', N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}}))$, we conclude that $\neg\exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : e'_s \sqsubseteq n' \wedge \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (h(e), g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S)$. Thus, we get $(h(e), g, e'_s) \in S_{\text{TCNC}}$ (definition). Define function $h' : \mathsf{reached} \to N$ with $h := h \cup \{e' \mapsto \text{cov}((h(e), g, e'_s))\}$. We need to show that h' fulfills the requirements of our induction hypothesis after the execution of line 9. Since before execution of line 9 state $e' \notin \mathsf{reached}$, h' is a total function. Due to transitivity of $\sqsubseteq$, $e' \sqsubseteq e'_s$, and $e'_s \sqsubseteq \text{cov}((h(e), g, e'_s))$, we get $e' \sqsubseteq h(e')$. Since $\neg\mathsf{cover}(e', N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}}))$ and cover consistent with partial order, we know that $e' \notin N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}})$. Furthermore, we already know that $e \in \mathsf{reached}, g \in G_{\text{CFA}} : (e, g, e') \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \wedge f_{e,h'(e),g}((e, g, e')) \in S_{\text{TCNC}}$ and $h'(e') = \text{cov}(f_{e,h'(e),g}((e, g, e')))$. From $e' \sqsubseteq h(e')$, $\mathsf{cover}(e', N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}})) = \mathit{false}$, and cover consistent with partial order (well-behaving), we know that $h(e') \notin N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}})$. We only need to show that h' is injective. By induction h is injective. Thus, to show that h' is injective, we need to show that $\neg\exists \widehat{e'} \in \mathsf{reached} : h'(\widehat{e'}) = h'(e')$. Assume that there exists $\widehat{e'} \in \mathsf{reached}$ with $h'(\widehat{e'}) = h'(e')$. If $\widehat{e'} = e'$ nothing needs to be shown. Let $\widehat{e'} \neq e'$. From $h'(e') \notin N_{\text{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, we conclude that there exists $\widehat{e} \in \mathsf{reached}, \widehat{g} \in G_{\text{CFA}} : \wedge((\widehat{e}, \widehat{g}, \widehat{e'}) \in \rightsquigarrow \wedge f_{\widehat{e},h(\widehat{e}),\widehat{g}}((\widehat{e}, \widehat{g}, \widehat{e'})) \in S_{\text{TCNC}} \wedge \text{cov}(f_{\widehat{e},h(\widehat{e}),\widehat{g}}((\widehat{e}, \widehat{g}, \widehat{e'}))) = h(\widehat{e'}))$. Since cov is injective, we get $f_{\widehat{e},h(\widehat{e}),\widehat{g}}((\widehat{e}, \widehat{g}, \widehat{e'})) = f_{e,h(e),g}((e, g, e')) = h'(e')$. The definition of the functions f give us $(h(\widehat{e}), \widehat{g}, \widehat{e'_s}) = (h(e), g, e'_s)$. We get $h(\widehat{e}) = h(e), \widehat{g} = g, \widehat{e'_s} = e'_s$. Since h is injective, we know that $\widehat{e} = e$. Since $f_{e,h(e),g}$ is injective and $f_{\widehat{e},h(\widehat{e}),\widehat{g}}((\widehat{e}, \widehat{g}, \widehat{e'})) = f_{e,h(e),g}((e, g, \widehat{e'})) = f_{e,h(e),g}((e, g, e')) = h'(e')$, we conclude that $\widehat{e'} = e'$. The induction hypothesis follows.

From induction, we know that at line 10 such a function h exists. From function h and definition of $\sqsubseteq$, we conclude that in line 10 $\mathsf{reached} \sqsubseteq N$ and $|\mathsf{reached}| \leq |N|$ (h injective). □

Lemma 4.10. *Let $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover}) = (D_{\mathbb{C}^{\mathcal{A}}}, \rightsquigarrow, \mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^{\mathcal{A}}$ and coverage check cover which is well-behaving, $\rightsquigarrow$ be monotonic, $R^P_{\mathbb{C}^{\mathcal{A}}}$ be an ARG for finite program P and enhancement $\mathbb{C}^{\mathcal{A}}$ of CPA $\mathbb{C}$, and $R^P_{\mathbb{C}^{\mathcal{A}}}$ be well-formed for $e \in E_{\mathbb{C}^{\mathcal{A}}}$. Algorithm 4 started with CCV $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and certificate $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ returns true.*

Proof. Let $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$. We know that $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) = (N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}), |N|)$ (definition). From the definition of ARG and $N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, we know that $\text{cert}_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$ is finite. From Lemma 4.6, we deduce that Algorithm 4 terminates. Prove by contradiction that Algorithm 4 returns true. Assume that Algorithm 4 returns false. Algorithm 4 may return false in line 2 or 10.

Assume that Algorithm 4 returns false in line 2. Since $R^P_{\mathbb{C}^{\mathcal{A}}}$ is well-formed $e \sqsubseteq root \in N$. Due to transitivity of partial order $\sqsubseteq$, we get $e_0 \sqsubseteq root$. From definition of $N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, we know that $root \in N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$. If Algorithm 4 returns false in line 2, $\mathsf{cover}(e_0, N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})) = \mathit{false}$. Since cover is well-behaving (consistent with partial order) and $e_0 \sqsubseteq root \in N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, $\mathsf{cover}(e_0, N_{\text{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})) = \mathit{true}$. Contradiction to assumption, Algorithm 4 does not return false in line 2.

Assume that Algorithm 4 returns false in line 10. If Algorithm 4 returns false in line 10, $|\mathsf{reached}| > |N|$ or $\exists(\cdot, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$. The previous lemma gives us that at line 10 $\mathsf{reached} \sqsubseteq N$ and $|\mathsf{reached}| \leq |N|$. We infer that the second condition must be

violated. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is safe (well-formed), $\neg\exists(\cdot,q) \in N : q = q_\top \vee q = q_{\text{err}}$. By definition of $\mathsf{reached} \sqsubseteq N$, for every $(e,q) \in \mathsf{reached} \exists(e',q') \in N : (e,q) \sqsubseteq (e',q')$. By definition of $\sqsubseteq$ and $\mathcal{Q}$ being a flat lattice, we deduce that $\forall(e,q) \in \mathsf{reached} : q = q_\bot \vee (q \in Q \wedge q \neq q_{\text{err}})$. Hence, $\neg\exists(\cdot,q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$. Contradiction to assumption, Algorithm 4 does not return false in line 10.

Thus, Algorithm 4 returns true. □

A.3.2 Outstanding Proofs for Partitioned Certificates

Lemma 4.13. *Let $parts_{\mathbb{C}^\mathcal{A}}$ be a set of partition elements which is a safe overapproximation for program $P = (L, G_{\text{CFA}}, l_0)$, property automaton $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and a set of initial states $I \subseteq C$. Then, every path $p \in paths_P(I)$ is safe w.r.t. $\mathcal{A}$.*

Proof. Show by induction over the path length that for every path $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P(I)$ a configuration sequence $(c_0,q_0)\dots(c_n,q_n)$ for p and $\mathcal{A}$ exists s.t. $\forall 0 \leq i \leq n : \exists(pn,bn) \in parts_{\mathbb{C}^\mathcal{A}} : \exists(e_i,q_i') \in pn : q_i \sqsubseteq q_i' \wedge c_i \in [\![(e_i,q_i')]\!]_{\mathbb{C}^\mathcal{A}}$.

Basis Let $c_0 \in paths_P(I)$ be a path of length 0. By definition of paths, $c_0 \in I$. By definition, (c_0,q_0) is a configuration sequence for path c_0 and $\mathcal{A}$. Since the initial states are covered (safe overapproximation), we know that $\exists(pn,bn) \in parts_{\mathbb{C}^\mathcal{A}} : \exists(e',q_0) \in pn : c_0 \in [\![(e',q_0)]\!]_{\mathbb{C}^\mathcal{A}}$. Since $\sqsubseteq$ is reflexive, $q_0 \sqsubseteq q_0$. The induction hypothesis follows.

Step Let $p := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \xrightarrow{g_i} c_i \in paths_P(I)$ be a path of length i. By definition of paths, $p_{\text{sub}} := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \in paths_P(I)$ be a path of length $i-1$ and $g_i \in G_{\text{CFA}}$. By induction, a configuration sequence $(c_0,q_0)\dots(c_{i-1},q_{i-1})$ for p and $\mathcal{A}$ exists s.t. $\forall 0 \leq j \leq i-1 : \exists(pn,bn) \in parts_{\mathbb{C}^\mathcal{A}} : \exists(e_j,q_j') \in pn : q_j \sqsubseteq q_j' \wedge c_j \in [\![(e_j,q_j')]\!]_{\mathbb{C}^\mathcal{A}}$. Let $(pn,bn) \in parts_{\mathbb{C}^\mathcal{A}}$ be a partition element with $(e_{i-1},q_{i-1}') \in pn$, $c_{i-1} \in [\![(e_{i-1},q_{i-1}')]\!]$ and $q_{i-1} \sqsubseteq q_{i-1}'$. From $parts_{\mathbb{C}^\mathcal{A}}$ being a safe overapproximation, $c_{i-1} \in [\![(e_{i-1},q_{i-1}')]\!]$, $c_{i-1} \xrightarrow{g_i} c_i$, and $g_i \in G_{\text{CFA}}$, we get that an abstract state $(\widehat{e},\widehat{q}) \in pn \cup bn$ exists with $\widehat{q} = q_\top$ or $\exists C_{\text{sub}} \subseteq C : (c_i \in C_{\text{sub}} \wedge (q_{i-1}', op, C_{\text{sub}}, \widehat{q}) \in \delta)$, and $c_i \in [\![(\widehat{e},\widehat{q})]\!]$. If $(\widehat{e},\widehat{q}) \in pn$, set $(e_i,q_i') = (\widehat{e},\widehat{q})$. If $(\widehat{e},\widehat{q}) \notin pn$, we know that $(\widehat{e},\widehat{q}) \in bn$. Since $parts_{\mathbb{C}^\mathcal{A}}$ being a safe overapproximation, we can conclude that $\exists(pn,bn) \in parts_{\mathbb{C}^\mathcal{A}} : \exists(e_i,q_i') \in pn : c_i \in [\![(e_i,q_i')]\!]_{\mathbb{C}^\mathcal{A}} \wedge \widehat{q} \sqsubseteq q_i'$. We fix such an (e_i,q_i') for the second case. If $q_i' = q_\top$, set q_i to any $q \in Q$ with $(q_{i-1}, op, C_{\text{sub}}, q) \in \delta$ and $c_i \in C_{\text{sub}}$. Due to completion of δ, at least one exists. If $q_i' \in Q$, we set $q_i = q_i'$ and we know that $q_i' = \widehat{q}$ (in the latter case from $\mathcal{Q}$ being a flatt lattice and $\widehat{q} \sqsubseteq q_i'$). For both cases, we conclude that $(c0,q0)\dots(c_{i-1},q_{i-1})(c_i,q_i)$ is a configuration sequence for p and $\mathcal{A}$ and $q_i \sqsubseteq q_i'$. The induction hypothesis follows.

Let $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P(I)$ be an arbitrary path. From induction, we know that a configuration sequence $(c_0,q_0)\dots(c_n,q_n)$ for p and $\mathcal{A}$ exists with $\forall 0 \leq i \leq n : \exists(pn,bn) \in parts_{\mathbb{C}^\mathcal{A}} : \exists(e_i,q_i') \in pn : q_i \sqsubseteq q_i'$. Since $parts_{\mathbb{C}^\mathcal{A}}$ is a safe overapproximation, we know that $\forall 0 \leq i \leq n : q_i \neq q_{\text{err}}$. Hence, p is safe w.r.t. $\mathcal{A}$. □

Corollary 4.15. *Let $\mathcal{PC}_{\mathbb{C}^\mathcal{A}} = (parts_{\text{sub}}, n)$ be a partitioned certificate which is valid for a program P, property automaton $\mathcal{A}$, and a set of initial states $I \subseteq C$ and let $I_{\text{sub}} \subseteq I$ be a subset of the initial states. Then, $\mathcal{PC}_{\mathbb{C}^\mathcal{A}}$ is also valid for P, $\mathcal{A}$, and I_{sub}.*

Proof. Since $\mathcal{PC}_{\mathbb{C}^\mathcal{A}} = (parts_{\text{sub}}, n)$ is valid for P, $\mathcal{A}$, and I, we know that $parts_{\text{sub}}$ can be extended to a set $parts_{\mathbb{C}^\mathcal{A}}$ s.t. $parts_{\mathbb{C}^\mathcal{A}}$ is a safe overapproximation for P, $\mathcal{A}$, and I, there exists total surjective function $m : parts_{\text{sub}} \to parts_{\mathbb{C}^\mathcal{A}} : \forall (pn, bn) \in parts_{\text{sub}} : m((pn, bn)) = (pn', bn') \implies pn \subseteq pn' \wedge bn \subseteq bn'$ and $|\bigcup_{(pn,\cdot)\in parts_{\mathbb{C}^\mathcal{A}}} pn| \leq n$. We reuse the same extension to show that $\mathcal{PC}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and I_{sub}. Hence, it remains to be shown that $parts_{\mathbb{C}^\mathcal{A}}$ is a safe overapproximation for P, $\mathcal{A}$, and I_{sub}. From $parts_{\mathbb{C}^\mathcal{A}}$ being a safe overapproximation for P, $\mathcal{A}$, and I, we conclude that $I_{\text{sub}} \subseteq I \subseteq \bigcup_{(e,q_0)\in pn\wedge(pn,\cdot)\in parts_{\mathbb{C}^\mathcal{A}}} [\![(e, q_0)]\!]_{\mathbb{C}^\mathcal{A}}$, $\forall (pn, bn) \in parts_{\mathbb{C}^\mathcal{A}} : \forall (e, q) \in pn : \forall c \in [\![(e, q)]\!]_{\mathbb{C}^\mathcal{A}}$, $(l, op, l') \in G_{\text{CFA}} : c \xrightarrow{(l,op,l)} c' \implies (q' = q_\top \vee q' \in Q \wedge \exists C_{\text{sub}} \subseteq C : c' \in C_{\text{sub}} \wedge (q, op, C_{\text{sub}}, q') \in \delta) \wedge \exists (e', q'') \in (pn \cup bn) : q' \sqsubseteq q'' \wedge c' \in [\![(e', q'')]\!]_{\mathbb{C}^\mathcal{A}}$, $\forall (pn, bn) \in parts_{\mathbb{C}^\mathcal{A}} : \forall (e, q) \in bn : [\![(e, q)]\!]_{\mathbb{C}^\mathcal{A}} \subseteq \{[\![(e', q')]\!]_{\mathbb{C}^\mathcal{A}} \mid q \sqsubseteq q' \wedge \exists (pn, \cdot) \in parts_{\mathbb{C}^\mathcal{A}} : (e', q') \in pn\}$, and $\forall (e, q) \in E^\mathcal{A}, (pn, bn) \in parts_{\mathbb{C}^\mathcal{A}} : (e, q) \in pn \implies q \neq q_{\text{err}} \wedge q \neq q_\top$. We conclude that $\mathcal{PC}_{\mathbb{C}^\mathcal{A}}$ is valid for P, $\mathcal{A}$, and I_{sub}. □

Lemma 4.16. *Let $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and enhanced CPA $\mathbb{C}^\mathcal{A}$ which is well-formed for $e_0 = (e, q_0) \in E_{\mathbb{C}^\mathcal{A}}$. The set $parts_{\mathbb{C}^\mathcal{A}} := \{(N, N)\}$ of partition elements is a safe overapproximation for P, $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$, and $[\![root]\!]$.*

Proof. Let $P = (L, G_{\text{CFA}}, l_0)$. Since ARG $R^P_{\mathbb{C}^\mathcal{A}}$ is safe (well-formed), we know that $root = (\cdot, q_r)$ and $q_r \neq q_\top$. From $R^P_{\mathbb{C}^\mathcal{A}}$ being rooted (well-formed), we get that $e_0 \sqsubseteq root$. The definition of $\sqsubseteq$ and $\mathcal{Q}$ being a flat lattice, gives us $q_r = q_0$. Furthermore, $root \in N$ (definition of ARG) lets us conclude that $root \in \{(\widetilde{e}, q_0) \in N\}$ and $[\![root]\!] \subseteq \bigcup_{(\widetilde{e},q_0)\in pn\wedge(pn,\cdot)\in parts_{\mathbb{C}^\mathcal{A}}} [\![(\widetilde{e}, q_0)]\!]$.

Let (pn, bn) be an arbitrary partition element in $parts_{\mathbb{C}^\mathcal{A}}$. By definition, we know that $(pn, bn) = (N, N)$. Let $(\widehat{e}, q) \in pn = N$ be an arbitrary abstract state and $c \in [\![(\widehat{e}, q)]\!]$, $(l, op, l') \in G_{\text{CFA}}$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is safe (well-formed), we know that $q \neq q_\top$. Let $c \xrightarrow{(l,op,l')} c'$ be a transition in $\to$. Due to $c \in [\![(\widehat{e}, q)]\!]$ and $c \xrightarrow{(l,op,l')} c'$, we know that there exists $((\widehat{e}, q), (l, op, l'), (e', q')) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ and $c' \in [\![(e', q')]\!]$ (overapproximation). Due to definition of $\rightsquigarrow_{\mathbb{C}^\mathcal{A}}$, we infer that $q' = q_\top$ or $\exists (q, op, C_{\text{sub}}, q') \in \delta$ with $c' \in C_{\text{sub}}$. From $R^P_{\mathbb{C}^\mathcal{A}}$ being complete (well-formed) and $((\widehat{e}, q), g, (e', q')) \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$, we know that $\exists (e'', q'') \in E_{\mathbb{C}^\mathcal{A}} : (e', q') \sqsubseteq (e'', q'') \wedge (((\widehat{e}, q), g, (e'', q'')) \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid ((\widehat{e}, q), g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}((e'', q''), S))$. By meaning of $\sqsubseteq$, definition of $\sqsubseteq$, soundness of termination check, and $\{n'' \in N_{\text{cov}} \mid ((\widehat{e}, q), g, n'') \in G_{\text{ARG}}\} \subseteq N$, we get that an abstract state $(e_n, q_n) \in N = bn = pn \cup bn$ exists with $c' \in [\![(e_n, q_n)]\!]$ and $q' \sqsubseteq q_n$. Thus, concrete successor configurations (c', q') of a partition node are considered by the same partition element as the partition node.

Since the set of partition nodes and the set of boundary nodes are both N, for each boundary node the same node exists in the set of partition nodes. Every boundary node is covered by a partition node considering the same automaton abstract state. Hence, we get that boundary nodes are covered by partition nodes considering the same or a more abstract automaton abstract state.

From $R^P_{\mathbb{C}^\mathcal{A}}$ being safe (well-formed), we conclude that $\forall (\widehat{e}, q) \in N : q \neq q_{\text{err}} \wedge q \neq q_\top$. Since the the set of partition nodes is N, the partition nodes are safe.

Finally, $parts_{\mathbb{C}^\mathcal{A}}$ is a safe overapproximation for P, $\mathcal{A}$ and $[\![e_0]\!]$. □

Lemma 4.20. *Let* $\mathsf{reached}'$ *denote the reached set at the state of termination of Algorithm 5. If Algorithm 5 started with configurable certificate validator* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}$ *for abstract domain* $D_{\mathbb{C}^{\mathcal{A}}}$ *enhanced with property automaton* $\mathcal{A} = (Q, \delta, q_0, q_{\text{err}})$*, program* P*, initial abstract state* $e_0 = (e, q_0) \in E_{\mathbb{C}^{\mathcal{A}}}$*, and partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *returns true, then the set* $parts_{\mathbb{C}^{\mathcal{A}}} := \{(\mathsf{reached}', \mathsf{reached}')\}$ *of partition elements is a safe overapproximation for* P*,* $\mathcal{A}$*, and* $[\![e_0]\!]$*.*

Proof. Let $P = (L, G_{\text{CFA}}, l_0)$ and $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}} = (parts_{\text{sub}}, n)$. Define $PN := \bigcup_{(pn,\cdot)\in parts_{\text{sub}}} pn$.

Since Algorithm 5 returns true, we know that $\mathsf{cover}(e_0, PN) = true$. The definition of a coverage check cover lets us infer that $[\![e_0]\!] = [\![(e, q_0)]\!] \subseteq \bigcup_{(e',q')\in PN \wedge q_0 \sqsubseteq q'} [\![(e', q')]\!]$. From Algorithm 5 returning true, we infer that $PN \subseteq \mathsf{reached}'$ and for all $(e, q) \in \mathsf{reached}' : q \neq q_{\text{err}} \wedge q \neq q_\top$. Thus, $[\![e_0]\!] \subseteq \bigcup_{(e',q')\in \mathsf{reached}' \wedge q' = q_0} [\![(e', q')]\!]$.

Let (pn, bn) be an arbitrary partition element in $parts_{\mathbb{C}^{\mathcal{A}}}$. By definition, we know that $(pn, bn) = (\mathsf{reached}', \mathsf{reached}')$. Let $(\widehat{e}, q) \in pn = \mathsf{reached}'$ be an arbitrary abstract state and $c \in [\![(\widehat{e}, q)]\!]$, $(l, op, l') \in G_{\text{CFA}}$. Let $c \xrightarrow{(l,op,l')} c'$ be a transition in $\rightarrow$. Due to $c \in [\![(\widehat{e}, q)]\!]$ and $c \xrightarrow{(l,op,l')} c'$, we know that there exists $((\widehat{e}, q), (l, op, l'), (e', q')) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ and $c' \in [\![(e', q')]\!]$ (overapproximation). Due to definition of $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$, we infer that $q' = q_\top$ or $\exists (q, op, C_{\text{sub}}, q') \in \delta$ with $c' \in C_{\text{sub}}$. Since every state in $\mathsf{reached}$ is also put in $\mathsf{waitlist}$ and $(\widehat{e}, q) \in \mathsf{reached}'$, we know that $(\widehat{e}, q)$ was put to $\mathsf{waitlist}$. States are only removed from $\mathsf{waitlist}$ in line 5 or in line 3. If Algorithm 5 returns true, the while loop in line 4 only terminates if $\mathsf{waitlist}$ is empty, no states are removed from $\mathsf{waitlist}$ in line 3. Furthermore, $\mathsf{waitlist}$ is empty if Algorithm 5 terminates. We infer that $(\widehat{e}, q)$ was removed from $\mathsf{waitlist}$ in line 5, and, hence $((\widehat{e}, q), (l, op, l'), (e', q')) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ was explored in lines 7 to 9 considering a partition $(pn', bn') \in parts_{\text{sub}}$. If (e', q') was added in line 9, $(e', q') \in \mathsf{reached}'$, nothing remains to show. If $(e', q') \notin \mathsf{reached}'$, we know that $\mathsf{cover}((e', q'), pn' \cup bn') = true$. By definition of coverage check, we get that $[\![(e', q')]\!] \subseteq \bigcup_{(e'',q'')\in(pn'\cup bn')\wedge q' \sqsubseteq q''} [\![(e'', q'')]\!]$. Hence, there exists a state $(e_n, q_n) \in pn' \cup bn'$ with $c' \in [\![(e_n, q_n)]\!]$ and $q' \sqsubseteq q_n$. Since Algorithm 2 explores all partition elements $(pn, bn) \in parts_{\text{sub}}$, added pn to $\mathsf{reached}$ in line 3 for each (pn, bn), and never removes a state from $\mathsf{reached}$, we get that $\forall (pn, bn) \in parts_{\text{sub}} : pn \subseteq \mathsf{reached}'$. Since Algorithm 5 returns true, we conclude from line 12 that $\forall (pn, bn) \in parts_{\text{sub}} : bn \subseteq \bigcup_{(pn',\cdot)\in parts_{\text{sub}}} \subseteq \mathsf{reached}'$. We conclude that a state $(e_n, q_n) \in \mathsf{reached}'$ exists with $c' \in [\![(e_n, q_n)]\!]$ and $q' \sqsubseteq q_n$. Thus, concrete successor configurations (c', q') of a partition node are considered by the node's partition element.

Since the set of partition nodes and boundary nodes are both $\mathsf{reached}'$, each boundary node exists in the set of partition nodes. Every boundary node is covered by a partition node considering the same automaton abstract state. Boundary nodes are covered by partition nodes considering the same or a more abstract automaton abstract state.

We know that $\mathsf{reached}$ in line 12 equals $\mathsf{reached}'$. Since Algorithm 5 returns true, which is only possible at line 12, we get from the check in line 12 that $\forall (\widehat{e}, q) \in \mathsf{reached}' : q \neq q_{\text{err}} \wedge q \neq q_\top$. Since the set of partition nodes is $\mathsf{reached}'$, the partition nodes are safe.

Finally, $parts_{\mathbb{C}^{\mathcal{A}}}$ is a safe overapproximation for P, $\mathcal{A}$, and $[\![e_0]\!]$. □

Lemma 4.23 (Termination)**.** *Let* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ *be a configurable certificate validator for CPA* $\mathbb{C}^{\mathcal{A}}$ *and coverage check* cover *and program* $P = (L, G_{\text{CFA}}, l_0)$ *be finite. Then, Algorithm 5 started with* $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$*,* P*, initial abstract state* $e_0 \in E_{\mathbb{C}^{\mathcal{A}}}$*, and finite partitioned certificate* $\mathcal{PC}_{\mathbb{C}^{\mathcal{A}}}$ *terminates.*

Proof. Algorithm 5 terminates if the number of partition elements in the partitioned certificate are finite, the number of program edges are finite, for each pair $(e, g) \in E_{\mathbb{C}^\mathcal{A}} \times G_{\text{CFA}}$ only finitely many elements $(e, g, e') \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ exists and in each iteration of the for loop in line 2 the while loop terminates. Due to the definition of a finite partitioned certificate, the partitioned certificate contains only finitely many partition elements. The number of program edges are finite (finite program). Since the transfer relation is a transfer relation of a CPA (definition of $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$), it follows from Eq. 2.3 that $\forall (e, g) \in \mathcal{C}_{\mathbb{C}^\mathcal{A}} \times G_{\text{CFA}} : \exists n \in \mathbb{N} : |\{(e, g, e') \in \rightsquigarrow\}| \leq n$. Algorithm 4 only adds states to waitlist which are also added to reached. Due to the condition of the while loop and Algorithm 5 never removing states from reached, Algorithm 5 can only add finitely many, different states to reached and, thus, to waitlist, or terminates. Since Algorithm 5 pops an element from waitlist in each while loop iteration and Algorithm 5 terminates if waitlist becomes empty in each iteration of the for loop in line 2, Algorithm 5 terminates if an abstract state is added at most once to waitlist in each iteration of the for loop in line 2. Elements are only added in lines 3 and 9. In line 3, Algorithm 5 adds a subset of reached to waitlist. Since no state is ever removed from reached and waitlist $\subseteq$ reached, in line 9 we only add states to waitlist which have not been added to reached and thus to waitlist. Hence, in each iteration of the for loop in line 2, we add each element from reached at most once to waitlist. Algorithm 5 terminates. □

Lemma 4.24. *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for finite program P and $\mathbb{C}^\mathcal{A}$, and $R^P_{\mathbb{C}^\mathcal{A}}$ be well-formed for $e \in E_{\mathbb{C}^\mathcal{A}}$. Furthermore, let $N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N_{\text{sub}} \subseteq N$ and $\text{partition}(N_{\text{sub}})$ a partition of N_{sub}. If Algorithm 5 started with CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ from $\text{partition}(N_{\text{sub}})$ and $R^P_{\mathbb{C}^\mathcal{A}}$, then at line 12* reached $\subseteq N$.

Proof. Let $P = (L, G_{\text{CFA}}, l_0)$ and $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}}) = (parts_{\text{sub}}, n)$. Algorithm 5 initializes reached with the empty set, obviously $\emptyset \subseteq N$, and only adds states to reached in line 3 and 9 (within iteration of for loop in line 2). Consider arbitrary iteration of for loop in line 2, assume that this iteration considers partition element (pn, bn) and show by induction over the changes to waitlist that for every element e' added to waitlist in this iteration a sequence $e_1, \ldots, e_n = e'$ exists s.t. $e_1 \in pn$ and $\forall 2 \leq j \leq n : e_j \notin N_{\text{sub}} \wedge \exists (e_{j-1}, \cdot, e_j) \in G_{\text{ARG}}$.

Basis Algorithm 5 initializes waitlist in line 3 with pn. For every $e_1 \in$ waitlist there exists a sequence e_1 with the desired property. The induction hypothesis follows.

Step After initialization in line 3, Algorithm 5 only adds states e' in line 9. We know that in line 5 Algorithm 5 popped element e_p from waitlist. Furthermore, we infer that there exists $(e_p, g, e') \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ and $g \in G_{\text{CFA}}$. Since in each iteration of for loop in line 2, Algorithm 5 only pops elements from waitlist which it added in the same iteration, by induction we know that $e_1, \ldots, e_n = e_p$ exists s.t. $e_1 \in pn$ and $\forall 2 \leq j \leq n : e_j \notin N_{\text{sub}} \wedge \exists (e_{j-1}, \cdot, e_j) \in G_{\text{ARG}}$. We infer that $e_p \in N$ (exists $(\cdot, \cdot, e_p) \in G_{\text{ARG}}$). Since Algorithm 5 added all elements of pn to reached in line 3, we know from condition in line 8 that $e' \notin pn$. We need to show that $e' \notin N_{\text{sub}}$ and an ARG edge $(e, \cdot, e') \in G_{\text{ARG}}$ exists. From completeness (well-formedness) of ARG $R^P_{\mathbb{C}^\mathcal{A}}$, we get that $\exists n' \in E_{\mathbb{C}^\mathcal{A}} : e' \sqsubseteq n' \wedge ((e_p, g, n') \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S))$. If $n' \neq e'$, we know that $n' \in N_{\text{sub}}$ or $\{n'' \in N_{\text{cov}} \mid (e_p, g, n'') \in G_{\text{ARG}}\} \subseteq N_{\text{sub}}$ (definition of N_{sub} and $N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}})$). In

both cases, we know there exists a path $e_0, \ldots, e_n, n'$ and $e_0, \ldots, e_n, n''$ in ARG $R^P_{\mathbb{C}^\mathcal{A}}$. From $e_0, n', n'' \in N_{\text{sub}}$, we get that (e_0, n'), (e_0, n'') are edges in $G_{N_{\text{sub}}}$ (construction of vertex contracted graph). We conclude that n' and n'', respectively, are contained in bn. Since cover adheres to $\mathsf{stop}_{\mathbb{C}^\mathcal{A}}$, is consistent with partial order, and monotonic (well-behaving), we get $\mathsf{cover}(e', pn \cup bn) = \mathit{true}$ if $n' \neq e'$. We conclude that $n' = e'$ and $(e_p, g, e') \in \text{ARG}$. If $e' \in N_{\text{sub}}$, we conclude that $e' \in bn$ (similar reasons as before). Since cover consistent with partial order, $e' \sqsubseteq e'$ and $\mathsf{cover}(e', pn \cup bn) = \mathit{false}$, we infer that $e' \notin N_{\text{sub}}$. For sequence $e_1, \ldots, e_n, e'$ we get $e_1 \in pn$ and $\forall 2 \leq j \leq n : e_j \notin N_{\text{sub}} \wedge \exists (e_{j-1}, \cdot, e_j) \in G_{\text{ARG}}$. The induction hypothesis follows.

By definition of partitioned certificate we know that $pn \subseteq N_{\text{sub}} \subseteq N$. Furthermore, $(e_j, \cdot, e_{j+1}) \in G_{\text{ARG}}$ implies $e_j, e_{j+1} \in N$. By induction on arbitrary iteration, we conclude that for every element e' added to waitlist that $e' \in N$. Since Algorithm 5 adds every state to waitlist which it adds to reached, we conclude by induction that it only adds states from N to reached. It follows that at line 12 $\mathsf{reached} \subseteq N$. □

Lemma 4.25. *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *which is well-behaving, $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for finite program P and $\mathbb{C}^\mathcal{A}$, and $R^P_{\mathbb{C}^\mathcal{A}}$ be well-formed for $e \in E_{\mathbb{C}^\mathcal{A}}$. Furthermore, let $N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N_{\text{sub}} \subseteq N$ and $\text{partition}(N_{\text{sub}})$ a partition of N_{sub}. Algorithm 5 started with CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ from $\text{partition}(N_{\text{sub}})$ and $R^P_{\mathbb{C}^\mathcal{A}}$ returns true.*

Proof. Let $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}}) = (parts_{\text{sub}}, n)$. From $N_{\text{sub}} \subseteq N$ and N being finite, we conclude that N_{sub} is finite. The definition of a partition of N_{sub} lets us conclude that $\exists k \in \mathbb{N} : |\text{partition}(N_{\text{sub}})| = k$. By definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, $parts_{\text{sub}}$ contains exactly k (finitely many) partition elements (pn, bn). Due to construction of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, we know for each $(pn, bn) \in parts_{\text{sub}}$ that $pn \subseteq N_{\text{sub}} \subseteq N$ and $bn \subseteq N_{\text{sub}} \subseteq N$. Hence, $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ is finite. From Lemma 4.23 we conclude that Algorithm 5 terminates.

It remains to show that Algorithm 5 returns true. Assume that Algorithm 5 returns false which is only possible in line 11 or 12.

If Algorithm 5 returns false in line 11, then $\mathsf{cover}(e_0, \bigcup_{(pn,\cdot) \in parts_{\text{sub}}} pn) = \mathit{false}$. By definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, $\text{partition}(N_{\text{sub}})$ being a partition of N_{sub}, we get $\bigcup_{(pn,\cdot) \in parts_{\text{sub}}} pn = N_{\text{sub}}$. From definition of $N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}})$ and $N_{\text{R}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N_{\text{sub}}$, we get that $root \in N_{\text{sub}}$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is rooted (well-formed), we know that $e \sqsubseteq root$. By transitivity of partial order $\sqsubseteq$ and $e_0 \sqsubseteq e$, we get $e_0 \sqsubseteq root$. From cover being consistent with partial order (well-behaving), we conclude that $\mathsf{cover}(e_0, \bigcup_{(pn,\cdot) \in parts_{\text{sub}}} pn) = \mathit{true}$ which is a contradiction. Thus, Algorithm 5 does not return false in line 11.

If Algorithm 5 returns false in line 12, then $|\mathsf{reached}| \leq n$, $\exists (e, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$ or $\bigcup_{(\cdot,bn) \in parts_{\text{sub}}} bn \not\subseteq \bigcup_{(pn,\cdot) \in parts_{\text{sub}}} pn$. From the previous lemma, we know that at line 12 $\mathsf{reached} \subseteq N$. From definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, we know that $n = |N|$. We get that $|\mathsf{reached}| \leq |N| = n$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is safe (well-formed), we know that $\neg \exists (e, q) \in N : q = q_\top \vee q = q_{\text{err}}$. Hence, $\neg \exists (e, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$. From definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$, we conclude that $\forall (\cdot, bn) \in parts_{\text{sub}} : bn \subseteq N_{\text{sub}}$. Thus, $\bigcup_{(\cdot,bn) \in parts_{\text{sub}}} bn \subseteq N_{\text{sub}} = \bigcup_{(pn,\cdot) \in parts_{\text{sub}}} pn$. Contradiction to Algorithm 5 returning false in line 12. Finally, we conclude that Algorithm 5 returns true. □

Lemma 4.26. *Let $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^{\mathcal{A}}$ and coverage check* cover *which is well-behaving, let transfer relation $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ be monotonic, $R^P_{\mathbb{C}^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ be an ARG for finite program P and $\mathbb{C}^{\mathcal{A}}$, and $R^P_{\mathbb{C}^{\mathcal{A}}}$ be well-formed for $e \in E_{\mathbb{C}^{\mathcal{A}}}$. If Algorithm 5 started with CCV $\mathbb{V}^{\mathbb{C}^{\mathcal{A}}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and highly reduced, partitioned certificate $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}})$ from $\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}))$ and $R^P_{\mathbb{C}^{\mathcal{A}}}$, then at line 12* reached $\sqsubseteq N$ *and* $|\mathsf{reached}| \leq |N|$.

Proof. Let $P = (L, G_{\mathrm{CFA}}, l_0)$ and $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}})), R^P_{\mathbb{C}^{\mathcal{A}}}) = (parts_{\mathrm{sub}}, n)$. Let $S_{\mathrm{TCNC}} := \{(n, g, e) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} | \; n \in N, g \in G_{\mathrm{CFA}} \wedge \neg\exists n' \in E_{\mathbb{C}^{\mathcal{A}}} : e \sqsubseteq n' \wedge ((n, g, n') \in G_{\mathrm{ARG}} \wedge n' \in N_{\mathrm{cov}} \vee \exists S \sqsubseteq \{n'' \in N_{\mathrm{cov}} \mid (n, g, n'') \in G_{\mathrm{ARG}}\} : \mathsf{stop}_{\mathbb{C}^{\mathcal{A}}}(n', S)\}$ be the set of abstract successor computations of abstract successors which are covered by non-covering nodes. Let $\mathrm{cov} : S_{\mathrm{TCNC}} \to N \setminus N_{\mathrm{cov}}$ be a total, injective function with $\forall (n, g, e) \in S_{\mathrm{TCNC}} : e \sqsubseteq \mathrm{cov}(e) \wedge (n, g, \mathrm{cov}(e)) \in G_{\mathrm{ARG}}$. (such a function exists because $R^P_{\mathbb{C}^{\mathcal{A}}}$ is well-covered (well-formed)). For $e, e' \in E_{\mathbb{C}^{\mathcal{A}}}$ s.t. $e \sqsubseteq e'$ and $g \in \mathcal{G}$ let $f_{e,e',g} : \{(e, g, \cdot) \in \rightsquigarrow\} \to \{(e', g, \cdot) \in \rightsquigarrow\}$ be a total, injective function with $\forall (e, g, e_s) \in \rightsquigarrow : f((e, g, e_s)) = (e', g, e'_s) \implies e_s \sqsubseteq e'_s$ (such functions exists because $\rightsquigarrow$ is monotonic).

Show by induction over the changes of reached and waitlist, that partial, surjective function $h : N \to$ reached and partial function $g : N \to \{pn \mid (pn, \cdot) \in parts_{\mathrm{sub}}\}$ exist s.t. for every $n \in N$, $h(n)$ is defined iff $g(n)$ is defined, $\forall n \in N : h(n)$ defined $\implies (h(n) \sqsubseteq n \wedge \exists e_0, g_1, e_1, \ldots, g_j, e_j : e_0 \in g(n) \wedge e_j = n \wedge \forall 1 \leq i \leq j : (e_{i-1}, g_i, e_i) \in G_{\mathrm{ARG}} \wedge e_i \notin N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \wedge (n \in N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \implies h(n) = n \wedge n \in g(n)) \wedge (n \notin N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \implies \exists n' \in N, g \in G_{\mathrm{CFA}} : h(n')$ defined $\wedge (h(n'), g, h(n)) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \wedge f_{h(n'),n',g}((h(n'), g, h(n))) \in S_{\mathrm{TCNC}} \wedge (n', g, n) \in G_{\mathrm{ARG}} \wedge \mathrm{cov}(f_{h(n'),n',g}((h(n'), g, h(n)))) = n))$ and if e is added to waitlist during exploration of partition element (pn, bn), then $\exists \widehat{n} \in h^{-1}(e) : g(\widehat{n}) = pn$.

Basis Algorithm 5 initializes reached and waitlist with the empty set in line 1. Since reached $= \emptyset$, no state need to be mapped. Since waitlist $= \emptyset$, no state is added to waitlist. No property need to be assured. The induction hypothesis follows.

Step After the initialization, states are added to reached and waitlist in line 3 or line 9. By induction, partial, surjective function $h : N \to$ reached and partial function $g : N \to \{pn \mid (pn, \cdot) \in parts_{\mathrm{sub}}\}$ exist s.t. for every $n \in N$, $h(n)$ is defined iff $g(n)$ is defined, $\forall n \in N : h(n)$ defined $\implies (h(n) \sqsubseteq n \wedge \exists e_0, g_1, e_1, \ldots, g_j, e_j : e_0 \in g(n) \wedge e_j = n \wedge \forall 1 \leq i \leq j : (e_{i-1}, g_i, e_i) \in G_{\mathrm{ARG}} \wedge e_i \notin N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \wedge (n \in N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \implies h(n) = n \wedge n \in g(n)) \wedge (n \notin N_{\mathrm{hR}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \implies \exists n' \in N, g \in G_{\mathrm{CFA}} : h(n')$ defined $\wedge (h(n'), g, h(n)) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}} \wedge f_{h(n'),n',g}((h(n'), g, h(n))) \in S_{\mathrm{TCNC}} \wedge (n', g, n) \in G_{\mathrm{ARG}} \wedge \mathrm{cov}(f_{h(n'),n',g}((h(n'), g, h(n)))) = n)))$ and if e is added to waitlist during exploration of partition element (pn, bn), then $\exists \widehat{n} \in h^{-1}(e) : g(\widehat{n}) = pn$.

If Algorithm 5 adds states pn to reached and waitlist in line 3, we know $\exists (pn, \cdot) \in parts_{\mathrm{sub}}$. By construction of a highly reduced, partitioned certificate, we know that $pn \subseteq N_{\mathrm{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}}) \subseteq N$. Define $h' : h \cup \{e_{\mathrm{hR}} \mapsto e_{\mathrm{hR}} \mid e_{\mathrm{hR}} \in pn\}$ and $g' : g \cup \{e_{\mathrm{hR}} \mapsto pn \mid e_{\mathrm{hR}} \in pn\}$. Obviously, h' is defined iff g' is defined and $\forall e_{\mathrm{hR}} \in pn : e_{\mathrm{hR}} \sqsubseteq e_{\mathrm{hR}}$ (reflexivity of partial order). Since we defined a projection for every element which is added to reached, h' is surjective. For each element $e_{\mathrm{hR}} \in pn$, sequence $e_0 = e_{\mathrm{hR}}$ fulfills the necessary requirements $e_0 \in g(e_{\mathrm{hR}})$ and $e_0 = e_{\mathrm{hR}}$. All further properties are fulfilled by construction, we only need to show for any $e_{\mathrm{hR}} \in pn$ that if $h(e_{\mathrm{hR}})$ and, thus, $g(e_{\mathrm{hR}})$ are defined, then $h(e_{\mathrm{hR}}) = e_{\mathrm{hR}}$ and $g(e_{\mathrm{hR}}) = pn$. By induction and $e_{\mathrm{hR}} \in N_{\mathrm{hr}}(R^P_{\mathbb{C}^{\mathcal{A}}})$, we get $h(e_{\mathrm{hR}}) = e_{\mathrm{hR}}$ and $e_{\mathrm{hR}} \in g(e_{\mathrm{hR}})$. The definition of the highly reduced, partitioned certificate, gives us that $g(e_{\mathrm{hR}})$ is an element

from partition($N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$). Since the elements from partition($N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$), which are sets, are disjoint, there exists only one element pn with $e_{\text{hR}} \in pn$. The induction hypothesis follows.

If Algorithm 5 adds state e' in line 9, we know that $(e, g, e') \in \rightsquigarrow_{\mathbb{C}^\mathcal{A}}$, $g \in G_{\text{CFA}}$ and $\mathsf{cover}(e', pn \cup bn) = \mathit{false}$. Since Algorithm 5 only adds elements from $\mathsf{reached}$ to $\mathsf{waitlist}$, we know $e \in \mathsf{reached}$. Furthermore, in each iteration of the for loop in line 2, Algorithm 5 pops only elements from $\mathsf{waitlist}$ which it added in the same iteration of that for loop. Assume that e is added in the iteration considering partition element (pn, bn). By induction, $\exists n \in h^{-1}(e) : g(n) = pn$, $h(n)$ defined, and $e = h(n) \sqsubseteq n$. Let $f_{h(n),n,g}((h(n), g, e')) = (n_1, g', e_s)$. The definition gives us $n_1 = n$, $g = g'$, and $e' \sqsubseteq e_s$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is complete and $n \in N$, we know that $\exists n' \in E_{\mathbb{C}^\mathcal{A}} : e_s \sqsubseteq n' \wedge ((n, g, n') \in G_{\text{ARG}} \vee \exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (n, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S))$. Assume $n' \in N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$ or $\exists S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_j, g, n'') \in G_{\text{ARG}}\} : \mathsf{stop}_{\mathbb{C}^\mathcal{A}}(n', S)$. By induction, we get that there exists $e_0, g_1, e_1, \ldots, g_j, e_j : e_0 \in g(n) \wedge e_j = n \wedge \forall 1 \leq i \leq j : (e_{i-1}, g_i, e_i) \in G_{\text{ARG}} \wedge e_i \notin N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. We conclude that $e_0, g_1, e_1, \ldots, g_n, e_n, g, n'$ and $e_0, g_1, e_1, \ldots, g_n, e_n, g, n''$ would be paths in each case and $\forall 1 \leq i \leq j : e_i \notin N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$, $e_0 \in pn \subseteq N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$ and $n', n'' \in N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. Hence, an edge (e_0, n'), (e_0, n'') would exist in $G_{N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})}$. By certificate construction, we get $n'', n' \in pn \cup bn$. Since $e' \sqsubseteq e_s \sqsubseteq n'$ and cover is consistent with the partial order, $\mathsf{cover}(e', pn \cup bn) = \mathit{true}$ in the first case. Since $\mathsf{stop}_{\mathbb{C}^\mathcal{A}} \implies \mathsf{cover}$, $e' \sqsubseteq n'$, $S \sqsubseteq \{n'' \in N_{\text{cov}} \mid (e_j, g, n'') \in G_{\text{ARG}}\} \sqsubseteq pn \cup bn$, and cover is monotonic, we get $\mathsf{cover}(e', pn \cup bn) = \mathit{true}$ in the latter case. The assumption was wrong, we get for all $(n, g, n') \in G_{\text{ARG}}$ with $e_s \sqsubseteq n'$ that $n' \in N \setminus N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N \setminus N_{\text{cov}}$. We get that $(n, g, e_s) = f_{h(n),n,g}((h(n), g, e')) \in S_{\text{TCNC}}$. We conclude that $(n, g, \text{cov}((n, g, e_s))) \in G_{\text{ARG}}$ and $e_s \sqsubseteq \text{cov}((n, g, e_s))$. From $e' \sqsubseteq e_s$ and $e_s \sqsubseteq \text{cov}((n, g, e_s))$, we get $e' \sqsubseteq \text{cov}((n, g, e_s))$. The sequence $e_0, g_1, e_1, \ldots, g_j, e_j, g_{j+1}, e_{j+1}$ with $g_{j+1} = g$ and $e_{j+1} = \text{cov}((n, g, e_s))$ fulfills $e_0 \in pn \wedge e_{j+1} = \text{cov}((n, g, e_s)) \wedge \forall 1 \leq i \leq j : (e_{i-1}, g_i, e_i) \in G_{\text{ARG}} \wedge e_i \notin N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. Define $h' : h \cup \{\text{cov}((n, g, e_s)) \mapsto e'\}$ and $g' : g \cup \{\text{cov}((n, g, e_s)) \mapsto pn\}$. It remains to be shown that $h(\text{cov}((n, g, e_s)))$ is not defined, and thus $g(\text{cov}((n, g, e_s)))$ is not defined. Assume that $h(\text{cov}((n, g, e_s)))$ is defined. Let $n* = \text{cov}((n, g, e_s))$. Since $n* = \text{cov}((n, g, e_s)) \notin N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$, there exists $\widehat{n} \in N, \widehat{g} \in G_{\text{CFA}} : h(\widehat{n})$ defined and $\text{cov}(f_{h(\widehat{n}),\widehat{n},\widehat{g}}((h(\widehat{e}), \widehat{g}, h(n*)))) = n*$. We get $n* = \text{cov}(f_{h(\widehat{n}),\widehat{n},\widehat{g}}((h(\widehat{e}), \widehat{g}, h(n*))))$ $= \text{cov}(f_{h(n),n,g}((h(n), g, e'))) = \text{cov}((n, g, e_s))$. Since function cov is injective, we get $f_{h(\widehat{n}),\widehat{n},\widehat{g}}((h(\widehat{n}), \widehat{g}, h(n*)))) = f_{h(n),n,g}((h(n), g, e')) = (n, g, e_s)$. We infer that $f_{h(\widehat{n}),\widehat{n},\widehat{g}}((h(\widehat{n}), \widehat{g}, h(n*)))) = (\widehat{n}, \widehat{g}, e_s) = f_{h(n),n,g}((h(n), g, e')) = (n, g, e_s)$. It follows that $n = \widehat{n}, g = \widehat{g}$, and thus $h(\widehat{n}) = h(n)$. Since $f_{h(e),n,g}$ is injective, we conclude that $(h(\widehat{n}), \widehat{g}, h(n*)) = (h(n), g, h(n*)) = (h(n), g, e')$. We get $h(n*) = e'$. Since $e' \notin \mathsf{reached}$ (condition in line 8), we get that $h(\text{cov}((n, g, e_s)))$ is not defined. The induction hypothesis follows.

By induction, at line 12 a partial, surjective function $h : N \to \mathsf{reached}$ with $\forall \widehat{e} \in N : h(\widehat{e})$ defined $\implies h(\widehat{e}) \sqsubseteq \widehat{e}$ exists. From h being surjective, we conclude that $\mathsf{reached} \sqsubseteq N$ (definition of $\sqsubseteq$) and $|\mathsf{reached}| \leq |N|$. □

Lemma 4.27. *Let $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$ be a configurable certificate validator for CPA $\mathbb{C}^\mathcal{A}$ and coverage check* cover *which is well-behaving, let transfer relation $\rightsquigarrow_{\mathbb{C}^\mathcal{A}}$ be monotonic, $R^P_{\mathbb{C}^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for finite program P and $\mathbb{C}^\mathcal{A}$, and $R^P_{\mathbb{C}^\mathcal{A}}$ be well-formed for $e \in E_{\mathbb{C}^\mathcal{A}}$. Algorithm 5 started with CCV $\mathbb{V}^{\mathbb{C}^\mathcal{A}}(\mathsf{cover})$, program P, initial abstract state $e_0 \sqsubseteq e$, and highly reduced, partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}})$ from $\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}))$ and $R^P_{\mathbb{C}^\mathcal{A}}$ returns true.*

Proof. Let highly reduced, partitioned certificate be $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}}) = (parts_{\text{sub}}, n)$. By definition of $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$, we get $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}) \subseteq N$. We conclude that $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$ is finite (N finite by definition of ARG). The definition of a partition of $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$ lets us conclude that $\exists k \in \mathbb{N} : |\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}))| = k$. By definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}})$, $parts_{\text{sub}}$ contains exactly k (finitely many) partition elements (pn, bn). Due to the construction of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}})$, we know for each $(pn, bn) \in parts_{\text{sub}}$ that $pn \subseteq N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$ and $bn \subseteq N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. Hence, $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{\mathbb{C}^\mathcal{A}})$ is finite. From Lemma 4.23 we conclude that Algorithm 5 terminates.

It remains to be shown that Algorithm 5 returns true. Assume that Algorithm 5 returns false, which is only possible in line 11 or 12.

If Algorithm 5 returns false in line 11, then $\mathsf{cover}(e_0, \bigcup_{(pn,\cdot)\in parts_{\text{sub}}} pn) = false$. By definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}})$, $\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$ being a partition of $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$, we get $\bigcup_{(pn,\cdot)\in parts_{\text{sub}}} pn = N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. From definition of $N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$, we get that $root \in N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is rooted (well-formed), we know that $e \sqsubseteq root$. By transitivity of partial order $\sqsubseteq$ and $e_0 \sqsubseteq e$, we get $e_0 \sqsubseteq root$. From cover being consistent with partial order (well-behaving), we conclude that $\mathsf{cover}(e_0, (\bigcup_{(pn,\cdot)\in parts_{\text{sub}}} pn) = true$ which is a contradiction. Thus, Algorithm 5 does not return false in line 11.

If Algorithm 5 returns false in line 12, then $|\mathsf{reached}| \leq n$, $\exists (e, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$ or $\bigcup_{(\cdot,bn)\in parts_{\text{sub}}} bn \not\subseteq \bigcup_{(pn,\cdot)\in parts_{\text{sub}}} pn$. From the previous lemma, we know that at line 12 it yields that $\mathsf{reached} \sqsubseteq N$ and $|\mathsf{reached}| \leq |N|$. Furthermore, from definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}})$, we know that $n = |N|$. We get that $|\mathsf{reached}| \leq |N| = n$. Since $R^P_{\mathbb{C}^\mathcal{A}}$ is safe (well-formed), we know that $\neg\exists (e, q) \in N : q = q_\top \vee q = q_{\text{err}}$. From $\mathsf{reached} \sqsubseteq N$ and $\mathcal{Q}$ being a flat lattice, it follows that $\neg\exists (e, q) \in \mathsf{reached} : q = q_\top \vee q = q_{\text{err}}$. From definition of $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})), R^P_{\mathbb{C}^\mathcal{A}})$, we conclude that $\forall (\cdot, bn) \in parts_{\text{sub}} : bn \subseteq N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}})$. Thus, $\bigcup_{(\cdot,bn)\in parts_{\text{sub}}} bn \subseteq N_{\text{hR}}(R^P_{\mathbb{C}^\mathcal{A}}) = \bigcup_{(pn,\cdot)\in parts_{\text{sub}}} pn$. Contradiction to Algorithm 5 returning false in line 12.

Finally, we conclude that Algorithm 5 returns true. □

A.4 Outstanding Proofs for Chapter 5 Programs from Proofs

Lemma 5.1. *If Algorithm 2 started with refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$, any initial precision $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$, and program P returns $(true, \cdot, (N, G_{\text{ARG}}, root, N_{\text{cov}}))$, then $\forall ((\cdot, e_1), \cdot) \in N : \text{acs}(e_1) \in \mathcal{L}$.*

Proof. Show by induction over the changes of reached that $\forall((e_2, e_1), q) \in \mathsf{reached} : \mathrm{acs}(e_1) \in \mathcal{L}$.

Basis Algorithm 2 initializes reached in line 1 with $e_0 = ((e_2^0, e_1^0), q_0)$. Since e_0 is a compatible, initial abstract, we know that $\mathrm{acs}(e_1^0) \in \mathcal{L}$. The induction hypothesis follows.

Step Algorithm 2 changes reached in line 12 or line 19.

If it changes reached in line 12, it will add $e_{\text{new}} = ((e_2^n, e_1^n), q^n)$. By definition of e_{new} in line 9, e_{new} is the result of a merge of e_{prec} and $e'' = ((e_2'', e_1''), q'')$. The requirements on merge in a refined property checking analysis give us that $\mathrm{acs}(e_1^n) = \mathrm{acs}(e_1'')$. Since $e'' \in \mathsf{reached}$, by induction we get $\mathrm{acs}(e_1^n) = \mathrm{acs}(e_1'') \in \mathcal{L}$.

If Algorithm 2 changes reached in line 19, it adds $e_{\text{prec}} = ((e_2^p, e_1^p), q^p)$. By definition of e_{prec} in line 7, it is the result of precision adjustment for e'. The definition of the precision adjustment operator and the requirements on the precision adjustment operator in a refined property analysis, let us conclude that $\mathrm{acs}(e_1^p) = \mathrm{acs}(e_1')$. Due to line 6, we know that $\exists(e, g, e') \in \rightsquigarrow$ and $(e, \cdot)$ was popped from waitlist in line 4. Since all abstract states in waitlist are also contained in reached, we get from induction and $e = ((e_2, e_1), q) \in \mathsf{reached}$ that $\mathrm{acs}(e_1) \in \mathcal{L}$. From the definition of the transfer relation of a refined property checking analysis, we know that $(\mathrm{acs}(e_1), g, \mathrm{acs}(e_1')) \in \rightsquigarrow_{\mathbb{L}}$. From $\mathrm{acs}(e_1) \in \mathcal{L}$ and the definition of $\rightsquigarrow_{\mathbb{L}}$, it follows that $\mathrm{acs}(e_1') \in \mathcal{L}$. Thus, $\mathrm{acs}(e_1^p) \in \mathcal{L}$.

The induction hypothesis follows.

By induction, we know that at line 29 $\forall((e_2, e_1), q) \in \mathsf{reached} : \mathrm{acs}(e_1) \in \mathcal{L}$ is true. In line 29, $N = \mathsf{reached}$. It follows that $\forall((e_2, e_1), q) \in N : \mathrm{acs}(e_1) \in \mathcal{L}$ □

Theorem 5.3 (Behavioral Equivalence)**.** *Let* $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\mathrm{ARG}}, root, N_{\mathrm{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *which is strongly well-formed for* $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$. *Then,* $paths_P([\![root]\!]) =_{\mathrm{nl}} paths_{\mathrm{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ *with* $I = \{c \mid c \in C \wedge \mathrm{cs}(c) = root \wedge \exists c' \in [\![root]\!] : \mathrm{ds}(c') = \mathrm{ds}(c)\}$.

Proof. Let $\mathrm{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) = (L', G'_{\mathrm{CFA}}, l'_0)$ and $P = (L, G_{\mathrm{CFA}}, l_0)$. Note that program locations $l' \in L'$ of program $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ are ARG nodes (construction of program). We need to show that for every path $p \in paths_P([\![root]\!])$ a path $p' \in paths_{\mathrm{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ exists with $p =_{\mathrm{nl}} p'$ (case $\subseteq$) and for every path $p' \in paths_{\mathrm{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ a path $p \in paths_P([\![root]\!])$ with $p' =_{\mathrm{nl}} p$ exists (case $\supseteq$).

$\subseteq$ Show by induction over the path length that for all paths $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P([\![root]\!])$ a path $p' := c'_0 \xrightarrow{g'_1} c'_1 \cdots \xrightarrow{g'_n} c'_n \in paths_{\mathrm{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ exists with $\forall 0 \leq i \leq n : c_i \in [\![\mathrm{cs}(c'_i)]\!]$ and $p =_{\mathrm{nl}} p'$.

Basis Let $c_0 \in paths_P([\![root]\!])$ a path of length 0. From definition of paths, $c_0 \in [\![root]\!]$. By program construction, $root = l'_0$. Define $c'_0 = (root, \mathrm{ds}(c_0))$. Since $c_0 \in C$, the assumptions on concrete states give us $c'_0 \in C$. By definition, $c'_0 \in I$. Hence, by definition $c'_0 \in paths_{\mathrm{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})}(I)$ and $c_0 =_{\mathrm{nl}} c'_0$. The induction hypothesis follows

Step Let $p := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \xrightarrow{g_i} c_i \in paths_P([\![root]\!])$ be a path of length i. By definition, $p_{\text{sub}} := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \in paths_P([\![root]\!])$ is a path of length $i-1$. By induction, a path $p'_{\text{sub}} := c'_0 \xrightarrow{g'_1} c'_1 \dots c'_{i-1} \in paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ exists with $\forall 0 \leq j \leq i-1 : c_j \in [\![\text{cs}(c'_j)]\!]$ and $p_{\text{sub}} =_{\text{nl}} p'_{\text{sub}}$. Since $c_{i-1} \xrightarrow{g_i} c_i$, $c_{i-1} \in [\![\text{cs}(c'_{i-1})]\!]$, there exists $(\text{cs}(c'_{i-1}), g_i, e) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ and $c_i \in [\![e]\!]$ (overapproximation of transfer relation). From $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ being complete ((strongly) well-formed), $\text{cs}(c'_{i-1}) \in N$, $g_i \in G_{\text{CFA}}$, and $(\text{cs}(c'_{i-1}), g_i, e) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, we know that there exists $(\text{cs}(c'_{i-1}), g_i, n') \in G_{\text{ARG}}$ s.t. $c_i \in [\![n']\!]$. Let $g_i = (\cdot, op_i, \cdot)$. By program construction, $(\text{cs}(c'_{i-1}), op_i, n') \in G'_{\text{CFA}}$. Define $c'_i = (n', \text{ds}(c_i))$. The assumptions on concrete states and $c_i \in C$, give us $c'_i \in C$. From the assumptions on transition relation $\rightarrow$, $c_{i-1} \xrightarrow{g_i} c_i$, $\text{cs}(c'_i) = n'$, we get $c'_{i-1} \xrightarrow{(\text{cs}(c'_{i-1}), op, n')} c'_i$. We conclude that $p' := c'_0 \xrightarrow{g'_1} c'_1 \dots c'_{i-1} \xrightarrow{(\text{cs}(c'_{i-1}), op, n')} c'_i \in paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ and $p =_{\text{nl}} p'$. The induction hypothesis follows.

$\supseteq$ Show by induction over the path length that for all paths $p' := c'_0 \xrightarrow{g'_1} c'_1 \cdots \xrightarrow{g'_n} c'_n \in paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ a path $p := c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths_P([\![root]\!])$ exists with $\forall 0 \leq i \leq n : c_i \in [\![\text{cs}(c'_i)]\!]$.

Basis Let $c'_0 \in paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ a path of length 0. From definition of paths, $c'_0 \in I$. The definition of I gives us $\text{cs}(c'_0) = root$ and $\exists c_0 \in [\![root]\!] : \text{ds}(c_0) = \text{ds}(c'_0)$. By definition, $c_0 \in paths_P([\![root]\!])$ and $c_0 =_{\text{nl}} c'_0$. The induction hypothesis follows.

Step Let $p' := c'_0 \xrightarrow{g'_1} c'_1 \dots c'_{i-1} \xrightarrow{g'_i} c'_i \in paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ be a path of length i. By definition, $p'_{\text{sub}} := c'_0 \xrightarrow{g'_1} c'_1 \dots c'_{i-1} \in paths_{\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})}(I)$ is a path of length $i-1$. By induction, a path $p_{\text{sub}} := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \in paths_P([\![root]\!])$ exists with $\forall 0 \leq j \leq i-1 : c_j \in [\![\text{cs}(c'_j)]\!]$ and $p'_{\text{sub}} =_{\text{nl}} p_{\text{sub}}$. Since $c'_{i-1} \xrightarrow{g'_i} c'_i$, we infer from the requirements on the transition relation $\rightarrow$ that $\exists op \in Ops : g'_i = (\text{cs}(c'_{i-1}), op, \text{cs}(c'_i))$. From the construction of the transformed program, we know there exists $(\text{cs}(c'_{i-1}), (l_p, op, l_s), \text{cs}(c'_i)) \in G_{\text{ARG}}$, $(l_p, op, l_s) \in G_{\text{CFA}}$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is strongly well-formed, we know that $\text{cs}(c'_{i-1}) = ((\cdot, e_1^{i-1}), \cdot)$ with $\text{acs}(e_1^{i-1}) \in \mathcal{L}$ and $\text{cs}(c'_i) = ((\cdot, e_1^i), \cdot)$ with $\text{acs}(e_1^i) \in \mathcal{L}$. From $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ being well-constructed ((strongly) well-formed), we know that there exists $(\text{cs}(c'_{i-1}), (l_p, op, l_s), e_t) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$. Due to soundness ((strong) well-formedness) of $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, we get $e_t \sqsubseteq \text{cs}(c'_i)$. Let $e_t = ((e_2^t, e_1^t), q^t)$. From the definition of the transfer relations $\rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ and $\rightsquigarrow_{\mathbb{L}}$ and $\text{acs}(e_{i-1}^1) \in \mathcal{L}$, we infer that $\text{acs}(e_1^{i-1}) = l_p$ and $\text{acs}(e_1^t) = l_s$. From the definition of $\sqsubseteq$, $e_t \sqsubseteq \text{cs}(c'_i)$ and $\text{acs}(e_1^i) \in \mathcal{L}$, we know that $\text{acs}(e_1^i) = l_s$. From the definition of concretization, $\text{acs}(e_1^{i-1}) = l_p$, and $c_{i-1} \in \text{cs}(c'_{i-1})$, we get $\text{cs}(c_{i-1}) = l_p$. Define $c_i = (l_s, \text{ds}(c'_i))$. The assumptions on concrete states and $c'_i \in C$, give us $c_i \in C$. From the assumptions on transition relation $\rightarrow$, $c'_{i-1} \xrightarrow{g'_i} c'_i$, $p'_{\text{sub}} =_{\text{nl}} p_{\text{sub}}$, $\text{ds}(c'_{i-1}) = \text{ds}(c_{i-1})$, $\text{cs}(c_i) = l_s$ and $\text{cs}(c_{i-1}) = l_p$, we get $c_{i-1} \xrightarrow{(l_p, op, l_s)} c_i$. From $c_{i-1} \in \text{cs}(c'_{i-1})$ and $c_{i-1} \xrightarrow{(l_s, op, l_p)} c_i$, we get $c_i \in [\![e_t]\!]$ (overapproximation

of transfer relation and transfer relation of refined property checking analysis being a function). The meaning of $\sqsubseteq$ and $e_t \sqsubseteq \text{cs}(c_i')$, let us conclude that $c_i \in [\![\text{cs}(c_i')]\!]$. We conclude that $p := c_0 \xrightarrow{g_1} c_1 \dots c_{i-1} \xrightarrow{(l_p, op, l_s)} c_i \in paths_P([\![root]\!])$ and $p' =_{\text{nl}} p$. The induction hypothesis follows.

□

Proposition 5.9. *Let $\mathbb{C}_1^{\mathcal{A}}$ be a property checking analysis and $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of $\mathbb{C}_1^{\mathcal{A}}$. Then, a CPA $\mathbb{C}$ exists such that $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ is an enhancement of CPA $\mathbb{C}$ and control state unaware property automaton $\mathcal{A}$.*

Proof. Assume $\mathbb{C}_1^{\mathcal{A}} = (D^{\mathcal{A}}, \Pi, \rightsquigarrow^{\mathcal{A}}, \mathsf{prec}^{\mathcal{A}}, \mathsf{merge}^{\mathcal{A}}, \mathsf{stop}^{\mathcal{A}})$. Then, $\mathbb{C}_1 = (D, \Pi, \rightsquigarrow_{\mathbb{C}_1}, \cdot, \cdot, \cdot)$. Define $\mathbb{C} = (D, \Pi, \rightsquigarrow_{\mathbb{C}_1}, \mathsf{prec}_{\mathbb{C}}, \mathsf{merge}_{\mathbb{C}}, \mathsf{stop}_{\mathbb{C}})$ with $\mathsf{prec}_{\mathbb{C}}(e, \pi, S) := (e, \pi)$, $\mathsf{merge}_{\mathbb{C}}(e, e') := e'$, and $\mathsf{stop}_{\mathbb{C}}(e, S) := \exists e' \in S : e \sqsubseteq e'$.

First, show that $\mathbb{C}$ is a CPA. We know that $\mathbb{C}_1$ is a CPA. By definition of $\mathbb{C}$, nothing remains to be shown for abstract domain D, set of initial precisions Π, or the transfer relation. Since $e \sqsubseteq e$, the requirements on the precision adjustment operator and the merge operator are fulfilled. It remains to be shown that the termination check operator is sound. We get $\mathsf{stop}_{\mathbb{C}}(e, S) = \mathit{true}$ implies $\exists e' \in S : e \sqsubseteq e'$. Let $e' \in S$ with $e \sqsubseteq e'$ be arbitrary. The requirements on the abstract domain let us conclude that $[\![e]\!] \subseteq [\![e']\!]$. Hence, $[\![e]\!] \subseteq [\![e']\!] \subseteq \bigcup_{e'' \in S} [\![e'']\!]$. Thus, $\mathbb{C}$ is a CPA.

Second, show that $\text{DFA}(\mathbb{C}_1^{\mathcal{A}}) = (D^{\mathcal{A}}, \Pi, \rightsquigarrow^{\mathcal{A}}, \mathsf{prec}_{\text{DFA}}, \mathsf{merge}_{\text{DFA}}, \mathsf{stop}_{\text{DFA}})$ is an enhancement of CPA $\mathbb{C}$ and property automaton $\mathcal{A}$. By definition of $\mathbb{C}$, $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, and $\mathbb{C}_1^{\mathcal{A}}$ being an enhancement of CPA $\mathbb{C}_1$ with $\mathcal{A}$, the requirements for the abstract domain, the set of precisions, and the transfer relation are met. By definition, $\mathsf{prec}_{\text{DFA}}((e, q), \pi, S) = ((e, q), \pi) = \mathsf{prec}^{\mathcal{A}}_{\mathbb{C},\text{mp}}((e, q), \pi, S)$. Moreover, we have $\mathsf{merge}_{\text{DFA}}((e, q), (e', q')) = ((e, q) \sqcup (e', q')) \sqsupseteq (e', q') = \mathsf{merge}^{\mathcal{A}}_{\mathbb{C},\text{mp}}((e, q), (e', q'))$ if $\text{acs}(e) = \text{acs}(e')$ and otherwise we have $\mathsf{merge}_{\text{DFA}}((e, q), (e', q')) = (e', q') = \mathsf{merge}^{\mathcal{A}}_{\mathbb{C},\text{mp}}((e, q), (e', q'))$. We further infer that $\mathsf{stop}^{\mathcal{A}}_{\mathbb{C},\text{mp}}((e, q), S) = \mathsf{stop}_{\mathbb{C}}(e, \{e' \mid (e', q') \in S \wedge q \sqsubseteq q'\}) = \exists e'' \in \{e' \mid (e', q') \in S \wedge q \sqsubseteq q'\} : e \sqsubseteq e'' = \exists (e'', q') \in S : (e, q) \sqsubseteq (e'', q')$. Hence, $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ is an enhancement of CPA $\mathbb{C}$ and property automaton $\mathcal{A}$. □

Lemma 5.10. *Let $\mathbb{C}_1^{\mathcal{A}}$ be a property checking analysis, and $P = (L, G_{\text{CFA}}, l_0)$ be a program. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ with $\text{acs}(e) \in L$, initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and P, then after line 1 always $\forall (e', q') \in \mathsf{reached} :$ $\text{acs}(e') \in L \wedge \neg \exists (e'', q'') \in \mathsf{reached} : (e'', q'') \neq (e', q') \wedge \text{acs}(e'') = \text{acs}(e')$.*

Proof. Show by induction over the changes of $\mathsf{reached}$ that after line 1 always $\forall (e', q') \in \mathsf{reached} : \text{acs}(e') \in L \wedge \neg \exists (e'', q'') \in \mathsf{reached} : (e'', q'') \neq (e', q') \wedge \text{acs}(e'') = \text{acs}(e')$.

Basis Algorithm 2 initializes $\mathsf{reached}$ in line 1 with a single element (e, q) and $\text{acs}(e) \in L$. The induction hypothesis follows.

Step Algorithm 2 changes $\mathsf{reached}$ only in line 12 and line 19.

If it change $\mathsf{reached}$ in line 12, it adds e_{new}. Let $e'' = (e_1'', q'')$ and $e_{\text{new}} = (e_1^n, q^n)$. From the definition of e_{new} in line 12 and the definition of $\mathsf{merge}_{\text{DFA}}$, we know that $\text{acs}(e_1^n) = \text{acs}(e_1'')$. By induction and $e'' \in \mathsf{reached}$, we get $\text{acs}(e_1^n) \in L$ and e'' is the only element in $\mathsf{reached}$ with location state $\text{acs}(e_1'')$. Since Algorithm 2 replaces e'', the only element in $\mathsf{reached}$ with location state $\text{acs}(e_1'')$, with e_{new}, only one element exists in $\mathsf{reached}$ considering location $\text{acs}(e_1'') = \text{acs}(e_1^n)$ after execution of line 12.

If Algorithm 2 changes reached in line 19, it adds e_{prec}. From definition of prec_{DFA}, we know that $e_{\text{prec}} = e'$. Let $e' = (e'_1, q')$. We know $(e, g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ and $g \in L \times Ops \times L$. Let $e = (e_1, q*)$. Since Algorithm 2 only adds $(\widehat{e}, \cdot)$ to waitlist if it adds $\widehat{e}$ in reached and it removes all $(\widehat{e}, \cdot)$ in waitlist if it removes $\widehat{e}$ from reached, we know that in line 4 $e \in$ reached. By induction, we get $\text{acs}(e_1) \in L$. From definition of transfer relation $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $g \in L \times Ops \times L$, and $\text{acs}(e_1) \in L$, we infer that $\text{acs}(e'_1) \in L$. We need to show that no element $(e'''_1, q''') \in$ reached exists with $\text{acs}(e'_1) = \text{acs}(e'''_1)$ before execution of line 19. Assume that such an element exists. Since in line 12 a state is replaced with a more abstract state which considers the same location, we infer that in line 7 a state $(e''_1, q'') \in$ reached exists with $\text{acs}(e'_1) = \text{acs}(e''_1)$. By definition of $\text{merge}_{\text{DFA}}$, e' is combined with a state e'' in line 12 and $e' \sqsubseteq \text{merge}(e', e'')$ (definition of merge). Due to the requirements on $\text{merge}_{\text{DFA}}$ (Eq. 2.5) and transitivity of partial order $\sqsubseteq$, we know that before execution of line 19 $\exists \widehat{e'} \in \text{reached} : e' \sqsubseteq \widehat{e'}$. Furthermore, at line 19 $\text{stop}_{\text{DFA}}(e', \text{reached}) = \mathit{false}$. Hence, $\neg\exists \widehat{e'} \in \text{reached} : e' \sqsubseteq \widehat{e'}$ (definition of stop_{DFA}). Contradiction to $\exists \widehat{e'} \in \text{reached} : e' \sqsubseteq \widehat{e'}$. We conclude that no such state exists.

The induction hypothesis follows.

□

Lemma 5.12. *Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ which is strongly well-formed for $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)}$. Furthermore, let $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$ be the generated program, $(e_1, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$, $n \in N$, and $g \in G_{\text{CFA}}$. If $((e_1, q), g, (e'_1, q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $\text{acs}(e_1) = n$, and $(e_1, q) \sqsubseteq n[n]$, then there exists $n' \in N$ with $\text{acs}(e'_1) = n'$ and $(e'_1, q') \sqsubseteq n'[n']$.*

Proof. Assume $((e_1, q), g, (e'_1, q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $\text{acs}(e_1) = n$ and $(e_1, q) \sqsubseteq n[n]$. By definition of the transfer relations $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ and $\rightsquigarrow_{\mathbb{L}}$, and $\text{acs}(e_1) = n \in \mathcal{L}$ (program construction and $n \in N$), we know that $g = (\text{acs}(e_1), op, \text{acs}(e'_1))$. From program construction, we get $(\text{acs}(e_1), (l_p, op, l_s), \text{acs}(e'_1)) \in G_{\text{ARG}}$ and $\text{acs}(e'_1) \in N$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is well-constructed and sound ((strongly) well-formed), we infer that $\exists (\text{acs}(e_1), (l_p, op, l_s), \widehat{e}) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} : \widehat{e} \sqsubseteq \text{acs}(e'_1)$. Let $\text{acs}(e_1) = ((e_2^n, e_1^n), q^n)$ and $\widehat{e} = ((\widehat{e_2}, \widehat{e_1}), \widehat{q})$. From definition of $\rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, we get $((e_1^n, q^n), (l_p, op, l_s), (\widehat{e_1}, \widehat{q})) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed, we know that $\text{acs}(e_1^n), \text{acs}(\widehat{e_1}) \in \mathcal{L}$. The definition of $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ and $\rightsquigarrow_{\mathbb{L}}$ let us conclude that $\text{acs}(e_1^n) = l_p$ and $\text{acs}(\widehat{e_1}) = l_s$. From definition of $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $n[n] = \text{acs}(e_1)[\text{acs}(e_1)]$, and $\widehat{e}[\text{acs}(e'_1)]$, we get $(n[n], g, \widehat{e}[\text{acs}(e'_1)]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. From $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ being a monotonic function and $(e_1, q) \sqsubseteq n[n]$, we infer that $(e'_1, q') \sqsubseteq \widehat{e}[\text{acs}(e'_1)]$. From definition of $\widehat{e}[\text{acs}(e'_1)]$, $\text{acs}(e'_1)[\text{acs}(e'_1)]$, $\widehat{e} \sqsubseteq \text{acs}(e'_1)$, and definition of $\sqsubseteq$, we conclude that $\widehat{e}[\text{acs}(e'_1)] \sqsubseteq \text{acs}(e'_1)[\text{acs}(e'_1)]$. Since partial order $\sqsubseteq$ is transitive, we get $(e'_1, q') \sqsubseteq \text{acs}(e'_1)[\text{acs}(e'_1)]$. Set $n' = \text{acs}(e'_1)$. We conclude that a $n' \in N$ exists with $\text{acs}(e'_1) = n'$ and $(e'_1, q') \sqsubseteq n'[n']$. □

Lemma 5.13. *Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ which is strongly well-formed for $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)}$. Furthermore, let $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$ the generated program. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state $e'_0 \sqsubseteq e_0[l_0]$, arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$ terminates, then it returns $(\cdot, \text{reached}, \cdot)$ with $\forall e \in \text{reached} : \exists n \in N : e \sqsubseteq n[n]$.*

Proof. Prove by induction over the changes of reached that $\forall (e_1, q) \in \mathsf{reached} : \exists n \in N : \mathrm{acs}(e_1) = n \wedge e \sqsubseteq n[n]$.

Basis Algorithm 2 initializes reached with e'_0 in line 2. From program construction, we know that $l_0 = root$ and $root \in N$ (definition of ARG). Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$ is rooted ((strongly) well-formed), we know that $e_0 \sqsubseteq root$. By definition of $\sqsubseteq$, $e_0[l_0]$, $root[l_0]$ and $e_0 \sqsubseteq root$, we infer $e_0[l_0] \sqsubseteq root[l_0]$. Since $e'_0 \sqsubseteq e_0[l_0]$ and $e_0[l_0] \sqsubseteq root[l_0]$, we get $e'_0 \sqsubseteq root[l_0] = root[root]$ The induction hypothesis follows.

Step After initialization Algorithm 2 changes reached in line 12 or line 19. Now, let $(e, g, e') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ be the transfer relation transition which is explored in the iteration of the for loop of line 6 in which reached is changed in line 12 and 19, respectively. Since Algorithm 2 only adds $(e, \cdot)$ to waitlist when it adds e in reached and it removes all $(e, \cdot)$ in waitlist when it removes e from reached, we know that in line 4 $e = (e_1, q) \in$ reached. Let $e = (e_1, q)$ and $e' = (e'_1, q')$. Hence, by induction $\exists n \in N : \mathrm{acs}(e_1) = n \wedge e \sqsubseteq n[n]$. From previous lemma, we conclude that $\exists n' \in N : \mathrm{acs}(e'_1) = n' \wedge e' \sqsubseteq n'[n']$. From definition of $\mathsf{prec}_{\mathrm{DFA}}$, we conclude that $e_{\mathrm{prec}} = e'$.

If Algorithm 2 changes reached in line 12, it replaces $e'' = (e''_1, q'') \in$ reached by $e_{\mathrm{new}} = (e^n_1, q^n)$. By induction, we know that $\exists n'' \in N : \mathrm{acs}(e''_1) = n'' \wedge e'' \sqsubseteq n''[n'']$. Since in line 12 $e'' \neq e_{\mathrm{new}}$, we know that $e_{\mathrm{new}} = e_{\mathrm{prec}} \sqcup e''$ and $\mathrm{acs}(e'_1) = \mathrm{acs}(e''_1)$ (definition of $\mathsf{merge}_{\mathrm{DFA}}$ and $e_{\mathrm{prec}} = e'$). Hence, $n' = n''$. From $\mathcal{E}_{\mathbb{C}_1^\mathcal{A}}$ being a join-semilattice, we conclude that $e_{\mathrm{new}} = e_{\mathrm{prec}} \sqcup e'' \sqsubseteq n'[n']$. From definition of $\sqcup$ and $\mathrm{acs}(e'_1) = \mathrm{acs}(e''_1)$, we get $\mathrm{acs}(e^n_1) = \mathrm{acs}(e'_1) = n'$. The induction hypothesis follows.

If Algorithm 2 changes reached in line 19, it adds e_{prec}. Since $e_{\mathrm{prec}} = e'$, the induction hypothesis follows.

□

Lemma 5.16. *Let $P = (L, G_{\mathrm{CFA}}, l_0)$ be a program. Furthermore, let $\mathrm{DFA}(\mathbb{C}_1^\mathcal{A})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^\mathcal{A}$ and $(e, q) \in E_{\mathbb{C}_1^\mathcal{A}}$ s.t. $\mathrm{acs}(e) \in L$. If Algorithm 2 started with $\mathrm{DFA}(\mathbb{C}_1^\mathcal{A})$, initial abstract state $(e, q) \in E_{\mathbb{C}_1^\mathcal{A}}$, arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^\mathcal{A}}$, and program P, then for every element $((e', q'), \pi')$ added to waitlist during iteration of the while loop a non-empty sequence $g_1 \ldots g_n$ exists with $g_1 = (\mathrm{acs}(e), \cdot, \cdot) \wedge \forall 1 \leq i \leq n : g_i \in G_{\mathrm{CFA}} \wedge (i = n \wedge g_n = (\cdot, \cdot, \mathrm{acs}(e')) \vee g_i = (\cdot, \cdot, l') \wedge g_{i+1} = (l', \cdot, \cdot))$.*

Proof. Prove by induction over the iterations i of the while loop that for every insertion of $((e', q'), \pi')$ in waitlist in iteration i a non-empty sequence $g_1 \ldots g_n$ exists with $g_1 = (\mathrm{acs}(e), \cdot, \cdot) \wedge \forall 1 \leq i \leq n : g_i \in G_{\mathrm{CFA}} \wedge (i = n \wedge g_n = (\cdot, \cdot, \mathrm{acs}(e')) \vee g_i = (\cdot, \cdot, l') \wedge g_{i+1} = (l', \cdot, \cdot))$.

Basis Let $((e', q'), \pi')$ be an element added in an arbitrary insertion of the first iteration of the while loop. Insertion is only possible in line 11 or line 19. We know that Algorithm 2 pops initial abstract state $((e, q), \cdot)$ in line 4. We get that $e_{\mathrm{prec}} = (e_t, q_t)$ with $((e, q), (l, op, l'), (e_t, q_t)) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ (definition of operators of dataflow analysis). From the requirements on the property checking analysis and $\mathrm{acs}(e) \in L$, we conclude that $\mathrm{acs}(e) = l$ and $\mathrm{acs}(e_t) = l'$. If $((e', q'), \pi')$ is added in line 11, we conclude that $(e', q') = e_{\mathrm{new}} \neq e''$. Thus, $e_{\mathrm{new}} = e_{\mathrm{prec}} \sqcup e'' = (e_t, q_t) \sqcup e''$. The definition of the merge operator gives us that $\mathrm{acs}(e') = \mathrm{acs}(e_t) = l'$. If $((e', q'), \pi')$ is added in line 19, we get that $(e', q') = e_{\mathrm{prec}} = (e_t, q_t)$. In both cases, the sequence (l, op, l') fulfills the requirements. The induction hypothesis follows.

Step Let $((e', q'), \pi')$ be an element added in an arbitrary insertion of the ith iteration of the while loop and $i > 1$. Insertion is only possible in line 11 or line 19. Assume Algorithm 2 pops abstract state $((e_p, q_p), \cdot)$ in line 4. We infer that $((e_p, q_p), \cdot)$ must be added in iteration $j < i$. We conclude that a non-empty sequence $g_1 \dots g_n$ exists with $g_1 = (\text{acs}(e), \cdot, \cdot) \wedge \forall 1 \leq i \leq n : g_i \in G_{\text{CFA}} \wedge (i = n \wedge g_n = (\cdot, \cdot, \text{acs}(e_p)) \vee g_i = (\cdot, \cdot, l') \wedge g_{i+1} = (l', \cdot, \cdot)$. Hence, $\text{acs}(e_p) \in L$. We get that $e_{\text{prec}} = (e_t, q_t)$ with $((e_p, q_p), (l, op, l'), (e_t, q_t)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ (definition of operators of dataflow analysis). From the requirements on the property checking analysis and $\text{acs}(e_p) \in L$, we conclude that $\text{acs}(e_p) = l$ and $\text{acs}(e_t) = l'$. If $((e', q'), \pi')$ is added in line 11, we conclude that $(e', q') = e_{\text{new}} \neq e''$. Thus, $e_{\text{new}} = e_{\text{prec}} \sqcup e'' = (e_t, q_t) \sqcup e''$. The definition of the merge operator gives us that $\text{acs}(e') = \text{acs}(e_t) = l'$. If $((e', q'), \pi')$ is added in line 19, we get that $(e', q') = e_{\text{prec}} = (e_t, q_t)$. In both cases, the sequence $g_1 \dots g_n(l, op, l')$ fulfills the requirements. The induction hypothesis follows.

□

Lemma 5.17. *Let $P = (L, G_{\text{CFA}}, l_0)$ be a program. Furthermore, let $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ and $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ s.t. $\text{acs}(e) \in L$. If Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state (e, q), arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and program P and $((e', q'), \pi')$ is popped in an iteration of the while loop, then for each $l' \in L$ at most $|\{(\text{acs}(e'), \cdot, l') \in G\}|$ elements $((e'', q''), \pi'')$ with $\text{acs}(e'') = l'$ are added to* waitlist *in the same iteration.*

Proof. Consider an arbitrary iteration of the while loop. Assume this iteration popped abstract state $((e', q'), \pi')$ from waitlist. Since waitlist only contains $((e', q'), \pi')$ when reached contains (e', q'), from Lemma 5.10 we know that $\text{acs}(e') \in L$. By definition of a dataflow analysis and the transfer relation $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, we know that $((e', q'), (l', op, l''), (e'', q'')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ only if $\text{acs}(e') = l'$ and $\text{acs}(e'') = l''$. From $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ being a function, it follows that per edge at most one abstract successor exists. It remains to be shown that in each iteration of the for loop in line 6, which explores transition $((e', q'), (l', op, l''), (e'', q'')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, only one element $((e''', q'''), \pi''')$ with $\text{acs}(e''') = l''$ is added. For these iterations, we know that Algorithm 2 adds elements $((e''', q'''), \pi''')$ in line 11 or line 19. Due to the definition of the merge and precision adjustment operator, we know that $\text{acs}(e''') = l''$. From Lemma 5.10 and the definition of the merge operator, at most one element is added in line 11. Furthermore, we know that if an element $((e''', q'''), \pi''')$ is added in line 11, then $(e'', q'') \sqsubseteq (e''', q''')$ (definition of merge) and at line 18 $(e''', q''') \in$ reached. Hence, if an element (e''', q''') is added in line 11, $\text{stop}_{\text{DFA}}((e'', q''), \text{reached}) = \mathit{true}$. We conclude that Algorithm 2 either adds an element in line 11 or in line 19. □

Proposition 5.18. *Let $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be the dataflow analysis of property checking analysis $\mathbb{C}_1^{\mathcal{A}}$ and $(e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ s.t. $\text{acs}(e) \in \mathcal{L}$. If program $P = (L, G_{\text{CFA}}, l_0)$ is finite and does not contain loops, and L is finite, then Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, initial abstract state (e, q), arbitrary initial precision $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and program P terminates.*

Proof. If $\text{acs}(e) \notin L$, we know that not exists $((e, q), g, (e', q')) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ s.t. $g \in G$ (definition of $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, requirements on property checking analysis and $\text{acs}(e) \in \mathcal{L}$). The while loop terminates after one iteration because the number of control flow edges is bounded (program P is finite) and no transfer successor exists for any edge.

Assume that $\text{acs}(e) \in L$. From Lemma 5.10 we infer that the size of the reached set is always restricted by $|L|$. The for each loop in line 8 always terminates. Since $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ is a

function (definition of DFA and property checking analysis), we infer that the for loop in line 6 always terminates. Now, we conclude that the for loop in line 5 always terminates (number of control flow edges bounded, because program finite). It remains to be shown that the while loop terminates.

Define $s : L \to \mathbb{N}$ with $s(l) = |\{g_1 \dots g_n \mid g_1 = (\text{acs}(e), \cdot, \cdot) \wedge \forall 1 \leq i \leq n : g_i \in G_{\text{CFA}} \wedge (i = n \wedge g_n = (\cdot, \cdot, l) \vee g_i = (\cdot, \cdot, l') \wedge g_{i+1} = (l', \cdot, \cdot)\}|$. Since P is loop-free, function s is well-defined, $s(\text{acs}(e)) = 0$ and $(l, \cdot, l') \in G_{\text{CFA}}$ implies $s(l) \leq s(l')$.

For every $l \in L$, we prove by induction over the value of $s(l)$ and the size of the longest path from $\text{acs}(e)$ to l that after line 1 Algorithm 2 adds for each $l \in L$ at most $s(l)$ many times an element $((e', q'), \pi')$ with $\text{acs}(e') = l$ to waitlist.

Basis Let $l \in L$ be an arbitrary location with $s(l) = 0$. From Lemma 5.16 and $s(l) = 0$, we conclude that during iteration of the while loop no element $((e', q'), \pi')$ with $\text{acs}(e') = l$ is added to waitlist. Hence, after line 1 no element $((e', q'), \pi')$ with $\text{acs}(e') = l$ is added to waitlist. The induction hypothesis follows.

Step Let $l \in L$ be an arbitrary location with $s(l) = i$, $i > 0$ and size of longest path j. Since $s(l) > 0$, we get $j > 0$. From Lemma 5.17, we know that Algorithm 2 only adds an element $((e'', q''), \cdot)$ to waitlist in iteration i considering element $((e', q'), \cdot)$ if an edge $(\text{acs}(e'), \cdot, \text{acs}(e'')) \in G$ exists. By definition of s, we know that for all $l' \in L$ s.t. an edge $(l', \cdot, l) \in G$ exists that $s(l') \leq i$ and the longest path from $\text{acs}(e)$ to l' is smaller than j (exists a path $s(l) > 0$ and program loop-free). By induction, we get that at most $s(l')$ many times an element $((e'', q''), \pi'')$ with $\text{acs}(e'') = l'$ is added to waitlist. Per addition to waitlist, an element can only be removed once. Hence, from Lemma 5.17 we get that for each such l' at most $s(l') \cdot |\{(\text{acs}(e'), \cdot, l) \in G\}|$ many elements $((e'', q''), \cdot)$ with $\text{acs}(e'') = l$ are added to waitlist. We conclude that at most $\sum_{(l', \cdot, l) \in G} s(l') \cdot |\{(\text{acs}(e'), \cdot, l) \in G\}|$ many elements $((e'', q''), \cdot)$ with $\text{acs}(e'') = l$ are added to waitlist. By definition of s, we have $\sum_{(l', \cdot, l) \in G} s(l') \cdot |\{(\text{acs}(e'), \cdot, l) \in G\}| = s(l)$. The induction hypothesis follows.

From Lemma 5.10, we know that never an element (e', q') with $\text{acs}(e') \notin L$ is added to reached. Thus, no element $((e', q'), \pi')$ with $\text{acs}(e') \notin L$ is added to waitlist. From induction and the insertion (e_0, π_0) in line 1, we know that at most $s_L := 1 + \sum_{l \in L} s(l)$ many times an element is added to waitlist. Since L is finite and each $s(l) \in \mathbb{N}$, we get $s_L \in \mathbb{N}$. Only finitely many often, elements are added to waitlist. The while loop removes one element in each iteration and terminates if the waitlist is empty. Hence, the while loop terminates. □

Lemma 5.19. *Let* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$. *Let* reached_B *denote the reached set in Algorithm 2 before a change and* reached_A *that reached set after a change. If Algorithm 2 is started with* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *arbitrary initial abstract state* $e_0 \in E_{\mathbb{C}_1^{\mathcal{A}}}$ *and precision* $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* $P = (L, G_{\text{CFA}}, l_0)$, *then after initialization of* reached *in line 1 for each change of* reached *it holds that* $\text{reached}_B \widehat{\sqsubset} \text{reached}_A$ *and* $|\text{reached}_B| \leq |\text{reached}_A| \leq |L + 1|$.

Proof. Let initial abstract be $e_0 = (e_1^0, q^0)$. If $\text{acs}(e_1^0) \notin L$ and $\text{acs}(e_1^0) \neq \top_{\mathbb{L}}$, we know that $\neg\exists((e_1^0, q^0), g, \cdot) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge g \in G_{\text{CFA}}$. Algorithm 2 only adds (e_1^0, q^0) to reached in line 1.

Now, consider $\text{acs}(e_1^0) \in L$ or $\text{acs}(e_1^0) = \top_{\mathbb{L}}$. Algorithm 2 initializes reached with e_0 and we get $|\text{reached}| = 1 = |\{l \mid (\widehat{e}, q) \in \text{reached} \wedge \text{acs}(\widehat{e}) = l\}|$. After initialization, reached is

changed only in line 12 or 19. For each change, we need to show that $\mathsf{reached}_B \widehat{\sqsubset} \mathsf{reached}_A$, $|\mathsf{reached}_B| \leq |\mathsf{reached}_A|$, and moreover $|\mathsf{reached}_B| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l\}|$ $= |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$ also implies that $|\mathsf{reached}_A| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l\}| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$.

If Algorithm 2 changes $\mathsf{reached}$ in line 12, it adds e_{new}. Let $e'' = (e_1'', q'')$ and $e_{\text{new}} = (e_1^n, q^n)$. From the definition of e_{new} in line 12 and the definition of $\mathsf{merge}_{\text{DFA}}$, we know that $\mathrm{acs}(e_1^n) = \mathrm{acs}(e_1'')$ and $e'' \sqsubseteq e_{\text{new}}$. Since Algorithm 2 replaces e'' with e_{new} ($e'' \neq e_{\text{new}}$), we get $e'' \sqsubset e_{\text{new}}$ and $\mathsf{reached}_B \widehat{\sqsubset} (\mathsf{reached}_B \setminus \{e''\}) \cup \{e_{\text{new}}\} = (\mathsf{reached}_B \cup \{e_{\text{new}}\}) \setminus \{e''\} = \mathsf{reached}_A$. Since it only replaces one element considering location $\mathrm{acs}(e_1^n)$ with an element with $\mathrm{acs}(e_1^n)$, we get $|\mathsf{reached}_B| = |\mathsf{reached}_A|$ and $|\mathsf{reached}_B| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l\}| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l \in L \cup \top_{\{}\mathbb{L}\}\}|$ implies that $|\mathsf{reached}_A| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l\}| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$.

If Algorithm 2 changes $\mathsf{reached}$ in line 19, it adds e_{prec}. From definition of $\mathsf{prec}_{\text{DFA}}$, we know that $e_{\text{prec}} = e'$. Let $e' = (e_1', q')$. We know $(e, g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ and $g \in G_{\text{CFA}}$. Let $e = (e_1, q)$. Due to the definition of the transfer relation, we get $\mathrm{acs}(e_1') \in L \cup \{\top_{\mathbb{L}}\}$. Since Algorithm 2 only adds $(\widetilde{e}, \cdot)$ to $\mathsf{waitlist}$ if it adds $\widetilde{e}$ in $\mathsf{reached}$ and it removes all $(\widetilde{e}, \cdot)$ in $\mathsf{waitlist}$ if it removes $\widetilde{e}$ from $\mathsf{reached}$, we know that in line 4 $e \in \mathsf{reached}$. We need to show that no element $(e_1''', q''') \in \mathsf{reached}$ exists with $\mathrm{acs}(e_1') = \mathrm{acs}(e_1''')$ before execution of line 19. Assume that such an element exists. Since in line 12 a state is replaced with a more abstract considering the same location, we infer that a state $(e_1'', q'') \in \mathsf{reached}$ exists with $\mathrm{acs}(e_1') = \mathrm{acs}(e_1'')$ in line 7. By definition of $\mathsf{merge}_{\text{DFA}}$, e' is combined with a state e'' in line 12 and $e' \sqsubseteq \mathsf{merge}(e', e'')$ (definition of merge). Due to the requirements on $\mathsf{merge}_{\text{DFA}}$ (Eq. 2.5) and transitivity of partial order $\sqsubseteq$, we know that before execution of line 19 $\exists \widehat{e'} \in \mathsf{reached}_B : e' \sqsubseteq \widehat{e'}$. Furthermore, at line 19 $\mathsf{stop}_{\text{DFA}}(e', \mathsf{reached}_B) = \mathit{false}$. Hence, $\neg \exists \widehat{e'} \in \mathsf{reached}_B : e' \sqsubseteq \widehat{e'}$ (definition of $\mathsf{stop}_{\text{DFA}}$). Contradiction to $\exists \widehat{e'} \in \mathsf{reached}_B : e' \sqsubseteq \widehat{e'}$. We conclude that no such state exists. Algorithm 2 added an element. Thus, $|\mathsf{reached}_B| \leq |\mathsf{reached}_A|$. Since not existed state with same location and location in $L \cup \{\top_{\mathbb{L}}\}$, we get $\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l\} \setminus \{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l\} = \{\mathrm{acs}(e_1')\}$ and $|\mathsf{reached}_B| < |\mathsf{reached}_B \cup \{e'\}| = |\mathsf{reached}_A|$. Algorithm 2 added only one state. Hence, we conclude that if $|\mathsf{reached}_B| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l\}| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(\widehat{e}) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$, then $|\mathsf{reached}_A| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l\}| = |\{l \mid (\widehat{e}, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(\widehat{e}) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$. Since $\mathsf{reached}_B \sqsubseteq \mathsf{reached}_B$, we get $\mathsf{reached}_B \sqsubseteq \mathsf{reached}_B \cup \{e'\} = \mathsf{reached}_A$ It follows that $\mathsf{reached}_B \widehat{\sqsubset} \mathsf{reached}_A$.

Since at the beginning we know that $|\mathsf{reached}| = 1 = |\{l \mid (e, q) \in \mathsf{reached} \wedge \mathrm{acs}(e) = l\}|$ and line 12 and line 19 never violate this property, we get $|\mathsf{reached}_B| = |\{l \mid (e, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(e) = l\}| = |\{l \mid (e, q) \in \mathsf{reached}_B \wedge \mathrm{acs}(e) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$ and $|\mathsf{reached}_A| = |\{l \mid (e, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(e) = l\}| = |\{l \mid (e, q) \in \mathsf{reached}_A \wedge \mathrm{acs}(e) = l \in L \cup \{\top_{\mathbb{L}}\}\}|$. We get that $|\mathsf{reached}_B| \leq |L| + 1$ and $|\mathsf{reached}_A| \leq |L| + 1$. □

Proposition 5.20. *Let* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$, $P = (L, G_{\text{CFA}}, l_0)$ *a finite program, and* L *being finite. If the join-semilattice* $\mathcal{E}_{\mathbb{C}_1^{\mathcal{A}}}$ *has finite height, Algorithm 2 started with* $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *arbitrary initial abstract state* $e_0 \in E_{\mathbb{C}_1^{\mathcal{A}}}$, *arbitrary initial precision* $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* P *terminates.*

Proof. From Lemma 5.19, we infer that the size of the reached set is always restricted by $|L + 1|$. The for each loop in line 8 always terminates. Since $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ is a function (definition of DFA and property checking analysis), we infer that the for loop in line 6 always terminates. Now, we conclude that the for loop in line 5 always terminates (number of control flow edges bounded, because program finite). It remains to show that the while

loop terminates. Algorithm 2 pops one element from waitlist in each while loop iteration and only adds $(e, \cdot)$ to waitlist if it adds e to reached. Since the number of states in reached is bounded, whenever reached is updated, it becomes more abstract, and every state can become more abstract only finitely many times (Lemma 5.19 plus finite lattice height), reached is updated finally many times only. Due to the fact that Algorithm 2 only adds a single element to waitlist each time and only finitely many insertions exist, waitlist will become empty after finitely many iterations. Algorithm 2 terminates. □

Lemma 5.22. *Let* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ *be a refined property checking analysis s.t. operators* $\mathsf{merge}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(e, e') = e'$, $\mathsf{prec}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(((e_2, e_1), q), \pi, S) = ((\cdot, e_1), q), \cdot)$, *and furthermore* $\forall e \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : e \in S \implies \mathsf{stop}_{\mathbb{C}_1}(e, \pi, S)$. *If Algorithm 2 started with* $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, *compatible, initial abstract state* $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, *a precision* $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, *and program* P *returns* $(true, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, \cdot)$, *and* $G^{l_{24}}_{\text{ARG}} \subseteq G_{\text{ARG}}$ *is the subset of ARG edges added in line 24 of the CPA algorithm, then for every* $n \in N$ *a sequence* $root, ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m) = n$ *exists s.t.* $\forall 1 \leq i \leq m : \exists g_i \in G_{\text{CFA}} : (((e_2^{i-1}, e_1^{i-1}), q^{i-1}), g_i, ((e_2^i, e_1^i), q^i)) \in G^{l_{24}}_{\text{ARG}} \wedge ((e_1^{i-1}, q^{i-1}), g_i, (e_1^i, q^i)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge \neg\exists 0 \leq j \leq m : i \neq j \wedge ((e_2^i, e_1^i), q^i) = ((e_2^j, e_1^j), q^j)$.

Proof. Due to definition of $\mathsf{merge}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, Algorithm 2 never executes lines 10-17. Hence, no states are removed from reached. The requirements on the termination check operator let us conclude that an abstract state is added at most once to reached. Show by induction over the number of insertions to reached that for every $n \in$ reached there exists a sequence of abstract states $e_0, ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m) = n$ s.t. $\forall 1 \leq i \leq m : \exists g_i \in G_{\text{CFA}} : (((e_2^{i-1}, e_1^{i-1}), q^{i-1}), g_i, ((e_2^i, e_1^i), q^i)) \in G^{l_{24}}_{\text{ARG}} \wedge ((e_1^{i-1}, q^{i-1}), g_i, (e_1^i, q^i)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge \forall 0 \leq j \leq m : i \leq j \implies ((e_2^i, e_1^i), q^i)$ was added to reached before $((e_2^j, e_1^j), q^j)$.

Basis The first state which is inserted to reached is e_0. The induction hypothesis follows.

Step Let e_{prec} be the ith state added to reached and $i > 1$. We know that e_{prec} is added in line 19. Since lines 10-17 are never executed, the requirements on the termination check let us infer that an abstract state is added at most once to reached, and, thus to waitlist. We know that $(e, \cdot)$ is popped from waitlist in line 4. Thus, we conclude that no element $(e, \cdot)$ exists in waitlist. Furthermore, an element is only contained in waitlist if it is contained in reached. Together with never executing lines 10-17, we conclude that $e \in$ reached. Let $e = ((e_2, e_1), q)$ and $e_{\text{prec}} = ((e_2^{\text{prec}}, e_1^{\text{prec}}), q^{\text{prec}})$. We conclude from Algorithm 2, the definition of the precision adjustment operator, and the definition of the transfer relation that $((e_1, q), g, (e_1^{\text{prec}}, q^{\text{prec}})) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ and $e \in$ reached before e_{prec} is added. By induction, a sequence of abstract states $e_0, ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m) = e$ exists s.t. $\forall 1 \leq i \leq m : \exists g_i \in G_{\text{CFA}} : (((e_2^{i-1}, e_1^{i-1}), q^{i-1}), g_i, ((e_2^i, e_1^i), q^i)) \in G^{l_{24}}_{\text{ARG}} \wedge ((e_1^{i-1}, q^{i-1}), g_i, (e_1^i, q^i)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge \forall 0 \leq j \leq m : i \leq j \implies ((e_2^i, e_1^i), q^i)$ was added to reached before $((e_2^j, e_1^j), q^j)$. Hence, if (e, g, e_{prec}) is never removed from G_{ARG}, then there exists $e_0, ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m), ((e_2^{m+1}, e_1^{m+1}), q^{m+1}) = e_{\text{prec}}$ s.t. $\forall 1 \leq i \leq m+1 : \exists g_i \in G_{\text{CFA}} : (((e_2^{i-1}, e_1^{i-1}), q^{i-1}), g_i, ((e_2^i, e_1^i), q^i)) \in G^{l_{24}}_{\text{ARG}} \wedge ((e_1^{i-1}, q^{i-1}), g_i, (e_1^i, q^i)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge \forall 0 \leq j \leq m+1 : i \leq j \implies ((e_2^i, e_1^i), q^i)$ was added to reached before $((e_2^j, e_1^j), q^j)$. It remains to be shown that Algorithm 2 never removes (e, g, e_{prec}) from G_{ARG}. Algorithm 2 may only remove (e, g, e_{prec}) in line 20 (lines 10-17 are not executed) if $\mathsf{stop}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(e, \text{reached}) = false$. Since $e \in$ reached

if the edge is added, e is not removed from $\mathsf{reached}$ and $\mathsf{stop}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(e, \mathsf{reached}) = \mathit{true}$ after $(e, g, e_{\text{prec}})$is added, that edge is never removed.

Due to ARG construction in Algorithm 2 and the definition of a refined property checking analysis, we know that $\mathit{root} = e_0$ and at line 29 it holds true that $\mathsf{reached} = N$. Furthermore, each abstract state is added at most once to $\mathsf{reached}$. It follows that or every $n \in N$ there exists a sequence of abstract states $\mathit{root}, ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m) = n$ s.t. $\forall 1 \leq i \leq m : \exists g_i \in G_{\text{CFA}} : (((e_2^{i-1}, e_1^{i-1}), q^{i-1}), g_i, ((e_2^i, e_1^i), q^i)) \in G_{\text{ARG}}^{l_{24}} \wedge ((e_1^{i-1}, q^{i-1}), g_i, (e_1^i, q^i)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge \neg\exists 0 \leq j \leq m : i \neq j \wedge ((e_2^i, e_1^i), q^i) = ((e_2^j, e_1^j), q^j)$. □

Theorem 5.24. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a refined property checking analysis s.t. operators $\mathsf{merge}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(e, e') = e'$, $\forall e \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}, S \subseteq E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} : e \in S \implies \mathsf{stop}_{\mathbb{C}_1}(e, \pi, S)$, and $\mathsf{prec}_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}(((e_2, e_1), q), \pi, S) = (((\cdot, e_1), q), \cdot)$. If Algorithm 2 started with refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, a precision $\pi \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, and program P returns $(\mathit{true}, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$, then Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, $e_0[l_0]$, a precision $\pi' \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, and $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and using a tree ordering for $\mathsf{waitlist}$ terminates in a single pass.*

Proof. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$ and to the ordering of the locations used by Algorithm 2 while checking the generated program.

Show by induction over the iterations of the while loop in line 3 of Algorithm 2 that before each iteration i of that while loop that

1. $\forall l \in L : \mathit{to}(l) > i \vee \exists (e_1, q) \in \mathsf{reached} : \text{acs}(e_1) = l$,
2. $\forall (e_1, q) \in \mathsf{reached} : (\mathit{to}(\text{acs}(e_1)) > i \vee (e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)]) \wedge (\mathit{to}(\text{acs}(e_1)) \geq i \Leftrightarrow \exists((e_1, q), \cdot) \in \mathsf{waitlist})$,
3. $\forall((e_1, q), \pi), ((e_1', q'), \pi') \in \mathsf{waitlist} : ((e_1, q), \pi) = ((e_1', q'), \pi') \vee \text{acs}(e_1) \neq \text{acs}(e_1')$, and
4. $\forall g = (l, op, l') \in G_{\text{CFA}} : \mathit{to}(l) < i \wedge \mathit{to}(l') > i \implies \forall (l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in \mathsf{waitlist} : e' \sqsubseteq e''$.

Basis Before the first iteration, $e_0[l_0]$ is added to $\mathsf{reached}$ and $(e_0[l_0], \pi_0)$ to $\mathsf{waitlist}$. Due to ARG construction in Algorithm 2, definition of refined property checking analysis and program generation, we know that $e_0 = \mathit{root} = l_0$. From Lemma 5.22, we know that every $n \in N$ can be reached from root. The program generation lets us conclude that every location $l \in L$ is reachable from l_0. Hence, by definition of ordering $\mathit{to}(l_0) = 1$. The induction hypothesis follows.

Step Consider arbitrary iteration $i > 0$. By induction, we know that before iteration i that $\forall (e_1, q) \in \mathsf{reached} : (\mathit{to}(\text{acs}(e_1)) > i \vee (e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)]) \wedge (\mathit{to}(\text{acs}(e_1)) \geq i \Leftrightarrow \exists((e_1, q), \cdot) \in \mathsf{waitlist})$. Since $((e_1, q), \cdot)$ only in $\mathsf{waitlist}$ if $(e_1, q) \in \mathsf{reached}$ and at the beginning of each iteration i we know that $\mathsf{waitlist} \neq \emptyset$, we infer that $i \leq |L|$. Next, show the four parts of the induction hypothesis.

By induction, we know that before execution of line 4 $\forall l \in L : \mathit{to}(l) > i \vee \exists (e_1, q) \in \mathsf{reached} : \text{acs}(e_1) = l$. Algorithm 2 only removes abstract states from $\mathsf{reached}$ in line 12. In this case, it replaces $(e_1', q') = e'' \in \mathsf{reached}$ by $e_{\text{new}} = (e_1^n, q^n) \neq e''$ and $e_{\text{new}} = e_{\text{prec}} \sqcup (e_1', q')$. By definition of the merge operator, $\text{acs}(e_1') =$

$\text{acs}(e_1^n)$. Hence, before iteration $i+1$ it holds that $\forall l \in L : to(l) > i \vee \exists (e_1, q) \in \mathsf{reached} : \text{acs}(e_1) = l$.

If $i \geq |L|$, for every location $l \in L$ we have $to(l) \leq i$. In this case, we already know that $\forall l \in L : to(l) > i+1 \vee \exists (e_1, q) \in \mathsf{reached} : \text{acs}(e_1) = l$.

Assume $i < |L|$ and let l_{i+1} be the location with $to(l_{i+1}) = i+1$ (exactly one exists because to bijective). Consider two cases.

In the first case, there exists an edge $(l', op, l_{i+1}) \in G_{\text{CFA}}$ with $to(l') < i$. Due to program generation, an ARG edge $(l', (l_p, op, l_s), l_{i+1}) \in G_{\text{ARG}}$ exists. Assume $l' = ((e_2^p, e_1^p), q^p)$. From Proposition 5.2, we know that an abstract transition $(l', (l_p, op, l_s), ((e_2^t, e_1^t), q^t)) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ exists (well-constructedness) and $\text{acs}(e_1^p) \in \mathcal{L}$. From the definition of the transfer relation, we conclude that $((e_1^p, q^p), (l_p, op, l_s), (e_1^t, q^t)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $\text{acs}(e_1^p) = l_p$, and $\text{acs}(e_1^t) = l_s$. Furthermore, the requirements on the transfer relation let us also conclude that $(l'[l'], (l', op, l_{i+1}), ((e_2^t, e_1^t), q^t)[l_{i+1}]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ By induction, $((e_1', q'), \cdot) \in \mathsf{waitlist}$ exists with $((e_2^t, e_1^t), q^t)[l_{i+1}] \sqsubseteq (e_1', q')$. We infer that $(e_1', q') \in \mathsf{reached}$ and with the help of Lemma 5.10 we get $\text{acs}(e_1') = l_{i+1}$.

In the second case, not exists an edge $(l', op, l_{i+1}) \in G_{\text{CFA}}$ with $to(l') < i$. From Lemma 5.22, we know that every $n \in N$ can be reached from *root* via edges added in line 24 of the CPA algorithm. The program generation lets us conclude that every location $l \in L$ is reachable from l_0. From $to(l_{i+1}) = i+1 > 1$ and definition of ordering to, we know that $to(l_0) = 1$ and $l_{i+1} \neq l_0$. Since l_{i+1} is reachable from l_0, an edge $(l', op, l_{i+1}) \in G_{\text{CFA}}$ exists with $l' \neq l_{i+1}$ and $to(l') < to(l_{i+1})$. Now, we get that not $to(l') < i$ and $to(l') < i+1$. Hence, $to(l') = i$. By induction, we know that before line 4, an element $(e_1^i, q^i) \in \mathsf{reached}$ exists with $to(\text{acs}(e_1^i)) = i$ and $\text{acs}(e_1^i) = l'$ (to bijective). Furthermore, $((e_1^i, q^i), \pi^i) \in \mathsf{waitlist}$ exists and it is the only element for location $\text{acs}(e_1^i)$ in $\mathsf{waitlist}$. Additionally, not exists $((e_1', q'), \pi') \in \mathsf{waitlist}$ with $to(\text{acs}(e_1')) < i$. Since Algorithm 2 uses a tree ordering and the ordering to is injective, we conclude in line 4 element $((e_1^i, q^i), \pi^i)$ is removed and $to(\text{acs}(e_1^i)) = i$. By induction, we get $(e_1^i, q^i) = \text{acs}(e_1^i)[\text{acs}(e_1^i)] = l'[l']$. Due to program generation, an ARG edge $(l', (l_p, op, l_s), l_{i+1}) \in G_{\text{ARG}}$ exists. Assume $l' = ((e_2^p, e_1^p), q^p)$. From Proposition 5.2, we know that there exists $(l', (l_p, op, l_s), ((e_2^t, e_1^t), q^t)) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ (well-constructedness) and $\text{acs}(e_1^p) \in \mathcal{L}$. From the definition of the transfer relation, we conclude that $((e_1^p, q^p), (l_p, op, l_s), (e_1^t, q^t)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $\text{acs}(e_1^p) = l_p$, and $\text{acs}(e_1^t) = l_s$. Due to the requirements on the transfer relation and $l'[l'] = (e_1^i, q^i)$, we get that $((e_1^i, q^i), (l', op, l_{i+1}), ((e_2^t, e_1^t), q^t)[l_{i+1}]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ Consider the iteration of the for loop in line 6 which considers this transfer relation element. Due to the definition of the precision adjustment, either (e_1^i, q^i) is added to $\mathsf{reached}$ in line 19 or $\mathsf{stop}((e_1^i, q^i), \mathsf{reached}) = \mathit{true}$. In the first case, $(e_1', q') = ((e_2^t, e_1^t), q^t)[l_{i+1}] \in \mathsf{reached}$ exists with $\text{acs}(e_1') = l_{i+1}$. In the second case, we know that $(e_1', q') \in \mathsf{reached}$ exists with $((e_2^t, e_1^t), q^t)[l_{i+1}] \sqsubseteq (e_1', q')$. Due to partial order and Lemma 5.10, we conclude that $\text{acs}(e_1') = l_{i+1}$.

Show that if there exists a point in time during the execution of the ith iteration of the while loop s.t. $(e_1', q') \in \mathsf{reached}$ with $\text{acs}(e_1') = l_{i+1}$, then this property is never violated by the remaining execution of iteration i. An element (e_1', q') can only be removed from $\mathsf{reached}$ in line 12. In this case, Algorithm 2 replaces $(e_1', q') = e'' \in \mathsf{reached}$ by $e_{\text{new}} = (e_1^n, q^n) \neq e''$ and $e_{\text{new}} = e_{\text{prec}} \sqcup (e_1', q')$. By definition of the merge operator, $\text{acs}(e_1') = \text{acs}(e_1^n)$. Thus, before execution of

iteration $i+1$ we know that $\forall l \in L : to(l) > i \vee \exists (e_1, q) \in \mathsf{reached} : \mathrm{acs}(e_1) = l$. We conclude that before iteration $i+1$ we have that $\forall l \in L : to(l) > i+1 \vee \exists (e_1, q) \in \mathsf{reached} : \mathrm{acs}(e_1) = l$.

By induction, we know that before execution of line 4 that $\forall (e_1, q) \in \mathsf{reached} : (to(\mathrm{acs}(e_1)) > i \vee (e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)]) \wedge (to(\mathrm{acs}(e_1)) \geq i \Leftrightarrow \exists ((e_1, q), \cdot) \in \mathsf{waitlist})$. Consider the removal in line 4. By induction, we know that before line 4, an element $(e_1^i, q^i) \in \mathsf{reached}$ exists with $to(\mathrm{acs}(e_1^i)) = i$ (to bijective, $i \leq |L|$). Furthermore, $((e_1^i, q^i), \pi^i) \in \mathsf{waitlist}$ exists and it is the only element for location $\mathrm{acs}(e_1^i)$ in $\mathsf{waitlist}$. Additionally, not exists $((e_1', q'), \pi') \in \mathsf{waitlist}$ with $to(\mathrm{acs}(e_1')) < i$ nor $to(\mathrm{acs}(e_1')) = i$ and $\mathrm{acs}(e_1') \neq \mathrm{acs}(e_1^i)$. Hence, in line 4 of iteration i of the while loop Algorithm 2 pops an element $((e_1^p, q^p), \pi^p)$ from $\mathsf{waitlist}$ with $\mathrm{acs}(e_1^p) = i$. After execution of line 4, we get $\forall (e_1, q) \in \mathsf{reached} : (to(\mathrm{acs}(e_1)) > i \vee (e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)]) \wedge (to(\mathrm{acs}(e_1)) \geq i+1 \Leftrightarrow \exists ((e_1, q), \cdot) \in \mathsf{waitlist})$.

First, show that the property $\forall (e_1, q) \in \mathsf{reached} : (to(\mathrm{acs}(e_1)) > i \vee (e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)])$ is never violated. A state (e_1, q) can only be removed from $\mathsf{reached}$ in line 12. Proof by contradiction that if (e_1, q) is removed from $\mathsf{reached}$ in line 12, then $\mathrm{acs}(e_1) > i$ or $(e_1, q) \neq \mathrm{acs}(e_1)[\mathrm{acs}(e_1)]$. Assume Algorithm 2 removes a state $(e_1, q) = e''$ in line 12 with $\mathrm{acs}(e_1) \leq i$ and $(e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)]$. Then, it adds a state (e_1^n, q^n) with $(e_1, q) \sqsubset (e_1^n, q^n)$. Let $\mathsf{reached}_f$ be the reached set returned by Algorithm 2 after checking the generated program. Due to transitivity of partial order $\sqsubseteq$, we know that (e_1^f, q^f) exists with $(e_1, q) \sqsubset (e_1^n, q^n) \sqsubseteq (e_1^f, q^f)$. Let $(e_1^f, q^f) \in \mathsf{reached}_f$ with $(e_1^n, q^n) \sqsubseteq (e_1^f, q^f)$ be arbitrary. Furthermore, from Proposition 5.2 and Lemma 5.13, we know that $n \in N$ exists with $(e_1^f, q^f) \sqsubseteq n[n]$. Let $n \in N$ with $(e_1^f, q^f) \sqsubseteq n[n]$ be arbitrary. From transitivity of partial order $\sqsubseteq$, we infer $(e_1, q) \sqsubset n[n]$. From definition of $\sqsubseteq$, we conclude that $n = \mathrm{acs}(e_1)$ and $(e_1, q) \neq n[n] = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)]$. Due to a contradiction, we get that (e_1, q) is removed from $\mathsf{reached}$ in line 12 only if $\mathrm{acs}(e_1) > i$ or $(e_1, q) \neq \mathrm{acs}(e_1)[\mathrm{acs}(e_1)]$. By induction, we know that for all $l \in L$ with $to(l) \leq i$ there exists $(e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)] = l[l]$ in $\mathsf{reached}$. We showed that these states are not deleted. During iteration i, it always holds that for all $l \in L$ with $to(l) \leq i$ there exists $(e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)] = l[l]$ in $\mathsf{reached}$. With the help of Lemma 5.10, we conclude that no state (e_1', q') with $\mathrm{acs}(e_1') \notin L$ or $to(\mathrm{acs}(e_1')) \leq i$ and $(e_1', q') \neq \mathrm{acs}(e_1')[\mathrm{acs}(e_1')]$ can be added to $\mathsf{reached}$. The property is never violated.

Second, show that $\forall (e_1, q) \in \mathsf{reached} : (to(\mathrm{acs}(e_1)) \geq i+1 \Leftrightarrow \exists ((e_1, q), \cdot) \in \mathsf{waitlist})$ is never violated. After execution of line 4 in iteration i elements $((e_1', q'), \pi')$ are only removed from $\mathsf{waitlist}$ if (e_1', q') is removed from $\mathsf{reached}$ and (e_1', q') is added to $\mathsf{reached}$ only if an element $((e_1', q'), \cdot)$ is added to $\mathsf{waitlist}$. It remains to be shown that if a state (e_1, q) is added to $\mathsf{reached}$ in iteration i, then $to(\mathrm{acs}(e_1)) \geq i+1$.

Consider insertion in line 19. With the help of Lemma 5.10, we conclude that elements (e_1, q) are only added if $\mathrm{acs}(e_1) \in L$ and either not exists $(e_1', q') \in \mathsf{reached}$ with $\mathrm{acs}(e_1) = \mathrm{acs}(e_1')$ or $\forall (e_1', q') \in \mathsf{reached} : \mathrm{acs}(e_1') = \mathrm{acs}(e_1) \implies (e_1, q) = (e_1', q)$. We already showed that during iteration i always for all $l \in L$ with $to(l) \leq i$ there exists $(e_1, q) = \mathrm{acs}(e_1)[\mathrm{acs}(e_1)] = l[l]$ in $\mathsf{reached}$. If not exists $(e_1', q') \in \mathsf{reached}$ with $\mathrm{acs}(e_1) = \mathrm{acs}(e_1')$, we get $to(\mathrm{acs}(e_1)) \geq i+1$.

Consider case $\forall(e_1', q') \in \mathsf{reached} : \text{acs}(e_1') = \text{acs}(e_1) \implies (e_1, q) = (e_1', q)$. Due to the definition of the termination check, we get $\mathsf{stop}((e_1, q), \mathsf{reached}) = \mathit{true}$. We conclude that (e_1, q) is not added in line 19.

Now consider insertions of elements (e_1, q) in line 12. If $to(\text{acs}(e_1)) \geq i+1$, nothing must be shown. Assume $to(\text{acs}(e_1)) < i+1$ or $\text{acs}(e_1) \notin L$. We know that $(e_1, q) = e_{\text{new}} \neq e''$ and at line 9 $e'' \in \mathsf{reached}$. Let $e'' = (e_1'', q'')$. Due to the definition of the merge operator and Lemma 5.10, we have $\text{acs}(e_1) = \text{acs}(e_1'') \in L$ and $to(\text{acs}(e_1)) < i+1$. We already proved that during iteration i it always holds that for all $l \in L$ with $to(l) < i+1$ there exists $(e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)] = l[l]$ in $\mathsf{reached}$ With the help of Lemma 5.10, we know that $e'' = \text{acs}(e_1'')[\text{acs}(e_1'')] = \text{acs}(e_1)[\text{acs}(e_1)] = (e_1, q)$. Contradiction to $e'' \neq (e_1, q)$. In line 12, (e_1, q) is not added if $to(\text{acs}(e_1)) < i+1$ or $\text{acs}(e_1) \notin L$. The property is never violated.

Before iteration $i+1$ of the while loop, it yields that $\forall(e_1, q) \in \mathsf{reached} : (to(\text{acs}(e_1)) > i \vee (e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)]) \wedge (to(\text{acs}(e_1)) \geq i+1 \iff \exists((e_1, q), \cdot) \in \mathsf{waitlist})$. It remains to be shown that before iteration $i+1$ of the while loop it is true that $\forall(e_1, q) \in \mathsf{reached} : to(\text{acs}(e_1)) = i+1 \implies (e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)]$.

If $i+1 > |L|$ nothing must be shown. Consider case $i+1 \leq |L|$. Since to is bijective, $l_{i+1} \in L$ exists with $to(l_{i+1}) = i+1$.

Due to program construction, we know that $l_{i+1} \in N$. From Lemma 5.22, we know that every $n \in N$ can be reached from $root$ via ARG edges added in line 24. The program generation lets us conclude that every location $l \in L$ is reachable from l_0. From $to(l_{i+1}) = i+1 > 1$ and definition of ordering to, we know that $to(l_0) = 1$ and $l_{i+1} \neq l_0 = root$. From Lemma 5.22, we know that a sequence $root = ((e_2^0, e_1^0), q^0), ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m) = l_{i+1}$ exists s.t. $\forall 1 \leq i \leq m : \exists g_i = (l_p^i, op_i, l_s^i) \in G_{\text{CFA}} : (((e_2^{i-1}, e_1^{i-1}), q^{i-1}), g_i, ((e_2^i, e_1^i), q^i)) \in G_{\text{ARG}}^{l_{24}} \wedge ((e_1^{i-1}, q^{i-1}), g_i, (e_1^i, q^i)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \wedge \neg\exists 0 \leq j \leq m : i \neq j \wedge ((e_2^i, e_1^i), q^i) = ((e_2^j, e_1^j), q^j)$ and $m > 0$. Fix such a sequence $root = ((e_2^0, e_1^0), q^0), ((e_2^1, e_1^1), q^1), \ldots, ((e_2^m, e_1^m), q^m) = l_{i+1}$ and corresponding edges g_i. Due to program construction, we know that $(((e_2^{m-1}, e_1^{m-1}), q^{m-1}), op_m, l_{i+1}) \in G_{\text{CFA}}$, element $((e_2^{m-1}, e_1^{m-1}), q^{m-1}) \in L$, $to(((e_2^{m-1}, e_1^{m-1}), q^{m-1})) < to(l_{i+1})$. Furthermore, $((e_1^{m-1}, q^{m-1}), g_m, (e_1^m, q^m)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. Let $l' := ((e_2^{m-1}, e_1^{m-1}), q^m)$. Since the ARG $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed (Proposition 5.2), $\text{acs}(e_1^{m-i}) \in \mathcal{L}$. Due to the requirements on the property checking analysis, especially on $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, we conclude that $\text{acs}(e_1^{m-i}) = l_p^m$ and $\text{acs}(e_1^m) = l_s^m$. Moreover, we infer that $(l'[l'], (l', op_m, l_{i+1}), l_{i+1}[l_{i+1}]) = (((e_2^{m-1}, e_1^{m-1}), q^{m-1})[l'], (l', op_m, l_{i+1}), ((e_2^m, e_1^m), q^m)[l_{i+1}]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$.

First, show that a point in time during execution i exists in which $l_{i+1}[l_{i+1}]$ is contained in $\mathsf{reached}$. If $to(l') < i$, by induction before the execution of line 4, we know that $((e_1'', q''), \cdot) \in \mathsf{waitlist}$ exists with $l_{i+1}[l_{i+1}] \sqsubseteq (e_1'', q'')$. We infer that $(e_1'', q'') \in \mathsf{reached}$. Since Algorithm 2 only deletes abstract states if it replaces them by more abstract states, we infer from Lemma 5.13 that a $n \in N$ exists with $l_{i+1}[l_{i+1}] \sqsubseteq (e_1'', q'') \sqsubseteq n[n]$. From the definition of $n[n]$ and $\sqsubseteq$, we conclude that $\text{acs}(e_1'') = n$. Since $n \in \mathcal{L}$, from the definition of $\sqsubseteq$ we get that $\text{acs}(e_1'') = l_{i+1}$. Hence, $l_{i+1}[l_{i+1}] \sqsubseteq (e_1'', q'') \sqsubseteq n[n] = l_{i+1}[l_{i+1}]$ and $(e_1'', q'') = l_{i+1}[l_{i+1}]$. If $to(l') = i$, we already showed that in line 4 of itera-

tion i Algorithm 2 pops an element $((e_1^p, q^p), \pi^p)$ from waitlist with $\text{acs}(e_1^p) = i$. From $((e_1^p, q^p), \pi^p) \in$ waitlist, we know that $(e_1^p, q^p) \in$ reached. By induction and *to* being injective, we know that $(e_1^p, q^p) = \text{acs}(e_1^p)[\text{acs}(e_1^p)] = l'[l']$. Hence, during iteration i Algorithm 2 explores $(l'[l'], (l', op_m, l_{i+1}), l_{i+1}[l_{i+1}])$ in some iteration of the for loop in line 6 ($e = l'[l']$). Due to the definition of the precision adjustment $e_{\text{prec}} = l_{i+1}[l_{i+1}]$. Consider two cases. In the first case, $\text{stop}(l_{i+1}[l_{i+1}], \text{reached}) = \mathit{false}$. Then, $l_{i+1}[l_{i+1}]$ is added to reached. In the second case, $\text{stop}(l_{i+1}[l_{i+1}], \text{reached}) = \mathit{true}$. By definition of the termination check, a state $(e_1^t, q^t) \in$ reached exists with $l_{i+1}[l_{i+1}] \sqsubseteq (e_1^t, q^t)$. Since Algorithm 2 only deletes abstract states if it replaces them by more abstract states, we infer from Lemma 5.13 that a $n \in N$ exists with $l_{i+1}[l_{i+1}] \sqsubseteq (e_1^t, q^t) \sqsubseteq n[n]$. From the definition of $n[n]$ and $\sqsubseteq$, we conclude that $\text{acs}(e_1^t) = n$. Since $n \in \mathcal{L}$, from the definition of $\sqsubseteq$ we get that $\text{acs}(e_1^t) = l_{i+1}$. Hence, $l_{i+1}[l_{i+1}] \sqsubseteq (e_1^t, q^t) \sqsubseteq n[n] = l_{i+1}[l_{i+1}]$ and $(e_1^t, q^t) = l_{i+1}[l_{i+1}]$.

Second, show that if $l_{i+1}[l_{i+1}]$ is contained in reached, it will never be removed. Algorithm 2 may only remove $l_{i+1}[l_{i+1}]$ from reached, if it replaces it by a more abstract state $(e_1^n, q^n) \sqsupset l_{i+1}[l_{i+1}]$. Since Algorithm 2 only deletes abstract states if it replaces them by more abstract states, we infer from Lemma 5.13 that a $n' \in N$ exists with $l_{i+1}[l_{i+1}] \sqsubset (e_1^n, q^n) \sqsubseteq n'[n']$. From the definition of $n'[n']$ and $\sqsubseteq$, we conclude that $\text{acs}(e_1^n) = n'$. Since $n' \in \mathcal{L}$, from the definition of $\sqsubseteq$ we get that $\text{acs}(e_1^n) = l_{i+1}$. Hence, $l_{i+1}[l_{i+1}] \sqsubset (e_1^n, q^n) \sqsubseteq n[n] = l_{i+1}[l_{i+1}]$ and $(e_1^n, q^n) = l_{i+1}[l_{i+1}]$. We conclude that $l_{i+1}[l_{i+1}]$ is never replaced by a more abstract state. If $l_{i+1}[l_{i+1}]$ is contained in reached during iteration i, it will never be removed.

From Lemma 5.10, we know that before iteration $i+1$ at most one element $(e_1, q) \in$ reached exists with $\text{acs}(e_1) = l_{i+1}$. We proved that exactly one element $(e_1, q) \in$ reached exists with $\text{acs}(e_1) = l_{i+1}$ and $(e_1, q) = l_{i+1}[l_{i+1}] \in$ reached. Since *to* injective, we conclude that $\forall (e_1, q) \in \text{reached} : to(\text{acs}(e_1)) = i+1 \implies (e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)]$.

Before iteration $i+1$ of the while loop, we know that $\forall (e_1, q) \in \text{reached} : (to(\text{acs}(e_1)) > i+1 \vee (e_1, q) = \text{acs}(e_1)[\text{acs}(e_1)]) \wedge (to(\text{acs}(e_1)) \geq i+1 \Leftrightarrow \exists ((e_1, q), \cdot) \in \text{waitlist})$.

By induction, we know that before execution of line 4 of the CPA algorithm it is valid that $\forall ((e_1, q), \pi), ((e_1', q'), \pi') \in \text{waitlist} : ((e_1, q), \pi) = ((e_1', q'), \pi') \vee \text{acs}(e_1) \neq \text{acs}(e_1')$. Show that this property is never violated when waitlist is changed during execution of iteration i. During iteration i of the while loop, Algorithm 2 changes waitlist in line 4, 11, and 19. The removal of an element in line 4 does not violate the property. Assume before execution of line 11 it is true that $\forall ((e_1, q), \pi), ((e_1', q'), \pi') \in \text{waitlist} : ((e_1, q), \pi) = ((e_1', q'), \pi') \vee \text{acs}(e_1) \neq \text{acs}(e_1')$. Due to the definition of the merge operator and $e'' \neq e_{\text{new}}$, we know that $e_{\text{new}} = e_{\text{prec}} \sqcup e''$. We conclude that $e_{\text{new}} = (e_1^n, q^n)$ with $\text{acs}(e_1^n) = \text{acs}(e_1'')$ (definition of merge operator). From Lemma 5.10, we know that $e'' = (e_1'', q'')$ and $\text{acs}(e_1'') \in L$ and e'' is the only element in reached considering that location. Hence, at most one element $(e'', \cdot) \in$ waitlist exists which refers to location $\text{acs}(e_1'')$ (waitlist contains $(e, \cdot)$ only if $e \in$ reached due to algorithm execution). Since elements $(e'', \cdot)$ are removed from waitlist, the property is not violated after the execution of line 11. Assume before execution of line 19, it is true that $\forall ((e_1, q), \pi), ((e_1', q'), \pi') \in \text{waitlist} : ((e_1, q), \pi) = ((e_1', q'), \pi') \vee \text{acs}(e_1) \neq \text{acs}(e_1')$. From Lemma 5.10, we know that at most one element per location is

contained in reached and all states consider concrete program locations. Hence, in line 19, $((e_1^{\text{prec}}, q^{\text{prec}}), \pi_{\text{prec}})$ is only added if not exists $(e_1', q') \in$ reached with $\text{acs}(e_1') = \text{acs}(e_1^{\text{prec}})$ and, thus, not exists $((e_1', q'), \pi') \in$ waitlist with $\text{acs}(e_1') = \text{acs}(e_1^{\text{prec}})$, or $(e_1^{\text{prec}}, q^{\text{prec}}) \in$ reached before execution of line 18. The property is not violated. It follows that before execution of iteration $i+1$ of the while it holds that $\forall((e_1, q), \pi), ((e_1', q'), \pi') \in$ waitlist $: ((e_1, q), \pi) = ((e_1', q'), \pi') \vee \text{acs}(e_1) \neq \text{acs}(e_1')$.

By induction, we know that before execution of line 4 it yields that $\forall g = (l, op, l') \in G_{\text{CFA}} : to(l) < i \wedge to(l') > i \implies \forall(l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in$ waitlist $: e' \sqsubseteq e''$. Hence, also $\forall g = (l, op, l') \in G_{\text{CFA}} : to(l) < i \wedge to(l') > i+1 \implies \forall(l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in$ waitlist $: e' \sqsubseteq e''$. This property can only be violated if an element is removed from waitlist, possible only in lines 4 or 11.

Consider the removal in line 4. By induction, we know that before line 4, an element $(e_1^i, q^i) \in$ reached exists with $to(\text{acs}(e_1^i)) = i$ (to bijective, $i \leq |L|$). Furthermore, $((e_1^i, q^1), \pi^i) \in$ waitlist exists and it is the only element for location $\text{acs}(e_1^i)$ in waitlist. Additionally, not exists $((e_1', q'), \pi') \in$ waitlist with $to(\text{acs}(e_1')) < i$. Hence, in line 4 of iteration i of the while loop Algorithm 2 pops an element $((e_1^p, q^p), \pi^p)$ from waitlist with $\text{acs}(e_1^p) = i$. After line 4, it still holds that $\forall(l, op, l') \in G_{\text{CFA}} : to(l) < i \wedge to(l') > i+1 \implies \forall(l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in$ waitlist $: e' \sqsubseteq e''$.

If (e'', π) is removed from waitlist in line 11, $(e_{\text{new}}, \pi_{\text{prec}})$ is added ($e'' \neq e_{\text{new}}$). From the requirements on the merge operator, we conclude that $e'' \sqsubseteq e_{\text{new}}$. Due to transitivity of partial order $\sqsubseteq$ and addition of add $(e_{\text{new}}, \pi_{\text{prec}})$ to waitlist, deleting a state from waitlist in line 11 does not violate the property. It follows that before execution of iteration $i+1$ of the while it holds that $\forall g = (l, op, l') \in G_{\text{CFA}} : to(l) < i \wedge to(l') > i+1 \implies \forall(l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in$ waitlist $: e' \sqsubseteq e''$.

It remains to be shown that before execution of iteration $i+1$ it yields that $\forall g = (l, op, l') \in G_{\text{CFA}} : to(l) = i \wedge to(l') > i+1 \implies \forall(l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in$ waitlist $: e' \sqsubseteq e''$

We already showed that in line 4, Algorithm 2 pops $((e_1^p, q^p), \pi^p)$ from waitlist with $to(\text{acs}(e_1^p)) = i$. By induction and Lemma 5.10, we know that $(e_1^p, q^p) = \text{acs}(e_1^p)[\text{acs}(e_1^p)]$. Since to is injective, not exists $l \in L$ with $l \neq \text{acs}(e_1^p)$ and $to(l) = i$. We need to show that $\forall g = (\text{acs}(e_1^p), op, l') \in G_{\text{CFA}} : to(l') > i+1 \implies \forall((e_1^p, q^p), g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists(e'', \cdot) \in$ waitlist $: e' \sqsubseteq e''$. During iteration i, a state (e_1', q') with $to(\text{acs}(e_1')) > i+1$ is removed from waitlist if it is replaced by a more abstract state $e_{\text{new}} = (e_1^n, q^n) \sqsupset (e_1', q')$ (requirements on merge operator) and $\text{acs}(e_1^n) = \text{acs}(e_1')$ (definition of merge), hence some element $(e_{\text{new}}, \cdot)$ is added to waitlist. Since reached, and thus waitlist, only contain elements which only consider concrete locations (Lemma 5.10 and $(e, \cdot) \in$ waitlist only if $e \in$ reached) and partial order $\sqsubseteq$ is transitive, we only need to show that for all $g = (\text{acs}(e_1^p), op, l') \in G_{\text{CFA}}$ with $to(l') > i+1$ if $((e_1^p, q^p), g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ exists, then either already exists an element $(e'', \cdot) \in$ waitlist with $e' \sqsubseteq e''$ before iteration i or an element $(e'', \cdot)$ with $e' \sqsubseteq e''$ is added to waitlist. Let $g = (\text{acs}(e_1^p), op, l') \in G_{\text{CFA}}$ with $to(l') > i+1$ and $((e_1^p, q^p), g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ be arbitrary. Assume $e' = (e_1', q')$. Due to the definition of the transfer relation and $\text{acs}(e_1^p) \in L$, we infer that $\text{acs}(e_1') = l'$. An iteration of the for loop in line 6 exists in which Algorithm 2 explores $((e_1^p, q^p), g, e')$ ($e = (e_1^p, q^p)$). By definition of the precision

adjustment, we know that $e_{\text{prec}} = e'$. Consider two cases. In the first case, $\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathit{false}$. Algorithm 2 adds $(e_{\text{prec}}, \pi_{\text{prec}})$ to $\mathsf{waitlist}$. Due to reflexivity of partial order $\sqsubseteq$, we know $e_{\text{prec}} = e' \sqsubseteq e'$. In the second case, $\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathsf{stop}(e', \mathsf{reached}) = \mathit{true}$. By definition of the termination check, there exists $e'' \in \mathsf{reached}$ with $e' \sqsubseteq e''$. If e'' is added to $\mathsf{reached}$ in iteration i, possible only in lines 12 or 19, then there exists $(e'', \cdot) \in \mathsf{waitlist}$ (after line 4 $(e'', \cdot)$ only removed from $\mathsf{waitlist}$ if e'' removed from $\mathsf{reached}$). Now, consider that $e'' = (e_1'', q'')$ is added to $\mathsf{reached}$ only before iteration i. From Lemma 5.10, we know that $\text{acs}(e_1'') \in L$. From $(e_1', q') = e' \sqsubseteq e''$ and $\text{acs}(e_1') = l'$, we conclude that $\text{acs}(e_1'') = l'$. By induction and $to(l') > i + 1$, we know that before iteration i there exists $(e'', \cdot) \in \mathsf{waitlist}$. We already proved that as long as e'' is not removed from $\mathsf{reached}$ during iteration i we can guarantee that $(e'', \cdot) \in \mathsf{waitlist}$..

It follows that before execution of iteration $i + 1$ of the while it holds that $\forall g = (l, op, l') \in G_{\text{CFA}} : to(l) < i + 1 \wedge to(l') > i + 1 \implies \forall (l[l], g, e') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} \exists (e'', \cdot) \in \mathsf{waitlist} : e' \sqsubseteq e''$.

The induction hypothesis is valid before the $i + 1$ execution of the while loop.

From Lemma 5.19, we infer that the size of the reached set is always restricted by $|L+1|$. The for each loop in line 8 always terminates. Since $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ is a function (definition of DFA and property checking analysis), we infer that the for loop in line 6 always terminates. From Proposition 5.6, we know that $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$ is finite. Now, we conclude that the for loop in line 5 always terminates (number of control flow edges bounded, because program is finite). From the induction hypothesis, $(e, \pi) \in \mathsf{waitlist}$ only if $e \in \mathsf{reached}$, and the definition of the ordering to, we know that $((e, q), \pi) \in \mathsf{waitlist}$ implies $(e, q) \in \mathsf{reached}$, $\text{acs}(e) \in L$ and $to(\text{acs}(e)) \leq |L|$. The program generation and N being finite due to ARG property, let us conclude that L is finite. The induction hypothesis gives us that after $|L|$ iterations the $\mathsf{waitlist}$ must be empty. We conclude that the loop body of the while loop in line 3 can be executed at most $|L|$ (finitely many) times. Algorithm 2 terminates.

From the induction hypothesis, we know that before the execution of line 4 in iteration i, $1 \leq i \leq |L|$, there exists $(e_1, q) \in \mathsf{reached}$ with $to(\text{acs}(e_1) = i$. From the induction hypothesis, it then follows that $\exists((e_1, q), \cdot) \in \mathsf{waitlist}$. Since not exists $((e_1', q'), \cdot) \in \mathsf{waitlist}$ with $to(e_1') < i$, in iteration i, $1 \leq i \leq |L|$, Algorithm 2 pops an element $((e_1^i, q^i), \cdot)$ from $\mathsf{waitlist}$ with $to(\text{acs}(e_1^i)) = i$. Due to at most $|L|$ while loop iterations, one abstract state per program location and no abstract states which do not consider concrete program locations are inspected. The transfer relation is a function and adheres to the control flow of the program, i.e., if an abstract state with concrete location l is explored, only transfer successors for CFA edges $(l, \cdot, \cdot)$ exist and for each of those edges at most one transfer successor exists. Since for any location l at most one abstract state which considers that location l and no abstract states considering location information $\top_{\mathbb{L}}$ is explored, one transfer successor per CFA edge is explored. Algorithm 2 terminates in a single pass. □

Lemma 5.25. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalence relation consistent, refined property checking analysis and $\sim$ an equivalence relation which shows that $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is equivalence relation consistent. If Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, arbitrary precision $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and program P, then after line 1 it always holds true that $\forall e, e' \in \mathsf{reached} : e = e' \vee e \not\sim e'$.*

Proof. Show by induction over the changes of $\mathsf{reached}$ that $\forall e, e' \in \mathsf{reached} : e = e' \vee e \not\sim e'$.

Basis In line 1, reached is initialized with a single abstract state e_0. The induction hypothesis follows.

Step After initialization of reached in line 1, reached is changed in line 12 or line 19.

If reached is changed in line 12, we know that $e_{\text{new}} \neq e''$, $e'' \in$ reached before execution of line 12, e'' is removed from reached, e_{new} is added to reached and $e_{\text{new}} = \mathsf{merge}(e', e'')$. The requirements on the merge operator give us, $e'' \sim e_{\text{new}}$. By induction, we know that e'' was the only element in reached from equivalence class $[e_{\text{new}}]_\sim$. Since the only element from equivalence class $[e_{\text{new}}]_\sim$ in reached is deleted, after adding e_{new}, the induction hypothesis follows.

If reached is changed in line 19, then $\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathit{false}$ and Algorithm 2 adds e_{prec} to reached. We need to show that before line 19, not exists $e_r \in$ reached with $e_r \sim e_{\text{prec}}$. Prove by contradiction. Assume there exists $e_r \in$ reached with $e_r \sim e_{\text{prec}}$. From $\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathit{false}$, we conclude that $e_{\text{prec}} \not\sqsubseteq e_r$. Since in line 12 abstract states e_{new} are only added, if a state e'' in reached exists which is in the same equivalence class, before the execution of the for loop in line 6 an element e_t with $e_t \sim e_{\text{prec}}$ existed in reached. During the execution of that for loop, e_t may only be replaced by more abstract states of the same equivalence class (definition of merge). Hence, we know that a state e'' exists with $e'' \sim e_{\text{prec}}$ and $e_{\text{new}} = e_{\text{prec}} \sqcup e''$. We get that $e_{\text{prec}} \sqsubseteq e_{\text{new}}$ and $e_{\text{prec}} \sim e_{\text{new}}$ (definition of merge). Furthermore, before execution of line 18, there exists $e'_r \in$ reached with $e_{\text{prec}} \sim e'_r$ and $e_{\text{prec}} \sqsubseteq e'_r$. Thus, $\mathsf{stop}(e_{\text{prec}}, \mathsf{reached}) = \mathit{true}$ (contradiction). The induction hypothesis follows.

□

Lemma 5.27. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalent consistent, refined property checking analysis and $\sim$ an equivalence relation which shows that $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ is equivalence relation consistent. If Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, a precision $\pi \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and program P returns $(\cdot, \cdot, (N, G_{\text{ARG}}, root, N_{\text{cov}}))$, then $\forall (n, g, n') \in G_{\text{ARG}} : \forall (n, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} : e' \sim n'$.*

Proof. Show by induction over the changes to G_{ARG} that for every $(n, g, n') \in G_{\text{ARG}}$ there exists $(n, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ with $n \sim e'$.

Basis The set G_{ARG} is initialized in line 2 with the empty set. The induction hypothesis follows.

Step After initialization, the set G_{ARG} is changed in lines 13, 20, 24, and 28.

Assume the set G_{ARG} is changed in line 13. Deletion of edges does not violate the property. Consider addition of edge (e_p, g, e_{new}). We know that an edge $(e_p, g, e'') \in G_{\text{ARG}}$ exists before the execution of line 13. By induction hypothesis, we know an abstract transition $(e_p, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ exists with $e' \sim e''$. Due to the definition of the merge operator, we know that $e_{\text{new}} \sim e''$. From $e_{\text{new}} \sim e''$ and $e' \sim e''$, we get $e_{\text{new}} \sim e'$. The induction hypothesis follows.

Assume the set G_{ARG} changed in line 20. Deletion of edges does not violate the property. After deletion of edges in line 20, the induction hypothesis follows.

Assume the set G_{ARG} is changed in line 24. From the definition of e_{prec} in line 7 and the requirements on the precision adjustment, we know $e' = e_{\text{prec}}$. Hence, if

(e, g, e_{prec}) is added in line 24, an abstract transition $(e, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ exists with $e' \sim e_{\text{prec}}$. The induction hypothesis follows.

Assume the set G_{ARG} is changed in line 28. Due to the definition of coveringSet in line 26 and the definition of the termination check, we have $e_r \in$ coveringSet implies $e_{\text{prec}} \sim e_r$. From the definition of e_{prec} in line 7 and the requirements on the precision adjustment, we know $e' = e_{\text{prec}}$ and thus $e' \sim e_r$. Hence, if (e, g, e_r) is added in line 28, an abstract transition $(e, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ exists with $e' \sim e_r$. The induction hypothesis follows.

Since the transfer relation $\rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is a function, and in line 29 the set G_{ARG} is equivalent with the returned set, we get that $\forall (n, g, n') \in G_{\text{ARG}} : \forall (n, g, e') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$: $e' \sim n'$. □

Lemma 5.28. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be an equivalence relation consistent, refined property checking analysis, $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ be a dataflow analysis for property checking analysis $\mathbb{C}_1^{\mathcal{A}}$, $e_p = ((e_2^p, e_1^p), q^p) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and $e_c = (e_1^c, q^c) \in E_{\mathbb{C}_1^{\mathcal{A}}}$. Furthermore, assume that Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, arbitrary precision $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, and program $P = (L^p, G_{\text{CFA}}^p, l_0^p)$ returns $(true, \cdot, R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$. Now, let ARG $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ and $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}) = (L, G_{\text{CFA}}, l_0)$ be the generated program. If $\text{acs}(e_1^c) \in L$, $\text{acs}(e_1^p) \neq \bot_{\mathbb{L}}$, and $e_p =_\sim e_c$, then there exists a bijective function $bt : \{(e_p, g, e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} | \ g \in G_{\text{CFA}}^p\} \to \{(e_c, g', e_c') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} | \ g' \in G_{\text{CFA}}\}$ with $bt((e_p, g, e_p')) = (e_c, g', e_c') \implies e_p' =_\sim e_c'$.*

Proof. Let $\text{acs}(e_1^c) = ((e_2^l, e_1^l), q^l)$. From $e_p =_\sim e_c$ and $\text{acs}(e_1^c) \in L$, we conclude that $(e_1^p, q^p) \sqsubseteq_{\mathbb{C}_1^{\mathcal{A}}} (e_1^l, q^l)$ and due to program construction $\text{acs}(e_1^c) \in N$. Proposition 5.2, $\text{acs}(e_1^c) \in N$, $\text{acs}(e_1^p) \neq \bot_{\mathbb{L}}$, and $(e_1^p, q^p) \sqsubseteq_{\mathbb{C}_1^{\mathcal{A}}} (e_1^l, q^l)$ let us infer that $\text{acs}(e_1^p) \in \mathcal{L}$.

First, we show that if $(e_p, (l_p, op, l_s), e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ and $(l_p, op, l_s) \in G_{\text{CFA}}^p$, there exists $(e_c, (l_p', op, l_s'), e_c') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ with $(l_p', op, l_s') \in G_{\text{CFA}}$ and $e_p' =_\sim e_c'$, and not exists $(e_p, (l_p'', op, l_s''), e_p'') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ with $e_p'' =_\sim e_c'$, $(l_p'', op, l_s'') \in G_{\text{CFA}}^p$, and $(e_p, (l_p, op, l_s), e_p') \neq (e_p, (l_p'', op, l_s''), e_p'')$.

Let $(e_p, (l_p, op, l_s), e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, $(l_p, op, l_s) \in G_{\text{CFA}}^p$, and $e_p' = ((e_2^{'p}, e_1^{'p}), q^{'p})$. Due to monotonicity of transfer relation $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, $(e_1^p, q^p) \sqsubseteq_{\mathbb{C}_1^{\mathcal{A}}} (e_1^l, q^l)$, equivalence relation consistency of $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, and $e_p \sim \text{acs}(e_1^c)$ $(e_p =_\sim e_c)$, an abstract transition $(\text{acs}(e_1^c), (l_p, op, l_s), e_r^p) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$, $e_p' \sqsubseteq e_r^p$ exists, and $e_p' \sim e_r^p$. Furthermore, let $e_r^p = ((e_{2,r}^p, e_{1,r}^p), q_r^p)$. Then, also $(e_1^{'p}, q^{'p}) \sqsubseteq_{\mathbb{C}_1^{\mathcal{A}}} (e_{1,r}^p, q_r^p)$. Since $\text{acs}(e_1^c) \in N$ and $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is complete and sound (strongly well-formed due to Proposition 5.2), there exists $(\text{acs}(e_1^c), (l_p, op, l_s), e_t^p) \in G_{\text{ARG}}$ with $e_r^p \sqsubseteq e_t^p$. Due to Lemma 5.27 also $e_t^p \sim e_r^p \sim e_p'$. Due to program construction, $(\text{acs}(e_1^c), op, e_t^p) \in G_{\text{CFA}}$. The requirements on the transfer relation and $\text{acs}(e_1^p) \in \mathcal{L}$, let us deduce that $\text{acs}(e_1^p) = l_p$, $\text{acs}(e_1^{'p}) = l_s$, and $((e_1^p, q^p), (l_p, op, l_s)(e_1^{'p}, q^{'p})) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. Thus, $(e_p[\text{acs}(e_1^c)], (\text{acs}(e_1^c), op, e_t^p), e_p'[e_t^p]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. We conclude that $e_p' =_\sim e_p'[e_t^p]$. Define $e_c' = e_p'[e_t^p]$. Now, $(e_c, (l_p', op, l_s'), e_c') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ with $(l_p', op, l_s') = (\text{acs}(e_1^c), op, e_t^p) \in G_{\text{CFA}}$ and $e_p' =_\sim e_c'$.

Let $(e_p, (l_p'', op, l_s''), e_p'') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ with $e_p'' =_\sim e_c'$. Let $e_p'' = ((e_2^{''p}, e_1^{''p}), q^{''p})$, $e_c' = ((e_2^{'c}, e_1^{'c}), q^{'c})$. Since $\text{acs}(e_1^p) \in L^p$, the requirements on the transfer relation give us that $\text{acs}(e_1^p) = l_p'' = l_p$ and $\text{acs}(e_1^{''p}) = l_s''$. Due to construction of e_c', we know that $\text{acs}(e_1^{'c}) = e_t^p$ and $\text{acs}(e_1^{'c}) = e_t^p \in N$. Let $e_t^p = ((e_{2,t}^p, e_{1,t}^p), q_t^p)$. From $e_p'' =_\sim e_c'$, we get

$(e_1^{''p}, q^{''p}) \sqsubseteq (e_{1,t}^p, q_t^p)$. Due to Proposition 5.2, $\text{acs}(e_1^{'c}) = e_t^p \in N$, $(e_1^{''p}, q^{''p}) \sqsubseteq (e_{1,t}^p, q_t^p)$, $\text{acs}(e_1^{''p}) = l_s''$, $(e_1^{'p}, q^{'p}) \sqsubseteq (e_{1,t}^p, q_t^p)$, and $\text{acs}(e_1^{'p}) = l_s$, we get that $\text{acs}(e_{1,t}) \in \mathcal{L}$ and, thus, $l_s'' = l_s$. Since the transfer relation is a function, $(e_p, (l_p, op, l_s), e_p') = (e_p, (l_p'', op, l_s''), e_p'')$.

Second, we show that if $(e_c, (l_p, op, l_s), e_c') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ and $(l_p, op, l_s) \in G_{\text{CFA}}$, there exists $(e_p, (l_p', op, l_s'), e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$ with $e_p' =_\sim e_c'$ and $(l_p', op, l_s') \in G_{\text{CFA}}^p$, and not exists $(e_c, (l_p'', op, l_s''), e_c'') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ with $e_p' =_\sim e_c''$, $(l_p'', op, l_s'') \in G_{\text{CFA}}$, and $(e_c, (l_p, op, l_s), e_c) \neq (e_c, (l_p'', op, l_s''), e_c'')$.

Let $(e_c, (l_p, op, l_s), e_c') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$, $(l_p, op, l_s) \in G_{\text{CFA}}$, and $e_c' = (e_1^{'c}, q^{'c})$. Due to the requirements on the transfer relation and $\text{acs}(e_1^c) \in L$, we get $\text{acs}(e_1^c) = l_p$ and $\text{acs}(e_1^{'c}) = l_s$. From program construction, we know that an ARG edge $(l_p, (l_p', op, l_s'), l_s) \in G_{\text{ARG}}$ exist, $l_p, l_s \in N$, and $(l_p', op, l_s') \in G_{\text{CFA}}^p$. From Proposition 5.2, we conclude that there exists $(l_p, (l_p', op, l_s'), l_s^r) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, $l_s^r \sqsubseteq l_s$. Additionally, from Lemma 5.27 we get $l_s^r \sim l_s$. Let $l_s = ((e_2^n, e_1^n), q^n)$. From Proposition 5.2 and $l_p = \text{acs}(e_1^c) = ((e_2^l, e_1^l), q^l)$, we know that $\text{acs}(e_1^n), \text{acs}(e_1^l) \in \mathcal{L}$. Since $(l_p, (l_p', op, l_s'), l_s^r) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, $l_s^r \sqsubseteq l_s$, and $\text{acs}(e_1^n), \text{acs}(e_1^l) \in \mathcal{L}$, we conclude from the requirements on the transfer relation that $\text{acs}(e_1^n) = l_s'$, $\text{acs}(e_1^l) = l_p'$. Since $(l_p', op, l_s') \in G_{\text{CFA}}^p$, $l_p', l_s' \in L^p$. Since $(e_1^p, q^p) \sqsubseteq (e_1^l, q^l)$ and $\text{acs}(e_1^p) \in \mathcal{L}$, we get $\text{acs}(e_1^p) = l_p'$. Since $e_p[\text{acs}(e_1^c)] = e_c$ ($e_p =_\sim e_c$), $(e_c, (l_p, op, l_s), e_c') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$, we know that there exists $((e_1^p, q^p), (l_p', op, l_s'), (e_1^{'p}, q^{'p})) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$, $\text{acs}(e_1^{'p}) = l_s'$, and $((\top_{\mathbb{C}_2}, e_1^{'p}), q^{'p})[\text{acs}(e_1^n)] = e_c'$. Due to monotonicity of $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$, definition of $\rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, $(l_p, (l_p', op, l_s'), l_s^r) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, $l_s^r \sqsubseteq l_s$, and $(e_1^p, q^p) \sqsubseteq (e_1^l, q^l)$, we know that $(e_1^{'p}, q^{'p}) \sqsubseteq (e_1^n, q^n)$. With $e_p \sim l_p$ ($e_p =_\sim e_c$ and $\text{acs}(e_1^c) = l_p$) and $(l_p, (l_p', op, l_s'), l_s^r) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, we conclude from equivalence relation consistency (requirement on transfer relations) that there exists $(e_p, (l_p', op, l_s'), e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$ with $e_p' = ((\cdot, e_1^{'p}), q^{'p})$ and $e_p' \sim l_s^r \sim l_s$. Thus, $e_p'[\text{acs}(e_1^{'c})] = e_c'$. We conclude that $e_p' =_\sim e_c'$.

Let $(e_c, (l_p'', op, l_s''), e_c'') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ with $e_p' =_\sim e_c''$. Let $e_c'' = (e_1^{''c}, q^{''c})$. Due to $\text{acs}(e_1^c) \in L$ and the requirements on the transfer relation, we get $l_p = \text{acs}(e_1^c) = l_p''$ and $\text{acs}(e_1^{''c}) = l_s''$. Due to program construction, $l_s'' \in N$ and $(l_p, (l_p^*, op, l_s^*), l_s'') \in G_{\text{ARG}}$. Let $l_s'' = ((e_2^{n''}, e_1^{n''}), q^{n''})$. From Proposition 5.2, we get $\text{acs}(e_1^{n''}) \in L^p$ and there exists an abstract transition$(l_p, (l_p^*, op, l_s^*), l_s^{r'}) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$ and $l_s^{r'} \sqsubseteq l_s''$. Additionally, the definition of the transfer relation lets us conclude that $l_p' = \text{acs}(e_1^p) = l_p^*$ and $\text{acs}(e_1^{n''}) = l_s^*$. From $(e_1^{'p}, q^{'p}) \sqsubseteq (e_1^{n''}, q^{n''})$ ($e_p' =_\sim e_c''$) and $\text{acs}(e_1^{'p}) = l_s'$, we get $l_s' = l_s^*$ From Proposition 5.2 (determinism), we infer that $l_s = l_s''$. Since the transfer relation is a function, we get $(e_c, (l_p'', op, l_s''), e_c'') = (e_c, (l_p, op, l_s), e_c'') = (e_c, (l_p, op, l_s), e_c')$.

We conclude that a bijective function $bt : \{(e_p, g, e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} |\ g \in G_{\text{CFA}}^p\} \to \{(e_c, g', e_c') \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}} |\ g' \in G_{\text{CFA}}\}$ exists with $bt((e_p, g, e_p')) = (e_c, g', e_c') \implies e_c' =_\sim e_p'$. □

Theorem 5.29. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ be an equivalence relation consistent, refined property checking analysis. Assume Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$, compatible initial abstract state $e_0 \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, any precision $\pi_0 \in \Pi_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, and program P returns $(true, \cdot, R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}})$. Now, let $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}) = (L, G_{\text{CFA}}, l_0)$. Then, exploration orders of G_{CFA} for each while loop iteration of Algorithm 2 and a management of* waitlist *exist s.t. Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^\mathcal{A})$, $e_0[l_0]$, any precision $\pi_0' \in \Pi_{\mathbb{C}_1^\mathcal{A}}$, and $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}})$ terminates.*

Proof. Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ and $P = (L^p, G_{\text{CFA}}^p, l_0^p)$. In the following we denote the execution of Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ as the producer analysis

and the execution of Algorithm 2 started with $\mathbb{C}_1^{\mathcal{A}}$ as the consumer analysis.

Let $\mathsf{reached}_P^i, \mathsf{waitlist}_P^i$ the reached set and waitlist used by the producer before iteration i of the while loop and $\mathsf{reached}_C^i, \mathsf{waitlist}_C^i$ those of the consumer. Without the indices we refer to the respective variables reached set, waitlist used by producer and consumer in Algorithm 2.

Show by induction that there exist exploration orders of G_{CFA} for the first $i-1$ while loop iterations of Algorithm 2 and a management of $\mathsf{waitlist}_C$ s.t. before each iteration i of the producer's while loop, there exists a bijective function $b : \mathsf{reached}_P^i \to \mathsf{reached}_C^i$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P^i| = |\mathsf{waitlist}_C^i|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P^i| : \mathsf{waitlist}_P^i(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C^i(j) = (b(e_p^j), \cdot)$.

Basis Before the first iteration, we have $\mathsf{reached}_P^i = \{e_0\}$, $\mathsf{waitlist}_P^i = \{(e_0, \pi_0)\}$, $\mathsf{reached}_C^i = \{e_0[l_0]\}$, and $\mathsf{waitlist}_P^i = \{(e_0[l_0], \pi_0')\}$. Due to program construction, we know that $l_0 = \mathit{root} \in N \subseteq E_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed (Proposition 5.2), we get $e_0 \sqsubseteq \mathit{root} = l_0$ (rootedness). Let $e_0[l_0] = (e_1, q)$. Due to definition, $\text{acs}(e_1) = l_0$. From construction of *root* and requirements on merge, we know that $e_0 \sim \mathit{root}$. We conclude that $e_0 =_\sim e_0[l_0]$. The induction hypothesis follows.

Step Consider iteration $i > 0$ of the producer analysis. We know that the execution of iteration i of the while loop is only possible if $\forall j \leq i : \mathsf{waitlist}_P^i \neq \emptyset$. By induction, we know that $\forall j \leq i : \mathsf{waitlist}_C^i \neq \emptyset$. The consumer analysis executes the while loop at least i-times. Thus, the consumer analysis also has an iteration i of the while loop.

By induction, there exist exploration orders of G_{CFA} for the first $i-1$ while loop iterations of Algorithm 2 and a management of $\mathsf{waitlist}_C$ s.t. before each iteration k of the producer's while loop, there exists a bijective function $b : \mathsf{reached}_P^k \to \mathsf{reached}_C^k$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P^k| = |\mathsf{waitlist}_C^k|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P^i| : \mathsf{waitlist}_P^k(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C^k(j) = (b(e_p^j), \cdot)$. Assume the producer pops the jth element of his $\mathsf{waitlist}_P$ in line 4. Let the consumer pop his jth element of the $\mathsf{waitlist}_C$ in line 4 (exists due to same size). After execution of line 4 in producer and consumer analysis, we know that the producer analysis popped (e_p, π_p), the consumer analysis popped (e_c, π_c), and $e_p =_\sim e_c$, and there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j), \cdot)$, assuming that the consumer reordered its elements in the same way as the producer. Note this is possible in general.

Next, we need to define an exploration order for the edges in G_{CFA} for iteration i. To define an exploration order, we first map the explored transfer relation elements in an appropriate way. We want to define a bijective function $bt : \{(e_p, g, e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} | \; g \in G^p_{\text{CFA}}\} \to \{(e_c, g', e_c') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} | \; g' \in G_{\text{CFA}}\}$ with $bt((e_p, g, e_p')) = (e_c, g', e_c') \implies e_p' =_\sim e_c'$. Due to Lemma 5.10 and the proof of Lemma 5.1, we know that the location information considered by the producer state is a program location and the location information considered by the consumer state is a program location of the generated program. From the previous lemma, we now know that such a bijective function exists. Now, let $G^p := \{g \mid (e_p, g, e_p') \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} | \; g \in G^P_{\text{CFA}}\}$ and $G^c := \{g' \mid (e_c, g', e_c') \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}} | \; g' \in G_{\text{CFA}}\}$. Since the transfer relations $\rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ and $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ are functions, for e_p, e_c and each explored edge by the producer and consumer analysis, respectively, at most one abstract successor exists. Hence, $|G^p| = |G^c|$

and from function bt we can easily define a bijective function $bg : G^p \to G^c$ with $\forall g \in G^p : \forall (e_p, g, e'_p) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} : bt((e_p, g, e'_p)) = (e_c, bg(g), \cdot)$.

Let bijective function $o : G^p_{\text{CFA}} \to \{1, \ldots, |G^p_{\text{CFA}}|\}$ describe the exploration order of the producer in iteration i ($o(g) = x$ means exploration of g in iteration x). With out loss of generality assume that $\forall g, g' \in G^p_{\text{CFA}} : \neg\exists(e_p, g, \cdot) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} \wedge \exists(e_p, g', \cdot) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}} \implies o(g) > o(g')$. Let $o' : G_{\text{CFA}} \to \{1, \ldots, |G_{\text{CFA}}|\}$ be a bijective function with $\forall g \in G_{\text{CFA}} : g \in G_c \implies o'(g) = o(bg^{-1}(g))$. Such a function o' can easily be generated as follows. If $g \in G^c \subseteq G_{\text{CFA}}$, then $o'(g) = o(bg^{-1}(g))$ and otherwise assign a value from $\{|G_c| + 1, |G_{\text{CFA}}|)$ which was not assigned before.

We know that before the execution of the for loop in line 5 there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e_p \in \mathsf{reached}_P : e_p =_\sim b(e_p)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$. Show that after each of the first $|G_p|$ iterations of the for loop in line 5 in the producer and consumer analysis the property holds if the property held before that iteration and proper management of $\mathsf{waitlist}_C$ and orders o and o' are used.

Consider arbitrary iteration $j \leq |G_p| = |G_c| \leq min(|G^p_{\text{CFA}}|, |G_{\text{CFA}}|)$. Such an iteration exists in both, producer and consumer, analyses. Assume that at the beginning of iteration j a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ exists with $\forall e_p \in \mathsf{reached}_P : e_p =_\sim b(e_p)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$ Due to the meaning of the order functions o, o', we know that the producer explores the only edge $g_p \in G^p_{\text{CFA}}$ with $o(g_p) = j$ and the the consumer explores the unique edge $g_c \in G_{\text{CFA}}$ with $o'(g_c) = j$. Due to the assumptions on o and the construction of o', we get $g_p \in G_p$ and $g_c = bg(g_p) \in G_c$. We know that transfer successors for e_p and g_p, and for e_c and g_c exist (definition of G_p and G_c). Since the transfer relations are functions, exactly one element $(e_p, g_p, e'_p) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$ and $(e_c, g_c, e'_c) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ exist. Now, the requirements on bg give us $bt((e_p, g, e'_p)) = (e_c, g_c, e'_c)$. We infer that $e'_p =_\sim e'_c$. Let $e'_c = (e_1'^c, q'^c)$ and $e'_p = ((e_2'^p, e_1'^p), q'^p)$. Due to the requirements on the precision adjustments, we get $e^p_{\text{prec}} = ((\cdot, e_1'^p), q'^p)$, $e'_p \sim e^p_{\text{prec}}$, and $e^c_{\text{prec}} = e'_c$. From $e'_p =_\sim e'_c$, it follows that $e^p_{\text{prec}} =_\sim e'_c$.

First, show that before the execution of the for loop in line 6 the following holds true: $\exists e''_p \in \mathsf{reached}_P : e''_p \sim e^p_{\text{prec}} \Leftrightarrow \exists (e_1''^c, q''^c) \in \mathsf{reached}_C : \text{acs}(e_1'^c) = \text{acs}(e_1''^c)$.

Assume a state $e''_p \in \mathsf{reached}_P$ exists with $e''_p \sim e^p_{\text{prec}}$. Then, there exists $b(e''_p) = (\widetilde{e_1}, \widetilde{q}) \in \mathsf{reached}_C$ and $e''_p =_\sim b(e''_p)$. With $e^p_{\text{prec}} =_\sim e'_c$ and $e''_p =_\sim b(e''_p)$, we get $\text{acs}(e_1'^c) \sim e^p_{\text{prec}} \sim e''_p \sim \text{acs}(\widetilde{e_1})$, $\text{acs}(e_1'^c), \text{acs}(\widetilde{e_1}) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, and, thus, $\text{acs}(e_1'^c), \text{acs}(\widetilde{e_1}) \in L$. Due to program construction, $\text{acs}(e_1'^c), \text{acs}(\widetilde{e_1}) \in N$. From Corollary 5.26, we get $\text{acs}(e_1'^c) = \text{acs}(\widetilde{e_1})$.

Assume a state $(e_1''^c, q''^c) \in \mathsf{reached}_C$ exists with $\text{acs}(e_2'^c) = \text{acs}(e_1''^c)$. Then, a state $b^{-1}((e_1''^c, q''^c)) \in \mathsf{reached}_P$ exists and $b^{-1}((e_1''^c, q''^c)) =_\sim (e_1''^c, q''^c)$. Hence, with $e'_p =_\sim e'_c$ and $b^{-1}((e_1''^c, q''^c)) =_\sim (e_1''^c, q''^c)$ we get $b^{-1}(e_1''^c, q''^c) \sim \text{acs}(e_1''^c) \sim \text{acs}(e_1'^c) \sim e^p_{\text{prec}}$.

If not exists $e''_p \in \mathsf{reached}_P$ with $e''_p \sim e^p_{\text{prec}}$, not exists $(e_1''^c, q''^c) \in \mathsf{reached}_C$ with $\text{acs}(e_2'^c) = \text{acs}(e_1''^c)$. Lines 11 and 12 are not executed by both analyses. Before execution of line 18 there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with

$\forall e_p \in \mathsf{reached}_P : e_p =_\sim b(e_p)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e_p^j), \cdot)$.

If $e_p'' = ((e_2^{''p}, e_1^{''p}), q^{''p}) \in \mathsf{reached}_P$ exists with $e_p'' \sim e_{\text{prec}}^p$, there exists $(e_1^{''c}, q^{''c}) \in \mathsf{reached}_C$ with $\text{acs}(e_1^{'c}) = \text{acs}(e_1^{''c})$. From Lemma 5.25, Lemma 5.10, and the requirements on the merge operators, we conclude that during the iteration of the for loop in line 6, lines 11 and 12 are executed at most once in both, producer and consumer analyses, and the removed elements e_p'' and $(e_1^{''c}, q^{''c})$ are contained in the respective reached set before the execution of the for loop in line 6.

First, we show that $(e_1^{''c}, q^{''c}) = b(e_p'')$. Let $b(e_p'') = ((\widetilde{e_1'}), \widetilde{q'})$. With $e_p'' =_\sim b(e_p'')$ and $e_{\text{prec}}^p =_\sim e_c'$, we get $\text{acs}(e_1^{'c}) \sim e_{\text{prec}}^p \sim e_p'' \sim \text{acs}(\widetilde{e_1'})$, $\text{acs}(e_1^{'c}), \text{acs}(\widetilde{e_1'}) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}}$, and, thus $\text{acs}(e_1^{'c}), \text{acs}(\widetilde{e_1'}) \in L$. Due to program construction, $\text{acs}(e_1^{'c}), \text{acs}(\widetilde{e_1}) \in N$. From Corollary 5.26, we get $\text{acs}(e_1^{'c}) = \text{acs}(\widetilde{e_1})$. Lemma 5.10 lets us conclude that $(e_1^{''c}, q^{''c}) = b(e_p'')$.

Next, prove that $e_{\text{prec}}^p \sqcup e_p'' =_\sim e_c' \sqcup b(e_p'')$. Let $e_c' \sqcup b(e_p'') = (e_1^m, q^m)$. Due to the definition of merge, $\text{acs}(e_1^m) = \text{acs}(e_1^{'c}) = \text{acs}(\widetilde{e_1})$. From $e_p'' \sim \text{acs}(\widetilde{e_1})$ $(e_p'' =_\sim b(e_p''))$ and the definition of merge, we have $e_{\text{prec}}^p \sqcup e_p'' \sim \text{acs}(e_1^m)$. Let $\text{acs}(\widetilde{e_1}) = ((\widetilde{e_2^l}, \widetilde{e_1^l}), \widetilde{q^l})$. From $(e_1^{''p}, q^{''p}) \sqsubseteq (\widetilde{e_1^l}, \widetilde{q^l})$ $(e_p'' =_\sim b(e_p''))$, $(e_1^{'p}, q^{'p}) \sqsubseteq (\widetilde{e_1^l}, \widetilde{q^l})$ $(e_{\text{prec}}^p =_\sim e_c', \text{acs}(\widetilde{e_1}) = \text{acs}(e_1^{'c}), e_{\text{prec}}^p = ((\cdot, e_1^{'p}), q^{'p}))$, and the definition of merge, we have $e_{\text{prec}}^p \sqcup e_p'' = ((\cdot, e_1^{'p} \sqcup e_1^{''p}), q^{'p} \sqcup q^{''p})$ and $(e_1^{'p} \sqcup e_1^{''p}, q^{'p} \sqcup q^{''p}) \sqsubseteq (\widetilde{e_1^l}, \widetilde{q^l})$. From $e_p''[\text{acs}(\widetilde{e_1'})] = b(e_p'')$ $(e_p'' =_\sim b(e_p''))$, $e_{\text{prec}}^p[\text{acs}(e_1^{'c})] = e_c'$ $(e_p' =_\sim e_c')$, we get $(e_{\text{prec}}^p \sqcup e_p'')[\text{acs}(e_1^m)] = e_{\text{prec}}^p[\text{acs}(e_1^{'c})] \sqcup e_p''[\text{acs}(\widetilde{e_1'})] = e_c' \sqcup b(e_p'')$. We conclude that $e_{\text{prec}}^p \sqcup e_p'' =_\sim e_c' \sqcup b(e_p'')$. Let $b' : (\mathsf{reached}_P \setminus \{e_p''\}) \cup \{e_{\text{prec}}^p \sqcup e_p''\} \to (\mathsf{reached}_C \setminus \{b(e_p'')\}) \cup \{e_c' \sqcup b(e_p'')\}$ with $b' := (b \setminus \{(e_p'', b(e_p''))\}) \cup \{(e_{\text{prec}}^p \sqcup e_p'', e_c' \sqcup b(e_p''))\}$. Due to Lemma 5.10, Lemma 5.25, $e_c' \sqcup b(e_p'') \neq e_c''$, and $e_{\text{prec}}^p \sqcup e_p'' \neq e_p''$, we know that $e_c' \sqcup b(e_p'') \notin \mathsf{reached}_C$ and $e_{\text{prec}}^p \sqcup e_p'' \notin \mathsf{reached}_P$. We infer that b' is a bijective function with $\forall e \in (\mathsf{reached}_P \setminus \{e_p''\}) \cup \{e_{\text{prec}}^p \sqcup e_p''\} : e =_\sim b'(e)$. From $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e_p^j), \cdot)$, we know that the producer analysis only deletes element j if the consumer analysis deletes element j and the producer and consumer analysis delete all elements $(e_p'', \cdot)$ and $(b(e_p''), \cdot)$. Furthermore, due to Lemma 5.10, Lemma 5.25, and the fact that any waitlist only contains $(e, \cdot)$ if $e \in \mathsf{reached}$, we know that not exists $(e_c' \sqcup b(e_p''), \cdot) \in \mathsf{waitlist}_C$ nor $(e_{\text{prec}}^p \sqcup e_p'', \cdot) \in \mathsf{waitlist}_P$. If the consumer analysis adds $(e_c' \sqcup b(e_p''), \pi_{\text{prec}}^c)$ to the same position as the producer analysis added $(e_{\text{prec}}^p \sqcup e_p'', \pi_{\text{prec}}^p)$ and the consumer reordered the non-added elements in the same way as the producer which is possible in general, after the execution of line 11, there exists a bijective function $b' : (\mathsf{reached}_P \setminus \{e_p''\}) \cup \{e_{\text{prec}}^p \sqcup e_p''\} \to (\mathsf{reached}_C \setminus \{b(e_p'')\}) \cup \{e_c' \sqcup b(e_p'')\}$ with $\forall e \in (\mathsf{reached}_P \setminus \{e_p''\}) \cup \{e_{\text{prec}}^p \sqcup e_p''\} : e =_\sim b'(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C(j) = (b'(e_p^j), \cdot)$. Since in line 12 $\{e_p''\}$ and $\{b(e_p'')\}$ are deleted from and $\{e_{\text{prec}}^p \sqcup e_p''\}$ and $\{e_c' \sqcup b(e_p'')\}$ are added to $\mathsf{reached}_P$ and $\mathsf{reached}_C$, respectively, after the execution of line 12, there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e_p^j), \cdot)$.

After the execution of the for loop in line 6, there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e_p^j, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e_p^j), \cdot)$.

Next, show that at line 18 it is valid that $\mathsf{stop}(e^p_{\text{prec}}, \mathsf{reached}^P) \Leftrightarrow \mathsf{stop}(e^c_{\text{prec}}, \mathsf{reached}^C)$. If $\mathsf{stop}(e^p_{\text{prec}}, \mathsf{reached}^P) = \mathit{true}$, then $\exists \widehat{e}_p \in \mathsf{reached}_P : e^p_{\text{prec}} \sqsubseteq \widehat{e}_p \wedge e^p_{\text{prec}} \sim \widehat{e}_p$. Consider $\widehat{e}_c := b(\widehat{e}_p)$. Let $\widehat{e}_c = (\widehat{e_1'}, \widehat{q'})$ We know that $\widehat{e}_p =_\sim \widehat{e}_c$, $e^p_{\text{prec}} =_\sim e'^c, e^c_{\text{prec}} = e'^c$. Hence, $e^p_{\text{prec}} =_\sim e^c_{\text{prec}}$, $\text{acs}(e_1'^c) \sim e^p_{\text{prec}} \sim \widehat{e}_p \sim \text{acs}(\widehat{e_1'})$, $\text{acs}(e_1'^c), \text{acs}(\widehat{e_1'}) \in E^{\mathcal{A}}_{(\mathbb{C}_2 \times \mathbb{C}_1)}$ and, thus, $\text{acs}(e_1'^c), \text{acs}(\widehat{e_1'}) \in L$. Due to program construction, $\text{acs}(e_1'^c), \text{acs}(\widehat{e_1'}) \in N$. From Corollary 5.26, we get $\text{acs}(e_1'^c) = \text{acs}(\widehat{e_1'})$. With the requirements on the merge operator, we can conclude that $e''_c = (\widehat{e_1''}, \widehat{q''}) \in \mathsf{reached}_C$ exists with $\text{acs}(\widehat{e_1''}) = \text{acs}(e_1'^c)$ before the execution of the for loop in line 6 and thus $e'^c_{\text{prec}} \sqsubseteq \widehat{e}_c$. The definition of the termination check gives us $\mathsf{stop}(e'^c_{\text{prec}}, \mathsf{reached}_C) = \mathit{true}$.

If $\mathsf{stop}(e^c_{\text{prec}}, \mathsf{reached}^C) = \mathsf{stop}(e'_c, \mathsf{reached}^C) = \mathit{true}$, then $\exists \widehat{e}_c = (\widehat{e_1}, \widehat{q}) \in \mathsf{reached}_C : e'_c \sqsubseteq \widehat{e}_c \wedge \text{acs}(\widehat{e_1}) = \text{acs}(e_1'^c)$. Consider $\widehat{e}_p := b^{-1}(\widehat{e}_c)$. We know that $\widehat{e}_p =_\sim \widehat{e}_c$, $e^p_{\text{prec}} =_\sim e^c_{\text{prec}}$, and $e^c_{\text{prec}} = e'^c$. Hence, $\widehat{e}_p \sim \text{acs}(\widehat{e_1}) \sim \text{acs}(\widehat{e_1'^c}) \sim e^p_{\text{prec}}$. From the requirements on the merge operator, we can conclude that $e''_p \in \mathsf{reached}_P$ exists with $e''_p \sim e^p_{\text{prec}}$ before the execution of the for loop in line 6 and thus $e^p_{\text{prec}} \sqsubseteq \widehat{e}_p$. The definition of the termination check gives us $\mathsf{stop}(e^p_{\text{prec}}, \mathsf{reached}_P) = \mathit{true}$.

If $\mathsf{stop}(e^p_{\text{prec}}, \mathsf{reached}_P) = \mathit{true}$, we know $\mathsf{stop}(e^c_{\text{prec}}, \mathsf{reached}_C) = \mathit{true}$. Both analyses do not execute line 19. There exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$.

If $\mathsf{stop}(e^p_{\text{prec}}, \mathsf{reached}_P) = \mathit{false}$, we know $\mathsf{stop}(e^c_{\text{prec}}, \mathsf{reached}_C) = \mathit{false}$. Hence, $e^p_{\text{prec}} \notin \mathsf{reached}_P$, $e^c_{\text{prec}} \notin \mathsf{reached}_C$, not exists $(e^p_{\text{prec}}, \cdot) \in \mathsf{waitlist}_P$, and not exists $(e^c_{\text{prec}}, \cdot) \in \mathsf{waitlist}_C$. Let $b' : \mathsf{reached}_P \cup \{e^p_{\text{prec}}\} \to \mathsf{reached}_C \cup \{e^c_{\text{prec}}\}$ with $b' := b \cup \{(e^p_{\text{prec}}, e^c_{\text{prec}})$. We conclude that b' is a bijective function which ensures that $\forall e \in \mathsf{reached}_P \cup \{e^p_{\text{prec}}\} : e =_\sim b'(e)$. Furthermore, we know that e^p_{prec} is added to $\mathsf{reached}_P$, e^c_{prec} is added to $\mathsf{reached}_C$, $(e^p_{\text{prec}}, \pi^p_{\text{prec}})$ is added to $\mathsf{waitlist}_P$, and $(e^c_{\text{prec}}, \pi^C_{\text{prec}})$ is added to $\mathsf{waitlist}_C$. If the consumer analysis adds $(e^c_{\text{prec}}, \pi^c_{\text{prec}})$ to the same position as the producer analysis added $(e^p_{\text{prec}}, \pi^p_{\text{prec}})$ and that the consumer reordered the non-added elements in the same way as the producer which is possible in general, after the execution of line 19, there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$.

Due to definition of G_p, G_c, and o', and the assumption on o, we know that in any iteration $i > |G_p| = |G_c|$ of the for loop in line 5 neither the producer analysis nor the consumer analysis executes the loop body of the for loop in line 6. We know that before execution $|G_p| + 1$ of the for loop in line 5 there exists bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$, we get after the execution of the for loop in line 5 there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$. Since iterations $i > |G_p| = |G_c|$ do not modify the waitlist or the reached set, we get after execution of the for loop in line 5 there exists a bijective function $b : \mathsf{reached}_P \to \mathsf{reached}_C$ with $\forall e \in \mathsf{reached}_P : e =_\sim b(e)$, $|\mathsf{waitlist}_P| = |\mathsf{waitlist}_C|$, and $\forall 1 \leq j \leq |\mathsf{waitlist}_P| : \mathsf{waitlist}_P(j) = (e^j_p, \cdot) \implies \mathsf{waitlist}_C(j) = (b(e^j_p), \cdot)$.

The induction hypothesis follows.

We know that Algorithm 2 started with $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, e_0, initial precision π_0, and program P terminates. The producer analysis terminates. Thus, after m iterations of the producer's while loop we have $\mathsf{waitlist}_P^{m+1} = \emptyset$ and before each iteration $i < m+1$ we know $\mathsf{waitlist}_P^i \neq \emptyset$.

From induction, we know that there exist exploration orders of G_{CFA} for the first m while loop iterations of Algorithm 2 and an exploration order of $\mathsf{waitlist}$ s.t. before each iteration $i \leq m+1$ of the consumer's while loop, $|\mathsf{waitlist}_P^i| = |\mathsf{waitlist}_C^i|$. We infer that $\mathsf{waitlist}_C^{m+1} = \emptyset$ and before each iteration $i < m+1$ we know $\mathsf{waitlist}_C^i \neq \emptyset$. There exist exploration orders of G_{CFA} for each while loop iteration of Algorithm 2 and a management of $\mathsf{waitlist}$ s.t. the consumer analysis, Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, $e_0[l_0]$, initial precision π_0', and $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})$, terminates. □

Proposition 5.30. *Let* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ *be the dataflow analysis of property checking analysis* $\mathbb{C}_1^{\mathcal{A}}$, $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}}$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}$, $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}}$ *is strongly well-formed for* $((e_2, e_1), q) \in E_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}}$, $\text{acs}(e_1) \in \mathcal{L}$, *and* $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}}) = (L', G'_{\text{CFA}}, l_0')$. *If Algorithm 2 started with* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *initial abstract state* $e_0 = (e, q) \in E_{\mathbb{C}_1^{\mathcal{A}}}$ *with* $\text{acs}(e) = \text{acs}(e_1)$, *arbitrary initial precision* $\pi_0 \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* P *returns* $(true, \cdot, R^P_{\mathbb{C}_1^{\mathcal{A}}})$, *then if Algorithm 2 started with* $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, *initial abstract state* e_0' *s.t.* $e_0' = (l_0', q)$ *if* $e \in E_{\mathbb{L}}$ *and* $e_0' = ((l_0', c), q))$ *if* $e_0 = (\cdot, c)$, *arbitrary initial precision* $\pi_0' \in \Pi_{\mathbb{C}_1^{\mathcal{A}}}$, *and program* $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}})$ *terminates, it returns* $(true, \cdot, \cdot)$.

Proof. Let $(e_1, q), (e_1', q') \in E_{\mathbb{C}_1^{\mathcal{A}}}$ be two abstract states. We write $(e_1, q) \sqsubseteq_{\text{nl}} (e_1', q')$ if $q \sqsubseteq q'$ and $\mathbb{C}_1 = \mathbb{L}$ or $\mathbb{C}_1$ is a composite CPA, $e_1 = (l, c)$, $e_1' = (l', c')$, $q \sqsubseteq q'$ and $c \sqsubseteq c'$.

Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}} = (N', G'_{\text{ARG}}, root', n'_{\text{cov}})$.

Since Algorithm 2 returned $(true, \cdot, R^P_{\mathbb{C}_1^{\mathcal{A}}})$, from Proposition 2.8 we conclude that $R^P_{\mathbb{C}_1^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ is an ARG for $P = (L, G_{\text{CFA}}, l_0)$ and $\mathbb{C}_1^{\mathcal{A}}$ which is well-formed for e_0. From Proposition 2.9, we get that $R^P_{\mathbb{C}_1^{\mathcal{A}}}$ is deterministic and sound. Prove by induction over the changes of $\mathsf{reached}$ during the execution of Algorithm 2 started with $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ on generated program $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}})$ that $\forall (e_1, q) \in \mathsf{reached} : \exists (e_1', q') \in N :$ $(e_1, q) \sqsubseteq_{\text{nl}} (e_1', q') \wedge \text{acs}(e_1) \neq \top_{\mathbb{L}} \wedge (\text{acs}(e_1) = ((\widehat{e}_2, \widehat{e}_1), \widehat{q}) \implies \text{acs}(\widehat{e}_1) = \text{acs}(e_1'))$.

Basis At line 1, $\mathsf{reached}$ is initialized with e_0'. Since $R^P_{\mathbb{C}_1^{\mathcal{A}}}$ is rooted (well-formed), we know that $e_0 = (e, q) \sqsubseteq root \in N$. By definition of $\sqsubseteq_{\text{nl}}$, $\sqsubseteq$, and definition of e_0', we infer that $e_0' \sqsubseteq_{\text{nl}} e_0$. Let $root' = ((e_2^r, e_1^r), q^r)$. From $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}}$ being strongly well-formed, we get $\text{acs}(e_1^r) \in \mathcal{L}$ and $((e_2, e_1), q) \sqsubseteq root'$. We infer that $\text{acs}(e_1^r) = \text{acs}(e_1) = \text{acs}(e)$. By definition of e_0' and $l_0' = root'$, the induction hypothesis follows.

Step After initialization, abstract states are added to $\mathsf{reached}$ in line 12 and line 19. Let $(e_p, g, e_s) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ be the transfer relation transition explored in line 12 and 19, respectively. We know that $g = (l_p, op, l_s) \in G'_{\text{CFA}}$. Since Algorithm 2 only adds $(\widehat{e}, \cdot)$ to $\mathsf{waitlist}$ when it adds $\widehat{e}$ in $\mathsf{reached}$ and it removes all $(\widehat{e}, \cdot)$ in $\mathsf{waitlist}$ when it removes $\widehat{e}$ from $\mathsf{reached}$, we know that in line 4 predecessor $e_p = (e_1^p, q^p) \in \mathsf{reached}$. Hence, by induction $\exists (e_n, q_n) \in N : e_p \sqsubseteq_{\text{nl}} (e_n, q_n) \wedge \text{acs}(e_1^p) \neq \top_{\mathbb{L}} \wedge (\text{acs}(e_1^p) = ((\widehat{e}_2, \widehat{e}_1), \widehat{q}) \implies \text{acs}(\widehat{e}_1) = \text{acs}(e_n))$. By construction of $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1')^{\mathcal{A}'}})$ and $g = (l_p, op, l_s) \in G'_{\text{CFA}}$, an ARG edge $(l_p, (l_p', op, l_s'), l_s) \in G'_{\text{ARG}}$ exists. Hence,

$(l'_p, op, l'_s) \in G_{\text{CFA}}$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}'_1)^{\mathcal{A}'}}$ is strongly well-formed and $(\mathbb{C}_2 \times \mathbb{C}'_1)^{\mathcal{A}'}$ is a refined property checking analysis, we get $l_p = ((\cdot, l^1_p), \cdot)$, $\text{acs}(l^1_p) = l'_p$, $l_s = ((\cdot, l^1_s), \cdot)$, and $\text{acs}(l^1_s) = l'_s$. Let $e_s = (e^s_1, q^s)$. Since $\text{acs}(e^p_1) \neq \top_{\mathbb{L}}$, $(e_p, g, e_s) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}_1}$, and $\mathbb{C}^{\mathcal{A}}_1$ is property checking analysis, we get $\text{acs}(e^p_1) = l_p = ((\cdot, l^1_p), \cdot)$ and $\text{acs}(e^s_1) = l_s = ((\cdot, l^1_s), \cdot) \neq \top_{\mathbb{L}}$. We infer that $\text{acs}(e_n) = \text{acs}(l^1_p) = l'_p$. Due to monotonicity and location transparency of $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$, $e_p \sqsubseteq_{\text{nl}} (e_n, q_n)$, and $\text{acs}(e_n) = l'_p \in \mathcal{L}$, we conclude that $((e_n, q_n), (l'_p, op, l'_s), (e_{n'}, q_{n'})) \in \rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$, $\text{acs}(e_{n'}) = l'_s = \text{acs}(l^1_s)$. Since $\rightsquigarrow_{\mathbb{C}^{\mathcal{A}}}$ is a monotonic function and $e_p \sqsubseteq_{\text{nl}} (e_n, q_n)$, we get that $e_s = (e^s_1, q^s) \sqsubseteq_{\text{nl}} (e_{n'}, q_{n'}) \wedge \text{acs}(e^s_1) \neq \top_{\mathbb{L}} \wedge \text{acs}(e^s_1) = ((\cdot, l^1_s), \cdot) \wedge \text{acs}(l^1_s) = l'_s = \text{acs}(e_{n'})$. Due to soundness of $R^P_{\mathbb{C}^{\mathcal{A}}_1}$, we know that there exists $(e'_n, q'_n) \in N : (e_{n'}, q_{n'}) \sqsubseteq (e'_n, q'_n)$. From Corollary 5.11 and definition of N, we know that $\text{acs}(e'_n) \in \mathcal{L}$, and, thus $\text{acs}(e'_n) = \text{acs}(e_{n'}) = l'_s$. From $(e_{n'}, q_{n'}) \sqsubseteq (e'_n, q'_n)$ and $e_s = (e^s_1, q^s) \sqsubseteq_{\text{nl}} (e_{n'}, q_{n'})$, it follows that $e_s = (e^s_1, q^s) \sqsubseteq_{\text{nl}} (e'_n, q'_n)$.

If $e_{\text{new}} = (e^{\text{new}}_1, q^{\text{new}})$ is added, we know that $e_{\text{new}} = e'' \sqcup e_s$, $e'' = (e''_1, q'')$ is an element from $\mathsf{reached}$ and $\text{acs}(e''_1) = \text{acs}(e^s_1) = \text{acs}(e^{\text{new}}_1)$ (definition of e_{new}, $e'' \neq e_{\text{new}}$ and definition of $\mathsf{merge}_{\text{DFA}}$). By induction $\exists (e''_n, q''_n) \in N : e'' \sqsubseteq_{\text{nl}} (e''_n, q''_n) \wedge \text{acs}(e''_1) \neq \top_{\mathbb{L}} \wedge \text{acs}(e''_1) = ((\widehat{e}_2, \widehat{e}_1), \widehat{q}) \implies \text{acs}(\widehat{e_1}) = \text{acs}(e''_n)$. From $\text{acs}(e''_1) = \text{acs}(e^s_1)$, we conclude that $\text{acs}(e''_1) = l_s = ((\cdot, l^1_s), \cdot)$ and $\text{acs}(e''_n) = \text{acs}(l^1_s) = l'_s$. From Corollary 5.11 and definition of N, we infer that $(e''_n, q''_n) = (e'_n, q'_n)$. Now, $e_s = (e^s_1, q^s) \sqsubseteq_{\text{nl}} (e'_n, q'_n)$ and $e'' \sqsubseteq_{\text{nl}} (e'_n, q'_n)$, give us $e_{\text{new}} = e'' \sqcup e_s \sqsubseteq_{\text{nl}} (e''_n, q''_n)$. The induction hypothesis follows.

If e_{prec} is added in line 19, we know that $e_{\text{prec}} = e_s$ (definition of $\mathsf{prec}_{\text{DFA}}$ and definition of e_{prec} in line 7). The induction hypothesis follows.

Since $R^P_{\mathbb{C}^{\mathcal{A}}_1}$ is safe (well-formed), we conclude that $\neg \exists (e'_1, q') \in N : q = q_\top \vee q = q_{\text{err}}$. If Algorithm 2 terminates with $\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}'_1)^{\mathcal{A}'}})$, then by induction, at line 29 we know $\forall (e_1, q) \in \mathsf{reached} : \exists (e'_1, q') \in N : (e_1, q) \sqsubseteq_{\text{nl}} (e'_1, q')$. Due to definition of $\sqsubseteq_{\text{nl}}$, we infer that $\forall (e_1, q) \in \mathsf{reached} : q \neq q_\top \wedge q \neq q_{\text{err}}$. Hence, Algorithm 2 returns true. □

A.5 Outstanding Proofs for Chapter 6 Integration of PfP and CPC

Lemma 6.1. *Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. Let $N_{\text{sub}} \subseteq N$ be a subset of nodes and $\text{partition}(N_{\text{sub}})$ be a partition of N_{sub}. Then, the transformation $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ of partition $\text{partition}(N_{\text{sub}})$ is a partition of $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$.*

Proof. Let $\text{partition}(N_{\text{sub}}) = \{p_1, \ldots, p_k\}$ be an arbitrary partition of N_{sub}. By definition $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})] = \{p_1[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})], \ldots, p_k[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]\}$. Let $p_i[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ be an arbitrary partition element.

By definition of the transformation and $p_i \neq \emptyset$ ($\text{partition}(N_{\text{sub}})$ being a partition), we get that $p_i[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})] \neq \emptyset$.

Let $e \in p_i[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ be arbitrary. By definition $\exists n \in p_i : e = n[n]$. Since $p_i \subseteq N_{\text{sub}}$, we get by definition of $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ that $e \in N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$. Let $e' \in N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$. By definition $\exists n' \in N_{\text{sub}} : e' = n'[n']$. Since $\text{partition}(N_{\text{sub}})$

of N_{sub}, there exists a subset $p_j \in \text{partition}(N_{\text{sub}})$ with $n' \in p_j$. By definition of $p_j[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$, we get $e' \in p_j[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. Thus, $\bigcup_{1\leq i\leq k} p_i[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$.

Let $p_k[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ be an arbitrary partition element with $i \neq k$. Let $e'' \in p_k[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ be arbitrary. By definition, an element $n'' \in p_k$ exists with $e'' = n''[n'']$. Since $\text{partition}(N_{\text{sub}})$ is a partition, $n'' \notin p_i$ (partition elements are disjoint). We infer that $e'' \notin p_i[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. Hence, the sets in $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ are disjoint.

We get that $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is a partition of $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. □

Lemma 6.2. *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ be an abstract reachability graph for program P and refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$. Then, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is an abstract reachability graph for program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ and $\text{DFA}(\mathbb{C}_1^\mathcal{A})$.*

Proof. Let $P = (L, G_{\text{CFA}}, l_0)$, $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{CFA}}, root, N_{\text{cov}})$, $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) = (L', G'_{\text{CFA}}, l'_0)$, and $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$. We need to show that $N' \subseteq E_{\mathbb{C}_1^\mathcal{A}}$, N' is finite, $G'_{\text{ARG}} \subseteq N' \times G'_{\text{CFA}} \times N'$, $root' \in N'$, $N'_{\text{cov}} \subseteq N'$.

Let $n' \in N'$ be arbitrary. By definition of N', there exists $n \in N$ with $n' = n[n]$. By definition of $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$, we get that $N = L' \subseteq \mathcal{L}$. We get that $n \in \mathcal{L}$. The definition of $n[n]$ lets us conclude that $n' = n[n] \in E_{\mathbb{C}_1^\mathcal{A}}$. Thus, $N' \subseteq E_{\mathbb{C}_1^\mathcal{A}}$. Due to the definition of N' and N being finite (property of ARG), we conclude that N' is finite.

Let $(n'_p, g', n'_s) \in G'_{\text{ARG}}$ be arbitrary. By definition of G'_{ARG}, there exists an ARG edge $(n_p, (l_p, op, l_s), n_s) \in G_{\text{ARG}}$ s.t. $n_p[n_p] = n'_p$, $n_s[n_s] = n'_s$, and $g' = (n_p, op, n_s)$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is an ARG, we conclude that $n_p, n_s \in N$, $(l_p, op, l_s) \in G_{\text{CFA}}$ and, hence, $op \in Ops$. By definition of $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$, we get that $g' \in G'_{\text{CFA}}$. By definition of N' and $n_p, n_s \in N$, we conclude that $n'_p, n'_s \in N'$. We infer that $G'_{\text{ARG}} \subseteq N' \times G'_{\text{CFA}} \times N'$.

By definition of $root'$, $root' = root[root]$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is an ARG, $root \in N$. By definition of N' and $root \in N$, we get $root' \in N'$.

Let $n'_c \in N'_{\text{cov}}$ be arbitrary. By definition of N'_{cov} and $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ being an ARG ($N_{\text{cov}} \subseteq N$), there exists $n_c \in N_{\text{cov}} \subseteq N$ with $n'_c = n_c[n_c]$. By definition of N' and $n_c \in N$, we get $n'_c \in N'$. Hence, $N'_{\text{cov}} \subseteq N'$.

We conclude that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is an ARG for program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ and $\text{DFA}(\mathbb{C}_1^\mathcal{A})$. □

Proposition 6.3. *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$. If $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is strongly well-formed for $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$, then $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is an ARG for $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ and $\text{DFA}(\mathbb{C}_1^\mathcal{A})$ which is well-formed for $e_0[root]$.*

Proof. Let transformed ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$ and generated program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) = (L', G'_{\text{CFA}}, l'_0)$. From Lemma 6.2 we know that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is an ARG for $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ and $\text{DFA}(\mathbb{C}_1^\mathcal{A})$. We need to show the following properties: rootednes, completeness, well-coveredness, safety and well-constructedness.

Rootedness By definition, $root' = root[root]$. From the definition of $e_0[root]$, $root' = root[root]$, the definition of $\sqsubseteq$ and $e_0 \sqsubseteq root$ ($\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})$ strongly well-formed), we get $e_0[root] \sqsubseteq root'$. ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is rooted.

Completeness Let $((e_1, q), (l'_p, op, l'_s), (e'_1, q')) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ be a transition in the transition relation $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ of $\text{DFA}(\mathbb{C}_1^\mathcal{A})$ s.t. $(e_1, q) \in N'$, $(l'_p, op, l'_s) \in G'_{\text{CFA}}$. Due to construction of N', we know that $\text{acs}(e_1) \in \mathcal{L}$. From definition of $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$, we get $\text{acs}(e_1) = l'_p$ and $\text{acs}(e'_1) = l'_s$. From definition of N', $\text{acs}(e_1) = l'_p$, and $(e_1, q) \in N'$, we infer that $(e_1, q) = l'_p[l'_p]$. Due to program construction and $(l'_p, op, l'_s) \in G'_{\text{CFA}}$, we conclude that $(l'_p, (l_p, op, l_s), l'_s) \in G_{\text{ARG}}$ exists. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is sound (strongly well-formed), for all $(l'_p, (l_p, op, l_s), ((e^t_2, e^t_1), q^t)) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$, $((e^t_2, e^t_1), q^t) \sqsubseteq l'_s$. Since $\rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is a function, at most one such transfer successor $((e^t_2, e^t_1), q^t)$ exists. From ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ being well-constructed (strongly well-formed), exactly one such transfer successor exists. Let $l'_p = ((e^p_2, e^p_1), q^p)$. From $l'_p[l'_p] = (e_1, q)$, we get $q^p = q$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is strongly well-formed, we know that $\text{acs}(e^p_1) \in \mathcal{L}$. The definition of refined property checking transfer relation, give us $\text{acs}(e^p_1) = l_p$, and, thus $\text{acs}(e^t_1) = l_s$. Furthermore, we get $((e^p_1, q^p), (l_p, op, l_s), (e^t_1, q^t)) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$. The requirements on $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ and the definitions of $l'_p[l'_p]$ and $((e^t_2, e^t_1), q^t)[l'_s]$, let us conclude that $(l'_p[l'_p], (l'_p, op, l'_s), ((e^t_2, e^t_1), q^t)[l'_s]) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$. Since $(e_1, q) = l'_p[l'_p]$ and $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ is a function, we get $(e'_1, q') = ((e^t_2, e^t_1), q^t)[l'_s]$. From the definition of $((e^t_2, e^t_1), q^t)[l'_s]$, $l'_s[l'_s]$, definition of $\sqsubseteq$, and $((e^t_2, e^t_1), q^t) \sqsubseteq l'_s$, we infer that $(e'_1, q') = ((e^t_2, e^t_1), q^t)[l'_s] \sqsubseteq l'_s[l'_s]$. By definition of G'_{ARG} and $(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}$, we know that $(l'_p[l'_p], (l'_p, op, l'_s), l'_s[l'_s]) \in G'_{\text{ARG}}$. ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ is complete.

Well-Coveredness Let $(n, g, e) \in S'_{\text{TCNC}}$. From the definition of S'_{TCNC}, we know that $n \in N'$, $g = (l_p, op, l_s) \in G'_{\text{CFA}}$, and $(n, g, e) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$. Due to the program generation, $(l_p, (l'_p, op, l'_s), l_s) \in G_{\text{ARG}}$. Now, the ARG transformation and $(n, g, e) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ let us conclude that $n = l_p[l_p]$. Let $n = (e^n_1, q^n)$ and $e = (e_1, q)$. We know that $\text{acs}(e^n_1) = l_p$ and $l_p = ((e^p_2, e^p_1), q^n)$. The definition of $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ give us $\text{acs}(e_1) = l_s$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is strongly well-formed, there exists $(l_p, (l'_p, op, l'_s), e') \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ (well-constructed) and $\text{acs}(e^p_1) \in \mathcal{L}$. Due to soundness (strongly well-formedness) of $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$, we infer that $e' \sqsubseteq l_s$. Let $e' = ((e'_2, e'_1), q')$. From the definition of $\rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$, we know that $((e^p_1, q^n), (l'_p, op, l'_s), (e'_1, q')) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$, $\text{acs}(e^p_1) = l'_p$, and $\text{acs}(e'_1) = l'_s$. We conclude that $(l_p[l_p], g, e'[l_s]) \in \rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$. Since $l_p[l_p] = n$ and $\rightsquigarrow_{\mathbb{C}_1^\mathcal{A}}$ is a function, we get $e = e'[l_s]$ Assume that $(l_p, (l'_p, op, l'_s), e') \notin S_{\text{TCNC}}$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}$ is deterministic (strongly well-formed), not exists $(l_p, (l'_p, op, l'_s), e'') \in G_{\text{ARG}}$ and $e'' \neq l'_s$. We conclude that $l'_s \in N_{\text{cov}}$. Hence, $l'_s[l'_s] \in N'_{\text{cov}}$ and $(n, g, l'_s[l'_s]) \in G'_{\text{ARG}}$ (definition of transformation). From the definition of $e'[l'_s]$, $l'_s[l'_s]$, definition of $\sqsubseteq$, and $e' \sqsubseteq l'_s$, we infer that $e = e'[l_s] \sqsubseteq l'_s[l'_s]$. It follows that $(n, g, e) \notin S'_{\text{TCNC}}$. Our assumption was wrong, we get $(l_p, (l'_p, op, l'_s), e') \in S_{\text{TCNC}}$. We get that for every $(n, g, e) \in S'_{\text{TCNC}}$ there exists $(l_p, g', e') \in S_{\text{TCNC}}$ with $n = l_p[l_p]$, $g = (l_p, op, l_s), g' = (\cdot, op, \cdot)$, and $(l_p, g', l_s) \in G_{\text{ARG}}$ s.t. $e = e'[l_s]$. Let $m : S'_{\text{TCNC}} \to S_{\text{TCNC}}$ with $\forall (n, g, e) \in S'_{\text{TCNC}} : m((n, g, e)) = (l_p, g', e')$ implies $n = l_p[l_p]$, $g = (l_p, op, l_s), g' = (\cdot, op, \cdot)$, and $(l_p, g', l_s) \in G_{\text{ARG}}$ exists s.t. $e = e'[l_s]$. Fix total, injective function $\text{cov} : S_{\text{TCNC}} \to N \setminus N_{\text{cov}}$ with $\forall (n_o, g_o, e_o) \in S_{\text{TCNC}} : e_o \sqsubseteq \text{cov}((n_o, g_o, e_o)) \wedge (n_o, g_o, \text{cov}((n_o, g_o, e_o))) \in G_{\text{ARG}}$. Define $\text{cov}' : S'_{\text{TCNC}} \to N' \setminus N'_{\text{cov}}$ as follows: $\text{cov}'((n, g, e)) = \text{cov}(m((n, g, e)))[\text{cov}(m((n, g, e)))]$. From definitions, we

infer $\text{cov}(m((n,g,e)))[\text{cov}(m((n,g,e)))] \in N' \setminus N'_{\text{cov}}$ Let $m((n,g,e)) = (l_p, g', e')$. We know $n = l_p[l_p]$, there exists $(l_p, g', l_s) \in G_{\text{ARG}}$ s.t. $e = e'[l_s]$, $g = (l_p, op, l_s)$, and $g' = (\cdot, op, \cdot)$, and $(l_p, g', \text{cov}((l_p, g', e'))) \in G_{\text{ARG}}$ and $e' \sqsubseteq \text{cov}((l_p, g', e'))$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is deterministic (strongly well-formed), $l_s = \text{cov}((l_p, g', e'))$. We already know that $e = e'[l_s] \sqsubseteq l_s[l_s] = \text{cov}(m((n,g,e)))[\text{cov}(m((n,g,e)))] = \text{cov}'((n,g,e))$. From $n = l_p[l_p]$, $g = (l_p, op, l_s)$, $g' = (\cdot, op, \cdot)$, and $(l_p, g', l_s) \in G_{\text{ARG}}$, we infer that $(n, g, l_s[l_s]) = (n, g, \text{cov}'((n,g,e))) \in G_{\text{ARG}}$. We need to show that cov' is injective. Let $\text{cov}'((n,g,e)) = \text{cov}'((\widehat{n}, \widehat{g}, \widehat{e}))$. From the definition of cov' we infer that $\text{cov}(m((n,g,e)))[\text{cov}(m((n,g,e)))] = \text{cov}(m((\widehat{n}, \widehat{g}, \widehat{e})))[\text{cov}(m((\widehat{n}, \widehat{g}, \widehat{e})))]$. We get $\text{cov}(m((n,g,e))) = \text{cov}(m((\widehat{n}, \widehat{g}, \widehat{e})))$. Since cov is injective $m((n,g,e)) = m((\widehat{n}, \widehat{g}, \widehat{e}))$. Let $m((n,g,e)) = (l_p, g', e')$. We know that $n = l_p[l_p] = \widehat{n}$ and $g = (l_p, op, l_s), g' = (\cdot, op, \cdot)$, $\widehat{g} = (l_p, op, \widehat{l_s})$, there exists $(l_p, g', l_s) \in G_{\text{ARG}}$ s.t. $e = e'[l_s]$ and there exists $(l_p, g', \widehat{l_s}) \in G_{\text{ARG}}$ s.t. $\widehat{e} = e'[\widehat{l_s}]$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is deterministic (strongly well-formed), $l_s = \widehat{l_s}$. We get that $g = \widehat{g}$. Since $n = \widehat{n}$, $g = \widehat{g}$, $(n,g,e), (\widehat{n}, \widehat{g}, \widehat{e}) \in \rightsquigarrow_{\mathbb{C}1^{\mathcal{A}}}$ $((n,g,e), (\widehat{n}, \widehat{g}, \widehat{e}) \in S'_{\text{TCNC}})$, it follows that $e = e'[l_s] = e'[\widehat{l_s}] = \widehat{e}$. Function cov' is injective. The transformed ARG is well-covered.

Safety By definition of N', we conclude that if $(e_1, q) \in N'$, then a node $((e_2, e'_1), q) \in N$ exists with $(e_1, q) = ((e_2, e'_1), q)[((e_2, e'_1), q)]$. Since N is safe ((strongly) well-formed), we know that $q \neq q_\top$ and $q \neq q_{\text{err}}$. Hence, $\forall (e', q') \in N' : q' \neq q_\top \wedge q' \neq q_{\text{err}}$. ARG $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ is safe.

Well-Constructedness Let $((e_1, q), (l'_p, op, l'_s), (e'_1, q')) \in G'_{\text{ARG}}$ be arbitrary. By definition, there exists $(l'_p, (l_p, op, l_s), l'_s) \in G_{\text{ARG}}$, $\text{acs}(e_1) = l'_p$, $\text{acs}(e'_1) = l'_s$, $(e_1, q) = l'_p[l'_p]$, and $(e'_1, q') = l'_s[l'_s]$. Let $l'_p = ((\widehat{e_2}, \widehat{e_1}), \widehat{q})$ and $l'_s = ((\widehat{e'_2}, \widehat{e'_1}), \widehat{q'})$. We know that $q = \widehat{q}$ and $\widehat{q'} = q'$ (definition of $l'_p[l'_p]$, $l'_s[l'_s]$, and $(e_1, q) = l'_p[l'_p]$, and $(e'_1, q') = l'_s[l'_s]$). Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is well-constructed ((strongly) well-formed), there exists $(l'_p, (l_p, op, l_s), ((e^t_2, e^t_1), q^t)) \in \rightsquigarrow_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$. The definition of a transfer relation of a refined property checking analysis lets us conclude $((\widehat{e_1}, \widehat{q}), (l_p, op, l_s), (e^t_1, q^t)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. Since $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed, we get $\text{acs}(\widehat{e_1}), \text{acs}(e^t_1) \in \mathcal{L}$. From the definition of a property checking analysis, we conclude that $\text{acs}(\widehat{e_1}) = l_p$ and $\text{acs}(e^t_1) = l_s$. By definition of $l'_p[l'_p]$, $((e^t_2, e^t_1), q^t)[l'_s]$ and the requirements on a transfer relation of a property checking analysis, we get $(l'_p[l'_p], (l'_p, op, l'_s), ((e^t_2, e^t_1), q^t)[l'_s]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ From the definition of $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and $(e_1, q) = l'_p[l'_p]$, we finally infer that ARG $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$ is well-constructed.

□

Lemma 6.7. *Let $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for program P and refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. Furthermore, let $N'_{\text{sub}} \subseteq N_{\text{sub}} \subseteq N$. Then,* $\text{bound}(N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})], VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})], R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]))$ $= \text{bound}(N'_{\text{sub}}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}))[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$.

Proof. Let transformed ARG $R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$.

Case "$\subseteq$": Let state $e_b \in \text{bound}(N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})], VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})], R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]))$ be arbitrary. From the definition of boundary nodes, we conclude that $e_b \in N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$, $e_b \notin N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}})]$, and there exists

$e_p \in N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ s.t. $(e_p, e_b) \in G_{N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]}$. By definition of vertex contraction, a sequence $e_p = e_1, \ldots, e_n = e_b$ of ARG nodes exists s.t. $n \geq 2$ and $\forall 2 \leq i \leq n : (e_{i-1}, (l_{i-1}, op, l_i), e_i) \in G'_{\text{ARG}}$ and $\forall 2 \leq i < n : e_i \notin N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. Due to ARG transformation, we get $e_{i-1} = l_{i-1}[l_{i-1}]$, $e_i = l_i[l_i]$, and $(l_{i-1}, (\cdot, op, \cdot), l_i) \in G_{\text{ARG}}$. From the definition of $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ and $N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$, we infer that $l_1 \in N'_{\text{sub}}$ $(e_p = l_1[l_1])$, $\forall 2 \leq i < n : l_i \notin N_{\text{sub}}$, $l_n \notin N'_{\text{sub}}$, and $l_n \in N_{\text{sub}}$. We conclude that $(l_1, l_n) \in G_{N_{\text{sub}}}$. Hence, $l_n \in \text{bound}(N'_{\text{sub}}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}))$. Finally, we get that $e_b = l_n[l_n] \in \text{bound}(N'_{\text{sub}}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}))[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$.

Case "$\supseteq$": Let $e_b \in \text{bound}(N'_{\text{sub}}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}))[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ be arbitrary. Then, $\exists l_n \in \text{bound}(N'_{\text{sub}}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})) : e_b = l_n[l_n]$. By definition of boundary nodes, there exists $l_0 \in N'_{\text{sub}}$, $(l_0, l_n) \in G_{N_{\text{sub}}}$, $l_n \in N_{\text{sub}}$ and $l_n \notin N'_{\text{sub}}$. Furthermore, there exists $l_0 = e_1, \ldots, e_m = l_n$ with $\forall 2 \leq i \leq m : (e_{i-1}, (\cdot, op, \cdot), e_i) \in G_{\text{ARG}}$ and $\forall 2 \leq i < m : e_i \notin N_{\text{sub}}$. By program construction and ARG transformation, $(e_{i-1}[e_{i-1}], (e_{i-1}, op, e_i), e_i[e_i]) \in G'_{\text{ARG}}$. Additionally, $\forall 2 \leq i < m : e_i[e_i] \notin N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$, $e_1[e_1] \in N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$, $e_m[e_m] \in N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$, and $e_m[e_m] \notin N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. Hence, $(e_1[e_1], e_m[e_m]) \in G_{N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]}$ and $e_b = l_n[l_n] = e_m[e_m] \in \text{bound}(N'_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]))$. □

Lemma 6.8. *Let* $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ *be an ARG for program* P *and refined property checking analysis* $(\mathbb{C}_2 \times \mathbb{C}_1)^\mathcal{A}$ *and* $\text{partition}(N_{\text{sub}}) = \{p_1, \ldots, p_n\}$ *be a partition of* $N_{\text{sub}} \subseteq N$. *Then,* $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = \text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})])$.

Proof. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$ be the transformed ARG. Let $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}) = (parts, n'')$, let transformed partitioned certificate $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}}), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = (parts_{\text{sub}}, n)$, and let $\text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]) = (parts'_{\text{sub}}, n')$. From the definition of the transformation of a partitioned certificate, we know that $n'' = n$. Furthermore, the construction of a partitioned certificate gives us $n = n'' = |N|$ and $n' = |N'|$. From Lemma 6.5, we infer that $n = n'$. Next, show that $parts_{\text{sub}} = parts'_{\text{sub}}$. By definition of partition transformation, we get $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = \{p_1[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], \ldots, p_n[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]\}$.

Case "$\subseteq$": Let $(pn, bn) \in parts_{\text{sub}}$ be arbitrary. By definition there exists $(\widehat{pn}, \widehat{bn}) \in parts$ and $pn = \widehat{pn}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ and $bn = \widehat{bn}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. The definition of a partitioned certificate lets us conclude that $\widehat{pn} \in \text{partition}(N_{\text{sub}})$, thus, $\widehat{pn} \subseteq N_{\text{sub}}$, and $bn = \text{bound}(\widehat{pn}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}))$. From Lemma 6.7, we further conclude that $bn = \widehat{bn}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = \text{bound}(\widehat{pn}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}))[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = \text{bound}(\widehat{pn}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]))$ $= \text{bound}(pn, VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]))$. Knowing that $\widehat{pn} \in \text{partition}(N_{\text{sub}})$, we infer that $pn \in \text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$. The definition of a partitioned certificate, $\text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ being a partition of the set $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})]$ (Lemma 6.1), and $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] \subseteq N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^\mathcal{A}})] = N'$ (definition of N', $N_{\text{sub}} \subseteq N$), gives us $(pn, bn) \in parts'_{\text{sub}}$.

Case "$\supseteq$": Let $(pn', bn') \in parts'_{\text{sub}}$ be arbitrary. Lemma 6.1 lets us infer that the transformed set partition$(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ is a partition of $N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \subseteq N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = N'$. The definition of a partitioned certificate, lets us conclude that the partition nodes $pn' \in \text{partition}(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ and boundary nodes $bn' = \text{bound}(pn', VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]))$. Due to the definition of partition$(N_{\text{sub}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$, we know there exists $\widehat{pn'} \in \text{partition}(N_{\text{sub}})$ with $pn' = \widehat{pn'}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ and $\widehat{pn'} \subseteq N_{\text{sub}}$. Furthermore, let us define $\widehat{bn'} := \text{bound}(\widehat{pn'}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))$. From the definition of a partitioned certificate, we conclude that $(pn', bn') \in parts$. Moreover, Lemma 6.7 lets us infer the following equivalence: $\widehat{bn'}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \text{bound}(\widehat{pn'}, VCG(N_{\text{sub}}, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \text{bound}(\widehat{pn'}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])) = \text{bound}(pn', VCG(N_{\text{sub}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])) = bn'$. We get that $(pn', bn') \in parts_{\text{sub}}$. □

Proposition 6.9. *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, root, N_{\text{cov}})$ be an ARG for a refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. Then,*

- $\text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$,
- $\text{cert}_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = \text{cert}_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$,
- *if* $\text{partition}(N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) = \text{partition}(N)[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$,

$$\begin{array}{rl} & \text{cert}_{\mathcal{PC}}(\text{partition}(N), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & \text{cert}_{\mathcal{PC}}(\text{partition}(N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \end{array} \text{, and}$$

- *if* $\begin{array}{l} \text{partition}(N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])) \\ = \text{partition}(N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \end{array}$

$$\begin{array}{rl} & \text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & \text{cert}_{\mathcal{PC}}(\text{partition}(N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]), R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \end{array} .$$

Proof. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = (N', G'_{\text{ARG}}, root', N'_{\text{cov}})$.

- From definition we get

$$\begin{array}{rl} & \text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & N' \\ = & \text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \ . \end{array}$$

- From definitions, Lemma 6.6 and Lemma 6.5, we conclude that

$$\begin{array}{rl} & \text{cert}_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \\ = & (N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})], |N|) \\ = & (N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]), |N|) \\ = & (N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]), |N'|) \\ = & \text{cert}_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \ . \end{array}$$

- Directly follows from Lemma 6.8, $N_{\text{sub}} := N$ and $\mathit{partition}(N)$ being a partition of N (definition of partitioned certificate).

- Directly follows from Lemma 6.8, $N_{\text{sub}} := N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$, $N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) \subseteq N$ (definition of highly reduced node set in combination with property of ARG), and $\mathit{partition}(N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))$ being a partition of $N_{\text{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ (definition of partitioned certificate).

□

Lemma 6.10. *Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$ be an ARG for program P and refined property checking analysis $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$. If $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}} = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$ is strongly well-formed for $e_0 \in E(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, then $N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \subseteq N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \subseteq N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$.*

Proof. Let $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = (N', G'_{\text{ARG}}, \mathit{root}', N'_{\text{cov}})$.

Let $n' \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})])$. Since $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ is an ARG for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$. By definition of reduced node set and definition of $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$, we know that $n' = \mathit{root}' \vee n' \in N'_{\text{cov}} \vee \exists n'' \in N' : (n'', g, n') \in G'_{\text{ARG}} \wedge (n'', g, n') \notin \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. If $n' = \mathit{root}'$, we know that $n' = \mathit{root}[\mathit{root}]$ and $\mathit{root} \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$. By definition, $n' = \mathit{root}[\mathit{root}] \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$. If $n' \in N'_{\text{cov}}$, we know that $\exists n_c \in N_{\text{cov}} : n' = n_c[n_c]$ and $n_c \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$. By definition, $n' = n_c[n_c] \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$. If $\exists n'' \in N' : (n'', g, n') \in G'_{\text{ARG}} \wedge (n'', g, n') \notin \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, by definition of G'_{ARG} there exists $(n_p, (l_p, op, l_s), n_s) \in G_{\text{ARG}}$ s.t. $n_p = ((e_2^p, e_1^p), q^p), n_s = ((e_2^s, e_1^s), q^s) \in N$, $n_p[n_p] = n''$, $n_s[n_s] = n'$, and $g' = (n_p, op, n_s)$. Let us assume that $(n_p, (l_p, op, l_s), n_s) \in \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$. By definition $((e_1^p, q^p), (l_p, op, l_s), (e_1^s, q^s)) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. From the definition of $\rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$, we get that either $\text{acs}(e_1^p) = l_p$ and $\text{acs}(e_1^s) = l_s$ or $\text{acs}(e_1^p) = \text{acs}(e_1^s) = \top_{\mathbb{L}}$. Since $n_p, n_s \in N$ and $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is strongly well-formed, we know that $\text{acs}(e_1^p) = l_p$ and $\text{acs}(e_1^s) = l_s$. From definition of $n_p[n_p]$, $n_s[n_s]$ and $n_p, n_s \in \mathcal{L}$, we get that $(n_p[n_p], (n_p, op, n_s), n_s[n_s]) \in \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$ (requirements on property checking analysis). Contradiction to $(n'', g, n') \notin \rightsquigarrow_{\mathbb{C}_1^{\mathcal{A}}}$. Hence, $(n_p, (l_p, op, l_s), n_s) \notin \rightsquigarrow_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ and $n_s \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$. Thus, $n' = n_s[n_s] \in N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$.

From $N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) \subseteq N$ (definition of reduced node set) and definition of transformation of $N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$, and transformation of $N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$, we infer that $N[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \supseteq N_{\text{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$. □

Theorem 6.11. *Let $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$ be a refined property checking analysis. Furthermore, let $\mathbb{V}^{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})})$ be the configurable certificate validator for $\text{DFA}(\mathbb{C}_1^{\mathcal{A}})$ and coverage check $\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}$. If Algorithm 2 started with CPA $(\mathbb{C}_2 \times \mathbb{C}_1)^{\mathcal{A}}$, compatible, initial abstract state $e_0 \in E_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, initial precision $\pi_0 \in \Pi_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$, and program P returns $(\mathit{true}, \cdot, R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}) = (N, G_{\text{ARG}}, \mathit{root}, N_{\text{cov}})$, then*

- *the validation algorithm for certificates (Algorithm 3, p. 54) started with configurable certificate validator $\mathbb{V}^{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\text{DFA}(\mathbb{C}_1^{\mathcal{A}})})$, initial abstract state $e_0[\mathit{root}]$, certificate $\text{cert}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$, and program $\text{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ returns true.*

- *the validation algorithm for reduced certificates (Algorithm 4, p. 82) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, initial abstract state* $e_0[root]$*,* $\mathrm{cert}_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ *or* $\mathrm{cert}_{\mathrm{hR}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$*, as well as program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*
- *the validation algorithm for partitioned certificates (Algorithm 5, p. 106) started with configurable certificate validator* $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$*, initial abstract state* $e_0[root]$*, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}}), R^P_{\mathbb{C}^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ *which is constructed from a full, partitioned, reduced, partitioned or highly reduced, partitioned certificate* $\mathrm{cert}_{\mathcal{PC}}(\mathrm{partition}(N_{\mathrm{sub}}), R^P_{\mathbb{C}^{\mathcal{A}}})$*, and program* $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ *returns true.*

Proof. For the transformation of a certificate, a highly reduced certificate, a full partitioned certificate, and a highly reduced, partitioned certificate, this theorem directly follows from Proposition 6.9 and Theorem 6.4.

From definition of $\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}$ and Corollary 3.4, we know that $\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}$ is a well-behaving coverage check. Hence, $\mathbb{V}^{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})}(\mathsf{stop}_{\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})})$ is a CCV. From Proposition 5.2, we infer that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is an ARG for program P and $(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}$ which is strongly well-formed for e_0. From Proposition 6.3, we conclude that $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = (N', G'_{\mathrm{ARG}}, root', N'_{\mathrm{cov}})$ is an ARG for program $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ and $\mathrm{DFA}(\mathbb{C}_1^{\mathcal{A}})$ which is well-formed for $e_0[root]$. Since Algorithm 2 terminates, when started with P, we know that P is finite. From Proposition 5.6 and P being finite, we infer that program $\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})$ is finite. Furthermore, $e_0[root] \sqsubseteq e_0[root]$ (reflexity of partial order $\sqsubseteq$).

Now consider the transformation of a reduced certificate and a reduced, partitioned certificate. From Lemma 6.10 and Lemma 6.5, we infer $\mathrm{cert}_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = (\mathcal{C}^{\mathrm{sub}}, n)$ with $N_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \subseteq \mathcal{C}^{\mathrm{sub}} \subseteq N[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = N'$. Finally, Lemma 4.8 lets us conclude the claim of this theorem for the transformation of a reduced certificate.

From Lemma 6.10, we furthermore infer that $N_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]) \subseteq N_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] \subseteq N[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})] = N'$. Now, from Lemma 6.8 we get that the transformation of a reduced, partitioned certificate from an arbitrary $\mathrm{partition}(N_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))$ and ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}$ is a partitioned certificate from partition $\mathrm{partition}(N_{\mathrm{R}}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}))[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$ and ARG $R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}}[\mathrm{prog}(R^P_{(\mathbb{C}_2\times\mathbb{C}_1)^{\mathcal{A}}})]$. From Lemma 4.25, we conclude the statement for the transformation of a reduced, partitioned certificate.

The claim of this theorem follows. □

B Evaluation Results

B.1 Results Basic Configurable Program Certification

Table B.1: Efficiency examination of certificate validation w.r.t. verification which looks at the best and total improvement of the execution time and memory consumption

CPA	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{R}$	342.95	2.16	159.05	3706.0	1311.7	0.35
	13384.58	480.65	27.85	494847.8	321089.4	0.65
$\mathbb{I}$	308.01	0.15	2071.32	3496.8	212.6	0.06
	18626.48	18510.39	1.01	879860.9	811305.6	0.92
$\mathbb{R}$	4.02	2.58	1.56	557.9	415.8	0.75
	2676.39	3914.84	0.68	324870.3	337489.9	1.04
$\mathbb{S}$	35.52	25.71	1.38	1138.1	888.3	0.78
	12465.84	12223.93	1.02	885247.4	882955.6	1.00
$\mathbb{U}$	5.78	5.89	0.98	4360.4	4059.0	0.93
	9068.62	12411.37	0.73	711295.6	731277.4	1.03
$\mathbb{V}$	8.23	5.86	1.41	1127.7	619.8	0.55
	10688.58	10078.60	1.06	878929.1	820267.0	0.93
$\mathbb{SI}$	403.09	0.17	2392.25	3498	206.2	0.06
	32917.31	30960.39	3.21	1097817.7	1057072.2	0.96
$\mathbb{VR}$	8.47	5.81	1.46	1963.9	1218.7	0.62
	5593.83	6193.96	0.90	565405.9	522515.5	0.92
$\mathbb{SI}$	1.74	0.14	11.98	2163.0	583.6	0.27
	19488.76	37146.18	0.52	1225311.2	1325337.7	1.08
$\mathbb{VR}$	336.23	63.26	5.32	3202.0	344.9	0.11
	9332.39	24637.32	0.38	564549.8	703871.4	1.25
$\mathbb{I}$	2.42	1.98	1.22	402.7	383.4	0.95
	32517.21	206369.95	0.16	995938.4	1383406.4	1.39

CPA	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{R}$	4.13	3.25	1.27	2112.6	1055.3	0.50
	4213.74	90155.45	0.05	210037.9	562500.9	2.68
$\mathbb{S}$	462.69	390.49	1.18	1096.2	990.7	0.90
	5991.17	6816.55	0.88	481160.7	536692.9	1.12
$\mathbb{U}$	4.98	5.77	0.86	4303.8	4048.4	0.94
	8698.32	12833.531	0.68	734606.1	755635.5	1.03
$\mathbb{V}$	168.87	64.50	2.62	2163.0	648.8	0.25
	25953.79	117885.82	0.22	758460.3	916245.9	1.21
$\mathbb{SI}$	31.07	22.15	1.41	3369.4	2425.1	0.72
	48296.36	208682.16	0.23	1380627.9	1529124.2	1.11
$\mathbb{VR}$	164.05	63.73	2.57	3201.6	975.4	0.31
	5712.04	15826.28	0.36	378929.6	621766.8	1.64
$\mathbb{O}$	63.10	3.80	16.61	3939.8	1135.5	0.29
	36293.32	63677.21	0.57	795084.3	684812.4	0.86
$\mathbb{P}$	14.78	5.64	2.62	600.3	3007.3	0.51
	22092.34	226677.71	0.01	408060.8	818166.4	2.01
$\mathbb{V}$	484.96	0.95	510.59	4304.9	270.8	0.06
	23505.46	7655.15	3.07	686101.9	448909.4	0.65

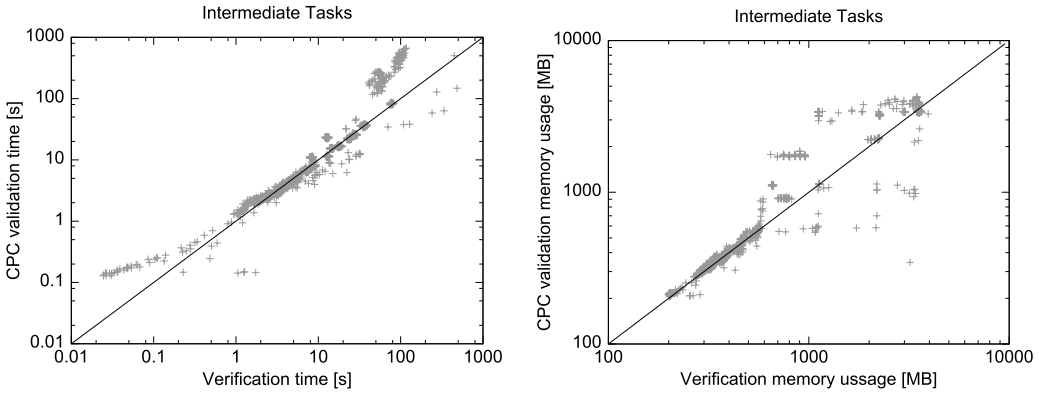

Figure B.1: Comparison of producer verification times of intermediate analysis tasks with consumer times for validation of the certificate from the producer's ARG (left) and comparison of memory consumption of the producer verification with memory consumption of the consumer for validation of the certificate from the producer's ARG (right)

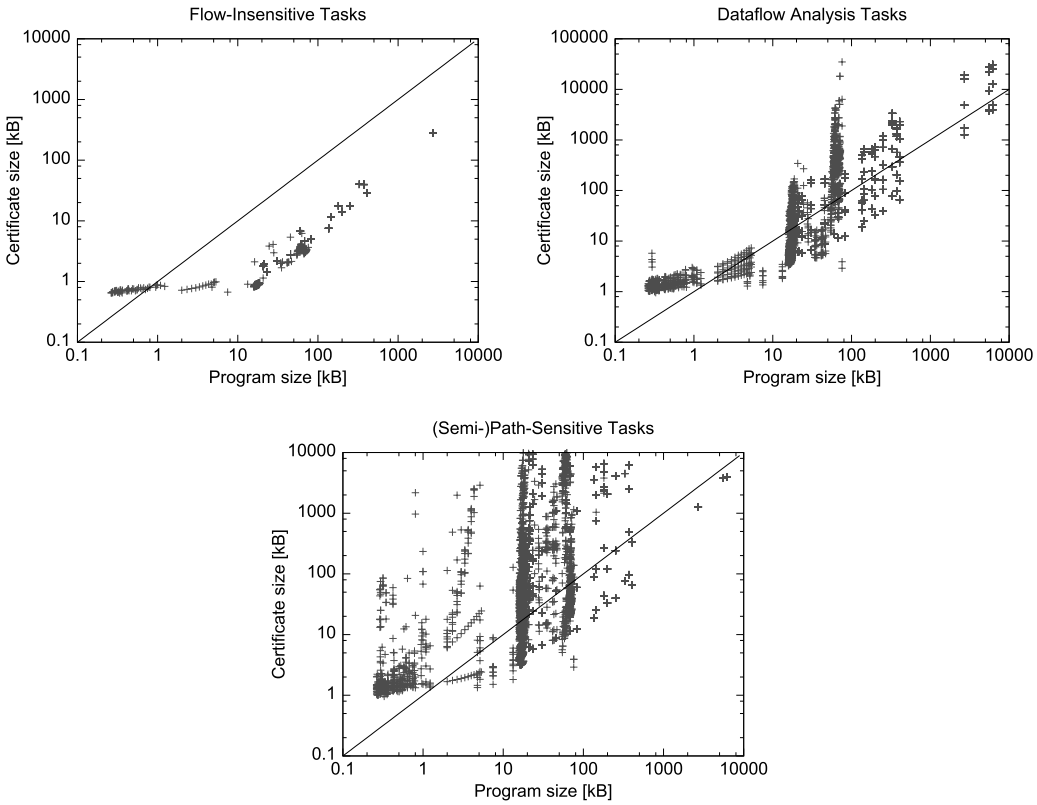

Figure B.2: Relation of the certificate file size (zip compressed) to the file size of the program (uncompressed) for flow-insensitive tasks (top left), dataflow analysis tasks (top right), and (semi-)path-sensitive tasks (bottom)

B.2 Results Optimized CPC

Table B.2: For each analysis, the choice of the partition strategy and the size of the partition element for the computation of full, partitioned certificates

		10	100	1000	10000	BEST_ FIRST	FM K-WAY	MULTI- LEVEL
DFA	$\mathbb{I}$	✓					✓	
	$\mathbb{R}$		✓			✓		
	$\mathbb{S}$	✓						✓
	$\mathbb{U}$	✓						✓
	$\mathbb{V}$	✓						✓
	$\mathbb{SI}$	✓					✓	
	$\mathbb{VR}$	✓				✓		
Inter.	$\mathbb{SI}$	✓						✓
	$\mathbb{VR}$		✓				✓	
MC	$\mathbb{I}$	✓				✓		
	$\mathbb{R}$		✓				✓	

		10	100	1000	10000	BEST_FIRST	FM K-WAY	MULTI-LEVEL
MC	$\mathbb{S}$		✓					✓
	$\mathbb{U}$		✓					✓
	$\mathbb{V}$		✓					✓
	$\mathbb{SI}$	✓						✓
	$\mathbb{VR}$		✓					✓
CEGAR	$\mathbb{O}$	✓				✓		
	$\mathbb{V}$		✓					✓

Table B.3: For each analysis, the choice of the partition strategy and the size of the partition element for the computation of reduced, partitioned certificates

		10	100	1000	10000	BEST_FIRST	FM K-WAY	MULTI-LEVEL
DFA	$\mathbb{I}$		✓			✓		
	$\mathbb{R}$	✓						✓
	$\mathbb{S}$	✓						✓
	$\mathbb{U}$	✓						✓
	$\mathbb{V}$		✓			✓		
	$\mathbb{SI}$		✓			✓		
	$\mathbb{VR}$	✓				✓		
Inter.	$\mathbb{SI}$	✓				✓		
	$\mathbb{VR}$		✓			✓		
MC	$\mathbb{I}$	✓				✓		
	$\mathbb{R}$	✓					✓	
	$\mathbb{S}$		✓			✓		
	$\mathbb{U}$		✓			✓		
	$\mathbb{V}$		✓				✓	
	$\mathbb{SI}$	✓						✓
	$\mathbb{VR}$	✓				✓		
CEGAR	$\mathbb{O}$	✓				✓		
	$\mathbb{V}$		✓			✓		

Table B.4: For each analysis, the choice of the partition strategy and the size of the partition element for the computation of highly reduced, partitioned certificates

		10	100	1000	10000	BEST_FIRST	FM K-WAY	MULTI-LEVEL
DFA	$\mathbb{I}$		✓			✓		
	$\mathbb{R}$		✓				✓	
	$\mathbb{S}$	✓						✓
	$\mathbb{U}$	✓						✓
	$\mathbb{V}$	✓						✓
	$\mathbb{SI}$		✓			✓		

		10	100	1000	10000	BEST_FIRST	FM K-WAY	MULTI-LEVEL
Inter.	$\mathbb{VR}$	✓						✓
	$\mathbb{SI}$			✓				✓
	$\mathbb{VR}$		✓			✓		
MC	$\mathbb{I}$			✓		✓		
	$\mathbb{R}$		✓					✓
	$\mathbb{S}$	✓						✓
	$\mathbb{U}$		✓					✓
	$\mathbb{V}$	✓					✓	
	$\mathbb{SI}$	✓					✓	
	$\mathbb{VR}$	✓					✓	
CEGAR	$\mathbb{O}$	✓					✓	
	$\mathbb{V}$	✓					✓	

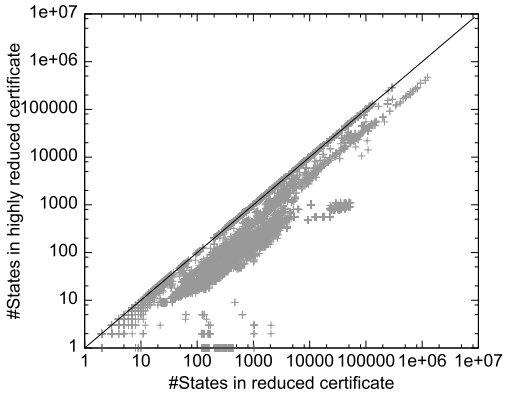

Figure B.3: Comparison of the number of abstract states stored in the reduced and highly reduced certificate

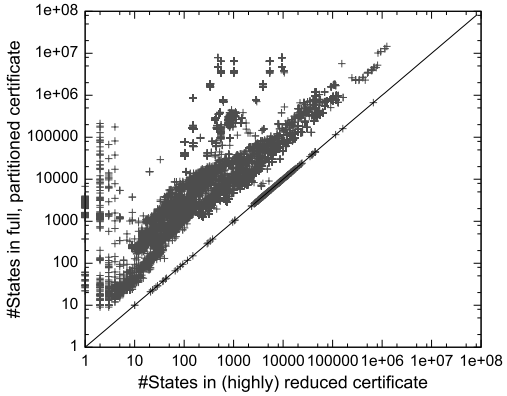

Figure B.4: Comparison of the number of abstract states stored in the best reduced certificate with those stored in the full, partitioned certificate

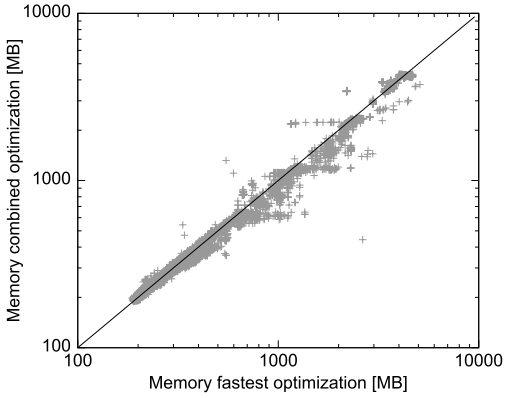

Figure B.5: Comparing the memory consumption of the fastest, optimized configurable program certificate approach with the combination of the two optimization approaches

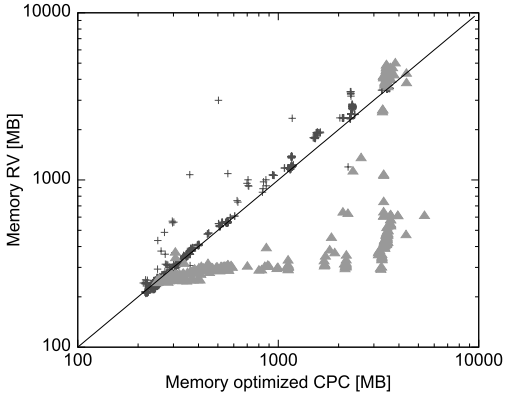

Figure B.6: Comparing the memory consumption of the fastest, optimized CPC validation with regression verification.

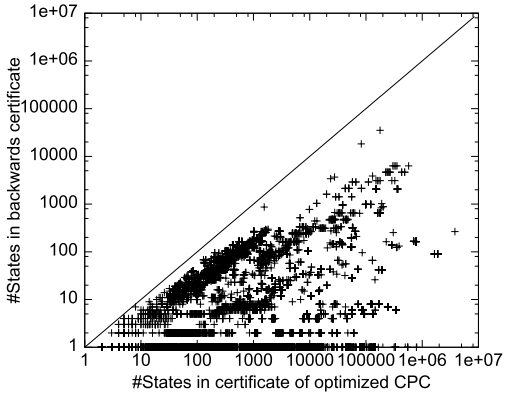

Figure B.7: Comparing the number of abstract states, the main part of the certificates, stored in that certificate whose validation was fastest among all CPC techniques for the task and the backwards certificate.

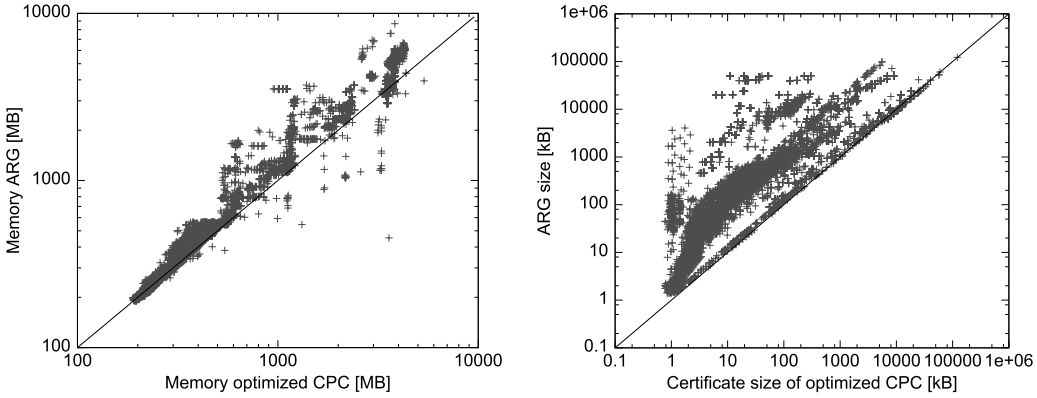

Figure B.8: Comparison of the memory consumption of the fastest, optimized CPC approach with the memory consumption of ARG validation (left) and comparison of the certificate size of the fastest, optimized CPC approach with the ARG size (right)

B.3 Results Programs from Proofs Approach

Table B.5: Comparison of the producer and consumer verification in the Programs from Proofs approach. Verification times of the producer V_P and consumer V_C are given in seconds and memory consumption, used heap plus used non-heap, is given in MB.

$\mathbb{C}_P$	program	#r	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
			Dataflow Analysis					
$\mathbb{PI}$	nosprintf*	3	0.59	0.07	8.47	243.90	210.20	0.86
	interproc*	2	0.32	0.05	6.07	241.00	210.00	0.87
	NetBSD*	4	0.68	0.08	8.39	245.20	214.90	0.88
	PfPb*	2	0.18	0.02	7.91	235.10	205.90	0.88
	PfPc*	2	0.22	0.03	6.97	229.70	202.60	0.88
	SLRb*	1	0.13	0.02	5.25	232.00	204.90	0.88
$\mathbb{OI}$	interproc	1	0.32	0.05	5.91	233.00	210.30	0.90
	SpamAssassin	1	0.93	0.16	6.02	244.00	214.30	0.88
	sendmail	1	0.64	0.24	2.66	241.10	218.10	0.90
	invertstring	1	0.56	0.20	2.84	231.80	215.70	0.93
	fibonacci*	1	0.23	0.05	5.08	222.50	200.10	0.90
	relax	1	0.42	0.19	2.29	234.00	214.90	0.92
$\mathbb{PS}$	condsum*	2	0.22	0.05	4.42	237.10	201.30	0.85
	propertyInFlag*	2	0.17	0.05	3.70	232.50	199.30	0.86
	PfP*	2	0.19	0.03	5.74	234.30	199.60	0.85
	PfPb*	2	0.17	0.03	6.37	232.90	196.20	0.84
	PfPc*	2	0.21	0.04	5.61	231.90	197.60	0.85
	liststatistics*	8	0.63	0.11	5.67	253.90	205.90	0.81
	harmonicMean*	2	0.43	0.14	3.08	241.70	207.10	0.86
	fraction	4	0.52	0.07	7.26	242.30	201.40	0.83
	SLR*	1	0.12	0.02	4.96	230.60	198.60	0.86
	facnegsum*	2	0.19	0.06	3.42	233.80	202.40	0.87

$\mathbb{C}_P$	program	#r	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{PS}$	SubMinSumDiv*	2	0.25	0.09	2.93	237.60	203.80	0.86
	lockfree3.0*	14	1.97	0.12	16.70	288.40	202.30	0.70
$\mathbb{OS}$	testlocks7	7	2.30	1.15	2.01	330.20	265.00	0.80
	testlocks8	8	9.30	2.07	4.50	566.00	283.10	0.50
	inf4*	2	0.45	0.32	1.41	225.90	216.70	0.96
	inf8	3	0.42	0.08	5.09	226.60	201.40	0.89
	propertyInFlag	1	0.14	0.05	2.69	215.30	202.20	0.94
	SLRb	1	0.11	0.03	3.98	217.30	198.30	0.91
	SubMinSumDiv*	1	0.15	0.08	1.82	215.10	201.50	0.94
	ESP	1	0.14	0.04	3.02	214.40	205.20	0.96
	ESPa	1	0.11	0.03	3.67	215.30	201.00	0.93
	ESPb	1	0.12	0.03	3.73	212.20	201.40	0.95
	ESPc	1	0.11	0.03	3.56	214.30	203.20	0.95
	addIteration*	1	0.09	0.03	2.97	212.60	202.00	0.95
$\mathbb{VS}$	SLRb*	2	0.10	0.03	4.04	217.80	197.40	0.91
	SLR*	1	0.08	0.02	3.41	217.30	200.60	0.92
	inf6*	4	0.28	0.03	8.23	230.50	201.50	0.87
	inf8*	3	0.37	0.07	5.43	228.20	202.50	0.89
	kundu*	10	3.00	0.36	8.41	346.00	214.90	0.62
	memslave1*	22	7.52	0.91	8.31	564.00	273.20	0.48
	memslave2	8	6.86	1.26	5.46	562.30	296.10	0.53
	PfPb*	2	0.14	0.03	5.57	222.00	199.20	0.90
	PfPc	2	0.26	0.04	6.74	222.70	199.70	0.90
	ESP	1	0.16	0.05	3.41	225.60	202.10	0.90
	ESPb*	1	0.11	0.03	3.33	217.70	202.10	0.93
	lockfree3.1*	3	0.44	0.12	3.73	230.80	204.00	0.88
$\mathbb{PSI}$	invertsorted*	21	9.92	0.67	14.76	565.70	233.80	0.41
	div	20	5.37	0.30	17.72	441.60	215.20	0.49
	fibonacci*	3.7	0.50	0.06	7.81	242.90	205.40	0.85
$\mathbb{OSI}$	fibonacci*	1.3	0.31	0.06	4.87	226.50	206.50	0.91
	palindrom	1	0.91	0.48	1.90	242.00	221.00	0.91
	invertarray	1	0.82	0.35	2.35	240.60	218.00	0.91
$\mathbb{PU}$	cdaudio	32	10.20	2.70	3.78	597.70	345.70	0.58
	pipeline2*	230	227.81	0.23	985.32	1792.80	211.50	0.12
	diskperf	2	2.00	1.37	1.47	314.60	269.90	0.86
$\mathbb{OU}$	pipeline	11	4.21	1.15	3.65	412.60	256.10	0.62
	s3srvr	1	0.95	0.27	3.56	257.10	215.30	0.84
	pipeline2	11	8.81	260.19	0.03	577.60	3971.20	6.88
$\mathbb{VU}$	s3srvr	2.8	2.76	0.62	4.43	339.30	236.50	0.70
	cdaudio	1	2.75	2.89	0.95	365.60	344.80	0.94
	pipeline2*	13	2.93	0.32	9.13	367.70	221.50	0.60
$\mathbb{PV}$	testlocks5*	30.2	2.64	0.30	8.93	304.60	214.60	0.70
	kbfiltr1*	2	0.90	0.41	2.17	268.40	230.50	0.86
	kbfiltr2*	2	1.29	0.77	1.68	281.90	244.20	0.87
	testlocks5d*	8.2	0.90	0.12	7.33	257.50	215.50	0.84
	testlocks12	18	329.79	8.23	40.05	1713.20	591.80	0.35
	testlocks12d	12	5.71	0.27	21.37	540.90	219.00	0.40

$\mathbb{C}_P$	program	#r	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{OV}$	memslave1	7.1	4.39	0.35	12.51	368.00	226.80	0.62
	memslave2	4	2.96	0.59	5.03	410.30	229.60	0.56
	testlocks6	6	1.45	0.36	3.99	280.50	226.40	0.81
	kbfiltr1	3.1	2.01	0.56	3.59	311.20	234.30	0.75
	kbfiltr2	2.9	2.91	0.96	3.04	354.10	251.60	0.71
	inf8	3	0.45	0.11	3.95	230.20	214.70	0.93
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	35	17.63	0.47	37.20	780.60	233.70	0.30
	memslave2	35	18.31	0.63	28.87	766.00	241.20	0.31
	kbfiltr2*	4	2.07	0.84	2.46	302.10	250.60	0.83
	testlocks6*	161	25.70	0.54	47.50	771.10	229.80	0.30
$\mathbb{O}\widetilde{\mathbb{V}}$	kbfiltr1	4	2.53	0.55	4.59	328.50	236.90	0.72
	testlocks5*	5	1.53	0.33	4.69	272.70	221.10	0.81
	testlocks5d*	1	0.33	0.12	2.83	229.40	214.10	0.93
Intermediate (Sep-Join)								
$\mathbb{PSI}$	fibonacci*	3	0.45	0.06	7.42	243.20	206.40	0.85
	palindrom*	18	6.02	0.48	12.65	475.70	221.20	0.46
	invertarray*	18	17.90	0.32	55.38	523.60	217.90	0.42
$\mathbb{OSI}$	invertsorted	1	1.27	0.81	1.57	261.70	237.70	0.91
	div	1	0.63	0.33	1.94	236.70	214.70	0.91
	palindrom	1	0.95	0.46	2.08	241.20	217.10	0.90
Model Checking								
$\mathbb{PL}$	locks*	2	0.15	0.01	10.85	232.80	199.70	0.86
	tokenring03	59	198.41	2.84	69.79	1733.20	299.20	0.17
	memslave1*	271	155.63	4.16	37.43	1793.00	368.80	0.21
	s3srvr	7	10.52	3.09	3.40	544.80	340.10	0.62
$\mathbb{OL}$	s3srvr	1.4	1.06	0.15	7.07	260.00	237.10	0.91
	transmitter01*	1	0.36	0.11	3.19	228.30	207.50	0.91
	transmitter02*	1	0.46	0.18	2.50	233.50	214.50	0.92
$\mathbb{VL}$	tokenring02	16	5.99	0.22	27.74	549.50	213.50	0.39
	tokenring04	22	14.15	43.05	0.33	1056.00	537.30	0.51
	transmitter03	1	0.59	0.23	2.55	241.30	215.00	0.89
$\mathbb{PS}$	kundu*	20	15.15	0.60	25.06	594.00	228.50	0.38
	transmitter01*	5	1.66	0.18	9.31	278.70	206.10	0.74
	transmitter02	6	8.63	5.47	1.58	494.70	553.10	1.12
$\mathbb{OS}$	transmitter01	1	0.89	0.18	5.05	251.90	205.20	0.81
	transmitter02	1	1.46	0.31	4.70	272.10	214.40	0.79
	powerapprox	1	9.86	10.73	0.92	1036.40	996.80	0.96
$\mathbb{VS}$	inf6*	4	0.31	0.03	9.24	228.50	204.10	0.89
	kundu*	10	6.61	0.14	47.92	481.10	202.60	0.42
	transmitter02*	5	5.07	0.21	23.98	438.10	206.20	0.47
$\mathbb{PU}$	pipeline*	15	6.25	0.22	28.65	474.90	215.70	0.45
	cdaudio	20.8	10.84	4.69	2.31	855.50	449.80	0.53
	diskperf	2	3.95	3.64	1.08	396.70	359.10	0.91
$\mathbb{OU}$	pipeline	11	3.90	1.18	3.30	419.00	260.20	0.62
	pipeline2	11	8.51	258.16	0.03	579.30	3903.00	6.74
	s3srvr	1	0.88	0.28	3.19	250.90	218.30	0.87
$\mathbb{VU}$	s3srvr*	6	3.81	0.61	6.23	385.60	231.90	0.60

$\mathbb{C}_P$	program	#r	V_P	V_C	$\frac{V_P}{V_C}$	M_P	M_C	$\frac{M_C}{M_P}$
$\mathbb{VU}$	cdaudio	1	3.00	3.63	0.83	388.80	380.00	0.98
	pipeline2*	14	3.05	0.31	9.82	370.90	220.20	0.59
$\mathbb{PV}$	testlocks5d*	0.9	2.22	0.60	3.70	308.10	231.90	0.75
	inf1*	1	0.20	0.07	2.72	238.70	214.90	0.90
$\mathbb{OV}$	testlocks5*	5	2.73	0.35	7.90	330.90	224.00	0.68
	testlocks12d*	1	135.46	0.26	520.62	1771.80	217.80	0.12
	testlocks5d*	1	0.66	0.11	5.78	240.30	216.10	0.90
$\mathbb{P}\widetilde{\mathbb{V}}$	testlocks5*	0	2.26	0.65	3.48	318.80	230.80	0.72
	interproc*	1	0.34	0.11	3.28	245.90	213.20	0.87
	nosprintf*	1	0.50	0.15	3.44	251.20	216.30	0.86
$\mathbb{O}\widetilde{\mathbb{V}}$	testlocks5d*	1	0.63	0.11	5.78	239.70	217.50	0.91
	relax*	1	0.88	0.47	1.87	255.00	230.40	0.90
	nosprintf	1	0.46	0.14	3.26	238.40	216.80	0.91
	Sum		1401.52	647.13		49526.30	38043.10	

Table B.6: Comparison of the original $\#\text{loc}_P, \text{bytes}_P$ and generated program $\#\text{loc}_C, \text{bytes}_C$ sizes based on the number of locations #loc and the program file size bytes in bytes

$\mathbb{C}_P$	program	$\#\text{loc}_P$	$\#\text{loc}_C$	$\frac{\#\text{loc}_C}{\#\text{loc}_P}$	bytes_P	bytes_C	$\frac{\text{bytes}_C}{\text{bytes}_P}$
	Datflow Analysis						
$\mathbb{PI}$	nosprintf*	70	71	1.01	2108	1644	0.78
	interproc*	53	49	0.92	1541	953	0.62
	NetBSD*	68	102	1.50	2087	2014	0.97
	PfPb*	16	14	0.88	150	139	0.93
	PfPc*	23	24	1.04	223	245	1.10
	SLRb*	18	15	0.83	156	142	0.91
$\mathbb{OI}$	interproc	66	49	0.74	1541	953	0.62
	SpamAssassin	130	120	0.92	3121	2247	0.72
	sendmail	100	163	1.63	2184	4003	1.83
	invertstring	60	151	2.52	646	2493	3.86
	fibonacci*	36	48	1.33	301	706	2.35
	relax	49	146	2.98	1466	2647	1.81
$\mathbb{PS}$	condsum*	23	34	1.48	278	446	1.60
	propertyInFlag*	27	28	1.04	258	335	1.30
	PfP*	20	21	1.05	191	229	1.20
	PfPb*	16	14	0.88	150	139	0.93
	PfPc*	23	24	1.04	223	245	1.10
	liststatistics*	138	73	0.53	4049	1550	0.38
	harmonicMean*	77	78	1.01	2266	1640	0.72
	fraction	44	62	1.41	1205	1372	1.14
	SLR*	16	14	0.88	141	135	0.96
	facnegsum*	22	34	1.55	206	422	2.05
	SubMinSumDiv*	34	49	1.44	311	649	2.09
	lockfree3.0*	119	67	0.56	2069	1637	0.79
$\mathbb{OS}$	testlocks7	117	1320	11.28	2723	22854	8.39

$\mathbb{C}_P$	program	$\#\text{loc}_P$	$\#\text{loc}_C$	$\frac{\#\text{loc}_C}{\#\text{loc}_P}$	bytes_P	bytes_C	$\frac{\text{bytes}_C}{\text{bytes}_P}$
	testlocks8	121	2868	23.70	2779	36225	13.04
	inf4*	46	231	5.02	614	4982	8.11
	inf8	64	71	1.11	762	1135	1.49
	propertyInFlag	27	27	1.00	258	318	1.23
	SLRb	18	14	0.78	156	125	0.80
$\mathbb{OS}$	SubMinSumDiv*	34	48	1.41	311	617	1.98
	ESP	25	31	1.24	289	545	1.89
	ESPa	23	19	0.83	269	257	0.96
	ESPb	23	19	0.83	266	257	0.97
	ESPc	21	19	0.90	241	257	1.07
	addIteration*	32	26	0.81	445	489	1.10
	SLRb*	18	15	0.83	156	142	0.91
	SLR*	16	14	0.88	141	135	0.96
	inf6*	41	24	0.59	360	244	0.68
	inf8*	64	61	0.95	762	1029	1.35
	kundu*	402	302	0.75	7841	6046	0.77
$\mathbb{VS}$	memslave1*	1080	967	0.90	28163	52921	1.88
	memslave2	1087	1225	1.13	28252	58872	2.08
	PfPb*	16	12	0.75	150	121	0.81
	PfPc	23	25	1.09	223	257	1.15
	ESP	25	33	1.32	289	577	2.00
	ESPb*	23	21	0.91	266	289	1.09
	lockfree3.1*	132	87	0.66	2212	1741	0.79
	invertsorted*	34	430	12.65	369	8601	23.31
$\mathbb{PSI}$	div	24	161	6.71	313	2803	8.96
	fibonacci*	29.7	47	1.58	301	689	2.29
	fibonacci*	35.3	48	1.36	301	706	2.35
$\mathbb{OSI}$	palindrom	33	310	9.39	244	5165	21.17
	invertarray	34	248	7.29	219	3651	16.67
	cdaudio	1637	5822	3.56	64510	143506	2.22
$\mathbb{PU}$	pipeline2*	637	421	0.66	13473	9509	0.71
	diskperf	864	2782	3.22	25267	79713	3.15
	pipeline	619	1932	3.12	13000	44884	3.45
$\mathbb{OU}$	s3srvr	518	641	1.24	43877	14887	0.34
	pipeline2	637	35207	55.27	13473	841659	62.47
	s3srvr	518	1362	2.63	43877	28696	0.65
$\mathbb{VU}$	cdaudio	1637	5798	3.54	64510	141393	2.19
	pipeline2*	637	677	1.06	13473	16070	1.19
	testlocks5*	81	227	2.80	1762	3374	1.91
	kbfiltr1*	400	368	0.92	19088	9553	0.50
$\mathbb{PV}$	kbfiltr2*	683	808	1.18	32845	22374	0.68
	testlocks5d*	73	78	1.07	1530	1019	0.67
	testlocks12	172	24646	143.29	3922	381105	97.17
	testlocks12d	150	246	1.64	3202	2964	0.93
	memslave1	1080	432	0.40	28163	7891	0.28
$\mathbb{OV}$	memslave2	1087	358	0.33	28252	8923	0.32
	testlocks6	103	360	3.50	2386	5589	2.34

$\mathbb{C}_P$	program	$\#\text{loc}_P$	$\#\text{loc}_C$	$\frac{\#\text{loc}_C}{\#\text{loc}_P}$	bytes_P	bytes_C	$\frac{\text{bytes}_C}{\text{bytes}_P}$
	kbfiltr1	400	468	1.17	19088	11322	0.59
$\mathbb{OV}$	kbfiltr2	683	941	1.38	32845	24951	0.76
	inf8	64	67	1.05	762	1108	1.45
	memslave1	1080	580	0.54	28163	18043	0.64
	memslave2	1087	585	0.54	28252	18186	0.64
$\mathbb{P}\widetilde{\mathbb{V}}$	kbfiltr2*	683	842	1.23	32845	22780	0.69
	testlocks6*	103	606	5.88	2386	10644	4.46
	kbfiltr1	400	468	1.17	19088	11955	0.63
$\mathbb{O}\widetilde{\mathbb{V}}$	testlocks5*	81	292	3.60	1762	3930	2.23
	testlocks5d*	73	71	0.97	1530	908	0.59
	Intermediate (Sep-Join)						
	fibonacci*	29	47	1.62	301	689	2.29
$\mathbb{PSI}$	palindrom*	26	306	11.77	244	5447	22.32
	invertarray*	26	243	9.35	219	3630	16.58
	invertsorted	42	491	11.69	369	8981	24.34
$\mathbb{OSI}$	div	29	163	5.62	313	2620	8.37
	palindrom	33	310	9.39	244	5166	21.17
	Model Checking						
	locks*	23	23	1.00	280	294	1.05
$\mathbb{PL}$	tokenring03	498	7195	14.45	8917	137888	15.46
	memslave1*	1080	12252	11.34	28163	285883	10.15
	s3srvr	518	9106	17.58	43877	221027	5.04
	s3srvr	571.6	821	1.44	43877	19715	0.45
$\mathbb{OL}$	transmitter01*	344	458	1.33	5702	8238	1.44
	transmitter02 *	446	744	1.67	7398	13524	1.83
	tokenring02	406	736	1.81	7254	14199	1.96
$\mathbb{VL}$	tokenring04	586	9233	15.76	10505	399662	38.04
	transmitter03	472	784	1.66	8602	14333	1.67
	kundu*	402	626	1.56	7841	15487	1.98
$\mathbb{PS}$	transmitter01*	322	205	0.64	5702	3793	0.67
	transmitter02	420	1761	4.19	7398	37652	5.09
	transmitter01	322	216	0.67	5702	4478	0.79
$\mathbb{OS}$	transmitter02	420	354	0.84	7398	7946	1.07
	powerapprox	19	7179	377.84	269	109438	406.83
	inf6*	41	23	0.56	360	225	0.63
$\mathbb{VS}$	kundu*	402	137	0.34	7841	5343	0.68
	transmitter02*	420	235	0.56	7398	4314	0.58
	pipeline*	619	521	0.84	13000	12369	0.95
$\mathbb{PU}$	cdaudio	1637	10526	6.43	64510	255961	3.97
	diskperf	864	6420	7.43	25267	168372	6.66
	pipeline	619	1932	3.12	13000	44884	3.45
$\mathbb{OU}$	pipeline2	637	35207	55.27	13473	841659	62.47
	s3srvr	518	653	1.26	43877	15709	0.36
	s3srvr*	518	1204	2.32	43877	28425	0.65
$\mathbb{VU}$	cdaudio	1637	6938	4.24	64510	166070	2.57
	pipeline2*	637	645	1.01	13473	15318	1.14
$\mathbb{PV}$	testlocks5d*	73	519	7.11	1530	7869	5.14

$\mathbb{C}_P$	program	$\#\text{loc}_P$	$\#\text{loc}_C$	$\frac{\#\text{loc}_C}{\#\text{loc}_P}$	bytes_P	bytes_C	$\frac{\text{bytes}_C}{\text{bytes}_P}$
$\mathbb{PV}$	inf1*	37	36	0.97	433	592	1.37
	testlocks5*	81	292	3.60	1762	3930	2.23
$\mathbb{OV}$	testlocks12d*	150	232	1.55	3202	2769	0.86
	testlocks5d*	73	71	0.97	1530	915	0.60
	testlocks5*	81	532	6.57	1762	8264	4.69
$\mathbb{P}\widetilde{\mathbb{V}}$	interproc*	53	63	1.19	1541	1278	0.83
	nosprintf*	70	113	1.61	2108	2422	1.15
	testlocks5d*	73	71	0.97	1530	915	0.60
$\mathbb{O}\widetilde{\mathbb{V}}$	relax*	40	283	7.08	1466	5683	3.88
	nosprintf	70	101	1.44	2108	2186	1.04
	Sum	37488	217265		1220010	5041533	

Table B.7: Comparison of the Consumer Validation Times in the PfP and PCC approaches. Validation times are given in seconds.

$\mathbb{C}_P$	program	V_{PfP}	V_{RV}	V_{ARG}	V_{CPC}	$V_{\text{CPC}^{\text{RP}}}$	$\frac{\min V_{\text{PCC}}}{V_{\text{PfP}}}$
	Dataflow Analysis						
	nosprintf*	0.07	0.25	F	F	0.38	3.61
	interproc*	0.05	0.17	0.39	0.39	0.28	3.18
$\mathbb{PI}$	NetBSD*	0.08	0.19	F	F	0.35	2.32
	PfPb*	0.02	0.12	0.30	0.26	0.23	5.08
	PfPc*	0.03	0.15	0.33	0.30	0.27	4.64
	SLRb*	0.02	0.10	0.28	0.24	0.23	4.04
	interproc	0.05	0.15	0.31	0.29	0.22	2.81
	SpamAssassin	0.16	0.31	0.60	0.52	0.40	1.98
$\mathbb{OI}$	sendmail	0.24	0.30	0.60	0.51	0.38	1.26
	invertstring	0.20	0.30	0.57	0.43	0.35	1.51
	fibonacci*	0.05	0.13	0.26	0.24	0.16	2.79
	relax	0.19	0.31	0.62	0.47	0.38	1.66
	condsum*	0.05	0.15	0.33	0.33	0.30	2.96
	propertyInFlag*	0.05	0.12	0.32	0.30	0.27	2.57
	PfP*	0.03	0.13	0.32	0.29	0.26	3.93
	PfPb*	0.03	0.12	0.30	0.27	0.25	4.66
	PfPc*	0.04	0.15	0.34	0.33	0.28	4.09
$\mathbb{PS}$	liststatistics*	0.11	0.27	F	F	F	2.46
	harmonicMean*	0.14	0.29	F	F	F	2.12
	fraction	0.07	0.24	0.45	0.45	0.44	3.33
	SLR*	0.02	0.09	0.27	0.23	0.23	3.65
	facnegsum*	0.06	0.15	0.32	0.30	0.28	2.58
	SubMinSumDiv*	0.09	0.15	0.33	0.32	0.30	1.78
	lockfree3.0*	0.12	0.46	0.80	0.70	0.50	3.90
	testlocks7	1.15	1.26	1.46	1.32	0.98	0.85
	testlocks8	2.07	7.15	5.31	6.01	3.82	1.85
$\mathbb{OS}$	inf4*	0.32	0.28	0.57	0.53	0.27	0.84
	inf8	0.08	0.16	0.38	0.33	0.25	1.91
	propertyInFlag	0.05	0.06	0.24	0.21	0.19	1.25

$\mathbb{C}_P$	program	V_{PfP}	V_{RV}	V_{ARG}	V_{CPC}	$V_{\text{CPC}^{\text{RP}}}$	$\frac{\min V_{\text{PCC}}}{V_{\text{PfP}}}$
$\mathbb{OS}$	SLRb	0.03	0.04	0.21	0.20	0.16	1.47
	SubMinSumDiv*	0.08	0.09	0.25	0.22	0.20	1.04
	ESP	0.04	0.06	0.24	0.23	0.18	1.41
	ESPa	0.03	0.05	0.22	0.20	0.15	1.63
	ESPb	0.03	0.05	0.22	0.20	0.15	1.60
	ESPc	0.03	0.04	0.21	0.19	0.15	1.42
	addIteration*	0.03	0.04	0.19	0.18	0.16	1.21
$\mathbb{VS}$	SLRb*	0.03	0.04	0.19	0.19	0.15	1.79
	SLR*	0.02	0.05	0.20	0.19	0.15	1.95
	inf6*	0.03	0.08	0.23	0.23	0.18	2.27
	inf8*	0.07	0.19	0.32	0.29	0.25	2.72
	kundu*	0.36	0.58	1.17	0.97	0.62	1.63
	memslave1*	0.91	0.99	1.42	1.33	0.77	0.85
	memslave2	1.26	2.50	2.99	2.63	1.60	1.27
	PfPb*	0.03	0.04	0.20	0.20	0.17	1.64
	PfPc	0.04	0.11	0.27	0.24	0.23	2.81
	ESP	0.05	0.10	0.25	0.24	0.20	2.04
	ESPb*	0.03	0.06	0.21	0.20	0.17	1.82
	lockfree3.1*	0.12	0.22	0.38	0.32	0.31	1.85
$\mathbb{PSI}$	invertsorted*	0.67	1.53	F	F	1.63	2.27
	div	0.30	0.82	1.10	1.76	1.09	2.70
	fibonacci*	0.06	0.19	0.40	0.42	0.28	2.97
$\mathbb{OSI}$	fibonacci*	0.06	0.15	0.27	0.26	0.20	2.41
	palindrom	0.48	0.49	0.73	0.78	0.42	0.87
	invertarray	0.35	0.45	0.71	F	0.39	1.14
$\mathbb{PU}$	cdaudio	2.70	2.68	4.28	3.99	3.82	1.00
	pipeline2*	0.23	2.30	2.97	2.52	1.68	7.27
	diskperf	1.37	1.45	2.50	2.36	2.61	1.06
$\mathbb{OU}$	pipeline	1.15	1.25	2.34	2.18	1.79	1.08
	s3srvr	0.27	0.71	1.65	1.53	0.74	2.67
	pipeline2	260.19	7.13	8.81	7.88	6.76	0.03
$\mathbb{VU}$	s3srvr	0.62	1.12	1.91	1.79	1.28	1.79
	cdaudio	2.89	2.38	3.73	3.24	2.63	0.82
	pipeline2*	0.32	0.72	1.43	1.24	0.64	2.00
$\mathbb{PV}$	testlocks5*	0.30	1.25	1.44	2.18	1.32	4.22
	kbfiltr1*	0.41	0.79	1.26	1.19	0.84	1.90
	kbfiltr2*	0.77	1.23	1.82	1.78	1.32	1.60
	testlocks5d*	0.12	0.48	0.80	1.12	0.53	3.90
	testlocks12	8.23	156.46	51.51	TO	54.90	6.26
	testlocks12d	0.27	1.78	2.12	2.27	2.05	6.66
$\mathbb{OV}$	memslave1	0.35	0.84	1.12	1.09	0.64	1.83
	memslave2	0.59	0.91	1.33	1.30	0.82	1.39
	testlocks6	0.36	0.76	1.10	0.95	0.58	1.61
	kbfiltr1	0.56	0.76	1.24	1.15	0.75	1.33
	kbfiltr2	0.96	1.12	1.63	1.61	1.22	1.17
	inf8	0.11	0.22	0.40	0.36	0.33	1.94
$\widetilde{\mathbb{PV}}$	memslave1	0.47	2.43	3.07	3.64	2.70	5.12

$\mathbb{C}_P$	program	V_{PfP}	V_{RV}	V_{ARG}	V_{CPC}	$V_{\text{CPC}^{\text{RP}}}$	$\frac{\min V_{\text{PCC}}}{V_{\text{PfP}}}$
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave2	0.63	2.36	3.11	3.67	2.79	3.72
	kbfiltr2*	0.84	1.26	1.83	1.68	1.34	1.49
	testlocks6*	0.54	2.85	2.70	10.68	2.44	4.50
$\mathbb{O}\widetilde{\mathbb{V}}$	kbfiltr1	0.55	0.83	1.26	1.10	0.89	1.51
	testlocks5*	0.33	0.72	1.36	1.10	0.73	2.20
	testlocks5d*	0.12	0.48	0.83	0.89	0.32	2.79
Intermediate (Sep-Join)							
$\mathbb{PSI}$	fibonacci*	0.06	0.19	0.38	0.42	0.27	3.16
	palindrom*	0.48	1.02	1.44	1.99	0.73	1.54
	invertarray*	0.32	2.30	F	F	1.36	4.20
$\mathbb{OSI}$	invertsorted	0.81	0.73	1.17	0.96	F	0.90
	div	0.33	0.35	0.74	0.58	0.33	1.02
	palindrom	0.46	0.53	0.77	0.76	0.51	1.11
Model Checking							
$\mathbb{PL}$	locks*	0.01	0.11	0.27	0.26	0.22	7.72
	tokenring03	2.84	10.44	11.29	17.14	10.61	3.67
	memslave1*	4.16	9.04	14.43	17.77	F	2.17
	s3srvr	3.09	8.97	15.94	26.38	13.70	2.90
$\mathbb{OL}$	s3srvr	0.15	0.52	0.94	0.88	0.50	3.34
	transmitter01*	0.11	0.17	0.56	0.38	0.27	1.51
	transmitter02*	0.18	0.23	0.72	0.56	0.35	1.25
$\mathbb{VL}$	tokenring02	0.22	0.66	1.07	1.03	0.45	2.08
	tokenring04	43.05	4.40	4.59	5.65	3.12	0.07
	transmitter03	0.23	0.42	1.01	0.74	0.40	1.73
$\mathbb{PS}$	kundu*	0.60	1.74	2.27	2.84	F	2.87
	transmitter01*	0.18	0.39	0.97	F	0.51	2.20
	transmitter02	5.47	2.22	2.83	3.27	2.88	0.41
$\mathbb{OS}$	transmitter01	0.18	0.23	0.69	0.61	0.30	1.27
	transmitter02	0.31	0.50	0.99	0.88	0.41	1.32
	powerapprox	10.73	9.14	4.56	4.77	F	0.42
$\mathbb{VS}$	inf6*	0.03	0.08	0.23	0.23	0.18	2.37
	kundu*	0.14	0.66	1.21	0.92	0.56	4.08
	transmitter02*	0.21	0.30	0.66	0.57	0.42	1.42
$\mathbb{PU}$	pipeline*	0.22	0.67	1.87	1.67	F	3.07
	cdaudio	4.69	5.09	9.38	8.92	9.49	1.08
	diskperf	3.64	3.14	5.83	5.02	5.12	0.86
$\mathbb{OU}$	pipeline	1.18	1.14	2.27	2.13	1.74	0.97
	pipeline2	258.16	5.63	8.65	8.17	6.61	0.02
	s3srvr	0.28	0.56	1.65	1.59	0.72	2.02
$\mathbb{VU}$	s3srvr*	0.61	1.22	2.02	1.84	1.07	1.75
	cdaudio	3.63	3.31	3.96	3.47	2.77	0.76
	pipeline2*	0.31	0.70	1.40	1.26	0.67	2.15
$\mathbb{PV}$	testlocks5d*	0.60	2.17	2.04	5.10	0.93	1.54
	inf1*	0.07	0.19	0.38	0.39	0.31	2.54
$\mathbb{OV}$	testlocks5*	0.35	0.77	1.34	1.49	0.73	2.11
	testlocks12d*	0.26	0.79	1.29	1.13	0.66	2.52

$\mathbb{C}_P$	program	V_{PfP}	V_{RV}	V_{ARG}	V_{CPC}	$V_{\text{CPC}^{\text{RP}}}$	$\frac{\min V_{\text{PCC}}}{V_{\text{PfP}}}$
$\mathbb{OV}$	testlocks5d*	0.11	0.28	0.53	0.42	0.41	2.46
$\mathbb{P}\widetilde{\mathbb{V}}$	testlocks5*	0.65	2.19	2.08	5.04	1.37	2.11
	interproc*	0.11	0.20	0.45	0.45	0.35	1.88
	nosprintf*	0.15	0.29	F	F	0.48	1.98
$\mathbb{O}\widetilde{\mathbb{V}}$	testlocks5d*	0.11	0.27	0.67	0.42	0.33	2.43
	relax*	0.47	0.40	0.94	0.74	0.53	0.84
	nosprintf	0.14	0.44	0.42	0.39	0.35	2.50
	Sum	647.13	304.71	259.05	1124.01	208.48	
		68.47	272.70	224.74	1090.84	183.51	

Table B.8: Comparison of consumer's memory consumption in PfP and PCC approaches. Memory consumption, used heap plus used non-heap, is displayed in MB.

$\mathbb{C}_P$	program	M_{PfP}	M_{RV}	M_{ARG}	M_{CPC}	$M_{\text{CPC}^{\text{RP}}}$	$\frac{M_{\text{PfP}}}{\min M_{\text{PCC}}}$
	Dataflow Analysis						
$\mathbb{PI}$	nosprintf*	210.2	242.4	F	F	234.7	0.90
	interproc*	210	235.9	F	F	231.1	0.91
	NetBSD*	214.9	240.5	243.2	229.6	242.2	0.94
	PfPb*	205.9	232.1	231.1	231.2	227.6	0.90
	PfPc*	202.6	232.6	232.6	233.6	229.8	0.88
	SLRb*	204.9	233.8	237.2	229.1	231.1	0.89
$\mathbb{OI}$	interproc	210.3	217.3	220.3	219.4	228.9	0.97
	SpamAssassin	214.3	227.7	228.8	224.4	233.5	0.95
	sendmail	218.1	230.3	226.1	218.6	230.1	1.00
	invertstring	215.7	228.7	229.8	221.0	228.0	0.98
	fibonacci*	200.1	216.1	219.7	218.5	226.0	0.93
	relax	214.9	229.8	231.8	223.0	231.7	0.96
$\mathbb{PS}$	condsum*	201.3	233.3	236.4	232.5	233.5	0.87
	propertyInFlag*	199.3	234.4	232.3	230.8	229.6	0.87
	PfP*	199.6	235.4	234.6	229.7	231.7	0.87
	PfPb*	196.2	231.0	233.5	230.0	230.7	0.85
	PfPc*	197.6	234.1	233.2	232.8	232.6	0.85
	liststatistics*	205.9	243.5	F	F	F	0.85
	harmonicMean*	207.1	238.7	F	F	F	0.87
	fraction	201.4	238.7	239.4	234.7	235.2	0.86
	SLR*	198.6	229.3	232.5	232.0	227.7	0.87
	facnegsum*	202.4	234.3	231.6	232.5	230.4	0.88
	SubMinSumDiv*	203.8	234.2	235.4	234.0	232.9	0.88
	lockfree3.0*	202.3	241.5	242.5	242.2	242.2	0.84
$\mathbb{OS}$	testlocks7	265.0	281.9	241.9	234.3	248.9	1.13
	testlocks8	283.1	557.0	388.9	555.7	367.9	0.77
	inf4*	216.7	218.3	217.5	218.4	224.5	1.00
	inf8	201.4	217.4	217.6	214.7	221.6	0.94
	propertyInFlag	202.2	214.0	215.0	211.7	217.8	0.96
	SLRb	198.3	208.8	212.9	206.5	215.0	0.96
	SubMinSumDiv*	201.5	213.8	218.3	210.7	217.5	0.96
	ESP	205.2	209.3	216.3	206.4	214.4	0.99

$\mathbb{C}_P$	program	M_{PfP}	M_{RV}	M_{ARG}	M_{CPC}	$M_{\text{CPC}^{\text{RP}}}$	$\frac{M_{\text{PfP}}}{\min M_{\text{PCC}}}$
$\mathbb{OS}$	ESPa	201.0	213.8	209.6	208.4	212.3	0.96
	ESPb	201.4	207.1	213.1	206.1	212.0	0.98
	ESPc	203.2	207.9	208.7	205.8	214.3	0.99
	addIteration*	202.0	213.6	214.1	204.0	212.7	0.99
$\mathbb{VS}$	SLRb*	197.4	214.7	216.7	214.0	220.4	0.92
	SLR*	200.6	215.8	216.6	216.7	222.0	0.93
	inf6*	201.5	215.7	214.9	215.5	222.9	0.94
	inf8*	202.5	226.7	227.4	216.3	226.8	0.94
	kundu*	214.9	236.0	248.0	245.8	246.4	0.91
	memslave1*	273.2	268.0	266.2	264.7	260.8	1.05
	memslave2	296.1	341.9	325.5	366.0	297.2	1.00
	PfPb*	199.2	216.0	217.5	217.4	225.3	0.92
	PfPc	199.7	214.8	218.0	218.9	232.4	0.93
	ESP	202.1	214.1	218.3	218.5	228.3	0.94
	ESPb*	202.1	217.1	223.3	213.6	224.5	0.95
	lockfree3.1*	204.0	228.3	234.9	224.2	235.4	0.91
$\mathbb{PSI}$	invertsorted*	233.8	271.6	F	F	252.3	0.93
	div	215.2	244.6	248.0	265.0	248.3	0.88
	fibonacci*	205.4	237.1	239.2	236.5	231.9	0.89
$\mathbb{OSI}$	fibonacci*	206.5	223.8	220.7	219.0	225.3	0.94
	palindrom	221.0	230.1	226.6	228.3	235.3	0.98
	invertarray	218.0	229.4	232.9	F	236.9	0.95
$\mathbb{PU}$	cdaudio	345.7	382.4	432.5	399.3	402.9	0.90
	pipeline2*	211.5	344.8	309.3	296.6	277.4	0.76
	diskperf	269.9	294.8	320.2	330.5	319.6	0.92
$\mathbb{OU}$	pipeline	256.1	272.9	276.7	267.7	266.9	0.96
	s3srvr	215.3	246.8	244.8	235.8	244.4	0.91
	pipeline2	3971.2	552.8	622.9	575.2	557.8	7.18
$\mathbb{VU}$	s3srvr	236.5	280.2	265.3	262.0	263.9	0.90
	cdaudio	344.8	378.8	400.5	393.0	344.4	1.00
	pipeline2*	221.5	244.8	254.8	242.6	251.6	0.91
$\mathbb{PV}$	testlocks5*	214.6	261.3	263.1	270.5	257.3	0.83
	kbfiltr1*	230.5	259.3	271.7	261	257.1	0.90
	kbfiltr2*	244.2	281.6	295.5	280.5	274.2	0.89
	testlocks5d*	215.5	247.6	249.6	244.3	240.4	0.90
	testlocks12	591.8	1548.0	1602.6	TO1243.1	1537.9	0.48
	testlocks12d	219.0	297.9	275.4	288.0	272.0	0.81
$\mathbb{OV}$	memslave1	226.8	256.5	255.6	257.5	266.5	0.89
	memslave2	229.6	259.5	272.1	275.1	256.7	0.89
	testlocks6	226.4	240.3	236.4	233.8	245.2	0.97
	kbfiltr1	234.3	247.2	253.8	247.8	257.6	0.95
	kbfiltr2	251.6	265.4	279.6	264.2	274.9	0.95
	inf8	214.7	223.4	229.2	222.2	232.1	0.97
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	233.7	325.7	304.6	333.1	310.4	0.77
	memslave2	241.2	337.3	306.3	332.7	313.7	0.79
	kbfiltr2*	250.6	281.7	294.8	284.0	274.8	0.91
	testlocks6*	229.8	307.2	289.8	544.4	292.7	0.79

$\mathbb{C}_P$	program	M_{PfP}	M_{RV}	M_{ARG}	M_{CPC}	$M_{\text{CPC}^{\text{RP}}}$	$\frac{M_{\text{PfP}}}{\min M_{\text{PCC}}}$
$\mathbb{O}\widetilde{\mathbb{V}}$	kbfiltr1	236.9	249.2	251.4	248.8	261	0.95
	testlocks5*	221.1	251.5	242.1	243.9	251.6	0.91
	testlocks5d*	214.1	229.7	237.3	230.0	236.2	0.93
Intermediate (Sep-Join)							
$\mathbb{PSI}$	fibonacci*	206.4	237.8	237.6	233.3	232.6	0.89
	palindrom*	221.2	251.4	253.4	278.5	243.0	0.91
	invertarray*	217.9	259.0	F	F	240.4	0.91
$\mathbb{OSI}$	invertsorted	237.7	241.2	234.7	228.6	F	1.04
	div	214.7	228.8	230.0	226.5	230.4	0.95
	palindrom	217.1	227.2	230.4	230.6	237	0.96
Model Checking							
$\mathbb{PL}$	locks*	199.7	230.3	230.1	230.5	230.1	0.87
	tokenring03	299.2	565.0	538.5	563.9	516.0	0.58
	memslave1*	368.8	564.9	565.9	547.5	F	0.67
	s3srvr	340.1	499.0	565.0	563.1	564.5	0.68
$\mathbb{OL}$	s3srvr	237.1	253.8	257.3	251.5	260.6	0.94
	transmitter01*	207.5	224.9	221.6	222.4	227.0	0.94
	transmitter02*	214.5	225.6	225.5	219.1	225.6	0.98
$\mathbb{VL}$	tokenring02	213.5	244.3	236.6	234.1	240.5	0.91
	tokenring04	537.3	545.5	546.4	572.1	426.1	1.26
	transmitter03	215	235.9	238.1	233.4	239.3	0.92
$\mathbb{PS}$	kundu*	228.5	284.6	290.2	286.1	F	0.80
	transmitter01*	206.1	242.9	248.8	F	245.6	0.85
	transmitter02	553.1	304.2	314.7	300.3	293.8	1.88
$\mathbb{OS}$	transmitter01	205.2	219.7	225.8	221.4	230.1	0.93
	transmitter02	214.4	230.7	230.8	233.2	233.7	0.93
	powerapprox	996.8	1077.9	360.0	552.6	F	2.77
$\mathbb{VS}$	inf6*	204.1	215.3	221.2	218.8	227.9	0.95
	kundu*	202.6	240.5	247.8	236.2	250.0	0.86
	transmitter02*	206.2	235.1	232.5	230.8	239.4	0.89
$\mathbb{PU}$	pipeline*	215.7	259.1	301.8	302.1	F	0.83
	cdaudio	449.8	483.7	568.1	566.7	565.4	0.93
	diskperf	359.1	378.4	455.6	401.2	421.8	0.95
$\mathbb{OU}$	pipeline	260.2	267.6	277.3	264.5	271.3	0.98
	pipeline2	3903.0	536.9	616.0	570.2	553.6	7.27
	s3srvr	218.3	271.5	247.3	240.4	251.1	0.91
$\mathbb{VU}$	s3srvr*	231.9	272.4	275.1	274.9	270.0	0.86
	cdaudio	380.0	377.0	420.2	394.6	361.2	1.05
	pipeline2*	220.2	241.3	251.4	241.6	249.7	0.91
$\mathbb{PV}$	testlocks5d*	231.9	313.9	276.5	299.8	248.7	0.93
	inf1*	214.9	238.9	240.9	236.2	237.2	0.91
$\mathbb{OV}$	testlocks5*	224	242.8	241.2	244.5	247.5	0.93
	testlocks12d*	217.8	249.5	244	241.5	251.6	0.90
	testlocks5d*	216.1	225.7	230.5	226.8	237	0.96
$\mathbb{P}\widetilde{\mathbb{V}}$	testlocks5*	230.8	312.8	287.4	310.3	260.5	0.89
	interproc*	213.2	237.8	245.6	240.7	237.7	0.90

$\mathbb{C}_P$	program	M_{PfP}	M_{RV}	M_{ARG}	M_{CPC}	$M_{\text{CPC}^{\text{RP}}}$	$\frac{M_{\text{PfP}}}{\min M_{\text{PCC}}}$
$\mathbb{P}\widetilde{\mathbb{V}}$	nosprintf*	216.3	F	F	240.2	245.1	0.90
	testlocks5d*	217.5	225.6	228.7	231.1	238.7	0.96
$\mathbb{O}\widetilde{\mathbb{V}}$	relax*	230.4	232.4	242.0	242.9	244.4	0.99
	nosprintf	216.8	230.7	233.0	231.9	238.3	0.94
	Sum	38043.1	36097.2	35728.1	35590.5	34857.9	
		27910	32628.3	32762.1	32532.8	32270.1	

B.4 Results Integration of PfP and CPC

Table B.9: Comparison of the verification and highly reduced, partitioned certificate validation on the program generated by the producer in the Programs from Proofs approach. Next to the validation times and the memory consumption, for the verification also the number of merges and the ratio of computed transfer successors to the size of the reached set are provided. All times are given in seconds and memory consumption, used heap plus used non-heap, is represented in MB.

$\mathbb{C}_P$	program	$\#\sqcup$	$\frac{\#\text{suc}}{\lvert N\rvert}$	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
				Dataflow Analysis					
$\mathbb{P}\mathbb{I}$	nosprintf*	0	0.36	0.07	0.15	0.46	210.2	227.1	1.08
	interproc*	0	0.30	0.05	0.14	0.36	210.0	218.0	1.04
	NetBSD*	64	1.11	0.08	0.15	0.53	214.9	225.1	1.05
	PfPb*	0	0.27	0.02	0.13	0.18	205.9	215.0	1.04
	PfPc*	1	0.29	0.03	0.14	0.23	202.6	221.7	1.09
	SLRb*	1	0.40	0.02	0.13	0.19	204.9	221.8	1.08
$\mathbb{O}\mathbb{I}$	interproc	0	0.25	0.05	0.15	0.36	210.3	219.3	1.04
	SpamAssassin	1	0.30	0.16	0.26	0.61	214.3	226.6	1.06
	sendmail	71	0.52	0.24	0.32	0.76	218.1	230.7	1.06
	invertstring	42	0.46	0.20	0.31	0.63	215.7	227.7	1.06
	fibonacci*	0	0.35	0.05	0.13	0.35	200.1	221.8	1.11
	relax*	7	0.37	0.19	0.35	0.53	214.9	228.6	1.06
$\mathbb{P}\mathbb{S}$	condsum*	4	0.46	0.05	0.15	0.33	201.3	215.9	1.07
	propertyInFlag*	2	0.44	0.05	0.15	0.31	199.3	211.3	1.06
	PfP*	0	0.34	0.03	0.14	0.23	199.6	206.7	1.04
	PfPb*	0	0.26	0.03	0.13	0.21	196.2	211.0	1.08
	PfPc*	1	0.28	0.04	0.14	0.27	197.6	212.5	1.08
	liststatistics*	27	0.47	0.11	0.18	0.63	205.9	215.7	1.05
	harmonicMean*	50	0.47	0.14	0.18	0.76	207.1	217.0	1.05
	fraction	1	0.37	0.07	0.18	0.39	201.4	214.3	1.06
	SLR*	1	0.37	0.02	0.13	0.19	198.6	204.4	1.03
	facnegsum*	14	0.72	0.06	0.14	0.40	202.4	210.6	1.04
	SubMinSumDiv*	21	0.83	0.09	0.16	0.53	203.8	214.0	1.05
	lockfree3.0*	49	0.27	0.12	0.17	0.67	202.3	214.2	1.06
$\mathbb{O}\mathbb{S}$	testlocks7	134	0.36	1.15	1.32	0.87	265.0	259.4	0.98
	testlocks8	0	0.19	2.07	1.99	1.04	283.1	297.0	1.05
	inf4*	27	0.31	0.32	0.35	0.91	216.7	221.3	1.02
	inf8	4	0.32	0.08	0.19	0.44	201.4	215.4	1.07

$\mathbb{C}_P$	program	$\#\sqcup$	$\frac{\#suc}{\lvert N\rvert}$	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
$\mathbb{OS}$	propertyInFlag	2	0.41	0.05	0.16	0.31	202.2	212.5	1.05
	SLRb	1	0.33	0.03	0.14	0.20	198.3	208.8	1.05
	SubMinSumDiv*	22	0.73	0.08	0.17	0.50	201.5	217.2	1.08
	ESP	3	0.43	0.04	0.16	0.29	205.2	209.3	1.02
	ESPa	1	0.32	0.03	0.14	0.22	201.0	209.2	1.04
	ESPb	1	0.32	0.03	0.14	0.22	201.4	205.7	1.02
	ESPc	1	0.36	0.03	0.14	0.23	203.2	205.0	1.01
	addIteration*	1	0.49	0.03	0.14	0.23	202.0	211.5	1.05
$\mathbb{VS}$	SLRb*	1	0.38	0.03	0.13	0.19	197.4	212.0	1.07
	SLR*	1	0.35	0.02	0.13	0.18	200.6	205.4	1.02
	inf6*	1	0.25	0.03	0.13	0.26	201.5	211.1	1.05
	inf8*	4	0.33	0.07	0.16	0.42	202.5	213.9	1.06
	kundu*	60	0.29	0.36	0.41	0.87	214.9	229.3	1.07
	memslave1*	1744	0.51	0.91	0.58	1.56	273.2	241.1	0.88
	memslave2	2261	0.19	1.26	0.78	1.62	296.1	243.8	0.82
	PfPb*	0	0.28	0.03	0.13	0.19	199.2	208.8	1.05
	PfPc	0	0.29	0.04	0.15	0.25	199.7	213.3	1.07
	ESP	3	0.45	0.05	0.19	0.25	202.1	208.1	1.03
	ESPb*	1	0.36	0.03	0.14	0.25	202.1	206.9	1.02
	lockfree3.1*	38	0.53	0.12	0.17	0.69	204.0	210.3	1.03
$\mathbb{PSI}$	invertsorted*	41	0.52	0.67	0.71	0.95	233.8	238.1	1.02
	div	21	0.52	0.30	0.39	0.77	215.2	226.5	1.05
	fibonacci*	0	0.38	0.06	0.15	0.43	205.4	223.0	1.09
$\mathbb{OSI}$	fibonacci*	0	0.35	0.06	0.15	0.42	206.5	224.2	1.09
	palindrom	21	0.45	0.48	0.58	0.82	221.0	235.7	1.07
	invertarray	0	0.37	0.35	0.43	0.82	218.0	228.4	1.05
$\mathbb{PU}$	cdaudio	149	0.53	2.70	2.79	0.97	345.7	354.2	1.02
	pipeline2*	96	0.31	0.23	0.32	0.73	211.5	226.0	1.07
	diskperf	22	0.53	1.37	1.55	0.88	269.9	283.4	1.05
$\mathbb{OU}$	pipeline	320	0.50	1.15	1.32	0.87	256.1	264.7	1.03
	s3srvr	6	0.34	0.27	0.36	0.75	215.3	229.3	1.07
	pipeline2	993k	13.60	Proof Construction Failed					
$\mathbb{VU}$	s3srvr	9	0.47	0.62	0.72	0.87	236.5	248.6	1.05
	cdaudio	190	0.51	2.89	2.90	1.00	344.8	351.9	1.02
	pipeline2*	163	0.55	0.32	0.42	0.76	221.5	231.7	1.05
$\mathbb{PV}$	testlocks5*	1	0.29	0.29	0.39	0.76	214.6	235.8	1.10
	kbfiltr1*	13	0.37	0.41	0.44	0.95	230.5	241.7	1.05
	kbfiltr2*	32	0.42	0.77	0.70	1.10	244.2	260.1	1.07
	testlocks5d*	1	0.16	0.12	0.23	0.54	215.5	228.1	1.06
	testlocks12	1	0.26	8.23	8.16	1.01	591.8	557.8	0.94
	testlocks12d	1	0.15	0.27	0.37	0.72	219.0	235.5	1.08
$\mathbb{OV}$	memslave1	19	0.20	0.35	0.32	1.09	226.8	238.6	1.05
	memslave2	255	0.16	0.59	0.29	2.03	229.6	233.2	1.02
	testlocks6	63	0.28	0.36	0.45	0.80	226.4	239.4	1.06
	kbfiltr1	13	0.36	0.56	0.54	1.03	234.3	244.4	1.04
	kbfiltr2	32	0.40	0.96	0.92	1.04	251.6	266.3	1.06
	inf8	4	0.33	0.11	0.22	0.51	214.7	228.9	1.07

$\mathbb{C}_P$	program	#⊔	$\frac{\#\text{suc}}{\lvert N\rvert}$	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
$\mathbb{P}\widetilde{\mathbb{V}}$	memslave1	114	0.09	0.47	0.34	1.39	233.7	241.7	1.03
	memslave2	256	0.15	0.63	0.34	1.88	241.2	243.5	1.01
	kbfiltr2*	34	0.43	0.84	0.71	1.19	250.6	257.9	1.03
	testlocks6*	64	0.22	0.54	0.60	0.91	229.8	247.5	1.08
$\mathbb{O}\widetilde{\mathbb{V}}$	kbfiltr1	14	0.33	0.55	0.52	1.05	236.9	247.4	1.04
	testlocks5*	0	0.22	0.33	0.42	0.78	221.1	238.4	1.08
	testlocks5d*	0	0.15	0.12	0.21	0.55	214.1	227.5	1.06
Intermediate (Sep-Join)									
$\mathbb{PSI}$	fibonacci*	0	0.38	0.06	0.15	0.40	206.4	224.0	1.09
	palindrom*	21	0.48	0.48	0.57	0.83	221.2	238.0	1.08
	invertarray*	0	0.37	0.32	0.39	0.82	217.9	230.4	1.06
$\mathbb{OSI}$	invertsorted	40	0.50	0.81	0.80	1.02	237.7	242.8	1.02
	div	21	0.45	0.33	0.40	0.82	214.7	230.5	1.07
	palindrom	21	0.45	0.46	0.56	0.82	217.1	232.2	1.07
Model Checking									
$\mathbb{PL}$	locks*	0	0.35	0.01	0.11	0.13	199.7	209.8	1.05
	tokenring03	0	0.38	2.84	3.01	0.95	299.2	302.4	1.01
	memslave1*	0	0.36	4.16	4.24	0.98	368.8	367.3	1.00
	s3srvr	0	0.50	3.09	3.06	1.01	340.1	351.6	1.03
$\mathbb{OL}$	s3srvr	0	0.31	0.15	0.22	0.69	237.1	250.3	1.06
	transmitter01*	0	0.45	0.11	0.19	0.58	207.5	223.4	1.08
	transmitter02*	0	0.51	0.18	0.28	0.66	214.5	224.8	1.05
$\mathbb{VL}$	tokenring02	0	0.21	0.22	0.29	0.74	213.5	224.5	1.05
	tokenring04	0	0.10	43.05	42.27	1.02	537.3	547.5	1.02
	transmitter03	0	0.43	0.23	0.33	0.69	215.0	236.4	1.10
$\mathbb{PS}$	kundu*	85	1.09	0.60	0.76	0.80	228.5	241.3	1.06
	transmitter01*	106	0.30	0.18	0.21	0.86	206.1	218.3	1.06
	transmitter02	18465	2.37	5.47	1.36	4.02	553.1	253.7	0.46
$\mathbb{OS}$	transmitter01	88	0.29	0.18	0.23	0.78	205.2	216.2	1.05
	transmitter02	120	0.25	0.31	0.30	1.03	214.4	220.6	1.03
	powerapprox	1022	0.51	10.73	7.39	1.45	996.8	553.7	0.56
$\mathbb{VS}$	inf6*	1	0.24	0.03	0.16	0.22	204.1	210.8	1.03
	kundu*	7	0.10	0.14	0.26	0.53	202.6	221.1	1.09
	transmitter02*	114	0.39	0.21	0.34	0.63	206.2	218.6	1.06
$\mathbb{PU}$	pipeline*	1	0.50	0.22	0.34	0.65	215.7	223.8	1.04
	cdaudio	203	0.48	4.69	5.07	0.93	449.8	456.8	1.02
	diskperf	53	0.48	3.64	4.00	0.91	359.1	372.4	1.04
$\mathbb{OU}$	pipeline	461	0.56	1.18	1.32	0.89	260.2	260.3	1.00
	pipeline2	993k	12.26	Proof Construction Failed					
	s3srvr	7	0.31	0.28	0.38	0.73	218.3	233.1	1.07
$\mathbb{VU}$	s3srvr*	16	0.34	0.61	0.70	0.87	231.9	245.9	1.06
	cdaudio	188	0.56	3.63	3.69	0.98	380.0	381.7	1.00
	pipeline2*	155	0.46	0.31	0.39	0.79	220.2	231.4	1.05
$\mathbb{PV}$	testlocks5d*	0	0.34	0.60	0.68	0.88	231.9	244.8	1.06
	inf1*	3	0.27	0.07	0.18	0.42	214.9	228.9	1.07
$\mathbb{OV}$	testlocks5*	0	0.29	0.35	0.42	0.82	224.0	237.9	1.06
	testlocks12d*	0	0.25	0.26	0.34	0.76	217.8	229.9	1.06

$\mathbb{C}_P$	program	$\#\sqcup$	$\frac{\#\text{suc}}{\lvert N\rvert}$	$V_{\mathbb{C}}$	$V_{\mathbb{V}}$	$\frac{V_{\mathbb{C}}}{V_{\mathbb{V}}}$	$M_{\mathbb{C}}$	$M_{\mathbb{V}}$	$\frac{M_{\mathbb{V}}}{M_{\mathbb{C}}}$
$\mathbb{OV}$	testlocks5d*	0	0.21	0.11	0.21	0.54	216.1	225.6	1.04
	testlocks5*	0	0.36	0.65	0.67	0.97	230.8	243.9	1.06
$\mathbb{P}\widetilde{\mathbb{V}}$	interproc*	0	0.27	0.11	0.22	0.47	213.2	229.8	1.08
	nosprintf*	0	0.33	0.15	0.28	0.51	216.3	231.0	1.07
	testlocks5d*	0	0.21	0.11	0.21	0.52	217.5	224.4	1.03
$\mathbb{O}\widetilde{\mathbb{V}}$	relax*	3	0.37	0.47	0.55	0.87	230.4	241.4	1.05
	nosprintf	0	0.34	0.14	0.26	0.54	216.8	226.3	1.04
	Sum			128.78	129.08		30168.9	30646.2	

Bibliography

[AAH12] Elvira Albert, Puebla Arenas, and Manuel Hermenegildo. Certificate size reduction in abstraction-carrying code. *Theory and Practice of Logic Programming*, 12(3):283–318, 2012.

[AAP06] Elvira Albert, Puri Arenas, and Germán Puebla. An incremental approach to abstraction-carrying code. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 377–391, Berlin, Heidelberg, 2006. Springer.

[AAPH06] Elvira Albert, Puri Arenas, Germán Puebla, and Manuel Hermenegildo. Reduced certificates for abstraction-carrying code. In Sandro Etalle and Mirosław Truszczyński, editors, *Logic Programming*, volume 4079 of *Lecture Notes in Computer Science*, pages 163–178, Berlin, Heidelberg, 2006. Springer.

[ACAE08] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Automatic certification of java source code in rewriting logic. In Stefan Leue and Pedro Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 200–217, Berlin, Heidelberg, 2008. Springer.

[ACAE09] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Automated certification of non-interference in rewriting logic. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 182–198, Berlin, Heidelberg, 2009. Springer.

[ACAE10] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Abstract certification of global non-interference in rewriting logic. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 105–124, Berlin, Heidelberg, 2010. Springer.

[ADZ+12] Torben Amtoft, Josiah Dodds, Zhi Zhang, Andrew Appel, Lennart Beringer, John Hatcliff, Xinming Ou, and Andrew Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, volume 7215 of *Lecture Notes in Computer Science*, pages 369–389, Berlin, Heidelberg, 2012. Springer.

[AF99] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, CCS '99, pages 52–62, New York, NY, USA, 1999. ACM.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 243–253, New York, NY, USA, 2000. ACM.

[AFPP14] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Program verification via iterated specialization. *Science of Computer Programming*, 95, Part 2:149–175, 2014.

[AGH+05] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–26, Berlin, Heidelberg, 2005. Springer.

[AL98] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 72–84, New York, NY, USA, 1998. ACM.

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, & tools*. Pearson Addison-Wesley, Boston, 2. ed. edition, 2007.

[AMA07] Wolfram Amme, Marc-André Möller, and Philipp Adler. Data flow analysis as a general concept for the transport of verifiable program annotations. *Electronic Notes in Theoretical Computer Science*, 176(3):97–108, 2007.

[APH05a] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. An abstract interpretation-based approach to mobile code safety. *Electronic Notes in Theoretical Computer Science*, 132(1):113–129, 2005.

[APH05b] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. Abstraction-carrying code. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397, Berlin, Heidelberg, 2005. Springer.

[APH08] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. Abstraction-carrying code: a model for mobile code safety. *New Generation Computing*, 26(2):171–204, 2008.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256, 2001.

[AR04] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[AS13] Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 25–42, Berlin, Heidelberg, 2013. Springer.

[ASS+16] Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. Efficiently intertwining widening and narrowing. *Science of Computer Programming*, 120:1–24, 2016.

[ASV13] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 377–386, New York, NY, USA, 2013. ACM.

[Bar12] John Barnes. *SPARK - The Proven Appraoch to High Integrity Software*. Altran Praxis Ltd, 2012.

[Bas91] David Basin. Extracting circuits from constructive proofs. In *Proceedings of ACM 1991 International Workshop on Formal Methods in VLSI Design*, 1991.

[BBS+98] Benl, Berger, Schwichtenberg, Seisenberger, and Zuber. Proof theory at work: Program development in the minlog system. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 of *Applied Logic Series*, pages 41–71, Dordrecht, 1998. Springer Netherlands.

[BBY14] Sandrine Blazy, David Bühler, and Boris Yakobowski. Improving static analyses of c programs with conditional predicates. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems*, volume 8718 of *Lecture Notes in Computer Science*, pages 140–154, Cham, 2014. Springer International Publishing.

[BBY16] Sandrine Blazy, David Bühler, and Boris Yakobowski. Improving static analyses of c programs with conditional predicates. *Science of Computer Programming*, 118:77–95, 2016.

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[BCH+04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast query language for software verification. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18, Berlin, Heidelberg, 2004. Springer.

[BCJ14] Frédéric Besson, Pierre-Emmanuel Cornilleau, and Thomas Jensen. Result certification of static program analysers with automated theorem provers. In *Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 304–325, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDD+15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 721–733, New York, NY, USA, 2015. ACM.

[BDDH16] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 326–337, New York, NY, USA, 2016. ACM.

[Ber03] Stefan Berghofer. Program extraction in simply-typed higher order logic. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 21–38, Berlin, Heidelberg, 2003. Springer.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *Lecture Notes in Computer Science*, pages 887–904, Berlin, Heidelberg, 2016. Springer.

[Bf15] Oliver Burn (founder). Checkstyle, access 07-28-2015. `http://checkstyle.sourceforge.net/`.

[BGKR06] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. In Kwangkeun Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317, Berlin, Heidelberg, 2006. Springer.

[BGKR09] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. *ACM Transactions on Programming Languages and Systems*, 31(5):18:1–18:45, July 2009.

[BGP08] Gilles Barthe, Benjamin Grégoire, and Mariela Pavlova. Preservation of proof obligations from java to the java virtual machine. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 83–99, Berlin, Heidelberg, 2008. Springer.

[BGS97] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering — ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, pages 361–377, Berlin, Heidelberg, 1997. Springer.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[BHKW12] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 57:1–57:11, New York, NY, USA, 2012. ACM.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 300–309, New York, NY, USA, 2007. ACM.

[BHRZ05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1):28–56, 2005.

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 532–546, Berlin, Heidelberg, 2006. Springer.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518, Berlin, Heidelberg, 2007. Springer.

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2008, pages 29–38. IEEE, 2008.

[BHTZ10] Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, and Damien Zufferey. Shape refinement through explicit heap analysis. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 263–277, Berlin, Heidelberg, 2010. Springer.

[BJP06] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006. Applied Semantics.

[BJT07] Frédéric Besson, Thomas Jensen, and Tiphaine Turpin. Small witnesses for abstract interpretation-based proofs. In Rocco Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 268–283, Berlin, Heidelberg, 2007. Springer.

[BK08] Gilles Barthe and César Kunz. Certificate translation in abstract interpretation. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382, Berlin, Heidelberg, 2008. Springer.

[BK11a] Gilles Barthe and César Kunz. An abstract model of certificate translation. *ACM Transactions on Programming Languages and Systems*, 33(4):13:1–13:46, July 2011.

[BK11b] Dirk Beyer and Mehmet Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190, Berlin, Heidelberg, 2011. Springer.

[BKKN10] Jörg Brauer, Volker Kamin, Stefan Kowalewski, and Thomas Noll. Loop refinement using octagons and satisfiability. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[BKW10] Dirk Beyer, Mehmet Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Formal Methods in Computer-Aided Design*, pages 189–197, Oct 2010.

[BL13] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162, Berlin, Heidelberg, 2013. Springer.

[BLN+13] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision reuse for efficient regression verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 389–399, New York, NY, USA, 2013. ACM.

[BLW15a] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In Bernd Fischer and Jaco Geldenhuys, editors, *Model Checking Software*, volume 9232 of *Lecture Notes in Computer Science*, pages 160–178, Cham, 2015. Springer International Publishing.

[BLW15b] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In Bernd Fischer and Jaco Geldenhuys, editors, *Model Checking Software*, volume 9232 of *Lecture Notes in Computer Science*, pages 20–38, Cham, 2015. Springer International Publishing.

[BLW15c] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Sliced path prefixes: An effective method to enable refinement selection. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 228–243, Cham, 2015. Springer International Publishing.

[BMMIM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[BN02] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40, Berlin, Heidelberg, 2002. Springer.

[BR01] Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 97–103, New York, NY, USA, 2001. ACM.

[BR02a] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[BR02b] Thomas Ball and Sriram K. Rajamani. SLIC: a specification language for interface checking (of c). TechReport MSR-TR-2001-21, Microsoft Research, 2002.

[Brö16] Henrik Bröcher. Evaluation von Graphpartitionierungsalgorithmen im Kontext von Konfigurierbarer Softwarezertifizierung. Bachelor thesis, Universität Paderborn, 2016.

[BRS06] Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In Theo Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126, Berlin, Heidelberg, 2006. Springer.

[BSI+08] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, Ou Wei, and Aarti Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In María Alpuente and Germán Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 238–254, Berlin, Heidelberg, 2008. Springer.

[BSIG09] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Refining the control structure of loops using static analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 49–58, New York, NY, USA, 2009. ACM.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium*

on Principles of Programming Languages, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[CC95] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In Pierre Wolper, editor, *Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 293–308, Berlin, Heidelberg, 1995. Springer.

[CCF+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300, Berlin, Heidelberg, 2007. Springer.

[CCH00] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 38(1):27–71, 2000.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distrubuted Systems Concepts and Design*. Addison Wesley, fourth edition, 2005.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Berlin, Heidelberg, 2000. Springer.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP02] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, Cambridge, Mass., 4. print. edition, 2002.

[CGR07] Patrick Cousot, Pierre Ganty, and Jean-François Raskin. Fixpoint-guided abstraction refinements. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 333–348, Berlin, Heidelberg, 2007. Springer.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107, Berlin, Heidelberg, 2013. Springer.

[Cha06a] Amine Chaieb. Proof-producing program analysis. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *Theoretical Aspects of Computing - ICTAC 2006*, volume 4281 of *Lecture Notes in Computer Science*, pages 287–301, Berlin, Heidelberg, 2006. Springer.

[Cha06b] Sagar Chaki. Sat-based software certification. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 151–166, Berlin, Heidelberg, 2006. Springer.

[Cha12] Kaustuv Chaudhuri. Compact proof certificates for linear logic. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 208–223, Berlin, Heidelberg, 2012. Springer.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254, Berlin, Heidelberg, 2012. Springer.

[CKV11] Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 333–348, Berlin, Heidelberg, 2011. Springer.

[CLCVH94] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 227–239, New York, NY, USA, 1994. ACM.

[CLRC07] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Stein Clifford. *Introduction to Algorithms Second Edition*. The MIT Press, Cambridge, Massachusetts, second edition, 2007. 8th printing.

[CM06] Robert Constable and Wojciech Moczydłowski. Extracting programs from constructive hol proofs via IZF set-theoretic semantics. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 162–176, Berlin, Heidelberg, 2006. Springer.

[CML13] Hong Yi Chen, Supratik Mukhopadhyay, and Zheng Lu. Control flow refinement and symbolic computation of average case bound. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis*, volume 8172 of *Lecture Notes in Computer Science*, pages 334–348, Cham, 2013. Springer International Publishing.

[CMZ15] Sylvain Conchon, Alain Mebsout, and Fatiha Zaïdi. Certificates for parameterized model checking. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 126–142, Cham, 2015. Springer International Publishing.

[Cor13] Pierre-Emmanuel Cornilleau. *Certification of static analysis in many-sorted first-order logic*. Theses, École normale supérieure de Cachan - ENS Cachan, March 2013.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Commununications of the ACM*, 54(5):88–98, May 2011.

[CSV07] Sagar Chaki, Christian Schallhart, and Helmut Veith. Verification across intellectual property boundaries. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 82–94, Berlin, Heidelberg, 2007. Springer.

[CSV13] Sagar Chaki, Christian Schallhart, and Helmut Veith. Verification across intellectual property boundaries. *ACM Transactions on Software Engineering and Methodology*, 22(2):15:1–15:12, March 2013.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 184–198, New York, NY, USA, 2000. ACM.

[DADY04] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 12–22, New York, NY, USA, 2004. ACM.

[DDY06] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-sensitive dataflow analysis with iterative refinement. In Kwangkeun Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 425–442, Berlin, Heidelberg, 2006. Springer.

[DFNP14] Emanuele De Angelis, Fabio Fioravanti, Jorge A. Navas, and Maurizio Proietti. Verification of programs by combining iterated specialization with interpolation. In Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko, and Valerio Senni, editors, *Proceedings First Workshop on Horn Clauses for Verification and Synthesis*, volume 169 of *EPTCS*, pages 3–18, 2014.

[DFPP13a] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verification of imperative programs by constraint logic program transformation. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, volume 129 of *EPTCS*, pages 186–210, 2013.

[DFPP13b] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verification of imperative programs through transformation of constraint logic programs. In Alexei Lisitsa and Andrei P. Nemytykh, editors, *First International Workshop on Verification and Program Transformation*, volume 16 of *EPiC Series in Computing*, pages 30–41, 2013.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DKFW10] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 271–274, Berlin, Heidelberg, 2010. Springer.

[DKP09] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 189–194, Dec 2009.

[DKP10] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. Proof-carrying hardware: Concept and prototype tool flow for online verification. *International Journal of Reconfigurable Computing*, 2010(180242), 2010.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[DM77] Nachum Dershowitz and Zohar Manna. The evolution of programs: A system for automatic program modification. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 144–154, New York, NY, USA, 1977. ACM.

[DP90] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.

[DZB14] Sun Ding, Hongyu Zhang, and Hee Beng Kuan Tan. Detecting infeasible branches based on code patterns. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 74–83, Feb 2014.

[Erl05] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[Fer14] Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *Lecture Notes in Computer Science*, pages 302–321, Berlin, Heidelberg, 2014. Springer.

[Fer16] Pietro Ferrara. A generic framework for heap and value analyses of object-oriented programming languages. *Theoretical Computer Science*, 631:43–72, 2016.

[FFJ12] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. Tval+ : Tvla and value analyses together. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 63–77, Berlin, Heidelberg, 2012. Springer.

[FHS09] Ansgar Fehnker, Ralf Huuck, and Sean Seefried. Incremental false path elimination for static software analysis. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 255–270, Berlin, Heidelberg, 2009. Springer.

[FHS10] Ansgar Fehnker, Ralf Huuck, and Sean Seefried. Counterexample guided path reduction for static program analysis. In Dennis Dams, Ulrich Hannemann, and Martin Steffen, editors, *Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 322–341, Berlin, Heidelberg, 2010. Springer.

[FJM05] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 227–236, New York, NY, USA, 2005. ACM.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of AMS Symposium on Applied Mathematics*, volume 19, pages 19–31, Providence R.I., 1967. American Mathematical Society.

[FM82] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Conference on Design Automation*, pages 175–181, June 1982.

[FMN15] Pietro Ferrara, Peter Müller, and Milos Novacek. Automatic inference of heap properties exploiting value domains. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 8931 of *Lecture Notes in Computer Science*, pages 393–411, Berlin, Heidelberg, 2015. Springer.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 286.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[FOW84] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 125–132, Berlin, Heidelberg, 1984. Springer.

[FPP05] Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Automatic proofs of protocols via program transformation. In *Monitoring, Security, and Rescue Techniques in Multiagent Systems*, volume 28 of *Advances in Soft Computing*, pages 99–116, Berlin, Heidelberg, 2005. Springer.

[GC05] Anubhav Gupta and Edmund Clarke. Reconsidering cegar: learning good abstractions without refinement. In *2005 International Conference on Computer Design*, pages 591–598, Oct 2005.

[GC10] Arie Gurfinkel and Sagar Chaki. Combining predicate and numeric abstraction for software model checking. *International Journal on Software Tools for Technology Transfer*, 12(6):409–427, 2010.

[GCNR08] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction*

and Analysis of Systems, volume 4963 of *Lecture Notes in Computer Science*, pages 443–458, Berlin, Heidelberg, 2008. Springer.

[GCNR10] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Refining abstract interpretations. *Information Processing Letters*, 110(16):666–671, 2010.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 375–385, New York, NY, USA, 2009. ACM.

[GKD+07] S. Glesner, J. Knoop, R. Drechsler, Jan Olaf Blech, and Arnd Poetzsch-Heffter. Proceedings of the workshop on compiler optimization meets compiler verification (cocv 2007) a certifying code generation phase. *Electronic Notes in Theoretical Computer Science*, 190(4):65–82, 2007.

[GKO12] Jan Friso Groote, Tim W. D. M. Kouters, and Ammar Osaiweran. Specification guidelines to avoid the state space explosion problem. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 7141 of *Lecture Notes in Computer Science*, pages 112–127, Berlin, Heidelberg, 2012. Springer.

[GPR10] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Non-monotonic refinement of control abstraction for concurrent programs. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis*, volume 6252 of *Lecture Notes in Computer Science*, pages 188–202, Berlin, Heidelberg, 2010. Springer.

[GR06a] Denis Gopan and Thomas Reps. Lookahead widening. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 452–466, Berlin, Heidelberg, 2006. Springer.

[GR06b] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 474–488, Berlin, Heidelberg, 2006. Springer.

[GR10] Roberto Giacobazzi and Francesco Ranzato. Example-guided abstraction simplification. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming*, volume 6299 of *Lecture Notes in Computer Science*, pages 211–222, Berlin, Heidelberg, 2010. Springer.

[GR14] Roberto Giacobazzi and Francesco Ranzato. Correctness kernels of abstract interpretations. *Information and Computation*, 237:187–203, 2014.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Berlin, Heidelberg, 1997. Springer.

[GT06] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 376–386, New York, NY, USA, 2006. ACM.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[HFH+99] Mark Harman, Chris Fox, Robert Hierons, David Binkley, and Sebastian Danicic. Program simplification as a means of approximating undecidable propositions. In *Seventh International Workshop on Program Comprehension*, pages 208–217, 1999.

[HG08] Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383, Berlin, Heidelberg, 2008. Springer.

[HHF05] Robert M. Hierons, Mark Harman, and Chris J. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

[HHH+04] Mark Harman, Lin Hu, Robert Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan 2004.

[HHP09] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 69–85, Berlin, Heidelberg, 2009. Springer.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. In Nachum Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 332–358, Berlin, Heidelberg, 2003. Springer.

[HK13] Shin Hong and Moonzoo Kim. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software*, 86(2):377–388, 2013.

[HN05] Matthew Harren and George C. Necula. Using dependent types to certify the safety of assembly code. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 155–170, Berlin, Heidelberg, 2005. Springer.

[HNJ+02] Thomas A. Henzinger, George C. Necula, Ranjit Jhala, Grégoire Sutre, Rupak Majumdar, and Westley Weimer. Temporal-safety proofs for systems code. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 526–538, Berlin Heidelberg, 2002. Springer.

[HNR16] Martin Hofmann, Christian Neukirchen, and Harald Rueß. Certification for μ-calculus with winning strategies. In Dragan Bošnački and Anton Wijs, editors, *Model Checking Software*, volume 9641 of *Lecture Notes in Computer Science*, pages 111–128, Cham, 2016. Springer International Publishing.

[HNTS16] Sönke Holthusen, Michael Nieke, Thomas Thüm, and Ina Schaefer. Proof-carrying apps: Contract-based deployment-time verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, volume 9952 of *Lecture Notes in Computer Science*, pages 839–855, Cham, 2016. Springer International Publishing.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Computer Science*, pages 102–116, Berlin, Heidelberg, 1971. Springer.

[How80] William A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, December 2004.

[HR81] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, Jan 1981.

[Hro11] Juraj Hromkovič. *Theoretische Informatik: Formale Sprachen, Berechenbarkeit, Komplexitätstheorie,Algorithmik, Kommunikation und Kryptographie*. Vieweg+Teubner, Wiesbaden, 2011.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[HST+03] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3):191–229, 2003.

[HT98] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In Giorgio Levi, editor, *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214, Berlin, Heidelberg, 1998. Springer.

[Hur09] Clément Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 52–68, Berlin, Heidelberg, 2009. Springer.

[HYD05] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 52–58, New York, NY, USA, 2005. ACM.

[ISO11] ISO/IEC JTC 1/SC 22. ISO/IEC 9899:2011 Information technology – Programming languages – C. Standard, International Organization for Standardization, 2011.

[Jak15] Marie-Christine Jakobs. Speed up configurable certificate validation by certificate reduction and partitioning. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, volume 9276 of *Lecture Notes in Computer Science*, pages 159–174, Cham, 2015. Springer International Publishing.

[JGVH95] Ralph E. Johnson, Erich Gamma, John Vlissides, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[JHFK12] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. Smt-based false positive elimination in static program analysis. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 316–331, Berlin, Heidelberg, 2012. Springer.

[JIG+06] Himanshu Jain, Franjo Ivančić, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 137–151, Berlin, Heidelberg, 2006. Springer.

[JM09a] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667, Berlin Heidelberg, 2009. Springer.

[JM09b] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

[JM12] Yier Jin and Yiorgis Makris. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *2012 IEEE 30th VLSI Test Symposium (VTS)*, pages 252–257, April 2012.

[JM14] Joxan Jaffar and Vijayaraghavan Murali. A path-sensitively sliced control flow graph. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 133–143, New York, NY, USA, 2014. ACM.

[JMNS12] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. Path-sensitive backward slicing. In Antoine Miné and David Schmidt, editors, *Static Analysis*, volume 7460 of *Lecture Notes in Computer Science*, pages 231–247, Berlin, Heidelberg, 2012. Springer.

[JOS+09] E.B. Johnsen, O. Owe, G. Schneider, Ando Saabas, and Tarmo Uustalu. The 19th nordic workshop on programming theory (NWPT 2007) proof optimization for partial redundancy elimination. *The Journal of Logic and Algebraic Programming*, 78(7):619–642, 2009.

[JS08] Lingxiao Jiang and Zhendong Su. Profile-guided program simplification for effective testing and analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 48–58, New York, NY, USA, 2008. ACM.

[JSS07] Bart Jacobs, Sjaak Smetsers, and Ronny Wichers Schreur. Code-carrying theories. *Formal Aspects of Computing*, 19(2):191–203, 2007.

[JW14] Marie-Christine Jakobs and Heike Wehrheim. Certification for configurable program analysis. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[JW15] Marie-Christine Jakobs and Heike Wehrheim. Programs from proofs of predicated dataflow analyses. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC 2015, pages 1729–1736, New York, NY, USA, 2015. ACM.

[JW17] Marie-Christine Jakobs and Heike Wehrheim. Programs from Proofs: A framework for the safe execution of untrusted software. *ACM Transactions on Programming Languages and Systems*, 39(2):7:1–7:56, March 2017.

[JYM13] Yier Jin, Bo Yang, and Yiorgos Makris. Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 99–106, June 2013.

[Ken78] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163–179, 1978.

[Ker11] Mehmet Erkan Keremoglu. *Towards scalable software analysis using combinations and conditions with CPACHECKER*. PhD thesis, Simon Fraser University, 2011.

[KH13] Safwan M. Khan and Kevin W. Hamlen. Computation certification as a service in the cloud. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 434–441, May 2013.

[KH14] Raimund Kirner and Walter Haas. Optimizing compilation with preservation of structural code coverage metrics to support software testing. *Software Testing, Verification and Reliability*, 24(3):184–218, 2014.

[KHC+05] Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10 pp.–254, Nov 2005.

[KHCL07] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 389–392, New York, NY, USA, 2007. ACM.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, December 1998.

[KK05] Karsten Klohs and Uwe Kastens. Memory requirements of java bytecode verification on limited devices. *Electronic Notes in Theoretical Computer Science*, 132(1):95–111, 2005.

[KKB05] Uwe Kastens and Hans Kleine Büning. *Modellierung Grundlagen und formale Methoden*. Carl Hanser Verlag, München Wien, 2005.

[Klo09] Karsten Klohs. *Validation of Data Flow Results for Program Modules*. PhD thesis, Universität Paderborn, 2009.

[KRA09] Ernst-Rüdiger Olderog Krzysztof R. Apt, Frank S. de Boer. *Verification of Sequential and Concurrent Programs*. Springer, London, 2009.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[KV04] Orna Kupferman and Moshe Y. Vardi. From complementation to certification. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 591–606, Berlin, Heidelberg, 2004. Springer.

[KV05] Orna Kupferman and Moshe Y. Vardi. From complementation to certification. *Theoretical Computer Science*, 345(1):83–100, 2005.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Springer International Series in Engineering and Computer Science*, pages 175–188, Boston, MA, 1999. Springer US.

[Ler02] Xavier Leroy. Bytecode verification on java smart cards. *Software: Practice and Experience*, 32(4):319–340, 2002.

[Let03] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219, Berlin, Heidelberg, 2003. Springer.

[LF11] Yanchuan Li and Gordon Fraser. Bytecode testability transformation. In Myra B. Cohen and Mel Ó Cinnéide, editors, *Search Based Software Engineering*, volume 6956 of *Lecture Notes in Computer Science*, pages 237–251, Berlin, Heidelberg, 2011. Springer.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 270–282, New York, NY, USA, 2002. ACM.

[LJM11] Eric Love, Yier Jin, and Yiorgos Makris. Enhancing security via provably trustworthy hardware intellectual property. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 12–17, June 2011.

[LJM12] Eric Love, Yier Jin, and Yiorgos Makris. Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. *IEEE Transactions on Information Forensics and Security*, 7(1):25–40, Feb 2012.

[LL09] Vincent Laviron and Francesco Logozzo. Refining abstract interpretation-based static analyses with hints. In Zhenjiang Hu, editor, *Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 343–358, Berlin, Heidelberg, 2009. Springer.

[LMS+14] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 93–104, New York, NY, USA, 2014. ACM.

[LRS05] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 519–533, Berlin, Heidelberg, 2005. Springer.

[LRS16] Jérôme Leroux, Philipp Rümmer, and Pavle Subotić. Guiding craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, 2016.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[LSvE12] Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen. Generating verifiable java code from verified pvs specifications. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 310–325, Berlin, Heidelberg, 2012. Springer.

[MBCC07] Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 419–436, Berlin, Heidelberg, 2007. Springer.

[MBH05] Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop*, pages 165–182, 2005.

[McC63] John McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress 62*, pages 21–28. North-Holland, 1963.

[MFH+07] Roman Manevich, John Field, Thomas A. Henzinger, G. Ramalingam, and Mooly Sagiv. Abstract counterexample-based refinement for powerset domains. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *Lecture Notes in Computer Science*, pages 273–292, Berlin, Heidelberg, 2007. Springer.

[MHM98] Sungdo Moon, Mary W. Hall, and Brian R. Murphy. Predicated array dataflow analysis for run-time parallelization. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 204–211, New York, NY, USA, 1998. ACM.

[MMM12] Damien Massé, Laurent Mauborgne, and Vivien Maisonneuve. Proceedings of the third international workshop on numerical and symbolic abstract domains, nsad 2011 convex invariant refinement by control node splitting: a heuristic approach. *Electronic Notes in Theoretical Computer Science*, 288:49–59, 2012.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[MN07] Peter Müller and Martin Nordio. Proof-transforming compilation of programs with abrupt termination. In *Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 39–46, New York, NY, USA, 2007. ACM.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ml. *Journal of Functional Programming*, 24:284–315, 5 2014.

[MP74] Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3(3):243–263, 1974.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20, Berlin, Heidelberg, 2005. Springer.

[MRS10] Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 71–99, Berlin, Heidelberg, 2010. Springer.

[MS14] Bogdan Mihaila and Axel Simon. Synthesizing predicates from abstract domain losses. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 328–342, Cham, 2014. Springer International Publishing.

[MSG09] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Extensible proof-producing compilation. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 2–16, Berlin, Heidelberg, 2009. Springer.

[MT04] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, San Francisco, 1997.

[MUSU06] P. Mosses, I. Ulidowski, Ando Saabas, and Tarmo Uustalu. Proceedings of the second workshop on structural operational semantics (sos 2005) a compositional natural semantics and hoare logic for low-level languages. *Electronic Notes in Theoretical Computer Science*, 156(1):151–168, 2006.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[Nam01] Kedar S. Namjoshi. Certifying model checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 2–13, Berlin, Heidelberg, 2001. Springer.

[NCM14] Martin Nordio, Cristiano Calcagno, and Bertrand Meyer. Certificates and separation logic. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, volume 8358 of *Lecture Notes in Computer Science*, pages 273–293, Cham, 2014. Springer International Publishing.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 229–243, New York, NY, USA, 1996. ACM.

[NL98a] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, pages 93–104, Jun 1998.

[NL98b] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 333–344, New York, NY, USA, 1998. ACM.

[NL98c] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91, Berlin, Heidelberg, 1998. Springer.

[NMM08] Martin Nordio, Peter Müller, and Bertrand Meyer. Proof-transforming compilation of eiffel programs. In Richard F. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 316–335, Berlin, Heidelberg, 2008. Springer.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis : with 51 tables*. Springer-Verlag, Berlin, 1. ed., corr. 2. print. edition, 2005.

[NR01] George C. Necula and Shree P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 142–154, New York, NY, USA, 2001. ACM.

[NRM04] Hemendra Singh Negi, Abhik Roychoudhury, and Tulika Mitra. Simplifying WCET analysis by code transformations. In *4th International Workshop on Worst-Case Execution Time Analysis*, WCET2004, pages 11–14, 2004. `http://www.irisa.fr/manifestations/2004/wcet2004/Papers/WCET2004.pdf`.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[PD08a] Heidar Pirzadeh and Danny Dubé. Encoding the program correctness proofs as programs in PCC technology. In *2008 Sixth Annual Conference on Privacy, Security and Trust*, pages 121–132, Oct 2008.

[PD08b] Heidar Pirzadeh and Danny Dubé. VEP: A virtual machine for extended proof-carrying code. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '08, pages 9–18, New York, NY, USA, 2008. ACM.

[PDHL10] Heidar Pirzadeh, Danny Dubé, and Abdelwahab Hamou-Lhadj. An extended proof-carrying code framework for security enforcement. In Marina L. Gavrilova, C. J. Kenneth Tan, and Edward David Moreno, editors, *Transactions on Computational Science XI*, volume 6480 of *Lecture Notes in Computer Science*, pages 249–269, Berlin, Heidelberg, 2010. Springer.

[PHG05] Arnd Poetzsch-Heffter and Marek Gawkowski. Proceedings of the 3rd international workshop on compiler optimization meets compiler verification (cocv 2004) towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics, DAIMI FN-19, computer science department, aarhus university, 1981.

[PMW93] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5):607–640, 1993.

[PPZ01] Doron Peled, Amir Pnueli, and Lenore Zuck. From falsification to verification. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 292–304, Berlin, Heidelberg, 2001. Springer.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, Berlin, Heidelberg, 1998. Springer.

[Pus02] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K. H. Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, volume 91 of *IFIP – The International Federation for Information Processing*, pages 163–172, Boston, MA, 2002. Springer US.

[PZ01] Doron Peled and Lenore Zuck. From model checking to a temporal proof. In Matthew Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 1–14, Berlin, Heidelberg, 2001. Springer.

[RD99] Martin C. Rinard and Marinov Darko. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, 1999.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Riv03] Xavier Rival. Abstract interpretation-based certification of assembly code. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2572 of *Lecture Notes in Computer Science*, pages 41–55, Berlin, Heidelberg, 2003. Springer.

[Riv04] Xavier Rival. Certification of compiled assembly code by invariant translation. *International Journal on Software Tools for Technology Transfer*, 6(1):15–37, 2004.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5), August 2007.

[Ros03] Eva Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3):303–334, 2003.

[RS59] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[RS09] Michael Ryabtsev and Ofer Strichman. Translation validation: From Simulink to C. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 696–701, Berlin, Heidelberg, 2009. Springer.

[SA15] Mohamed Nassim Seghir and David Aspinall. Evicheck: Digital evidence for android. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis*, volume 9634 of *Lecture Notes in Computer Science*, pages 221–227, Cham, 2015. Springer International Publishing.

[Sch73] Paul B. Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM Annual Conference*, ACM '73, pages 106–113, New York, NY, USA, 1973. ACM.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 38–48, New York, NY, USA, 1998. ACM.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[SDDA11] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 703–719, Berlin, Heidelberg, 2011. Springer.

[SI13] Sriram Sankaranarayanan and Franjo Ivančić. NECLA static analysis benchmarks (necla-static-small) v1.1, access 12-19-2013. `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`.

[SISG06] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In Kwangkeun Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17, Berlin, Heidelberg, 2006. Springer.

[SL12] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest*. dpunkt.verlag, Heidelberg, 5., überarbeitete und aktualisierte edition, 2012.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 346–364, Berlin, Heidelberg, 1991. Springer.

[Ste96] Bernhard Steffen. Property-oriented expansion. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41, Berlin, Heidelberg, 1996. Springer.

[Stu09] Aaron Stump. Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science*, 228:121–133, 2009.

[SYK05] Elisabeth A. Strunk, Xiang Yin, and John C. Knight. Echo: A practical approach to formal verification. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, pages 44–53, New York, NY, USA, 2005. ACM.

[SYY03] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In Atsushi Ohori, editor, *Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245, Berlin, Heidelberg, 2003. Springer.

[SYYH07] Sunae Seo, Hongseok Yang, Kwangkeun Yi, and Taisook Han. Goal-directed weakening of abstract interpretation results. *ACM Transactions on Programming Languages and Systems*, 29(6), October 2007.

[TA10] Ali Taleghani and Joanne M. Atlee. Search-carrying code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 367–376, New York, NY, USA, 2010. ACM.

[TC02] Li Tan and Rance Cleaveland. Evidence-based model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 455–470, Berlin, Heidelberg, 2002. Springer.

[TG08] Aditya Thakur and R. Govindarajan. Comprehensive path-sensitive data-flow analysis. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 55–63, New York, NY, USA, 2008. ACM.

[Thé10] Grégory Théoduloz. *Software Verification by Combining Program Analyses of Adjustable Precision*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, CH, 2010.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[Tsu00] Yasuyuki Tsukada. Mobile codes with interactive proofs: an approach to provably safe evolution of distributed software systems. In *Proceedings International Symposium on Principles of Software Evolution*, pages 23–27, 2000.

[Tsu05] Yasuyuki Tsukada. Interactive and probabilistic proof of mobile code safety. *Automated Software Engineering*, 12(2):237–257, 2005.

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, 1949. Univ. Math. Lab.

[VC05] Joseph C. Vanderwaart and Karl Crary. Automated and certified conformance to responsiveness policies. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '05, pages 79–90, New York, NY, USA, 2005. ACM.

[vEWY04] Robert van Engelen, David Whalley, and Xin Yuan. Automatic validation of code-improving transformations on low-level program representations. *Science of Computer Programming*, 52(1-3):257–280, 2004. Special Issue on Program Transformation.

[vG90] Rob J. van Glabbeek. The linear time - branching time spectrum. In Jos C. M. Baeten and Jan W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Berlin, Heidelberg, 1990. Springer.

[VM08] Aytekin Vargun and David R. Musser. Code-carrying theory. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 376–383, New York, NY, USA, 2008. ACM.

[WAS03] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pages 264–274, New York, NY, USA, 2003. ACM.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[WDP14] Tobias Wiersema, Stephanie Drzevitzky, and Marco Platzner. Memory security in reconfigurable computers: Combining formal verification with monitoring. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 167–174, Dec 2014.

[Wil06] Martin Wildmoser. *Verified Proof Carrying Code*. PhD thesis, Technische Universität München, 2006.

[WN05] Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 326–341, Berlin, Heidelberg, 2005. Springer.

[WNKN04] Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping proof carrying code. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, volume 155 of *IFIP International Federation for Information Processing*, pages 333–347, Boston, MA, 2004. Springer US.

[WP16] Tobias Wiersema and Marco Platzner. Verifying worst-case completion times for reconfigurable hardware modules using proof-carrying hardware. In *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8, June 2016.

[WSF02] Michael Whalen, Johann Schumann, and Bernd Fischer. Synthesizing certified code. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME 2002:Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 431–450, Berlin, Heidelberg, 2002. Springer.

[WSW13] Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim. Programs from Proofs – a PCC alternative. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 6480 of *Lecture Notes in Computer Science*, pages 912–927, Berlin, Heidelberg, 2013. Springer.

[WT15] Heike Wehrheim and Oleg Travkin. TSO to SC via symbolic execution. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, volume 9434 of *Lecture Notes in Computer Science*, pages 104–119, Cham, 2015. Springer International Publishing.

[WW12] Daniel Wonisch and Heike Wehrheim. Predicate analysis with block-abstraction memoization. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 332–347, Berlin, Heidelberg, 2012. Springer.

[WYGI07] Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivančić. Using counterexamples for improving the precision of reachability computation with polyhedra. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 352–365, Berlin, Heidelberg, 2007. Springer.

[WZH+13] Kirsten Winter, Chenyi Zhang, Ian J. Hayes, Nathan Keynes, Cristina Cifuentes, and Lian Li. Path-sensitive data flow analysis simplified. In Lindsay Groves and Jing Sun, editors, *Formal Methods and Software Engineering*, volume 8144 of *Lecture Notes in Computer Science*, pages 415–430, Berlin, Heidelberg, 2013. Springer.

[XH04] Songtao Xia and James Hook. Certifying temporal properties for compiled c programs. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 161–174, Berlin, Heidelberg, 2004. Springer.

[YHS03] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for pcc: Dynamic storage allocation. In Pierpaolo Degano, editor, *Programming Languages and Systems*, volume 2618 of *Lecture Notes in Computer Science*, pages 363–379, Berlin, Heidelberg, 2003. Springer.

[YHS04] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for pcc: dynamic storage allocation. *Science of Computer Programming*, 50(1):101–127, 2004.

[YKNW08] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis. In Michael D. Harrison and Mark-Alexander Sujan, editors, *Computer Safety, Reliability, and Security*, volume 5219 of *Lecture Notes in Computer Science*, pages 305–319, Berlin, Heidelberg, 2008. Springer.

[YKW09] Xiang Yin, John Knight, and Westley Weimer. Exploiting refactoring in formal verification. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 53–62, June 2009.

[ZNY13] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 365–376, New York, NY, USA, 2013. ACM.

[ZZ07] Sai Zhang and Jianjun Zhao. On identifying bug patterns in aspect-oriented programs. In *31st Annual International Computer Software and Applications Conference*, volume 1, pages 431–438, July 2007.