Slicing Integrated Formal Specifications for Verification

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Dipl.-Inform. Ingo Brückner

Oldenburg, 11. März 2008

Mitglieder der Promotionskommission:

- Prof. Dr. Heike Wehrheim (Vorsitzende, Gutachterin)
- Prof. Dr. Ernst-Rüdiger Olderog (Gutachter)
- Prof. Dr. Stefan Böttcher
- Prof. Dr. Wilhelm Schäfer
- Dr. Theodor Lettmann

Eingereicht: 18. Dezember 2007
Tag der mündlichen Prüfung: 11. März 2008

Abstract

Safety-critical electronic systems are becoming increasingly complex and simultaneously ubiquitous. As in other engineering disciplines, computer science needs to offer viable approaches to the correct design of such systems. Especially important within this design process is the phase of initial *specification*, since its results have impact on all subsequent stages of development. The most extensive and reliable analysis of system specifications is offered by using *formal methods* that allow us to obtain mathematical proofs of system correctness by applying automatic verification techniques.

In the area of formal system specification there is, however, not one single general-purpose notation, that would be equally well suited for all system aspects. Instead, *integrated formal methods* are investigated, which combine different specification languages, exploiting their individual strengths, while still maintaining a common semantic foundation for subsequent verification. One such notation is the high-level specification language *CSP-OZ-DC*, combining the process algebra *Communicating Sequential Processes* (CSP) for expressing behavioural aspects, the state-based notation *Object-Z* (OZ) for expressing data aspects, and the real-time logic *Duration Calculus* (DC) for expressing real-time aspects of systems.

The main obstacle for successful application of automatic verification, however, is the problem of *state space explosion*, i.e., the exponential blow-up in the number of system states to be analysed. Many techniques have been proposed to tackle this problem, one of them being the method of *slicing* that has its origins in the area of program analysis where it is used to compute those parts of a program that are relevant with respect to a specific analysis task.

Within this thesis, we develop a slicing approach for integrated formal specifications that is custom-tailored to the rich syntactical structure of CSP-OZ-DC specifications and that is applicable in the context of their verification with respect to real-time requirements. The slicing approach essentially consists of three steps: First, the specification is analysed with respect to dependences between its syntactical elements with several new types of dependence being defined such as synchronisation and timing dependence. Second, these dependences as a whole are used to identify those specification parts that are relevant for the given verification property. Third, the specification slice is computed, i.e., a reduced version of the full specification that does not contain any elements without influence on the verification property.

A *correctness proof* shows that verification can be carried out on the slice instead of the full original specification without changing the verification result. The proof is based on a notion of *projection* between a specification and its slice. The existence

of such a projection relation is shown to be guaranteed by the slicing approach. The particular logic used to express verification properties is then shown to be stuttering-invariant, i.e., provided that the projection relation exists, it cannot distinguish between the slice and the original specification such that the verification result will in both cases always be the same.

Furthermore, we present tool support that has been implemented for developing, slicing, and verifying CSP-OZ-DC specifications along with several case studies and experimental results for evaluating the effectiveness of the slicing approach.

Zusammenfassung

Sicherheitskritische elektronische Systeme werden immer komplexer und gleichzeitig immer allgegenwärtiger. Wie andere Ingenieurwissenschaften muss auch die Informatik gangbare Herangehensweisen anbieten, um den korrekten Entwurf solcher Systeme zu gewährleisten. Besonders wichtig innerhalb des Entwurfsprozesses ist die Phase der initialen *Spezifikation*, da ihre Ergebnisse Auswirkungen auf alle nachfolgenden Entwicklungsschritte haben. Die umfassendste und zuverlässigste Analyse von Systemspezifikationen kann durch die Verwendung formaler Methoden erreicht werden, die es durch den Einsatz automatischer Verifikationstechniken ermöglichen, einen mathematischen Beweis der Systemkorrektheit zu erhalten.

Im Bereich der formalen Spezifikation gibt es jedoch keine einzelne universell einsetzbare Notation, die gleichermaßen geeignet für alle Aspekte von Systemen wäre. Stattdessen werden *integrierte formale Methoden* erforscht, die unterschiedliche Spezifikationssprachen kombinieren, um ihre individuellen Stärken auszunutzen und gleichzeitig eine gemeinsame semantische Basis für die anschließende Verifikation aufrechtzuerhalten. Eine solche Notation ist die Spezifikationssprache *CSP-OZ-DC*, in der die Prozessalgebra *Communicating Sequential Processes* (CSP) zur Beschreibung von Verhaltensaspekten, die zustandsbasierte Notation *Object-Z* (OZ) zur Beschreibung von Datenaspekten und die Realzeit-Logik *Duration Calculus* (DC) zur Beschreibung von Realzeitaspekten von Systemen vereinigt sind.

Das hauptsächliche Hindernis für die erfolgreiche Anwendung automatischer Verifikationsmethoden ist jedoch das Problem der *Explosion des Zustandsraums*, also der exponentiellen Vergrößerung der Anzahl der zu analysierenden Systemzustände. Zahlreiche Techniken zur Bewältigung dieses Problems wurden bereits vorgeschlagen, unter ihnen die des *Slicing*, das seinen Ursprung im Gebiet der Programmanalyse hat, wo es zur Berechnung derjenigen Programmteile verwendet wird, die im Hinblick auf eine bestimmte Fragestellung relevant sind.

In der vorliegenden Dissertation wird eine Herangehensweise zum Slicing integrierter formaler Spezifikationen entwickelt, die maßgeschneidert für die reichhaltige syntaktische Struktur von CSP-OZ-DC-Spezifikationen ist, und die gleichzeitig im Rahmen ihrer Verifikation bezüglich Realzeitanforderungen einsetzbar ist. Der Slicing-Ansatz besteht im Wesentlichen aus drei Schritten: Erstens wird die Spezifikation im Hinblick auf verschiedene Typen von Abhängigkeiten zwischen ihren syntaktischen Elementen analysiert, wobei mehrere neue Abhängigkeitstypen wie Synchronisations- und Zeitabhängigkeit definiert werden. Zweitens wird die Gesamtheit dieser Abhängigkeiten genutzt, um diejenigen Teile der Spezifikation zu identifizieren, die relevant für die gegebene Verifikationseigenschaft sind. Drittens wird der Slice der Spezifikation berechnet, also eine reduzierte Version der vollen

Spezifikation, in der alle Elemente ohne Einfluss auf die Verifikationseigenschaft entfernt sind.

Ein *Korrektheitsbeweis* zeigt, dass anstelle der ursprünglichen Spezifikation nun der Slice für eine Verifikation verwendet werden kann, ohne das Verifikationsergebnis zu verändern. Der Beweis basiert auf dem Begriff der *Projektion* zwischen einer Spezifikation und ihrem Slice. Es wird gezeigt, dass der entwickelte Slicing-Ansatz die Existenz einer solchen Projektionsbeziehung garantiert. Darauf aufbauend wird gezeigt, dass die jeweilige Logik zur Beschreibung von Verifikationseigenschaften stotter-invariant ist, das heißt, unter der Voraussetzung der Existenz einer Projektionsrelation kann sie nicht zwischen Slice und ursprünglicher Spezifikation unterscheiden, sodass das Verifikationsergebnis in beiden Fällen das gleiche sein wird.

Schließlich wird die Werkzeugunterstützung vorgestellt, die zur Entwicklung, zum Slicing und zur Verifikation von CSP-OZ-DC-Spezifikationen implementiert wurde, ergänzt durch mehrere Fallstudien und experimentelle Ergebnisse zur Evaluierung der Effektivität des Slicing-Ansatzes.

Acknowledgements

This research has evolved from my work at the University of Oldenburg within the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (AVACS). My acknowledgement thus goes to the German Research Council (DFG) for funding AVACS as well as to the AVACS team as a whole for providing such a stimulating work background.

In addition to that, I want to thank a number of persons individually who have supported me and my work during the recent years in various ways.

First of all, I am deeply indebted to Heike Wehrheim for introducing me to this research topic and for supervising it in a very committed and supportive way with many fruitful discussions and impulses into the right direction.

Furthermore, my acknowledgements go to Ernst-Rüdiger Olderog for the opportunity of being a member of his group "Correct System Design", for many helpful advises at various points, and for simply being a marvellous boss.

The great work atmosphere that I enjoyed during the last years was, of course, also due to my (former) colleagues whom I want to thank sincerely: Henning Dierks, Johannes Faber, Sibylle Fröschle, Andrea Göken, Jochen Hoenicke, Roland Meyer, Michael Möller, Margarethe Muhle, André Platzer, Jan-David Quesel, Holger Rasch, Henning Rohlfs, Andreas Schäfer, Walter Schulz, and Tim Strazny.

My special acknowledgements also go to the persons who made the experimental evaluation of my slicing approach possible: the students of the Syspect project group (Janna Arnold, Dominik Denker, Christian Günther, Niels Hapke, Jürgen Happe, Patrick Kuballa, Sven Linker, Florian Marwede, Jan-David Quesel, Henning Rohlfs, Christian Wenzel) together with their supervisors Ernst-Rüdiger Olderog, Michael Möller and Andreas Schäfer for developing Syspect, the platform for my slicing experiments; Ulrich Hobelmann for integrating the ARMC-based verification into Syspect; Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim for the joint work on developing SLAB; and, last but not least, Sven Linker, who did an excellent job in implementing my slicing approach as a Syspect plug-in.

Furthermore, I thank the co-authors of my publications related to this thesis, which are Klaus Dräger, Bernd Finkbeiner, Björn Metzler, and Heike Wehrheim. Our joint work was a great pleasure for me.

Finally, I would like to take the opportunity to thank my parents Frauke and Ralf Brückner as well as my sister Heike Brückner. Your support in all aspects of life is very important to me.

My deepest thanks go to Imke Fischer. You know why.

Ingo Brückner
March 2008

Contents

1 Introduction

1.1 Flawless Design of Complex Systems

In recent decades or even centuries, all fields of engineering have been confronted with systems of immense complexity, may they be large building structures as they are devised in architecture and civil engineering, sophisticated plants as arising in industrial and mechanical engineering or large scale hardware and software systems as they are subject of computer science and software engineering. In each of these fields, the more complex the systems are, the more difficult and error-prone becomes the task of designing and developing such systems correctly.

However, an important aspect that distinguishes the latter and simultaneously youngest of the aforementioned engineering disciplines, namely the field of computer science, is that its complexity has not only reached an impressive level already decades ago, but has kept on growing exponentially since its foundation, proving Moore's Law being valid year after year, simultaneously spreading to more and more areas of application.

This observation clearly underlines the special importance of being able to manage the error-prone task of designing and developing complex hardware and software systems in such a way that errors in the resulting systems are avoided as much as possible. This claim becomes even more important with complex hardware and software systems being deployed in more and more safety-critical areas such that system failures not only have economic impact on the companies involved but also pose a threat to the life of people depending on the correct functioning.

But even without considering this additional aspect, the pure financial impact that is involved with errors introduced into a system design at an early stage seems to make any consideration worthwhile that allows to avoid them, since the more time it needs to identify a problem, the more follow-up decisions might have been based on the flawed design and might thus need to be revised when the problem is identified.

Therefore, the most important flaws to find are firstly those which are introduced very early in the design process and which thus have the maximum possible lifetime; secondly, they are those which are the least obvious, i.e., which arise only under very special and unexpected circumstances and which therefore have an increased probability of remaining undiscovered.

These conclusions firstly suggest that most worthwhile for additional investments in system correctness will be those stages of system development where the initial

specifications of a system are done, since this is the earliest possible point for correctness considerations; secondly, the observations suggest that in order to detect the least obvious problems, it is clearly necessary to analyse the specifications as completely as possible, thus decreasing the probability of any flaw remaining hidden in the specification.

1.2 Formal Specifications and their Verification

The usual methods applied at the previously described especially important early stages of the design process are informal (such as natural-language descriptions) or semi-formal (such as documents containing structured text) but to no extent formal and exact in the sense of mathematical objects on which mathematical proofs can be carried out or which could be completely analysed with mathematical, i.e., verification methods.

The key to enable such an analysis is to formalise the specification documents, i.e., to transform the informal documents into mathematical objects on which mathematical proofs and automatic mathematical analysis can then be performed. Such analysis methods (e.g., model checking or theorem proving) comprise the formulation of requirements, i.e., properties that the specified systems are supposed to exhibit. The system can then be analysed with respect to these requirements, be it semi-automatically as it is usually the case in theorem proving or be it fully-automatically as it is usually the case during model-checking. Both techniques either give us evidence of the analysed system's correctness with respect to the requirements as it does a proof obtained from a theorem prover; or they give us evidence of the system's incorrectness as it does a counterexample obtained from a model checker.

In any case, the rigid mathematical description of the system, the formulation of its essential properties and the detailed analysis of the system with respect to these properties altogether decrease the probability of fundamental problems remaining present in the system design and therefore not only prevent errors from being detected lately but also increase the trust in the correctness of the system design.

Many of such formal approaches to specification have been proposed and a large variety of different notations have been developed during the last decades. Some of these formal methods have even been applied with great success in industrial projects [TWC01, BA05, Abr06].

However, there has not yet emerged any general-purpose technique that is equally well suited for all specification purposes. Instead, each of the notations has its own advantages and disadvantages, mostly due to a focus on some specific aspect of system specification such as system data or behaviour. Therefore, the obvious idea is to combine different individual specification notations into one consistent formalism that exploits the advantages of each of its constituting sub-notations.

These are subject of the recently very active research area of *integrated formal specifications*.

One proposal for such a specification formalism is the high-level specification language *CSP-OZ-DC* [Hoe06] that allows us to define all essential characteristics of complex real-time systems on a level high above subsequent implementations, and thus ideally suited for early system design. Furthermore, CSP-OZ-DC has already been shown to be appropriate for modelling industrially relevant specifications such as safety-critical parts of the European Train Control System (ETCS [FM06]), an industry standard from the railway domain.

CSP-OZ-DC combines three individually well-researched formalisms: *Communicating Sequential Processes* (CSP [Hoa85]) to specify system behaviour in terms of the admissible ordering of events and communication between processes; *Object-Z* (OZ [Smi00]) to define a system's state space and modifications associated with the occurrence of events; *Duration Calculus* (DC [HZ97]) to define real-time properties over certain events or states. In CSP-OZ-DC, a common semantic basis is given to these three formalisms by extending the DC semantics such that it also covers the CSP and the OZ part. Moreover, CSP-OZ-DC provides a compositional translation into phase event automata, a variant of timed automata which is suitable for subsequent verification by fully automatic model-checking [FM06, PR06, BDFW07].

One of the main obstacles for automated verification, however, is the problem of state space explosion, i.e., the exponential blow-up in the number of system states to be analysed. This is for instance induced by additional levels of concurrency of system components or by additional system variables leading to additional state space dimensions.

Many techniques have been proposed to tackle this problem and the frontier of systems being amenable to model checking has been pushed forward again and again throughout the last years by the—often complementary—application of numerous sophisticated techniques. Among these approaches to state space reduction are techniques for efficient state space representation of the model generated from the specifications, techniques for compositional verification dividing the properties to be verified into sub-properties which might then be verified more easily than the top level property, or techniques such as partial-order reduction [Pel98], predicate abstraction [CGJ+00], cone of influence reduction [Kur94, CGP99], or heuristic search [ELLL04].

1.3 Slicing for Verification

Aiming also in the direction of mitigating the effect of the state space explosion problem is the method of *slicing*. It was originally introduced in the late seventies by Mark Weiser [Wei79] in the context of program analysis in order to determine those parts of a program which are relevant with respect to a specific debugging

task. Weiser observed this technique to be intuitively applied by programmers during debugging: When analysing a certain problem inside a program, the focus is naturally put only upon the currently relevant program parts while other parts of the source code are mentally hidden. This intuitive analysis can of course be automated and has become—based on data flow analysis and on dependence graphs—a well known method in the area of program analysis [HRB90, Tip95]. In the past decades, slicing has found numerous further areas of application [XQZ+05], among them the area of *software verification* where it has successfully been applied to various targets such as Java [HDZ00], Promela [MT00], or SAL [GSS99]. As a syntax-based approach that operates at the source code level, slicing can—in contrast to techniques working at lower levels—exploit additional knowledge about the system structure. It has hence been shown to be effective in addition to complementary techniques working on the semantic level of the models generated from the source code [DHH+06].

Slicing in the context of verification is usually done in two steps. First, a *dependence graph* is constructed representing control and data dependences present inside the source code. This first preparatory step is independent from the actual verification property and therefore only needs to be performed once for a given verification target. Second, a backwards analysis is performed on the dependence graph with the verification property as *slicing criterion*, i.e., a starting point to identify elements that directly or indirectly affect the property to be verified. Computing the slice of the specification that is relevant to the verification property allows us to omit the irrelevant parts in the subsequent verification step such that the state space to be examined is already reduced before the verification actually starts. An important requirement in this context is the *correctness* of the slicing approach, i.e., the verification result must remain the same, regardless of whether the verification is performed on the original verification target or on the slice.

In this thesis we apply slicing to Object-Z, CSP-OZ, and ultimately to CSP-OZ-DC specifications as a preparatory step for their subsequent verification with respect to stuttering-invariant logics such as the next-free projection of linear temporal logic (LTL), LTL_{-X}, the state/event interval-logic SE-IL [BW05b], and *test formulae* [MFR06], which form a subset of DC that is amenable to model-checking. The rich syntactical structure of CSP-OZ-DC specifications and their clear separation in different parts addressing different system aspects makes them an ideal target for the syntax-oriented technique of slicing. We exploit the special structure of CSP-OZ-DC specifications by introducing several new types of dependences such as *predicate*, *synchronisation*, and *timing* dependences into the dependence graph. In comparison to conventional dependence graphs these dependences yield additional information about the specification allowing us to construct a more precise dependence graph and thus a more precise slicing outcome. Integrating previous work [BW05b, BW05a, Brü07], we show correctness of our approach not only with respect to test formulae, but, more generally, with respect to any logic which is in-

Citation	Level	Timing Aspects	Verification
[HW97, Bol04]	Individual Specifications (HSM, Z)	—	—
[HDZ00, MT00]	Programs (Java, Promela)	—	✓
[JJ04, BDFW07]	Models (Timed Automata, Transition Constraint Systems)	✓	✓
This thesis ([BW05b, BW05a, Brü07])	Integrated Specifications (Object-Z, CSP-OZ, CSP-OZ-DC)	✓	✓

Table 1.1: Contributions compared to related works on slicing

variant under stuttering, i.e., which cannot distinguish between interpretations that are equivalent up to some stuttering steps (defined by sets of irrelevant variables and events obtained from slicing).

1.4 Contributions

On the one hand the integration of formal specifications is becoming an essential instrument in order to combine the specification of several aspects of large systems, while the verification of such specifications of practically relevant sizes still suffers from the problem of state space explosion. On the other hand the technique of conventional program slicing has for long been established as a well-researched reduction technique in the field of program analysis. Therefore, the main contribution of this thesis becomes self-evident, namely exploiting the benefits that are achievable by the integration of slicing into the verification process for integrated formal specifications and the resulting mitigation of the state space explosion in model checking.

In detail, the contributions of this thesis can thus be seen along the following three dimensions, as also depicted in Table 1.1:

Slicing target: Although slicing has mainly been applied to conventional programs such as Java programs [HDZ00] or Promela programs [MT00], there also exist a few works on slicing of formal specifications such as on slicing of hierarchical state machines (HSM, [HW97]) or on slicing of Z specifications [Bol04]. These, however, only deal with individual specification notations and do not consider *integrated* specification notations such as CSP-OZ-DC, combining

different aspects of systems within one consistent specification formalism. Furthermore, the existing slicing approaches for formal specification are not aiming at the context of formal verification but rather at specification comprehension (as in [Bol04]) or without explicitly mentioning a specific purpose. Therefore, these works do not consider a proof of correctness of their approach as we do it within this thesis.

Real-time aspects: There are only very few slicing approaches that consider real-time aspects such as [JJ04] does. There, however, slicing takes place on a completely different level than in this thesis, namely with timed automata as the slicing target, i.e., on the level of models on which the actual verification algorithms work. The main motivation of this thesis, however, is the problem of state space explosion. As stated before, this problem frequently occurs already during model generation. There it often prevents successful verification due to the generated models becoming too large to be handled by the verification infrastructure. Thus, with this motivation in mind, the application of slicing on the level of generated models might simply be too late for successful verification. However, both approaches can of course be applied complementary: In spite of reductions achieved by an initial slicing on the level of specifications, slicing on the level of timed automata might still achieve some further reductions, especially, when combined with additional state space reduction techniques that are applied at an intermediate stage in between both slicing approaches.

Slicing context: Most slicing approaches have a different application background than verification, i.e., reductions of programs or specifications are aimed at with motivations such as facilitating the task of debugging in software development environments (as usually in conventional program slicing) or better comprehension of large specification documents (as in [Bol04]). Slicing approaches which are indeed motivated by the application within a formal verification process, however, are usually very near to the actual verification engine, i.e., they work on a relatively low level such as [JJ04], working on timed automata, or [BDFW07] working on transition constraint systems, or they work on implementation level such as [HDZ00] on the level of Java programs. Therefore, these can be seen as complementary to our approach which applies slicing at the earliest possible point in the development process, i.e., at the stage of high-level specifications.

In addition to these dimensions, this thesis aims at the experimental assessment of the effects that can be achieved by the application of slicing, i.e., the presented slicing approach for CSP-OZ-DC specifications has been implemented and experimental results for a variety of specifications have been achieved.

1.5 Thesis Structure

The thesis is structured as follows. The next Chapter 2 “Background: Program Slicing” gives some background on conventional slicing and variants thereof. It also gives an overview on other techniques aiming at state space reduction in the context of formal verification.

Chapter 3 “Integrated Formal Specifications” introduces the specification formalisms considered in this thesis, i.e., the specification languages Object-Z, CSP-OZ, and CSP-OZ-DC, which form the objects of the slicing approach.

The core Chapters 4–5 of this thesis address each of the steps involved in the slicing approach: Chapter 4 “Dependence Analysis” presents the dependence analysis for each of the specification languages, including the definition of the various types of dependences assembling the dependence graph, which is the main result of this first step of the slicing approach; Chapter 5 “Specification Slices” defines how to use the previously constructed dependence graph such that for a given slicing criterion directly and indirectly relevant nodes can be computed, leading to the main result of this second step of slicing, the identification of a remaining set of irrelevant nodes; moreover, this chapter defines how to use this set of irrelevant nodes of the dependence graph in order to compute a reduced version of the original specification, which is then the overall result of the slicing approach.

Chapter 6 “Slicing Correctness” presents the correctness proof of the slicing algorithm for CSP-OZ-DC which subsumes correctness of slicing approaches for Object-Z and CSP-OZ.

Chapter 7 “Tool Support and Experimental Evaluation” introduces Syspect, the graphical modelling environment for CSP-OZ-DC, which serves as the platform to perform slicing experiments. Furthermore, this chapter gives an account on experimental results obtained from the application of slicing to a variety of specifications, ranging from the running examples presented in the preceding chapters about slicing up to a larger case study dealing with an industrially relevant specification.

Chapter 8 concludes the thesis with a discussion of the results, comments on some additional work conducted in the context of slicing formal specifications and elaborating on further perspectives.

2 Background: Program Slicing

Contents

In the introduction we have motivated the idea of applying program slicing to integrated formal specifications in the context of verification. In order to explore the background of program slicing in more detail, this chapter starts with an overview of its origins in the field of program analysis and debugging.

Several extensive overview papers on program slicing exist [Kam95, Tip95, BG96, HG98, HH01, Luc01, BH04, XQZ+05, DBG+06, MMK06], as well as several web sites with collections of resources on program slicing [Lyl95, Upc97, Hie04, HR05, Har07b, Har07a], so the intention of this section is not to add another complete overview but rather pick out some essential ideas that will illustrate the evolution of program slicing.

Thus, the next section begins with addressing foundational work on program slicing, which started in the late seventies with the PhD thesis of Mark Weiser [Wei79] and saw continuous increase of interest ever since.

The following Section 2.2 introduces some of the numerous variants of slicing that have been proposed for different purposes, and some of the various fields of applications that it has found so far.

(a) Original program:

```
 1   read(x)
 2   read(y)
 3   p=0
 4   s=0
 5   if x<=1 then
 6     s=y
 7   else
 8     read(z)
 9     p=x*y
10   fi
11   write(p)
12   write(s)
```

(b) Slice on z at line 12:

```
 1   read(x)
 5   if x<=1 then
 6   else
 8     read(z)
10   fi
```

(c) Slice on p at line 12:

```
 1   read(x)
 3   p=0
 5   if x<=1 then
 7   else
 9     p=x*y
10   fi
```

(d) Slice on x at line 9:

```
1   read(x)
```

(e) Slice on s at line 12:

```
 1   read(x)
 2   read(y)
 4   s=0
 5   if x<=1 then
 6     s=y
 7   else
10   fi
```

Figure 2.1: Weiser's example program fragment (a) and slices thereof (b-e)

The chapter concludes with Section 2.3 on different techniques serving the same purpose as the slicing approach presented in this thesis, namely the support of automatic verification by mitigating the problem of state space explosion.

2.1 Foundations of Program Slicing

The term of *program slicing* was initially coined by Mark Weiser in his PhD thesis [Wei79] and subsequent publications [Wei81, Wei82, Wei84]. Therefore, we next present Weiser's initial program slicing approach based on *data flow equations*, which are also used within several further slicing approaches [LR87, KL88, GL91]. Afterwards, we give an overview of the nowadays most common slicing approach based on *program dependence graphs*, as for example used in [HRB88, HRB90, AH90, Bin93, BH93a, CF94].

2.1.1 Slicing Based on Data Flow Equations

The example program fragment that Weiser originally used to illustrate his concept of slicing [Wei81] is depicted in Figure 2.1a. It computes variables s and p, denoting the result of a sum and a product calculation based on input to variables x and y. In addition, it contains an input statement that is obviously unrelated with the remaining statements, potentially modifying variable z.

Weiser's key observation was the following: When analysing the program fragment for the value of a given variable during execution of the program up to a given line, a task as it usually occurs during debugging, one does not need to consider all

elements of the program, since not all of its elements contribute to the question at hand.

Supposed we are interested in the value of variable `z` at the end of the example program fragment. Assignments to variable `s` and `p` obviously do not contribute to this question. However, input to variable `x` is essential, since the value of variable `x` determines via the branching condition at line 5 whether control flow reaches the point at line 8 where input to variable `z` takes place. For this specific purpose it is therefore sufficient to examine the *slice* of the original program depicted in Figure 2.1b, which contains only statements of the original program that might influence the value of `z` at line 12.

An even larger portion of the program code may be ignored when analysing the program for the value of variable `x` at line 9. The only statement with influence to this question is the input statement at line 1 such that all remaining statements may be removed, resulting in the slice depicted in Figure 2.1d.

Note that even the branching statement surrounding line 9 may be removed, since it does not contribute to the value of `x`, regardless of whether control flow reaches the statement at line 9 or not.

Weiser's last examples, depicted in Figure 2.1c and 2.1e, show the code relevant to the question of what value variables `p` and `s` have at line 12, i.e., at the end of the program fragment. Here, only assignments and input statements to the variables `z` and `s` or `p`, respectively, have been removed in comparison to the original program, since anything else, i.e., input to `x`, and branching statements depending on the value of `x` have influence on the final values of `p` and `s`.

To summarise the examples, each of the given analysis question requires a thorough analysis of the program code with respect to the contribution of each of the statements to the question at hand. These contributions might be more or less obvious, i.e., the relevance of a statement containing a direct assignment to the variable of interest is easy to recognise, while the indirect contribution of another variable via a sequence of mutually depending assignment statements or via control flow structure depending on it might be more difficult to detect, especially when the programs become longer than the tiny example considered here.

After observing the usefulness of computing program slices, Weiser also conducted experiments with programmers in order to examine whether program transformations of this kind are done intuitively [Wei82]. Within these experiments, programmers were confronted with the task to analyse some program with respect to a given debugging question. Upon completion of this task, the programmers were given several tiny fragments of the analysed program and were asked whether they encountered them during their previous analysis. Some of these fragments were related to the debugging task, while others were not.

The answers he obtained from the programmers confirmed Weiser's hypothesis: programmers do some kind of intuitive slicing during debugging, i.e., they recognised most of the relevant fragments, while they intuitively ignored irrele-

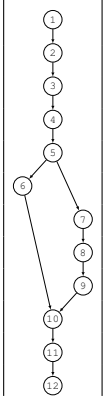
Program	**Graph**	**DEF**	**REF**	**INFL**	R_C^0	S_C^0	B_C^0	R_C^1	S_C^1	**C-Slice**
`read(x)`		`x`							✓	`read(x)`
`read(y)`		`y`				✓		`x`	✓	`read(y)`
`p=0`		`p`			`y`			`x,y`		
`s=0`		`s`			`y`	✓		`x,y`	✓	`s=0`
`if x≤1 then`			`x`	6,8,9	`s,y`		✓	`s,x,y`	✓	`if x≤1 then`
`  s=y`		`s`	`y`		`y`	✓		`y`	✓	`  s=y`
`else`							✓		✓	`else`
`  read(z)`		`z`			`s`			`s`		
`  p=x*y`		`p`	`x,y`		`s`			`s`		
`fi`					`s`		✓	`s`	✓	`fi`
`write(p)`			`p`		`s`			`s`		
`write(s)`			`s`		`s`			`s`		

Figure 2.2: Example of data flow equation computation with slicing criterion $C = \langle 12, \mathrm{s} \rangle$

vant fragments such that after completing the debugging task they mostly did not remember having encountered them before.

Based on these results, Weiser proposed a solution to automate the intuitive slicing in order to simplify debugging tasks by reducing the code to only its currently relevant portion, such that programmers might focus their examination to this without being distracted with irrelevant code fragments.

Firstly, Weiser's solution formalises the given analysis question. Since he previously identified slicing to be applicable in the context of debugging, the question driving the slicing approach is the same as for the usual debugging task: the value of a given variable at a given point in the program. This pair of parameters forms the *slicing criterion* and serves as the starting point for the subsequent slicing approach.

As a further ingredient for slicing, Weiser defined *flow graphs*, using nodes to represent a program's statements and using edges between these nodes to represent the flow of control between these statements. Figure 2.2 shows the flow graph for the previously introduced example program fragment in the second column.

For each flow graph node n, Weiser then defines sets DEF(n) of variables that are *altered* by the associated statement and sets REF(n) of variables that are *referenced* at the statement. Furthermore, for any branching statement, the set INFL(n) contains those statements that are influenced by the branching statement by being *control dependent* on it. The resulting sets for the example program fragment are shown in Figure 2.2.

Using these initial sets of variables, Weiser then sets up data flow equations defined recursively along the previously constructed flow graph. These data flow equations capture the relation between referenced and modified variables such that iterative solutions with respect to a given slicing criterion yield for each statement

those variables which are of interest with respect to the slicing criterion.

The calculation starts at the flow graph node as defined by the slicing criterion. For this node, the associated set R^0_C of *directly relevant variables* is that defined by the slicing criterion. Thus, the starting point of the example calculation in Figure 2.2 is line 12, with $R^0_C(12)$ containing only variable `s`, since the slice is computed with respect to slicing criterion $C = \langle 12, \mathtt{s} \rangle$.

From thereon, the sets R^0_C are computed for each flow graph node that precedes the node of the slicing criterion, following the flow graph backwards up to its entry node. For a given node n, the set $R^0_C(n)$ contains all relevant variables of its respective successor node m, unless they are defined at n, i.e., contained within DEF(n). Instead of such modified variables, $R^0_C(n)$ will contain all variables referenced at n, i.e., those from REF(n).

The main result of this first iteration is the set S^0_C, containing all statements with assignments to variables that are relevant at a successor node, i.e., for which DEF(n) $\cap\, R^0_C(m) \neq \varnothing$ holds with m being a successor of n. In the example calculation, the only such statements are the assignments to variable `y` and `s` in lines 2, 4 and 6.

Another result of the first iteration is the set B^0_C of *relevant branching statements*, i.e., those branching statements that have influence on directly relevant statements of S^0_C. The only such statement in the example calculation is the conditional construct in lines 5 through 10, since its first branch contains the directly relevant assignment to variable `s` of line 6.

The next iteration starts with the computation of another set $R^1_C(n)$ of indirectly relevant variables for each flow graph node. This computation is now based on the previously calculated sets R^0_C. At relevant branching statements, i.e., at those from B^0_C, the set $R^1_C(n)$ contains additionally all referenced variables. From R^1_C we can then compute another set of relevant statements S^1_C.

Having determined these, the iteration can be repeated as before until a fixed point is reached, i.e., until the set of relevant statements does not grow anymore. In the example calculation, this is already the case for S^1_C, such that this set of statements forms the final result of the calculation, namely the slice of the program with respect to slicing criterion $C = \langle 12, \mathtt{s} \rangle$.

2.1.2 Slicing Based on Dependence Graphs

Only little later than Weiser, the first graph-based approaches to program slicing came up in the works of Linda and Karl Ottenstein [OO84]. The concept of a program dependence graph had already been introduced before that point [KKP+81], but with a different motivation, namely its use in compiler construction and optimisation.

In fact, in addition to serving as a basis for program slicing, the program dependence graph can be used for a number of purposes within the field of program analysis such as data-flow analysis problems, shape approximation of heap-allocated

structures ("shape analysis"), or flow-insensitive points-to analysis, which can all be reduced to dependence graph reachability problems [Rep98a, Rep98b].

Ottenstein and Ottenstein [OO84], however, are the first to use the concept of dependence graphs for slicing and—like Weiser—aim at the application of slicing in the context of debugging in order to show programmers only relevant code, e.g., with respect to a given breakpoint within a software development environment. However, their work remains restricted to an informal explanation of the procedure of how to construct a slice. Moreover, they discuss the use of slicing in general under different execution models, for incrementally updating internal program representations and for program complexity metrics.

Subsequently, Karl Ottenstein together with Ferrante and Warren further elaborates the program-dependence-graph-based slicing concept in [FOW87], although in that work slicing is only one of the possible application scenarios of the program dependence graph they construct, next to conventional applications in compiler constructions such as parallelism detection, node splitting, code motion, or loop fusion.

The mentioned works contain already the elements that are still present in current graph-based approaches to program slicing: Initially, an abstract representation of a program's control flow is constructed, similar to Weiser's flow graphs along the lines of standard control flow graph construction as in compiler construction and optimisation [WG84, ASU97, WM97, NNH99, Muc00].

Based on such a control flow graph, however, not data flow equations are set up, but a program dependence graph is constructed, which re-uses the nodes of the control flow graph, but contains a revised set of edges of different types: first, control dependence edges between nodes for which some kind of control dependence exists, then data dependence edges between nodes for which data dependence edges exist. Instead of iteratively computing solutions to data flow equations, as in Weiser's original approach, the slicing problem is then reduced to a reachability problem within the program dependence graph.

Since the program dependence graph captures all mutual influences between elements of a program, everything that might affect a given slicing criterion (represented by some node in the program dependence graph) can be identified by tracing back along the graph's dependence edges. Consequently, elements which are not reachable in this way are not relevant with respect to the slicing criterion and can be removed from the program in the given context.

An example of the dependence graph of the previous example program fragment due to Weiser can be found in Figure 2.3. Each node of the dependence graph corresponds to one of the statements of the program fragment. Control dependence edges connect nodes representing branching statements with those nodes representing the statements within their branches whose execution directly depends on the branching condition. The start node can be regarded as an artificially introduced branching statement, representing the global condition that control flow

```
 1   read(x)
 2   read(y)
 3   p=0
 4   s=0
 5   if x<=1 then
 6     s=y
 7   else
 8     read(z)
 9     p=x*y
10   fi
11   write(p)
12   write(s)
```

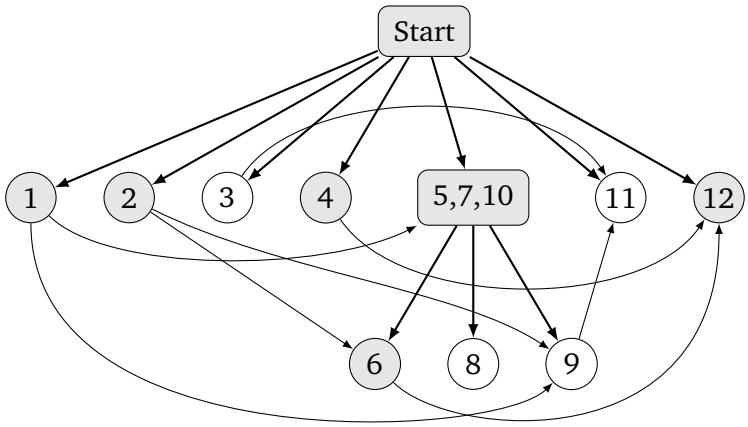


Figure 2.3: Program dependence graph for the example program fragment. Thick edges represent control dependence, thin edges represent data dependence, and shaded nodes represent statements associated with the slice with respect to $C = \langle 12, \mathrm{s} \rangle$.

has entered the program fragment. Thus, the start node is directly connected via control flow edges with any statement that will definitely be executed, i.e., with any node outside conditional statements.

Note that, apart from the start node, the example contains only one node that is source of control dependence edges, namely the node representing the only conditional construct of the program fragment within lines 5–10. Data dependence edges lead from any node representing an assignment to some variable to any of its control flow graph successor nodes where a reference to the same variable might take place.

The slice computation with respect to slicing criterion $C = \langle 12, \mathrm{s} \rangle$ starts at node 12 as defined by the slicing criterion. It then identifies all nodes that are reachable from this initial node when moving backwards along arbitrary data or control dependence edges. The nodes identified in this way have indirect influence on the slicing criterion and therefore need to remain in the slice. Nodes 3, 8, 9, and 11, on the other hand, are not reachable in this way, and may therefore be removed from the original program to obtain its slice.

In comparison to the slice computation based on data flow equations as presented in the previous section, this slice is less precise: it contains the additional statement of line 12 and is thus larger than necessary, since this output statement does not actually contribute to the value of s at line 12.

The reason for this different outcome can be found in the different starting points of each slicing approach: The starting point for dependence-graph-based slicing is a single program dependence graph node. From thereon, any arriving data

dependence is followed backwards, such that essentially all variables referenced at this node are regarded as being relevant for the slice.

Thus, the presented basic form of dependence-graph-based slicing simply ignores the second component of the slicing criterion, i.e., the set of variables. Instead, it only computes a slice with respect to the first component of the slicing criterion, i.e., with respect to the given statement, represented by its associated dependence graph node.

However, the program-dependence-graph-based slicing can be easily refined such that it achieves the same granularity of slicing results as data-flow-equation-based slicing does: Instead of using the single node fixed by the slicing criterion as a starting point, one can determine the set of nodes that define variables fixed by the slicing criterion, such that the definitions might reach the initial node. The backwards reachability analysis is then carried out with respect to this previously computed set of nodes, leading to the same slicing result as presented in the previous section.

In the recent decades, several such variants of dependence-graph-based slicing along with a lot of variants of the program dependence graph have been developed, with motivations that range from achieving a more efficient construction of the graph [HMR93], over the special treatment of additional language features such as complex control-flow (i.e., unconditional jumps [Bal93]) or special constructs to support parallelism [SS94] up to generalisations that capture several separate notions of the program dependence graph into a single consistent concept [Sta00].

2.2 Classification of Slicing Approaches

The previous section introduced both of the most common but fundamentally different approaches to slice computation, namely data-flow-equation-based and dependence-graph-based slice computation. In addition to the type of computation they use, slicing approaches can be classified with respect to a large number of criteria. This section characterises some of these criteria (partly following [Kam95], [Tip95], and [XQZ+05]) and gives for each of these criteria an overview of some representative associated slicing approaches.

2.2.1 Type of Slicing: Static or Dynamic

When executing a program, its operation usually depends on inputs. Thus, most of the behaviour of a program cannot be predicted at compile-time. Therefore, we can distinguish between slicing approaches yielding *static slices*, computed independently from the actual execution of the program, i.e., at compile-time, as it was the case in all previous examples, and slicing approaches yielding *dynamic slices*, computed under consideration of program input, i.e., with assumptions on the environment during run-time. Therefore, the latter can also be seen as a variant

(a) Original program:

```
   begin
 1   read(X)
 2   if (X < 0) then
 3      Y := f1(X);
 4      Z := g1(X);
     else
 5     if (X = 0) then
 6       Y := f2(X);
 7       Z := g2(X);
       else
 8       Y := f3(X);
 9       Z := g3(X);
       end_if;
     end_if;
10   WRITE(Y);
11   WRITE(Z);
   end.
```

(b) Static slice on `Y` at line 10:

```
   begin
 1   read(X)
 2   if (X < 0) then
 3     Y := f1(X);
     else
 5     if (X = 0) then
 6       Y := f2(X);
       else
 8       Y := f3(X);
 9     end_if;
     end_if;
   end.
```

(c) Dynamic slice on `Y` at line 10 with input X=-1:

```
   begin
 1   read(X)
 2   if (X < 0) then
 3     Y := f1(X);
     end_if;
   end.
```

Figure 2.4: Comparison between static and dynamic slicing

of slicing with an extended slicing criterion that comprises a series of inputs in addition to the usual combination of values of variables at certain program points. In the context of program debugging and testing, the obvious advantage of dynamic slicing [AH90, BDG+04] is thus to be better adaptable to specific analysis purposes such as a given execution history of a program.

The example program fragment that Agrawal and Horgan used to illustrate their concept of dynamic slicing is depicted in Figure 2.4. The additional knowledge about input to variable `X` allows us to ignore a complete branch of the conditional statement of the example. Thus, dynamic slicing yields a subset of the slice obtained from static slicing. Therefore, dynamic slicing can in many situations achieve more precise slices than static slicing, provided that the inputs to the program are known.

Related to the presented approach of dynamic slicing are the concept of *program conditioning* and the resulting slicing variant of *conditioned slicing* that will be subject of Section 2.3.1.

2.2.2 Direction of Slicing: Forward or Backward

The starting point of all previously introduced slicing approaches was the slicing criterion, from whereon the slicing algorithm performed a backwards-oriented computation, whether via data flow equations or via edges of the program dependence graph, since the aim was to identify sources of influence on the slicing criterion. Therefore, all previous approaches fall into the class of *backward slicing* approaches.

However, the same concepts can be applied in the opposite direction, giving rise to *forward slicing* approaches, such as mainly used in conjunction with dynamic

slicing [KY94, ABGS+01, ZGZ04] with the motivation of computing slices of the program that might be affected by the slicing criterion, i.e., an execution chain of the program starting at the point defined by the slicing criterion. Usually this is applied on-the-fly, i.e., during program execution with the current point of control being the slicing criterion, such that the effect of the currently executed statements can be analysed interactively without recording the execution trace.

Another motivation for the computation of forward slices can be seen in the area of verification under *property inheritance* [Weh04, Weh05, OW05]. When existing classes, for which a range of properties has already been successfully verified, are subsequently modified, e.g., in the process of refinement or for the purpose of sub-typing, the obvious question is, to what extent the previously proven properties remain valid for the modified class, i.e., whether the resulting classes inherit these properties or not.

In this context, forward slices are useful by yielding answers to the question of property inheritance between such modified classes. To this end, the dependence graph is used to compute the effect of the changes introduced in the modified class. The validity of a given property can then be determined by comparing its atomic propositions with those influenced by the newly introduced modifications. If the intersection is empty, the property remains valid for the modified class and the verification run does not need to be repeated, otherwise the property needs to be verified again.

2.2.3 Type of Slice: Executable or Non-Executable

So far, the goal of each of the previous slicing approaches was to compute *partially equivalent programs* such that the slice is (a) executable and (b) its behaviour with respect to the slicing criterion is equivalent to that of the full program, i.e., when executing the slice, the relevant variables at the program point defined by the slicing criterion have the same values as when executing the original program.

If ignoring condition (a) and thus weakening the slicing approaches to also allow *non-executable* slices as a result, i.e., if the goal is only to determine the set of statements of the full program that might affect the slicing criterion [HRB90, Ven91, Bin93], the consequence is that also condition (b) will not be satisfied any more, since it does not make very much sense to reason about the behaviour of non-executable programs.

Of course, such slices will therefore not be useful under the usual view on slicing as a technique to compute partially equivalent programs. However, when considering different purposes, such as incremental program optimisation [FOW87] or program comprehension [RLG02], only this weaker notion of a slice is needed, such that the computation of non-executable slices might then make very much sense, especially, since it allows obtaining reductions that conventional slicing can not achieve. Examples of such reductions are the removal of syntactical elements

(a) Original program:

```
1   i := 0;
2   while i < 10 do
3     if i=0 then h := 10;
4     i := i+1;
5   od;
6   write(h);
```

(b) Non-executable slice on `h` at line 6:

```
1   i := 0;
2   while i < 10 do
3     if i=0 then h := 10;

5   od;
6   write(h);
```

Figure 2.5: Comparison between executable and non-executable slices

that are needed to obtain well-formed and thus executable programs or reductions leading to non-termination such as the one illustrated in Figure 2.5, an example of non-executable slices due to [Ven91].

When applying the usual notion of slicing to the example program with respect to the given slicing criterion, we obtain no reduction at all. In contrast to that, the weaker notion of slicing only collects those statements that have a direct influence on the slicing criterion. This yields the reduction depicted in Figure 2.5. However, there is a price to pay in order to achieve this reduction: the resulting slice in this example is a non-executable program, since it will not terminate anymore.

2.2.4 Type of Slicing Criterion

When comparing different slicing approaches with respect to the kind of slicing criterion that are used as starting points, we can distinguish the following two main categories of slicing criteria: (1) Slicing criteria that explicitly name a specific program location as the point of interest with respect to that the slicing procedure is carried out, and (2) slicing criteria that do not explicitly refer to such a distinct program location, but rather perform slicing with respect to a whole range of program points that are implicitly characterised by the slicing criterion.

The former category (1) of slicing criteria can further be divided into the following types, which are distinguished according to the additional side conditions that they impose on the slicing approach:

- Static slicing criterion (v, p) as for instance used by [HRB90] with the side condition of a direct connection between variables of interest v and program location p: Here, any variable of interest v must be defined or referenced at the given program point p of the slicing criterion.

- Static slicing criterion (v, p) as for instance used by [Wei84, GL91] without the side condition of a connection between variables of interest v and program location p: In this more general variant of slicing criterion, a variable of interest v can be an arbitrary variable, without the side condition that it must be defined or referenced at the given program point p of the slicing criterion.

- Dynamic slicing criterion (v, p, x): In addition to some variable of interest v and a program location p, this variant of slicing criterion also defines a program input x, giving rise to dynamic slicing approaches.

Common to the latter category (2) of implicitly given slicing criteria is that they require an additional step in order to compute the set of program locations that actually serve as the starting point for the slicing algorithm. Examples of this category of slicing criteria are

- the Bandera Specification Language [CDHR02] that defines *observable propositions* used to specify properties of Java programs,
- the predicate-based slicing criteria used by [RLG02] for dynamic slicing of message passing programs, or
- temporal logics formulae such as state/event interval logic formulae or test formulae that we will later introduce within Chapter 5 of this thesis to reason about temporal properties and real-time properties of integrated formal specifications.

Within each of these latter slicing approaches, the slicing criterion is initially used to identify locations that have direct relevance by modifying variables mentioned within the slicing criterion or by affecting program/system states described by the slicing criterion.

2.2.5 Target Language

So far, we only saw examples in which slicing was applied to program fragments written in pseudo code. However, there is a broad spectrum of real-world *programming languages* for which slicing approaches have been devised such as in the following examples:

- Java programs [Zha99, HDZ00, Nan01, CX01, RH04], in the case of [HDZ00] used in the context of model construction for Java programs and their subsequent verification with respect to temporal logic properties.
- ANSI C programs [KK95, Kri03], where, in addition to the usual purpose of debugging and user support, slicing in [Kri03] is also used for clone detection within programs, based on identifying similar subgraphs in program dependence graphs.
- Promela programs [MT00], written in the input language of the SPIN model checker, mainly used for specifying communication protocols and their explicit-state verification.

- VHDL programs [INIY96, CFR+99, CFR+02, VABT03], addressing issues that are specific for hardware description languages, such as their implicit concurrency, their non-halting reactive nature, based on processor cycles, and further specific language constructs, such as signals and ports, or concurrent assignment.

In addition to these rather straightforward applications of slicing, several slightly less obvious slicing targets have been identified, including

- slicing of logic programs [SD96, Vas98, SGM02] usually based on the Prolog language,
- slicing of functional languages [AH00], using abstract interpretation techniques,
- slicing of class hierarchies in C++ [TCFR96] with focus on slicing hierarchy declarations instead of executable program statements,
- slicing of general-purpose syntax trees [SH96] instead of programs with a concrete given syntax, and
- specification-based slicing of programs enriched with assertions on pre and post conditions of program statements [CLYK01, LCYK01].

A number of slicing approaches have also considered slicing of *formal specifications* such as

- the Z specification language [OA93, CR94, Bol04, WY04], where the focus of [OA93] is on debugging, modification, and re-use of specifications, while [CR94] and [Bol04] aim at the comprehension of specifications, and [WY04] develops slicing for Z specifications without naming a specific application purpose,
- Petri nets in general [LKCK00] and marked Petri nets in the context of their falsification and verification with respect to LTL formulae [Rak07],
- hierarchical state machines [HW97],
- the Wide Spectrum Language (WSL) [War02], a specification language that intends to cover several stages of the development process, ranging from high-level-specifications down to real programming languages and even assembler code, using WSL transformations [War89] for connecting the stages,
- the algebraic specification notation OBJ [WA98], or
- the software architecture description language WRIGHT [Zha98].

For the support of *concurrent* structures within the target language, as it will also be needed for CSP-OZ-DC specifications, several approaches have developed suitable concepts of slicing.

One of the first works that consider concurrency explicitly with respect to slicing is Cheng [Che93], whose slicing approach is tailored to the analysis of OCCAM programs. In order to represent the concurrent nature of such programs, the notion of *process dependence net* is defined. This dependence net essentially forms an extension of the program dependence graph that contains additional types of dependence such as communication dependence representing the relation between two processes that exchange data. A later extension of the approach [ZCU96] also includes the coverage of object-oriented concepts under concurrency. A very similar approach is proposed in [Zha99] that deals with several aspects of concurrency which are specific to multi-threaded Java programs.

An approach leading to smaller and therefore more precise slices is explored by Krinke [Kri98, Kri03], who uses threaded ANSI C programs as its target language and thus needs to deal with implicitly given data interference due to variables shared between processes. In his slicing approach Krinke introduces expensive additional computations in order to avoid the introduction of non-transitive interference data dependence edges. However, this work does not yet cover explicit thread synchronisation, a shortcoming which is amended within closely related works [HCD+99, NR00, Nan01, CX01] on slicing concurrent Java programs.

One of the first approaches that explicitly consider slicing of *real-time systems* is [GH97], whose slicing approach operates on a relatively low level close to actual implementation of real-time programs. Therefore, their slicing algorithm is tailored to the setting of real-time scheduling analysis, working on sets of periodically executed tasks that share a common processing unit. Furthermore, their slicing approach is closely integrated into a dedicated scheduling strategy, allowing transforming previously unschedulable applications into schedulable ones.

More recent approaches have also considered slicing for timed systems on the comparably more abstract level of real-time automata [BGO02, JJ03, JJ04]. In the approach of [BGO02], the slicing target consists of a set of timed automata in parallel composition, which are subject of verification runs with respect to an observer automaton, also composed in parallel with the remaining automata. Slicing criteria are the states of the observer automaton. The result of the slicing approach is not a conventional reduction of the original system but rather an optimised system where components and clocks are reduced according to their relevance with respect to the current state of the observer automaton.

The slicing approach developed in [JJ03, JJ04] also aims at the application to timed automata. However, slicing is actually carried out on the level of an intermediate language which corresponds to the set of timed automata defining the system, and their slicing criterion consists of a set of operations which is assumed to be derived from a given verification property.

2.2.6 Area of Application

The original area of application of slicing as identified by Weiser [Wei82] can be found in the support of debugging activities, or, more general, in the support of measures for quality assurance, which includes—in addition to debugging aids—approaches of slice-based testing and slice-based differencing. However, slicing has soon found further possibilities of application, ranging from more general employments in quality assurance such as for instance, the general problem of *program comprehension*, not necessarily aiming at debugging or testing, over additional tasks in program analysis, such as using slicing approaches to define new *program metrics*, up to completely new application areas such as in software maintenance and re-engineering or in model generation in the context of *formal verification*. In the following, we introduce some examples of each of these application areas of slicing.

Quality Assurance: Debugging, Testing, and Differencing

Since being its original area of application, quality assurance has also received the most interest in recent years when developing new variants of slicing and therefore extending its possible applications.

Agrawal, Demillo, and Spafford, for instance, have developed an approach of *semi-automatic debugging* by applying dynamic slicing in combination with backtracking of program executions [Agr91, ADS93] along the possible lines of control flow defined by the slice and thus allow the interactive analysis of faulty program behaviour.

The concept of *chopping* is introduced by Jackson and Rollins [JR94] and further developed by Reps and Rosay [RR95] as a generalisation of slicing that combines forward and backward slicing: first, a forward slice is computed with respect to some source location; then—based on this forward slice—a backward slice is computed with respect to some target location, yielding the *chop* of the given program that contains all relevant information with respect to the question of how the source location might affect the target location.

For the purpose of testing, slicing has mostly been applied to enhance *regression testing*, a subject on which even a dedicated survey paper exists [Bin98]. Some of the approaches apply Weiser's original slicing technique based on data flow equations [GL91, GHS92], while most of them apply dependence-graph-based slicing [BH93b, RH96, Bin97] in order to identify program parts that might be affected by a newly introduced change in the program, such that this reduced version of the program can then be used for the computation of the test cases forming the regression test.

Moreover, slicing has been used in *mutation testing* [HFH+99, HHD99, HHD00], where program changes are purposely introduced in order to use them for the

assessment of test cases: the more of such artificially injected mutations are discovered by a test case, the better it will also be suited for identifying errors. In this context, program slicing can as well help in identifying *equivalent mutants*, i.e., in avoiding mutations that yield the same results for a given test case, as it can help in *generating* suitable *test cases* that cover a maximum number of mutants and are therefore expected to also be best suited for discovering further bugs.

Another relatively recent application of slicing within testing has been proposed for *partition-base testing* [HH00, HHF+02], where the input domain of a program is partitioned into disjoint subdomains such that the observable behaviour of the program remains relatively uniform within each of these subdomains. The partitions can be characterised by suitable conditions on the input domain, which can in turn be used by *conditioned slicing* to compute slices of the program not only with respect to traditional slicing criteria, but also with respect to the satisfaction of the premises given in form of conditions on the program input.

An application of slicing closely related to testing is that for *program differencing* [Hor90, Bin92, Bin95, BCRS01, Bin02], i.e., for the computation of differences between two given versions of a program. In these approaches, both forward and backward slicing are used in order to identify the difference between both programs, i.e., those program parts that are either directly modified, added, or removed, or which are affected by these changes.

Comprehension

More or less each of the previously mentioned approaches that employ program slicing for the support of quality assurance are based on human interaction, i.e., their goal is to support the individual software developer at the specific task at hand such as debugging or testing. Therefore, the common denominator of all these slicing-based approaches can be seen in supporting *human program comprehension* [MBPRR01, KNNI02] in general, without explicitly naming a specific comprehension purpose.

This is exactly the perspective that has driven the development of a lot of variants of slicing, i.e., slicing approaches that try to improve human understanding of programs without focusing on a specific analysis purpose. They achieve this by computing slices that in some sense are more precise or better adapted to human comprehension.

Slicing variants that have arisen from this rather general perspective include *quasi-static slicing* [Ven91, HDS95a] as well as *constrained slicing* [FRT95] and *conditioned slicing* [dLFM96, HHF+01]. All of these variants of slicing extend conventional slicing by trying to mimic the intuitive proceeding of programmers in program comprehension even closer than previous slicing approaches: instead of simply following mechanical dependences between elements of a given program as in static slicing, or following one specific execution path as in dynamic slicing, these

approaches impose additional assumptions on the program environment without limiting it to one single execution. Consequently, these approaches can achieve a more focused view on the given analysis task, i.e., the resulting slices obtained are often smaller and therefore better comprehensible.

Another related approach can be seen in *amorphous slicing* [HD97, HBD03], which lifts the relation between a program and its slice from the syntactic to the semantic level: an amorphous slice does not necessarily need to be a subset of statements of the original program, but its semantics rather has to be a projection of the original program's semantics. This allows additional transformations on the syntactic level of the program that might facilitate program comprehension in comparison to traditional slicing. Furthermore, amorphous slicing can be seen as a further imitation of how human analysts proceed, who intuitively build an abstract model of program behaviour instead of computing merely syntactical projections resulting from simple statement deletion [BRSH00].

Of course, the role of slicing within program comprehension is not only restricted to such general comprehension approaches, but has also been considered in the context of specific comprehension purposes. Concrete examples of such applications are the analysis of properties of dynamic memory access of programs [HSD98, SHS02], the identification of duplicated C code fragments [KH01], the understanding of database architectures [HEH+98], or the general understanding of program execution [KR97].

Maintenance and Re-Engineering

Many of the tasks that arise during software maintenance or software re-engineering can be supported by slicing approaches.

Decomposition slicing [GL91, GO01, Ton03] is used to determine slices with respect to only variables as a slicing criterion, however, without the limitation of considering only one single location of the given variable. This technique is used to support *software maintenance* activities in the assessment of changes proposed for a given location by identifying what effect such modifications might induce onto other components at different locations.

Program integration is the task of merging different versions of a common base program into one new version, sharing the common features of both previous versions and adding additional features from both, supposed that the newly introduced modifications do not interfere. The application of conventional tools such as `diff` give no guarantee, whether adding the syntactical changes derived from the new versions into the base program yields a sound result. This can be improved by applying a program-slicing-based integration technique [HPR89, RY89, HR92, BHR95] that takes into account the mutual influences of newly introduced modifications coming from both of the modified program versions. Therefore, the slicing-based approach produces a sound integration of both versions, provided that an interme-

diate analysis ensures that the modifications from both sides do not interfere.

A number of approaches have applied slicing techniques on the level of *system architectures* in order to compute reduced versions of the architecture descriptions with respect to specific analysis purposes [Zha98, KSCH99b, KSCH99a, KSCH00], aiming at but not necessarily being restricted to the context of software maintenance and software re-use. The high-level architecture descriptions considered by these approaches are dedicated to larger scale software systems and constitute an abstract view onto the system that hides details of the underlying data structures and algorithms. The main motivation of applying slicing on this level is the process of *impact analysis*, which assesses the effect of modifications of a given component onto different components of the system. Consequently, the slicing criterion in this context is much coarser grained than in traditional slicing approaches, i.e., instead of a variable at some program location, a set of system resources or system events serve as the slicing criterion.

The identification of *reusable functions* and their extraction from existing code has been the driving motivation for the approach of *transform slicing* [LV97] that basically ignores the control flow defined by a program in order to only collect those statements of a program that directly or indirectly contribute to a set of resulting variables defined by the slicing criterion, i.e., especially input statements and branching conditions are not included in the slice. However, the resulting slices only serve as possible candidates for the intended functions, i.e., they still have to be manually evaluated in order to assess, whether they form a sound version of the expected function that indeed satisfies the overall purpose of code re-use.

A related approach, not aiming at re-use but rather at *restructuring* programs has been pursued by [LD98] who devise a three-step "tuck" transformation that employs slicing only in its first step to compute a desired subset of statements, which is then, in a second step, split from the remaining program and finally folded into a new function.

The application of *interface slicing* for the support of *reverse engineering* has first been proposed in [BE93, Bec93] on the level of modules for the purpose of component re-use. Interface slicing is meant to enhance conventional slicing by operating on a coarser granularity than on the level of statements. An interface slice is computed for a component with respect to a subset of the functionality offered by the component. The resulting slice is then a custom-tailored subset of the original module with all necessary functionality to provide the selected interface elements.

Measurement

Among the various features of software that can be quantified by some kind of metrics, such as its complexity or its quality, the attribute of *cohesion* of software is the one where program slicing has most frequently been applied. A cohesion metrics yields a quantification for how closely related the elements of a given

software unit are, i.e., how many dependences exist between them, an analysis purpose for which the program dependence graph as used for slicing obviously contains all ingredients needed to directly obtain suitable answers.

This natural relationship between slicing and module cohesion has soon been recognised by [OT89, Ott92, OT93, BO94, OB98], who defined a qualitative and quantitative metric for module cohesion based on *slice profiles*, which are essentially a number of slices with respect to each of the output variables of a given module. The comparison of such slice profiles is then used to define the overall cohesion of the given module: the more these profiles have in common, i.e., the more similar they are, the greater is the overall cohesion of the module.

Such slicing-based approaches to measurement of software cohesion have been extended with respect to several aspects, such as a finer grained analysis of the relationships between slice profiles [HDS+95b], the extension to object-oriented software [OBKM95, Gup97], or the application on a higher level of abstraction instead of program code like in the context of system architecture design [KB96] or system specifications [Lem94].

Another related attribute of software that is amenable to slice-based measurement is the *coupling* between its constituting modules, i.e., a measure for the degree to which each module relies on other modules by sending or receiving information from one to another. Coupling can therefore be regarded as being complementary to cohesion, since a low degree of inter-module dependences often goes along with a higher degree of cohesion within each involved modules and vice versa.

Probably due to this close relationship between cohesion and coupling, most of the previous works refer to the potential application of their approaches also to the measurement of cohesion, but only a few works really focus on this application of slicing such as [HOSD97], who use basically the same approach as for cohesion measurement, but introduce a finer grained definition of data dependence for computing more precise slice profiles and therefore a better suited metrics for coupling, and [Li01], who develop an object-oriented extension to slicing-based measurement of coupling.

Compiler Optimisation

In addition to its application on source code level, slicing can—like many other techniques coming from the area of program analysis—also be applied at the level below of directly human comprehensible source code, or rather during transformation between both levels, i.e., for the purpose of *compiler optimisation*. Possible application scenarios that have already been proposed by [FOW87] include parallelism detection, node splitting, code motion, loop fusion, branch deletion, loop peeling, and unrolling.

Elaborate examples of such applications are presented in [BG97], where slicing is applied for an improved approach to elimination of dead source code, i.e., the

identification and removal of unreachable program fragments, and in [PF01], where slicing is applied to extract garbage collection mechanisms from existing software in order to move its execution into a separate thread and thus to achieve higher performance and stability.

Verification

The area of automatic verification is probably the one, where slicing has seen the largest interest in recent years. With model checking suffering from the problem of state space explosion, any feasible state space reduction technique is most welcome. This applies all the more with slicing being a relatively cheap technique in terms of memory and time requirements. In addition to that, slicing can be applied at the earliest possible stage, i.e., before verification models have been generated. Therefore, slicing has the potential to facilitate all subsequent stages of verification.

Among the earliest works in this direction are [MT98, MT00] whose application of slicing is located on the border between traditional program slicing used for understanding of protocols and an actual extension of the application of slicing dedicated to verification purposes. The slicing targets of their approach are programs written in *Promela*, the input language of the model checker SPIN, while their slicing criteria correspond to the different possibilities of Promela for expressing properties about the programs. This means that slicing can be carried out with respect to *assertions* contained at arbitrary points within Promela programs, as well as with respect to *never claims*, i.e., the Promela construct used to define undesired system behaviour.

On the lower level of models that are generated from actual program code, the work on slicing associated with the Bandera tool set for verification of Java programs [HDZ00] has probably had the highest impact. There the slicing targets are programs written in a flowchart language (FCL) which can actually be regarded as models of the underlying Java program, from which they have been derived. Slicing criteria are temporal logic formulae that simultaneously form the verification properties with respect to which the program is intended to be verified.

Recently, experimental evaluations on the effectiveness of slicing for verification of Java programs with respect to intermediate assertions and with respect to freedom of deadlocks have been conducted [DHH+06], confirming that this kind of software verification indeed benefits very much from the application of slicing, even in presence of other reduction methods such as partial order and thread symmetry reductions.

Comparable slicing approaches have also been used for reducing SAL programs [GSS99, BGL+00, dMOR+04], the input to the explicit-state model checkers (SAL 1) and symbolic and bounded model checkers (SAL 2) of the same name with respect to temporal logic formulae. The slicing algorithm of SAL operates on transition systems computed from SAL programs and uses sets of variables as its

slicing criteria.

Slicing has also been carried out in the context of formal verification of SDL design specifications [BFG00, BGM01, JG01, BGO+04]. Within this slicing approach, SDL design specifications are first translated into an intermediate representation within the IF environment, an open validation platform for asynchronous timed systems. There, the actual slicing takes place with respect to verification properties such as freedom of deadlocks and response properties expressed in temporal logic formulae.

2.3 Further Techniques Aiming at State Space Reduction

In the application of automatic verification techniques, one can usually distinguish two different phases: a first phase comprising the specification of the system model in some high-level description notation, which is then, in a second phase, translated into a lower level system model, on which the actual verification techniques operate, such as model checking does.

With state space explosion being a fundamental problem of any such automatic verification technique, a lot of research effort has been invested into the development of methods that can limit the impact of building and searching the complete state space when analysing a given model with respect to the validity of a formula.

This section presents some of these techniques which are mostly applicable in addition to slicing and might thus complement the achievable reduction. We distinguish between *high-level techniques* on the one hand, working on the actual high-level description of the systems under examination, and *low-level techniques* on the other hand, working on the level of the models generated from the previously designed system specification.

2.3.1 High-Level Techniques

If trying to contain the effect of the state space explosion problem, it seems to be an obvious advantage if the resulting reduction can be obtained as early as possible, such that the associated benefit carries over to each of the subsequent stages of verification. This claim is not only the main motivation for the application of slicing in the context of verification, but also for a number of other techniques such as the three methods that will be presented next: program conditioning, aspect-oriented programming, and compositional verification.

Program Conditioning

The idea of *program conditioning* was first proposed as an additional pre-processing step to enhance the outcome of traditional slicing. Therefore, program conditioning has its origin, as program slicing, in the field of program analysis and was first used to support program debugging and program understanding [CCL98].

The starting point of program conditioning is an initial condition, which is assumed to hold at a particular point of a given program. The act of *conditioning* the program with respect to this assumption yields then a reduced version of the program, containing only statements that will be executed under this assumption. This allows removing conditional branches from the program, which are unreachable under the given assumption. Traditional slicing can then be applied to this reduced version of the program in order to obtain further reductions with respect to ordinary slicing criteria.

In different approaches to conditioned program slicing, several ways to obtain conditioning properties have been proposed, such as *weakest preconditions* [CH96], specifications enriched with intermediate *assertions* [LCYK01], *error traces* obtained from model checking [JM05], or *antecedents* of verification properties following the pattern of implications [VEA07],

Usually, the application of theorem proving and constraint solving techniques is necessary for the propagation of the initially injected conditions. Several such implementations have been developed, such as the close integration of slicing with constraint solving techniques within the *VALSOFT* tool for ANSI C programs [Sne96, KS98, SRK06], the implementation of conditioning based on the theorem prover SVC, operating on a limited subset of C code within the conditioned program slicer *ConSIT* [FDHH04, WZH05], or the symbolic execution algorithm based on the FermaT simplify decision procedure for the Wide Spectrum Language (WSL, [War02]), implemented within the light-weight program conditioner *ConSUS* [DDF+05].

Further variants of program conditioning include the approach of *pre/post conditioned slicing* [HHF+01], where a concept of forward and backward conditioning is developed and used to reduce programs with respect to pre and post conditions, respectively; also the approach of *abstract slicing* [HLS05] can be regarded as an integration of program conditioning into a slicing approach, since the abstractions therein are computed with respect to complementary predicates, i.e., each abstraction corresponds to reduced versions of the given program, as also obtained from program conditioning.

Aspect-Oriented Programming

The fundamental motivation of *aspect-oriented programming* [KLM+97] is to allow the separation of concerns right from the beginning of software development. The overall goal is to avoid a mixture of code dedicated to different purposes. Within a given software component, for instance, the aspects of technical infrastructure facilities, such as logging or error recovery, should not intermingle with the core functionality and should, moreover, be consistent within the system as a whole. Aspect-oriented programming allows for the separation of such cross-cutting concerns and thus improves maintenance and re-use of the resulting software.

Another desirable side effect of the aspect-oriented programming paradigm is the possible benefit obtained for software verification: the clean separation between different aspects allows easily obtaining models of the software, i.e., abstractions that are dedicated to a particular aspect, with respect to which the software needs to be analysed.

Therefore, aspect-oriented programming can be regarded as the anticipatory counterpart to program slicing: where program slicing intends to compute the part of a program relevant to a specific analysis purpose *after* an analysable version of the program has been finished, the intention of aspect-oriented programming is to allow this separation right from the start, i.e., the analysis purposes need to be clear *before* the actual analysis takes place [KFG04, LKR05].

Furthermore, in addition to the complementary and competing nature of both techniques, aspect-oriented programming has also been used as a basis for the enhanced computation of traditional program slices [IKI03].

Compositional Verification

In short, the idea of *compositional verification* [CLM89, HQR98] is to make the verification of large systems with respect to a global property feasible by breaking the system into manageable parts, which can then be verified with respect to suitable sub-properties, such that the conjunction of the individual sub-properties implies the desired global property for the full system.

The relation between slicing and compositional verification can be seen in two directions: on the one hand, slicing techniques are applicable to enhance the verification of components within a compositional verification framework just as they are applicable within any general verification approach in order to compute reduced version of the verification target with respect to the given verification property, which will be a sub-property of the verification framework.

On the other hand, however, slicing can even be used to enhance a compositional verification approach by computing decompositions of a larger system with respect to decompositions of the top-level verification property. Once the global property has been divided into sub-properties, these can then be used as a slicing criterion, such that the associated slices correspond to decompositions of the complete system. Such a decomposition approach using slicing is especially attractive, since only the decomposition of the verification property needs to be computed manually, while the computation of the decomposition slice can be carried out completely automatically [Met07].

2.3.2 Low-Level Techniques

In addition to the previously introduced high-level techniques, working on the level of human-understandable descriptions of systems, a lot of research effort

has also been spent on the improvement of handling the actual models which are automatically generated from high-level descriptions. Verification algorithms usually operate on this lower level and thus there is a very direct correlation between any optimisations introduced on this level and the efficiency of verification algorithms. While some of these enhancements are quite similar to slicing, such as cone-of-influence reduction or abstract interpretation, most of them are obviously orthogonal to the application of slicing, such as efficient model representation, partial order reduction, and predicate abstraction.

Efficient Model Representation

One of the most important techniques to achieve an efficient representation of the models, on which automatic verification algorithms operate, is the encoding of systems based on binary decision diagrams (BDD, [McM92]) and variants thereof that simultaneously represent a breakthrough for the application of model checking, since its introduction increased the tractable model sizes by several orders of magnitude. BDDs achieve this not only by representing the models in a way that is space-efficient for many practically relevant systems, but by also providing logical operations on these representations that have advantages with respect to time-efficiency.

With respect to slicing, the efficient representation of models can be regarded as being complementary in terms of run-time improvements, as is also confirmed by experimental evaluation [DHH+06]. The main reason for this complementary nature is that the removal of any system element by applying slicing before model generation does of course improve subsequent verification by also removing the necessity to represent the system element in the model; conversely, for any element that can not be removed by slicing, its efficient representation is all the more important.

Partial-Order Reduction

In short, the basic idea of *partial order reduction* [God95, Pel98, CGP99, Sto00] is to limit the analysis of concurrently operating system parts to the minimal necessary number of runs that are representative for all other possible system runs. To this end, the commutativity of concurrently executed system actions is exploited by not analysing each of their possible permutations, but only a single one representative of these, provided that the concurrent system actions can indeed take place independently from each other.

Therefore, also this low-level optimisation technique can be regarded as being complementary to slicing, since the analysis of interleaved parts of a system that remain present in the slice will certainly be improved by the application of partial order reduction, while the successful removal of such interleavings before model

generation removes the necessity to apply partial order reduction at this point at all and will thus further improve the analysis. This mutually useful relationship between slicing and partial order reduction techniques has also been experimentally investigated and confirmed [DHH+06].

Moreover, the application of a close integration of basic slicing techniques and a technique similar to partial order reduction has successfully been explored for the parallel composition of low level processes [BSV93].

Cone-of-Influence Reduction

A technique very similar to program slicing is the *cone-of-influence reduction* or *localisation reduction* first proposed by Kurshan [Kur94, CGP99]. All of these techniques are based on the computation of dependences between elements of the model to be analysed. These dependences are then exploited in order to determine those parts of the analysed system that are relevant with respect to the analysis purpose.

As also observed by [CFR+99], the main difference between slicing and cone-of-influence reduction, however, and therefore the reason for both techniques being complementary, are again the different stages at which they are applied. While the application of slicing on source code level allows achieving more complex reductions, the cone-of-influence reduction working on bit-level allows more localised and thus finer grained reductions, such that both techniques achieve reductions, even in addition to the presence of the respective other.

Predicate Abstraction and Abstract Interpretation

The application of *predicate abstraction* [GS97] within model checking approaches aims at computing finite versions of infinite state spaces by partitioning the state space based on suitable predicates over state variables. For the verification of certain kinds of properties it suffices then to check their validity for upper approximations obtained from predicate abstractions of the concrete systems. If the properties hold for the abstract version of the system, this implies then their validity for the concrete system. If the property does not yet hold for the abstract system, it might either be the case that the property does not hold for the original system either, or it might be necessary to refine the abstraction, giving rise to the iterative approach to counterexample-guided abstraction refinement model checking [CGJ+00, CGJ+03].

Clearly, the kind of reductions achievable by predicate abstraction can not be obtained from the application of slicing. On the other hand, however, the complete removal of dimensions of the state space by complete elimination of variables, as it is achievable by the application of slicing, will also not be possible by using predicate abstraction. Furthermore, predicate abstraction mainly aims at a reduction of the data state space of systems, while slicing also allows reducing the state space in

terms of the possible control flow.

The same observations hold for the concept of *abstract interpretation* [CC77, CC99], which can be regarded as a generalisation of predicate abstraction by defining a general framework in which firstly abstractions of the domain are computed, on which the concrete system operates and which additionally provides resulting abstract versions (interpretations) of the actions available in the concrete system.

However, integrations of slicing with predicate abstraction have been proposed [HLS05, BDFW07] that combine the advantages of both techniques by first computing predicate-based abstractions of system models which are then incrementally refined, following the counterexample-guided abstraction refinement approach to model checking, enhanced by intermediate slice computations.

3 Integrated Formal Specifications

Contents

Modelling complex systems as they arise in the area of safety-critical application domains usually involves the description of different views on the system. In the UML this is facilitated by providing designers with a large number of different diagram types, each covering a different of the various possible aspects of a system.

The same situation applies to formal specifications: although one single specification notation might be well suited for the expression of aspects that it was originally designed for, the specification of additional aspects might not at all be possible, or it might only be feasible by using some very unnatural or awkward constructions. Therefore, a similar approach as in the UML is also widespread in the area of formal modelling notations, where *integrated formal methods* allow for a convenient specification of different views onto a system.

Integrated formalisms combine different existing notations into one new formalism, while still giving a consistent semantics to the combination and thus preserving the formal rigour in a design, which is the key precondition for allowing a subsequent exact mathematical analysis of the specified system model.

Models of complex systems in integrated specification formalisms usually contain views describing state-based aspects as well as views describing the dynamic behaviour of the system, and often also another view, which is especially important in the area of safety-critical systems, namely their real-time requirements. These three most important aspects of complex safety-critical systems will be covered by the specification notations that we introduce in this chapter.

The presentation of the specification notations proceeds incrementally, following the chronology of their emergence as well as the degree of integration that they offer in terms of the number of different system aspects that can be naturally specified within the respective notation.

The most basic of these languages is Object-Z, which is mainly dedicated to covering the aspect of specifying the data that a system comprises and the data manipulations associated with the execution of system events. Object-Z extends the set-based specification language Z with object-oriented concepts like encapsulation of methods and attributes within classes and inheritance relations between classes.

Next, we introduce CSP-OZ, an integration of Object-Z (OZ) with the process algebra Communicating Sequential Processes (CSP), offering the possibility to define orderings of events by means of process definitions, which yields a much more intuitive way of specifying the additional aspect of system behaviour than only using the concepts available within pure Object-Z.

The final extension presented in this chapter is the specification language CSP-OZ-DC, which expands the vocabulary offered by CSP-OZ by means for expressing timing aspects of systems in terms of real-time properties that can be defined within a separate Duration Calculus (DC) part. This DC part contains so-called counterexample formulae, a subset of DC, allowing us to define exact timing relations between certain events or states by referring to events and variables that have been defined in the CSP and OZ part of the given class.

The following sections introduce each of the aforementioned specification languages by a running example which will appear again in the subsequent chapters during the further development of the slicing approach. Furthermore, they define syntax and semantics of the respective specification language and give hints on related work regarding comparable specification formalisms.

3.1 Object-Z Specifications

When starting to specify a system, the most basic question to answer is what attributes of the system need to be defined in order to obtain a complete characterisation. As soon as these attributes have been identified, the next obvious question is that about the facilities that the system offers in order to use it and, moreover, which effect these operations have on the previously defined attributes.

Several formal notations for the exact specification of such requirements have been proposed, as for instance the B method developed by Abrial [Abr96] or the notation of abstract state machines (ASM) proposed by Börger [BS03], Another such state-based approach is the language Z, a mathematical notation developed by Abrial and others at the Programming Research Group of the Oxford University Computing Laboratory (OUCL) since the late 1970s. Important milestones of the development of Z have been Spivey's reference manual [Spi89] and the ISO/IEC

0	1	2
3	4	5
6	7	8

Figure 3.1: Tic-Tac-Toe board with 9 positions in a 3-by-3 array

standardisation [ISO02] of the Z language.

This section introduces an object-oriented extension of Z, namely the notation of Object-Z [Smi92, DR00], or more precisely, a subset of Object-Z that corresponds to the Object-Z part of CSP-OZ specifications [Fis97]. In contrast to regular Z specifications, the Object-Z specifications that we consider here follow the object-oriented concept of encapsulation of data and methods within classes.

Furthermore, each of the operations offered by these classes is defined by means of two separate schemas that (1) define an enabling guard that has to be satisfied before the associated operation may take place and (2) define the effect on the state space that is associated with the operation upon its occurrence by relating class variables in their pre and post state, i.e., before and after the operation has taken place. However, we do not yet consider the object-oriented concept of inheritance relationships between classes nor phenomena associated with object instantiation during run-time.

Next, we will show how to specify systems with Object-Z by means of a small example specification, which we later also use for illustrating our slicing approach.

3.1.1 Example: Tic-Tac-Toe

The example that we use to illustrate the way of specifying systems with Object-Z is inspired by the specification of a Tic-Tac-Toe game presented by [DR00], but has been slightly modified to serve later on as a good example for slicing. Tic-Tac-Toe is a board game involving two players (which will be called black and white) and a board with nine positions in a three-by-three array as depicted in Figure 3.1.
The players take turns to move. A move consists of choosing a free position and adding it to the player's owned positions. The goal (in our modified version) is to obtain as many diagonal, vertical, or horizontal lines with three positions as possible. This is the difference to the usual Tic-Tac-Toe, where the player with the first line of three positions wins. The game ends when all positions are occupied. Only then the number of lines occupied by each player is evaluated and the final result of the game is determined.

The type *Posn* models the nine available fields that can be occupied by the players

on the Tic-Tac-Toe board,

$$Posn == 0..8$$

the function *inLine* determines, whether a set of positions contains a horizontal, vertical, or diagonal line with three positions,

$$\begin{array}{|l} inLine : \mathbb{P}\, Posn \rightarrow \mathbb{B} \\ \hline \forall\, ps : \mathbb{P}\, Posn \bullet \\ \quad inLine(ps) \Leftrightarrow \\ \quad\quad \exists\, s : \{\{0,1,2\},\{3,4,5\},\{6,7,8\},\{0,3,6\}, \\ \quad\quad\quad \{1,4,7\},\{2,5,8\},\{0,4,8\},\{2,4,6\}\} \bullet s \subseteq ps \end{array}$$

the function *lines* counts the number of currently present three-position lines within a given set of positions.

$$\begin{array}{|l} lines : \mathbb{P}\, Posn \rightarrow \mathbb{N} \\ \hline \forall\, ps : \mathbb{P}\, Posn \bullet \\ \quad lines(ps) = \\ \quad \#\{x,y,z : Posn \mid \{x,y,z\} \subseteq ps \wedge inLine(\{x,y,z\}) \bullet \{x,y,z\}\} \end{array}$$

The game has three possible outcomes:

$$Result ::= black\_wins \mid white\_wins \mid draw$$

Depicted in Figure 3.2 is the actual Object-Z specification of the class *TicTacToe*. The unnamed state schema defines the specification's state space by declaring the involved variables with their associated types. Moreover, it contains predicates

$$free = Posn \setminus (bposn \cup wposn)$$

and

$$over \Leftrightarrow (moves = 9)$$

which define state invariants, i.e., these predicate have to be satisfied initially as well as by any subsequent modification of the state space that might be associated with execution of the methods.

The initial configuration of the state space is defined in the `Init` schema by a set of predicates. These only need to hold initially, but may be falsified by any subsequent state space modification due to the execution of methods during further operations of the system.

As already mentioned, the notation that we use is not exactly Object-Z, but rather the Object-Z part of a CSP-OZ [Fis97] specification. The difference can be found

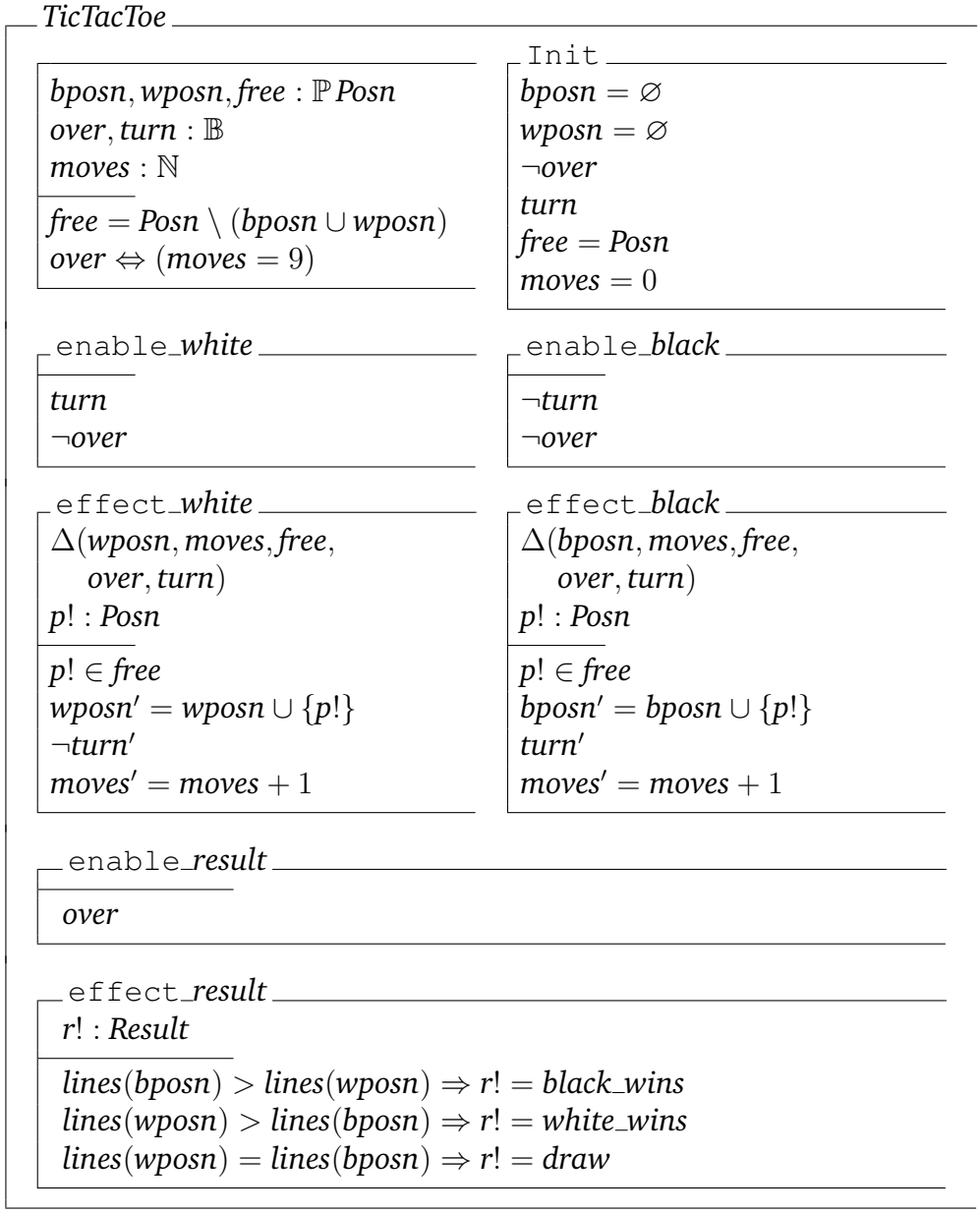


Figure 3.2: Tic-Tac-Toe specification

in the schemas for methods: a method *m* in CSP-OZ classes can be specified by giving an `enable` schema defining a guard to the execution of the method plus an `effect` schema defining the actual method execution.

Note in particular that no object references are present due to the communications that are used in CSP-OZ. Therefore, the aliasing problem does not occur.

Within each `effect` schema, a Δ-list can be given, which declares the set of variables, whose valuation might be modified by the associated method. Restrictions for such modifications can be given within the schema's predicates that relate the modified variables in their pre and post state by referring to them in primed (post state) and unprimed (pre state) versions. Furthermore, each `effect` schema might declare input variables (those decorated with a ?) and output variables (decorated with a !) which can then be used within the schema's predicates, either by referring to them or by imposing restrictions on the valuation of output variables.

For the rich vocabulary of mathematical symbols that the Z notation and thus also its object-oriented extension Object-Z offer, we refer to the Z reference manual [Spi89] and the ISO/IEC standard document for Z [ISO02].

3.1.2 Semantics of Object-Z Specifications

The usual semantics of Object-Z specifications is their history semantics [Smi95, Smi00], which defines the possible evolutions of the specified system in terms of its initial state space valuation along with a sequence of events taking place and the associated state space modifications induced by the events. This history semantics can then for instance be used as the basis for formal verification such as model checking Object-Z specifications with respect to temporal logic formulae [KS01].

With respect to slicing and the ultimate goal of extending the slicing approach to full CSP-OZ-DC, we now use a notion of labelled Kripke structures as the specifications' semantics that are defined according to the history semantics associated with an Object-Z class, similar to the approach that has been used in several works of Winter and Smith [WS03, WS02, SW03]. Labelled Kripke structures fit nicely into the framework of interpretations that will define the semantics of full CSP-OZ-DC specifications, since—as we will see later—untimed projections of such interpretations correspond directly to runs of the labelled Kripke structures that will now be defined as the semantics for Object-Z classes.

Labelled Kripke Structures

The temporal logic used to express verification properties, i.e., the formulae that we use later on as our slicing criterion for Object-Z specifications, will be interpreted on Kripke structures. Therefore, we will next define a Kripke structure semantics for Object-Z classes. The temporal logic will talk about both states *and* the execution of events (viz. methods). In contrast to ordinary Kripke structures, transitions of

labelled Kripke structures are thus labelled with events:

Definition 3.1.1 (Labelled Kripke structure)**.** *Let AP be a nonempty set of atomic propositions, E an alphabet of events (or method names).*

An (event-)labelled Kripke structure $K = (S, S_0, \rightarrow, L)$ *over AP and E consists of*

- *a finite set of states S,*
- *a set of initial states* $S_0 \subseteq S$*,*
- *a transition relation* $\rightarrow \subseteq S \times E \times S$*, and*
- *a labelling function* $L : S \rightarrow 2^{AP}$*.*

An infinite sequence of events and states $s_0 e_1 s_2 e_3 s_4 \ldots$ *is a* path *of the Kripke structure K iff* $s_0 \in S_0$ *and* $(s_i, e_{i+1}, s_{i+2}) \in \rightarrow$ *holds for all* $i \geq 0, i$ *even.*

A path $\pi = s_0 e_1 s_2 e_3 s_4 \ldots$ *is* fair *with respect to a set of events* $E' \subseteq E$ *(or* E'-fair*) iff* $inf(\pi) \cap E' \neq \varnothing$*, where*

$$inf(\pi) = \{e \in E \mid \exists \text{ infinitely many } i \in \mathbb{N} : e_i = e\}$$

denotes the set of events that occur infinitely often within π*.*

By convention we assume that paths are always infinite. This can be achieved by augmenting states s with no outgoing transitions by an extra transition $s \xrightarrow{\tau} s$, where τ is an internal event (e.g., as in the process algebras CSP and CCS).

Moreover, we will in the following only consider paths that are fair [CGP99] with respect to some set of events E'. This fairness requirement can be seen as an assumption on an environment, which infinitely often has to call methods (viz. events) from E'. Since an Object-Z class is not executing methods without a client calling them anyway, such fairness requirements are reasonable assumptions.

The Kripke structure semantics for an Object-Z class is obtained by taking all possible valuations of variables as *states* and state changes via execution of methods as *transitions*. The set of atomic propositions *AP* are the predicates over the class' variables, e.g., for class *TicTacToe* predicates $moves = 3$ and $free \neq \varnothing$ are possible atomic propositions. The set of events *E* are those which can be built from the methods by filling in values for inputs and outputs, e.g., the method *white* gives rise to events $white.3$, $white.4$, etc.

For convenience we will not make an explicit distinction between methods and events here and will not treat inputs and outputs. Therefore, we say that each class has a set of events *E* and for every such event there might be an *enable* and an *effect* schema.

Definition 3.1.2 (Object-Z class Kripke structure semantics)**.** *The* Kripke structure semantics *of an Object-Z class*

$$C = (State, Init, (\texttt{enable\_}e)_{e\in E}, (\texttt{effect\_}e)_{e\in E})$$

is the labelled Kripke structure $K = (S, S_0, \rightarrow, L)$ *over AP and E with*

- $S = State$*,*
- $S_0 = \{s \in S \mid Init(s)\}$ *a set of initial states,*
- *the transition relation* $\rightarrow = \rightarrow' \cup \{(s, \tau, s) \mid \nexists\, s' \bullet \nexists\, e : s \xrightarrow{e}{}' s'\}$*, where*

  $$\rightarrow' = \{(s, e, s') \mid \texttt{enable\_}e(s) \wedge \texttt{effect\_}e(s, s')\}, \text{ and}$$

- $L(s) = \{p \mid p \in AP \wedge s \Rightarrow p\}$.

In the further development of the slicing approach, we only consider Object-Z classes that satisfy the following two further assumptions: First, we assume the set of initial states to be nonempty ($\exists\, State \bullet Init$) and second, we assume for any *enable* schema to imply the pre-condition of its *effect* schema ($\forall\, e \in E : \texttt{enable\_}e \Rightarrow \text{pre}\, \texttt{effect\_}e$).

3.2 CSP-OZ Specifications

The notation of Object-Z that was introduced in the previous section does already offer a rich specification language that allows us to conveniently describe data aspects of systems based on a system's state space and its modifications that are associated with the possible events occurring within a system.

However, another important view on a complex system will be difficult to describe within Object-Z, which is the system evolution in terms of the admissible ordering of events, i.e., the system's *dynamic behaviour*. Although it might be feasible to express such requirements within Object-Z, as for instance by introducing auxiliary variables (similar to program counters) and associated predicates that enforce the intended system evolution, such an approach seems to be more of a workaround and would not offer the most natural way of specifying the intended system behaviour. Furthermore, the resulting specification would contain a mixture of specifying data aspects and behavioural aspects instead of maintaining a clean separation between both.

Therefore, an extension of the state-based notation of Object-Z is desirable that addresses the additional aspect of system behaviour in a natural way and simultaneously allows the separation of concerns, while it still provides a consistent semantics for the full formalisms.

A similar motivation has driven the emergence of a number of approaches that have been proposed in recent years. Mostly they are integrations that combine state-based notations like Z or B for describing data-oriented views on systems with process algebras like CCS or CSP for describing behavioural views on systems. Here we only mention some of the numerous examples of such combinations: Taguchi and Araki proposed the extension of CCS with Z expressions [TA97], Grieskamp, Heisel, and Dörr proposed the extension of state charts with Z expressions [GHD98], Schneider and Treharne employed CSP processes for controlling operations of specifications in the B notation [TS99, TS00, ST02, ST03], the International Organization for Standardization (ISO) fixed the enhancement of the LOTOS specification language by algebraic data types into E-LOTOS [ISO01], and, last but not least, Smith and Derrick [Smi97, SD01] as well as Woodcock and Cavalcanti [WC02] proposed combinations of Object-Z and CSP which are heading in the very same direction as the notation of CSP-OZ developed by Fischer [Fis97, Fis00] that we will consider here.

The integrated specification language CSP-OZ is the result of equipping Object-Z specifications with an additional CSP part that makes it—compared to pure Object-Z—much easier to also express behavioural properties of systems by defining the intended ordering of events within CSP process equations as will next be illustrated by means of a small introductory example specification.

3.2.1 Example: Untimed Air Conditioner System

For illustrating the approach of specifying systems using the notation of CSP-OZ we use the specification of a simple air conditioner system, or, to be more precise, the controller of such a system that will later on also serve as an example for slicing CSP-OZ specifications.

The air conditioner can operate in two modes, either heating or cooling. Initially the air conditioner is turned off. When it is switched on during the execution of event *workswitch*, it starts to run. While running, the air conditioner either heats or cools its environment and simultaneously allows the user to switch the mode with event *modeswitch*, to refill fuel with event *refill* or to switch it off again with event *workswitch*. Cooling or heating is modelled by a consumption of one unit of fuel during event *consume* and an emission of hot or cold air associated with event *dtemp*.

For the specification we first define the possible modes of operating of the air conditioner

$$Mode ::= heat \mid cool$$

and a type for specifying amounts of fuel:

$$Fuel == 0..100$$

The actual CSP-OZ specification of the air conditioner class is then depicted in Figure 3.3. The first part of the class defines its interface towards the environment in terms of a number of typed channels, over which the system may communicate with its environment via input parameters (those decorated with a ?) and output parameters (those decorated with a !).

The next part of the class specifies its dynamic behaviour, i.e., the allowed ordering of method execution. It is defined via a set of CSP process equations describing the cyclic workflow of the air conditioner. Its work cycle starts with an initial activation (`main`), followed by a process representing the activated air conditioner (*On*). This process in turn contains two interleaved processes modelling how the air conditioner can be controlled (*Operate*) and modelling the effect of its operation (*Work*). Only when both of these interleaved processes have terminated, the air conditioner can start with another iteration of its work cycle.

The CSP operators appearing in the example specification are

- prefixing $\rightarrow$ (denoting a process that communicates the prefixed event and then behaves like the attached process),
- sequential composition $;$ (denoting a process that initially behaves like the first process and after its termination behaves like the second process),
- interleaving $|||$ (denoting the special case of parallel composition with an empty synchronisation alphabet), and
- external choice $\Box$ (denoting the choice according to an external environment event that determines which of the processes is activated).

Further operators available in the CSP part but not appearing in the example include

- internal choice $\sqcap$, which represents that non-deterministic internal choice determines, which of the processes is activated,
- generalised parallel composition $\underset{A}{\|}$, which represents synchronisation of the involved processes on events from alphabet A, and
- alphabetised parallel composition ${}_A\|_B$, which represents synchronisation of the involved processes on events from the intersection of A and B, i.e., synchronisation on $A \cap B$.

The general syntax of CSP processes follows the grammar that is depicted in Figure 3.4.

The third and concluding part of a CSP-OZ class describes the attributes and the methods of the class. As for Object-Z specifications, attributes are defined within

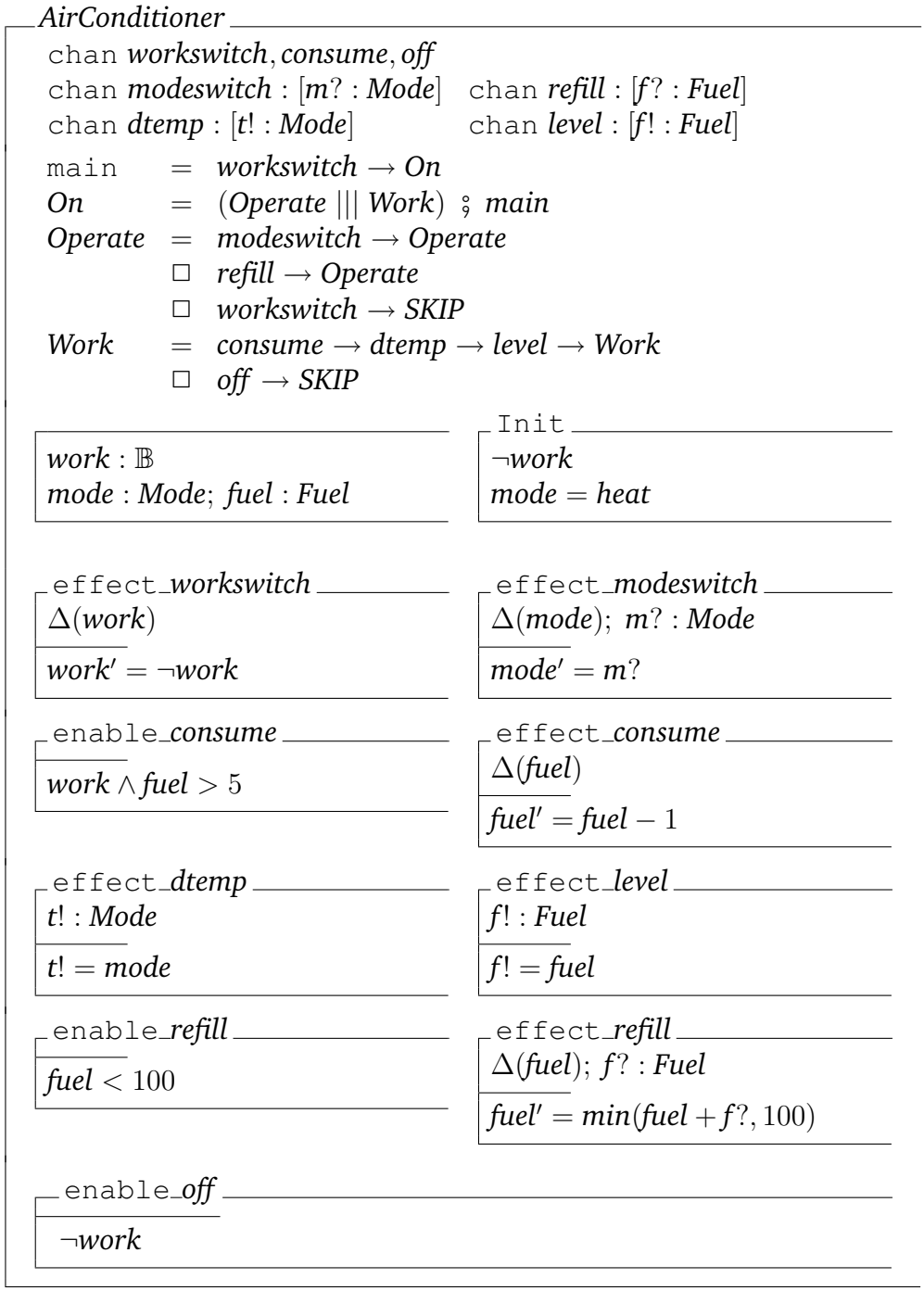


Figure 3.3: Untimed air conditioner specification

P	$::=$	$\texttt{Stop}$	deadlock
	$\mid$	$\texttt{Skip}$	termination
	$\mid$	$a \rightarrow P$	prefix operator
	$\mid$	$P_1 \sqcap P_2$	internal choice
	$\mid$	$P_1 \Box P_2$	external choice
	$\mid$	$P_1 \fatsemi P_2$	sequential composition
	$\mid$	$P_1 \underset{A}{\parallel} P_2$	parallel composition
	$\mid$	$P \setminus A$	hiding
	$\mid$	X	process call

Figure 3.4: Syntax of CSP processes with $a \in \mathit{Events}$ being an event and $A \subseteq \mathit{Events}$ being an alphabet, i.e., a set of events.

the unnamed state schema; in addition to that, the state schema may also define invariants that must hold throughout the execution of the class.

For every method appearing in the class interface and in the CSP part we might then have an `enable` schema fixing a guard for the method execution (enabling schemas equivalent to *true* are left out) and an `effect` schema describing the effect of a method upon execution. The valuations of input and output parameters that are declared within the class interface can be referenced or restricted by predicates within the associated method's `effect` schema.

For instance, for method *consume* the enabling schema tells us that the air conditioner has to be turned on and a minimal amount of fuel is necessary for *consume* to take place, while its `effect` schema expresses that upon execution of *consume* one unit of fuel is consumed. The method *level* on the other hand is always enabled; it just displays the level of fuel by setting the output parameter $f!$ to the current value of variable *fuel*. This output is then sent to the environment along the channel *level* that has been declared in the class interface.

Methods that are not referenced within the CSP part may occur at any arbitrary point of execution, supposed that the guard defined by predicates within their associated `enable` schema is satisfied.

3.2.2 Semantics of CSP-OZ Specifications

We define the operational semantics of CSP-OZ specifications in terms of labelled Kripke structures that result from a parallel composition of the labelled Kripke structures defined as the semantics for Object-Z specifications in the previous section and labelled Kripke structures that we will define next according to the operational semantics of the CSP part. The latter are comparable to the labelled transition systems defined by Fischer [Fis00] for CSP_Z, an extension of CSP that Fischer as well as Hoenicke [Hoe06] used to embed Z expressions into CSP when

developing their approaches of CSP-OZ and CSP-OZ-DC specifications, respectively.

The reason for using labelled Kripke structures rather than the definition used by Fischer is our ultimate goal of CSP-OZ-DC specifications. The semantics of CSP-OZ-DC will be defined in terms of sets of interpretations that satisfy the specification. Untimed projections of these interpretations, as they are appropriate for the untimed CSP and Object-Z parts of CSP-OZ-DC specifications, correspond directly to runs of labelled Kripke structures. The notion of labelled Kripke structures is the same as in the previous section on Object-Z specifications (cf. Definition 3.1.1).

The Kripke structure for a CSP-OZ class is derived in two steps: first, we separately compute the semantics of the CSP and the Object-Z part. In a second step, we combine the Kripke structure of the components by parallel composition.

In the following we assume a global set of atomic propositions AP and events E which are built over method names $m \in M$, i.e., an event e has the form $m.i.o$ where m is the name of a method and i and o are (potential) values for input and output parameters. The transition relation for the CSP part is computed via the operational semantics of CSP [Ros97].

Definition 3.2.1 (CSP part Kripke structure semantics)**.** *The* Kripke structure semantics *of the CSP part* `main` *of a CSP-OZ class is the labelled Kripke structure* $K^{CSP} = (\mathcal{L}^{CSP}, \{\texttt{main}\}, \rightarrow^{CSP}, L^{CSP})$ *with* $\mathcal{L}^{CSP}$ *being the set of all CSP terms,* $\rightarrow^{CSP}$ *the transition relation derived via the operational semantics of CSP and* $L^{CSP}(P) = AP$ *for all* $P \in \mathcal{L}^{CSP}$.

In the states of the Kripke structure for the CSP part all atomic propositions hold, since the CSP part makes no restrictions on values of attributes of the class.

Definition 3.2.2 (Object-Z part Kripke structure semantics)**.** *The* Kripke structure semantics *of the Object-Z part*

$$C = (State, Init, (\texttt{enable\_}m)_{m \in M}, (\texttt{effect\_}m)_{m \in M})$$

of a CSP-OZ class is the labelled Kripke structure $K^{OZ} = (State, Init, \rightarrow^{OZ}, L^{OZ})$ *with the transition relation*

$$\rightarrow^{OZ} = \{(s, m.i.o, s') \mid \texttt{enable\_}m(s,i) \wedge \texttt{effect\_}m(s,i,o,s')\},$$

and the labelling function $L^{OZ}(s) = \{p \in AP \mid s \models p\}$.

The states of the Kripke structure are simply the set of bindings of the state schema. These two Kripke structures are then combined via parallel composition. In the following we assume the alphabet of the CSP part and the set of methods in the Object-Z part to be equal, thus synchronisation takes places on all methods. Only one event remains which is executed by the CSP part alone, the invisible event τ which might arise out of internal choices in CSP processes.

Definition 3.2.3 (Parallel composition of labelled Kripke structures)**.** *The* parallel composition *of two labelled Kripke structures $K_i = (S_i, S_{0,i}, \rightarrow_i, L_i)$, $i \in \{1, 2\}$ over the same set of atomic propositions AP and events E, denoted by $K_1 \mid\mid K_2$, is the Kripke structure $K = (S, S_0, \rightarrow, L)$ with*

- $S = S_1 \times S_2$, $S_0 = S_{0,1} \times S_{0,2}$,
- $$\rightarrow = \left\{ ((s_1, s_2), e, (s_1', s_2')) \;\middle|\; \begin{array}{l} (s_1 \xrightarrow{e}_1 s_1' \wedge s_2 \xrightarrow{e}_2 s_2') \\ \vee\ (s_1 \xrightarrow{\tau}_1 s_1' \wedge s_2' = s_2) \\ \vee\ (s_2 \xrightarrow{\tau}_2 s_2' \wedge s_1' = s_1) \end{array} \right\}$$
- $L(s) = L(s_1) \cap L(s_2)$, *where* $s = (s_1, s_2)$.

Note that our definition is symmetric in general, while for the special case of parallel composition of CSP and Object-Z Kripke structures we assume that only the CSP side has τ transitions.

3.3 CSP-OZ-DC Specifications

The specification notations introduced so far allow us to specify data aspects of systems in terms of Object-Z data state space, invariants and methods operating on the data space as well as to specify behavioural aspects of systems in terms of CSP process definitions that define the admissible ordering of system events. One important aspect, however, is still missing, which is the specification of real-time properties that the specified system should obey, i.e., central requirements especially in the area of safety-critical systems. This concluding dimension is addressed in this section by adding another extension to CSP-OZ specifications, namely a Duration Calculus (DC) part, giving rise to the combined specification notation CSP-OZ-DC that was proposed by Ernst-Rüdiger Olderog and Jochen Hoenicke [Hoe01, HO02a, HO02b] and fully developed within the PhD thesis of Jochen Hoenicke [Hoe06].

Duration Calculus (DC) is a logic defined over real-time intervals that was developed within the ProCoS project [BHL+96] for mathematically describing the behaviour of real-time systems. The DC part of CSP-OZ-DC specifications is restricted to a subset of the full DC because the full language is too powerful to be checked automatically and thus would impede our goal of carrying out automatic verification of required properties for CSP-OZ-DC specifications [HM05a, HM05b]. In particular, the integration operator, a central element of Duration Calculus, is excluded from the DC formulae appearing in CSP-OZ-DC specifications, but also further restrictions apply in order to stay in the range of model-checkable specifications.

A number of comparable approaches to specifying real-time properties have been proposed, such as the extension of CSP with real-time operators, leading to

Timed CSP [Sch90, Dav93, DS94, Sch99], the extension of Object-Z with facilities to express real-time properties in terms of durations of operations, resulting in Real-Time Object-Z [SH02], an integration of Timed CSP with Object-Z [MD98, MD99, MD00], an integration of Timed CSP and Z [Sü02], or the extension of Circus ([WC02], a combination of Object-Z and CSP) with real-time operators similar to those of Timed CSP, resulting in Timed Circus [SJCS05, She06].

3.3.1 Example: Timed Air Conditioner System

For illustrating how to specify real-time systems with the notation of CSP-OZ-DC, we use a CSP-OZ-DC specification of an air conditioner system, which extends the air conditioner presented in the previous section by the following aspects: First, this extended specification does not consist of only one single class, but *parallel composition of classes* is now taken into account by combining the air conditioner class with a corresponding environment class. Second, both involved classes comprise the specification of *real-time properties* of the system.

Therefore, the main part of the *AirConditioner* class of the timed air conditioner system depicted in Figure 3.5 is identical to that of the untimed air conditioner system that we have seen in the previous section.

The only difference to pure CSP-OZ specifications emerges in the concluding *Duration Calculus* (DC) part of CSP-OZ-DC specification. This DC part defines real-time properties of the system within a number of DC *counterexample formulae*, i.e., formulae of a subset of DC which is amenable for later verification. A counterexample formula describes a single behaviour of the system that must *not* occur. Therefore, it not only defines the system's real-time properties, but it may also impose further restrictions on the admissible ordering of events.

Within the DC part of the *AirConditioner* specification, the only counterexample formula

$$\neg\Diamond(\lceil \mathit{work} = 1 \rceil \frown \updownarrow \mathit{workswitch} \frown \boxminus \mathit{off} \wedge \ell > 1)$$

specifies a kind of reaction property by negating an undesired behaviour. Positively, the formula expresses the following: whenever the air conditioner has been turned on for some non-zero time interval ($\lceil \mathit{work} = 1 \rceil$), which is followed by the occurrence of an event *workswitch*, an event *off* must follow within at most one time unit ($\neg(\ldots \ell > 1)$).

Note that the actually intended behaviour is specified by defining the opposite, namely an undesired trace of the system. Therefore, its negation allows only system behaviour that does not include the forbidden trace.

The expression $\lceil \mathit{work} = 1 \rceil$ denotes a non-empty time interval throughout which the predicate $\mathit{work} = 1$ holds. With $\updownarrow \mathit{workswitch}$ we refer to a point interval at which event *workswitch* occurs, while $\boxminus \mathit{off}$ represents a non-empty time interval

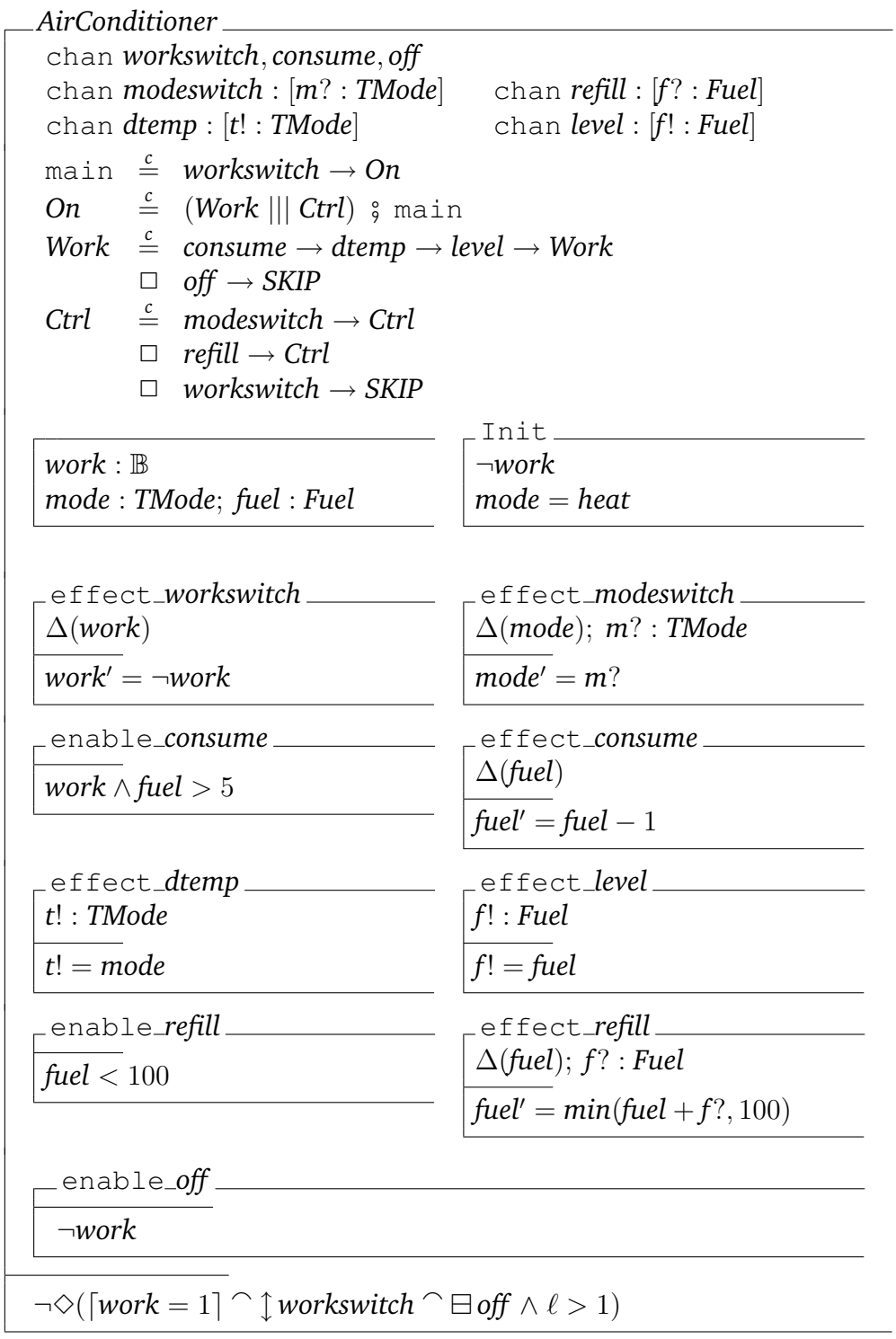


Figure 3.5: Timed air conditioner specification

CE	$::=$	$\neg(Ph \frown (Ph \mid Ev) \frown \ldots \frown (Ph \mid Ev) \frown true)$	counterexample traces, consisting of phase expressions Ph and event expressions Ev
Ph	$::=$	$(true \mid \lceil p \rceil)$	phase invariant, optionally with...
		$\lceil \wedge\ \ell \sim t \rceil$	... time bound ($\wedge\ \ell \sim t$) and ...
		$(\wedge \boxminus ev)^*$	... definition of forbidden events ($\boxminus ev$)
$\sim$	$::=$	$\leq \mid < \mid > \mid \geq$	time bound operator
Ev	$::=$	$\updownarrow ev$	required event: a zero-time (point) interval where event ev takes place
	$\mid$	$\not\updownarrow ev$	forbidden event: event ev must not take place at the given point interval
	$\mid$	$Ev \vee Ev$	disjunction of required/forbidden events
	$\mid$	$Ev \wedge Ev$	conjunction of required/forbidden events

Figure 3.6: Syntax of counterexample formulae CE with $ev \in Events$ being an event and p being a predicate over a set of state variables V.

without any occurrence of event *off*. The symbol ℓ abbreviates the length of the current interval.

The chop operator $\frown$ connects all three intervals, i.e., for each chop operator there must be some point in time such that prior to this point, the formula on the chop operator's left-hand side holds, while after this point, the formulae on the chop operator's right-hand side holds.

The diamond operator ($\Diamond$, "eventually") is an abbreviation for embracing the counterexample trace with true phases

$$\Diamond(\ldots) \equiv (true;\ \ldots;\ true)$$

and thus allows the trace to be located at an arbitrary point in time.

In general, DC counterexample formulae consist of a finite sequence of phase expressions and event expressions, which are embraced by an outer negation, following the grammar depicted in Figure 3.6.

An important restriction of counterexample formulae is to avoid exact time bounds of the form $\ell = t$. Without this restriction, the initiation of a measuring procedure would be required at any point in time that might be the beginning of a suitable interval of exact length. Since there might be infinitely many of such points in time, this would render the application of automatic verification methods infeasible.

The environment of the air conditioner is specified within a second class as depicted in Figure 3.7.

$LMode ::= brighten \mid darken$

Environment

```
chan tchange, lchange
chan dtemp : [t? : TMode]
chan dlight : [l? : LMode]
main  =  Temp ||| Light
Temp  =  dtemp → tchange → Temp
Light =  dlight → lchange → Light
```

Init

$temp = 0 \land light = 0$

$temp, light, dt, dl : \mathbb{Z}$

effect_dtemp

$t? : TMode;\ \Delta(dt)$

$t? = cool \Rightarrow dt' = -1$
$t? = heat \Rightarrow dt' = 1$

effect_dlight

$l? : LMode;\ \Delta(dl)$

$l? = darken \Rightarrow dl' = -1$
$l? = brighten \Rightarrow dl' = 1$

effect_tchange

$\Delta(temp)$

$temp' = temp + dt$

effect_lchange

$\Delta(light)$

$light' = light + dl$

$\neg\Diamond(\updownarrow dlight \frown \boxminus lchange \land \ell > 1)$

Figure 3.7: Timed air conditioner environment specification

This class communicates with the *AirConditioner* class via the channel *dtemp*, on which it receives information about the temperature difference that is induced by the air conditioner. According to this signal it then computes a new temperature value. In addition to the modelling of the actual physical temperature of the environment, this class simultaneously models the lighting situation via type *LMode*, which is possibly determined by some further components beyond the scope of our small example. Furthermore, the environment class also contains a DC part with again only one single counterexample formula. This formula specifies a requirement on the maximum time it should take for the environment to react with an event *lchange* to a preceding event *dlight*.

Finally, parallel composition of the air conditioner class and the environment

class with synchronisation on the set of common events (i.e., event *dtemp*, which transmits the temperature difference that is induced by the air conditioner) defines our complete example system:

$$System = AirConditioner \underset{\{dtemp\}}{\parallel} Environment$$

Intuitively, it is already quite obvious that the additional aspect of lighting in this specification is completely independent from the temperature. In Chapter 5 we will see how to automatically exploit this observation as part of the slicing result.

3.3.2 Semantics of CSP-OZ-DC Specifications

The compositional semantics of CSP-OZ-DC specifications has been defined by Hoenicke [Hoe06] in such a way that it consistently integrates the trace semantics for CSP [Hoa85], the history semantics for Object-Z [Smi00], and the semantics of Duration Calculus formulae [HZ97] by embedding the semantics for Object-Z and for CSP into the Duration Calculus semantics.

The standard semantics of Duration Calculus formulae is based on the definition of *state variables* (or *observables*) $P, Q, R, \ldots$, which are associated with *interpretations* that devise for each state variable a mapping from the time domain ($Time == \mathbb{R}^+$) to the values of $\{0, 1\}$, denoting the current valuation of the respective state variable.

For defining the semantics of CSP-OZ-DC specifications, the extended notion of interpretations defined by Hoenicke [Hoe06] is useful, such that the evolution of Z expressions and predicates over time becomes more directly expressible. The definition of these CSP-OZ-DC-interpretations is based on the following sets as defined in the ISO/IEC standard for Z [ISO02]:

- *NAME* denotes the set of all valid Z identifiers including generic Z symbols.
- $\mathbb{U}$ denotes the universe, i.e., the set of all possible semantic values, including values for generic Z symbols.
- $\mathbb{W}$ denotes the world, i.e., the set of all values from $\mathbb{U}$ that are not values of generic symbols.
- Model denotes the set of models with $\mathsf{Model} == NAME \pfun \mathbb{U}$.

Using these sets we can then define CSP-OZ-DC-interpretations as follows:

Definition 3.3.1 (CSP-OZ-DC-interpretation)**.** *A* CSP-OZ-DC-interpretation *is a function* $\mathcal{I} : \mathsf{Time} \to \mathsf{Model}$ *mapping the time domain* $\mathsf{Time} == \mathbb{R}^+$ *to the set of Z models. We require for such interpretations to have only a finite variability, i.e., on every finite interval* $[0, t]$*, the Z model obtained from* $\mathcal{I}$ *may change only a finite number of times.*

This notion of interpretation can be represented within usual Duration Calculus semantics by defining an additional state variable S for each Object-Z state expression S, such that the newly introduced state variable only has a value of 1 at those points in time, where our extended interpretation yields a model that satisfies the given state expression (with $\mathcal{I}_{DC}$ being the usual DC notion of interpretation):

$$\mathcal{I}_{DC}[\![S]\!](t) = \begin{cases} 1 & \mathcal{I}(t) \models S \\ 0 & \text{otherwise} \end{cases}$$

Consequently, we extend the notion of *observables* in a similar way as the notion of interpretation: As mentioned, ordinary Duration Calculus *observables* are Boolean-valued, while we are interested in the valuations of Z expressions and predicates. Therefore, the notion of *CSP-OZ-DC-observables* is useful, which are associated with elements of CSP-OZ-DC specifications, mapping the time domain to the current valuation of the respective elements.

Thus, the values of *CSP-OZ-DC-observables* depend on the point in time and may not only include Boolean values (as for example needed for channel variables, indicating the occurrence of events) but also include the domains of valuation of any element involved in CSP-OZ-DC specifications (as for instance valuations of arbitrary Object-Z variables), which is necessary to support complex data types that are used in the Object-Z part. We assume each CSP-OZ-DC-observable to have the same name as the specification element that it defines.

In the following, when using the terms “interpretation” and “observable”, we will not refer to the standard DC notions of interpretation and observable but rather to their extended versions that we have defined here.

The operational semantics of the CSP part requires us to be able to reason about CSP events which take place instantaneously. The straightforward translation into the setting of DC would be to define a DC state variable for each CSP channel that has a value of 1 only at points in time, where the associated event occurs and 0 otherwise.

The problem with such a definition, however, is that DC state variables can only be used for stable variables that keep their value for non-empty intervals. Thus, a different approach is necessary: In spite of this restriction, each CSP channel can indeed be represented by an associated DC state variable. However, the occurrence of an event needs to be defined in a different way, namely not via the actual value of the associated state variable, but via a change of its value at exactly that instant at which the event takes place.

An interpretation of a CSP-OZ-DC class then defines a set of *CSP-OZ-DC-observables*, i.e., time-dependent functions yielding valuations for

- all variables and constants that are used in the CSP-OZ-DC class, including variables declared in the state schema of the class and in axiomatic definitions in the CSP-OZ-DC class,

- the model of the environment, in which the class is declared,
- Boolean *channel variables* for all channels of the CSP-OZ-DC class, changing its value at each point in time when the associated event occurs,
- *parameter variables* for all channels equipped with parameters containing the parameter values at the point in time when the associated event occurs.

For satisfaction of the Object-Z part and the CSP part, the timing information included in interpretations is not relevant. Only information about the ordering of events and the sequence of the associated state space modifications is needed. Therefore, the following definition yields an abstract view of interpretations where time is not taken into account.

Definition 3.3.2 (Untimed projection of an interpretation)**.** *Let $\mathcal{I}$ be an interpretation, changing its valuation at points in time*

$$0 = t_0 < t_1 < t_2 < \ldots$$

from model $M_{i-1} \in \mathsf{Model}$ to $M_i \in \mathsf{Model}$ due to events e_i occurring at t_i, $i \geq 1$. Then the corresponding sequence of alternating states and events

$$\mathsf{Untime}(\mathcal{I}) = \langle M_0, e_1, M_1, e_2, M_2, \ldots \rangle$$

is the untimed projection *of $\mathcal{I}$.*

An interpretation is fair *with respect to a set of events $E' \subseteq$ Events (or E'-*fair*) iff $\mathit{inf}(\mathsf{Untime}(\mathcal{I})) \cap E' \neq \varnothing$ where*

$$\mathit{inf}(\mathsf{Untime}(\mathcal{I})) = \{e \in \mathsf{Events} \mid \exists \text{ infinitely many } i \in \mathbb{N} \colon e_i = e\}.$$

Note that this untimed sequence of alternating states and events corresponds directly to runs of labelled Kripke structures as defined in the previous sections on the semantics of Object-Z and CSP-OZ specifications.

The semantics of a CSP-OZ-DC class C, composed of its three parts C_{CSP}, C_{OZ}, and C_{DC}, is then provided by the set of interpretations that satisfy the given class, i.e., by interpretations $\mathcal{I}$ that simultaneously satisfy all three parts comprising the class as follows:

CSP part: $\mathcal{I} \models C_{CSP}$ iff $\mathsf{Untime}(\mathcal{I})$ corresponds to a run of the labelled transition system that is defined by the operational semantics of the CSP part [Hoa85].

Object-Z part: $\mathcal{I} \models C_{OZ}$ iff $\mathsf{Untime}(\mathcal{I})$ is in the history semantics of the Object-Z part [Smi00], i.e., its first valuation satisfies the *Init* schema of the Object-Z part, all its valuations satisfy the *State* schema of the Object-Z part, and all its events together with their pre- and post-states satisfy the `enable` and `effect` schemas of the associated method.

DC part: $\mathcal{I} \models C_{DC}$ iff $\mathcal{I}$ satisfies each of the DC formulae according to the semantics of DC [HZ97].

Parallel composition of two CSP-OZ-DC classes C_1 and C_2 is defined by conjunction of both classes $C_1 \wedge C_2$. To determine the semantics of such a parallel composition, we therefore first compute interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ that are in the semantics of the individual classes. If these individual interpretations agree on their common domain (such as the alphabet on which they have to synchronise, i.e., the valuation of channel variables and associated parameter variables), i.e., if $\mathcal{I}_1(t) \cup \mathcal{I}_2(t)$ yields a valid model, the interpretation $\mathcal{I}(t) = \mathcal{I}_1(t) \cup \mathcal{I}_2(t)$ is within the semantics of the parallel composition of both classes.

To argue about the events taking place at a given point in time within a given interpretation, we use the following function.

Definition 3.3.3 (Events taking place at a point in time)**.** *Let* $\mathcal{I} : \mathsf{Time} \to \mathsf{Model}$ *be an interpretation and t a point in time.* $\mathsf{TakesPlace}(\mathcal{I}, t)$ *is the set of events that take place in* $\mathcal{I}$ *at the point in time t:*

$$\mathsf{TakesPlace}(\mathcal{I}, t) = \{e \in \mathsf{Events} \mid \exists\, \varepsilon > 0 : \forall\, t_l \in [t-\varepsilon, t), t_r \in [t, t+\varepsilon] : \mathcal{I}(t_l)(e) \neq \mathcal{I}(t_r)(e)\}$$

Note that the definition of this function exploits the finite variability that we required for the interpretations of CSP-OZ-DC specifications: since an interpretation only changes finitely often within a finite interval, a suitable ε-interval around the point in time of an event's occurrence can always be found.

The next definition allows us to refer to a CSP process term that remains in a given interpretation at a given point in time.

Definition 3.3.4 (Residual CSP process term)**.** *Let* `main` *be the CSP part of a CSP-OZ-DC specification C and* $\mathcal{I}$ *an interpretation satisfying C with*

$$0 = t_0 < t_1 < t_2 < \ldots$$

being the points in time where $\mathcal{I}$ *changes and* $e_i \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$ *for* $i > 0$*. Then a* residual CSP process term *associated with a point in time, denoted by* $\mathsf{CSP}_C(\mathcal{I}, t_i)$*, is defined as a CSP process* P_i *with*

$$\texttt{main} \equiv P_0 \xrightarrow{e_1} P_1 \xrightarrow{e_2} \ldots \xrightarrow{e_i} P_i$$

being a valid transition sequence according to the operational semantics of the CSP part of C.

Note that $\mathsf{CSP}_C(\mathcal{I}, t)$ does not necessarily have to be unique. This, however, is no problem, since we will consider each possible result obtained from $\mathsf{CSP}_C(\mathcal{I}, t)$ at the points where we will use this definition for referring to residual CSP process terms.

4 Dependence Analysis

Contents

This chapter presents the analysis of the previously introduced notations with respect to the different types of dependences between the various elements involved in the respective specification formalism. The main result of this dependence analysis is the *dependence graph*, which contains all of the identified dependences and which is the foundation for the further steps of the slicing approach.

Each of the dependences defined in this chapter represents some kind of relation between specification elements, mostly following the principle of cause and effect, i.e., the source of a dependence edge represents the point, where the cause of the underlying incident is located, while the target of the same edge represents the point, where the associated effect will become evident. The simplest examples of this kind of relationship are data dependence edges: such an edge's source node represents the modification of some variable, while its target node references the same variable, such that the effect of the modification will then be visible.

Therefore, the dependences as a whole guarantee that, when tracing all dependence edges that leave from a certain specification element in a forward-oriented direction, we will find all specification elements, which are directly affected by the initial element. Conversely, they guarantee that, when tracing the edges that arrive

at the given element in a backward-oriented direction, we will find all specification elements that have some kind of influence on the initial element.

The conventional program dependence graph (PDG) usually represents data and control dependences that are present inside a program. For the dependence graph of our specification languages we derive several additional types of dependence from the rich syntactical structure of the Object-Z, CSP-OZ, and CSP-OZ-DC specifications, among them *predicate dependence* representing connections between schemas and associated predicates, *synchronisation dependence* representing mutual communication relations between processes or classes in parallel composition, and *timing dependence* representing timing relations between events or variables, which are derived from counterexample formulae within the DC part. Furthermore, the conventional data and control dependences are supplemented by several additional subtypes that are caused, for instance, by synchronisation relations or timing restrictions.

We proceed with the dependence analysis in an incremental way, dealing separately with each of the specification notations, each time taking previous definitions into account. We start again with Object-Z specifications, where control dependences will have a fixed structure, while data dependences are most important.

For CSP-OZ specifications, the control flow structure changes, since in addition to the fixed structure as defined for Object-Z specifications, the CSP part will be reflected within the control flow structure, resulting in new types of control dependences, but also leading to additional types of data dependences that were not needed previously.

Finally, for CSP-OZ-DC specifications we also have to consider the real-time part of specifications which will be represented by associated timing dependences. Here we also deal with parallel composition of classes that not only applies to CSP-OZ-DC specifications but also holds for both of the previous formalisms.

4.1 Object-Z Specifications

The dependence analysis for Object-Z specifications is based on a fixed control flow structure, since any method of an Object-Z class may take place at any arbitrary point of execution, provided its preconditions are satisfied. Therefore, as we will see in the next sections, control dependences are important for Object-Z specifications, but only need to be defined once in their general structure. Afterwards, this structure can be assumed for any Object-Z specification and no further individual analysis will be necessary, while the main object of the dependence analysis for Object-Z specifications will be the presence of data dependences.

Note that the general aspect of *parallel composition of several classes* will be covered within this chapter's concluding Section 4.3 on dependence analysis for CSP-OZ-DC classes, since it applies in the same way to all notations presented in

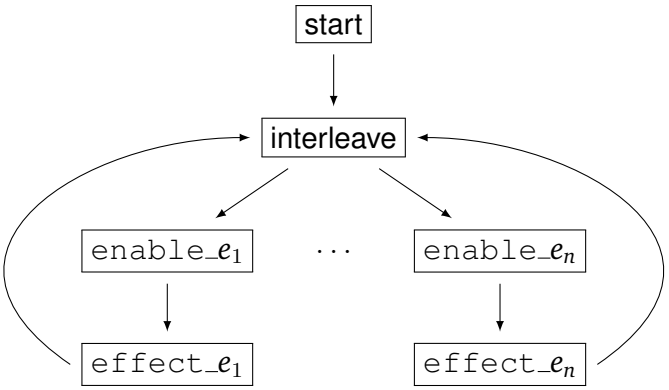


Figure 4.1: Control flow graph of a class

this chapter.

4.1.1 Control Flow Graph

The construction of the program dependence graph starts with the construction of the control flow graph (CFG). Each Object-Z class has a CFG structured as depicted in Figure 4.1.
Assuming E to be the set of methods (or events) of the class, the CFG then contains the following nodes:

- one dedicated entry node n_{start} labelled start, representing the initialisation of the class according to its `Init` schema,

- one node $n_{\text{interleave}}$ labelled interleave, representing the interleaving between any of the methods offered by the class that precedes every execution step (note that the interleaving takes place at the level of method execution, not at the level of schema evaluation, i.e., associated schemas `enable_`e and `effect_`e will always be simultaneously evaluated without interference of any further method f or any of its associated schemas `enable_`f or `effect_`f),

- for each method/event $e \in E$ two schema nodes n_{en_e} and n_{eff_e} labelled `enable_`e and `effect_`e, representing the method's associated `enable` and `effect` schemas.

These nodes are connected by the following *control flow edges* as illustrated in Figure 4.1:

- one edge leads from the unique start node to the interleave node

- for each method/event $e \in E$ three edges are introduced that lead from the interleave node via the `enable_`e and the `effect_`e nodes back to the interleave node.

Since Object-Z specifications do not impose further restrictions on the control flow, the CFG for each arbitrary Object-Z specification can be constructed according to this pattern. The nodes contained within the CFG also appear in the dependence graph that will be defined in the next section, while the control flow edges are not directly transferred, but their existence will be used to determine certain types of dependence.

4.1.2 Dependence Graph

In addition to the nodes of the CFG (N_{CFG}), the (program) dependence graph (PDG) contains nodes for each predicate $p \in Pred$ (where *Pred* is the set of predicates over V) inside a specification schema:

$$N_{pred} = \{p_x \mid p \in Pred \text{ is a predicate of schema node } x\}$$

Therefore, the set of nodes of the dependence graph is

$$N_{PDG} = N_{CFG} \cup N_{pred}.$$

Another important difference between both graphs is the set of edges they have. An edge connects two dependence graph nodes, if predicate, control, or data dependences exist between these nodes according to the definitions given below.

Before continuing with the construction of the dependence graph we first introduce some further abbreviations. When reasoning about paths inside the CFG, we let $path_{CFG}(n, n')$ denote the set of sequences of CFG nodes that are visited when walking along CFG edges from node n to node n'. When we refer to schemas or predicates associated with a dependence graph node n, we let

- V denote the set of all variables of the class,
- $out(n) \subseteq V$ denote all output variables of n (those decorated with a !),
- $in(n) \subseteq V$ denote all input variables of n (those decorated with a ?),
- $mod(n) \subseteq V$ denote $out(n)$ plus all variables of n being modified (those appearing in the Δ-list of the schema or in primed form in a predicate) plus—only for nodes n associated with the *Init* schema—all variables referenced by the *Init* schema,
- $ref(n) \subseteq V$ denote $in(n)$ plus all variables referenced by n (those appearing in unprimed form), and

- $vars(n) \subseteq V$ with $vars(n) = mod(n) \cup ref(n)$ denote all variables of n.

Next, we proceed with definitions of the further kinds of dependence types that establish the program dependence graph for Object-Z specifications.

Predicate Dependence

Each predicate that occurs within a CSP-OZ specification is located inside some schema. More precisely, each schema only contains exactly one predicate. This top-level schema predicate can in turn consist of the logical conjunction of several sub-predicates. This logical conjunction can either be given explicitly by combining the sub-predicates with the logical and-operator such as

enable_a

$$a = b \wedge c = d$$

or it can be given implicitly by enumerating the involved sub-predicates within several individual lines of the schema, each containing one of the sub-predicates such as

enable_a

$$a = b$$
$$c = d$$

In the following, we exploit this explicit or implicit decomposition of top-level schema predicates into sub-predicates. Consequently, when referring to the predicates of a schema, we do actually not refer to the top-level predicate such as $a = b \wedge c = d$, but rather to its sub-predicates such as $a = b$ and $c = d$, whose logical conjunction yields the top-level schema predicate. In particular, the previously introduced set of predicate nodes N_{pred} does only contain nodes representing such sub-predicates.

The idea of *predicate dependence* edges

$$\xrightarrow{pred} \subseteq (N_{CFG} \times N_{pred} \cup N_{pred} \times N_{CFG})$$

is to transfer this relation between schemas and the (sub-)predicates they contain by connecting the associated schema nodes with the (sub-)predicate nodes they contain.

For predicates of `enable` schemas, these edges lead from the `enable` node to its predicates and vice versa, while for predicates of `effect` schemas there are only edges in the direction from the `effect` schema to its predicates, i.e., two

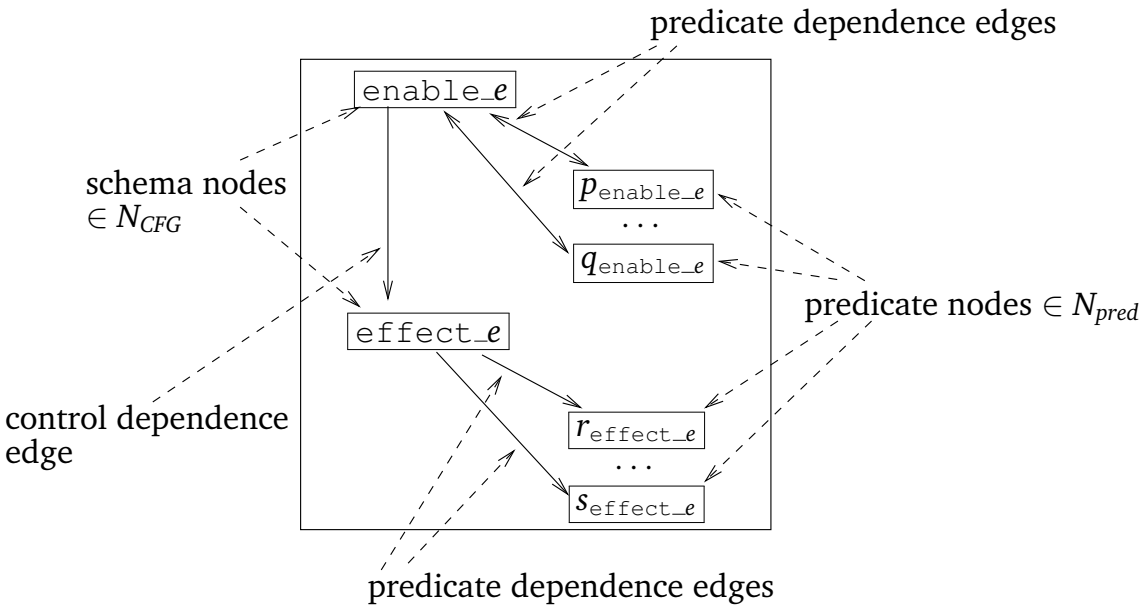


Figure 4.2: Predicate nodes and predicate dependence edges

nodes n and n' are connected by a predicate dependence edge $n \xrightarrow{pred} n'$ iff

$$\begin{aligned} & (n = x \wedge n' = p_x) && \text{[enable and effect schema predicates]} \\ \vee\ & (n = p_x \wedge n' = x \wedge \exists e \in E : x \equiv \texttt{enable\_}e). \\ & && \text{[enable schema predicates only]} \end{aligned}$$

The different treatment of `enable` and `effect` schema predicates, as illustrated in Figure 4.2, provides a way to represent the tighter connection between `enable` schemas and its predicates: `enable` predicates do not only depend on the event they are associated with, but also serve as the event's guard, i.e., a mutual dependence exists, while this is not the case for events of an `effect` schema.

Predicate nodes belonging to the `Init` schema are attached to the associated `Init` node in the way like predicate nodes belonging to an `effect` schema are attached to the associated `effect` node. This reflects the initial restriction of the state space according to the `Init` schema.

Predicate nodes associated with the *state invariant* are replicated and attached to each `effect` node. This represents a kind of normalisation and reflects the invariant restriction of the state space according to the state invariant that must hold at each occurrence of an event.

If a variable appears in the Δ-*list* of an `effect` schema, but is not referred to inside the remainder of the schema, a fresh predicate node is introduced for this variable and attached to the corresponding `effect` schema. This reflects the unrestricted modification of the corresponding variable by the associated event.

Control Dependence

The further construction of the dependence graph starts with the introduction of *control dependence* edges:

$$\xrightarrow{cd} \subseteq N_{CFG} \times N_{CFG}$$

The idea behind these edges is to represent the fact that an edge's source node controls whether its target node will be executed. In particular, a node cannot be control dependent on itself. The main type of control dependence is the following:

- A *control dependence edge due to nontrivial precondition* exists between an `enable` node and its associated `effect` node iff its associated `enable` schema is non-empty (i.e., not equivalent to true), i.e., for two nodes n and n' a control dependence edge due to nontrivial precondition $n \xrightarrow{cd} n'$ exists iff

$$\exists e \in E: \quad n \equiv \texttt{enable\_}e \wedge n' \equiv \texttt{effect\_}e \\ \wedge \exists p_e : n \xleftrightarrow{pd} p_e.$$

Therefore, only nodes representing events with non-trivial guards are source of this kind of control dependence edges.

In addition to this main type of control dependence, one further type of control dependence edge is introduced in order to achieve a well-formed graph:

- A *control dependence edge due to start* exists between the start node and its immediate CFG successor, i.e., the interleave node.

Finally, all previously defined (direct) control dependence edges are extended to CFG successor nodes as long as they do not bypass existing control dependence edges:

- An *indirect control dependence edge* exists between two nodes n and n' iff

$$\exists \pi \in path_{CFG}(n, n') : \forall m, m' \in \operatorname{ran} \pi : m \xrightarrow{cd} m' \Rightarrow m = n.$$

The idea of this latter definition is to integrate indirectly dependent nodes into the dependence graph that would otherwise remain isolated. The effect of indirect control dependence edges is illustrated in Figure 4.3 that also contains examples of the previous types of control dependence.

The graphs in Figure 4.3 are supposed to be derived from a specification that contains only two methods a and b. Moreover, we suppose that the specification's `enable_`a schema contains some predicate, while the specification's `enable_`b schema is supposed to be an empty schema. This leads to the control dependence edges as shown in Figure 4.3b:

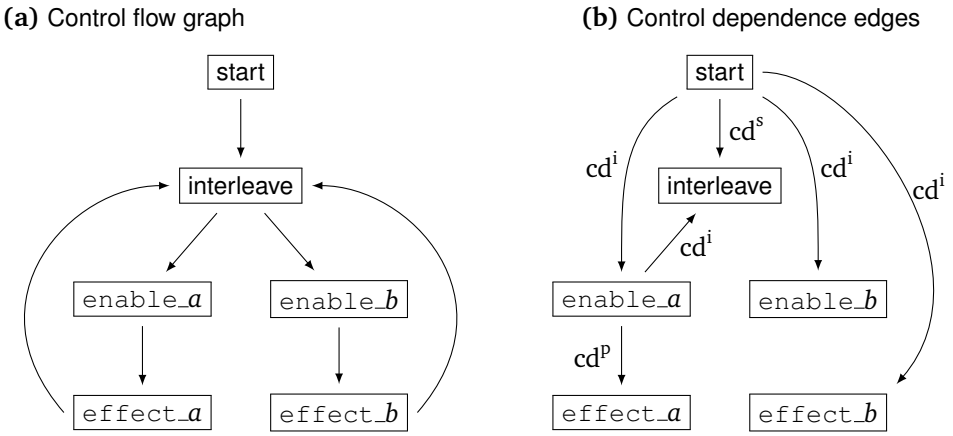


Figure 4.3: Example of different types of control dependence: control dependence edge due to start (cd^s), control dependence edge due to nontrivial precondition (cd^p), indirect control dependence (cd^i).

- One *control dependence edge due to start* (cd^s), leading from start to interleave.
- One *control dependence edge due to nontrivial precondition* (cd^p), leading from enable_a to effect_a.
- Three *indirect control dependence edges* (cd^i), leading from start to enable_a, enable_b, and effect_b. These edges result from extending the direct control dependence edge between start and interleave to its indirectly dependent CFG successors enable_a, enable_b, and effect_b.
- One *indirect control dependence edge* (cd^i), leading from enable_a to interleave. This edge results from extending the direct control dependence edge between enable_a and effect_a to its indirectly dependent CFG successor interleave.

The motivation for this relatively complicated definition of indirect control dependence will become clearer when dealing with CSP-OZ specifications in the next section. For now, this definition affects all `enable` nodes and `effect` nodes of events with empty `enable` schemas. The definition causes that these nodes are attached to the start node by means of a control dependence edge.

Data Dependence

The idea of *data dependence* edges

$$\xrightarrow{dd} \subseteq N_{pred} \times N_{pred}$$

is to represent the influence that one predicate might have on a different predicate by modifying some variable that the second predicate references. Therefore, the source node always represents a predicate located inside an `effect` schema, while the target node may also represent a predicate located inside an `enable` schema. We distinguish the following types of data dependence edges:

- *Direct data dependence* exists between two predicate nodes p_x and q_y (appearing in schemas x and y, respectively) iff there is a relevant variable that is modified by predicate p_x and referenced by predicate q_y and there is a CFG path between both associated schema nodes without any further modification of the relevant variable, i.e., iff

$$\exists v \in (mod(p_x) \cap ref(q_y)), \exists \pi \in path_{CFG}(x,y): \\ \forall m \in \operatorname{ran} \pi: v \in mod(m) \Rightarrow (m = x \vee m = y)$$

Note that the CFG path needs only to be considered in order to avoid data dependence edges leading from `effect` schema predicates back to predicates of the `Init` schema.

- *Symmetric data dependence* exists between two nodes p_x and q_y iff they are associated with the same schema and share modified variables, i.e., iff

$$mod(p_x) \cap mod(q_y) \neq \varnothing \wedge x = y$$

4.1.3 Example: Tic-Tac-Toe Dependence Graph

The program dependence graph of the class *TicTacToe* can be found in Figure 4.4.

For better readability, predicate dependence edges are abbreviated by the introduction of super nodes: schema edges form the title of these super nodes, and all predicate nodes belonging to the respective schema are contained within the super node. The different types of predicate dependence edges (bidirectional for `enable` schemas and unidirectional for `effect` schemas) and their associated predicate nodes are not represented explicitly, but are only implicitly present according to the type of schema a super node represents.

Control dependence edges between two super nodes stand for control dependence edges between the respective schema nodes. It can be seen that due to normalisation the effect schemas have two extra predicates which appeared in the original specification as the state invariant.

The example contains all previously defined types of dependences except for indirect control dependence edges between `enable` nodes and the start node which have been left out for better readability:

- Control dependence due to *nontrivial precondition* exists between all `enable` nodes and their associated `effect` nodes, since the example does not contain any empty `enable` schema.

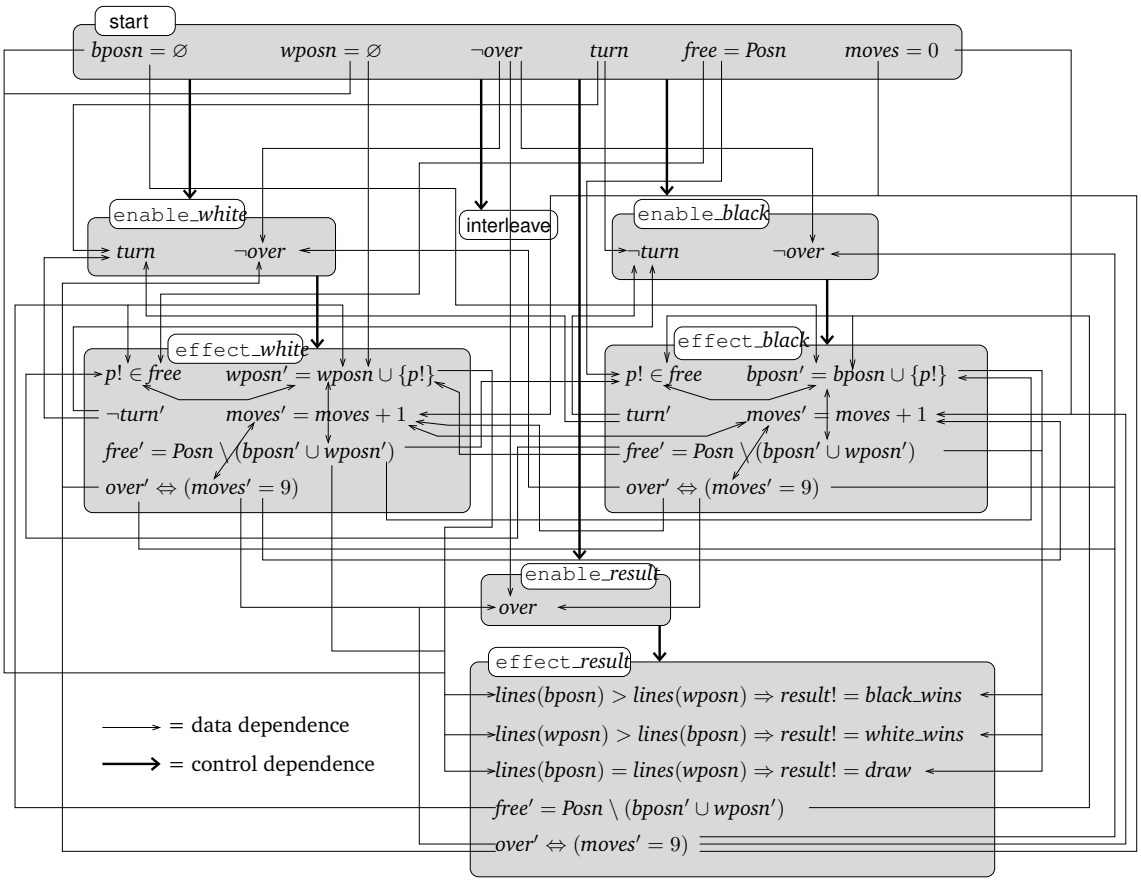


Figure 4.4: Dependence graph of the Object-Z class *TicTacToe*

- *Indirect* control dependence edges appear between the start node and each of the `enable` nodes.
- *Direct data* dependence exists between various predicate nodes, connecting nodes of the `Init` schema with subsequent references as well as connecting nodes of all `effect` schemas with nodes of other `enable` and `effect` schemas.
- *Symmetric data* dependence exists between predicate nodes within the `effect` schemas of the *white* and the *black* method.

The example dependence graph illustrates the tight connection between the various specification elements. In spite of this seemingly completeness of mutual connections, not all specification elements are in touch with each other, such that we nevertheless achieve a result by applying slicing to this example, as we will see in the next chapters.

4.2 CSP-OZ Specifications

In comparison to pure Object-Z specifications, the main difference that we need to consider within CSP-OZ specifications is the *variable control flow structure* that results from the CSP part. The control flow structure of Object-Z specifications was fixed, such that the same pattern of control flow graph was applicable to any arbitrary Object-Z specification. In contrast to that, the control flow graph for a CSP-OZ specification does not follow a fixed pattern, but needs to be newly computed for each CSP-OZ specification, such that it correctly represents the execution order of the specification's schemas according to the specification's CSP process definitions.

The subsequent *dependence analysis* for CSP-OZ specifications then relies on this control flow graph in order to compute the actual control and data dependences. In comparison to Object-Z specifications, the new and variable structure of the control flow graph of CSP-OZ specifications also leads to further types of dependences that arise in the dependence analysis.

Note that the general aspect of *parallel composition of several classes* will be covered within this chapter's concluding Section 4.3 on dependence analysis for CSP-OZ-DC classes, since it applies in the same way to all notations presented in this chapter.

4.2.1 Control Flow Graph

The control flow graph (CFG) for CSP-OZ specifications is derived inductively from the specification's CSP part. The only exceptions, where the Object-Z part is taken into account, are pure Object-Z methods that are only defined in the Object-Z part but not referenced within the CSP part.

The CFG nodes $n \in N_{CFG}$ and edges $\longrightarrow_{CFG} \subseteq N_{CFG} \times N_{CFG}$ forming the CFG $G_{CFG} = (N_{CFG}, \longrightarrow_{CFG})$ are thus mainly derived from the syntactical elements occurring in the CSP part. As we will see in this section, CFG nodes either correspond

- to CSP events (which, of course, correspond to schemas of the Object-Z part like `enable_`e and `effect_`e),
- to operators in the CSP part (like nodes interleave and uninterleave for operator $|||$, nodes extch and unextch for operator $\Box$, or nodes par_S and unpar_S for operator $\underset{S}{\|}$), or
- to the structuring of the CSP process definitions (like start.P and term.P for entry and exit points of a CSP process P, or call.P and ret.P for call and return points of references to a process P).

For *multiple occurrences* of Object-Z methods or CSP operators inside the CSP process definitions unique CFG nodes are introduced. This is achieved by a suitable

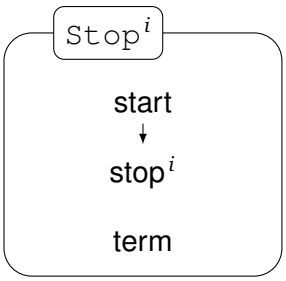


Figure 4.5: Control flow graph for the CSP deadlock operator `Stop`

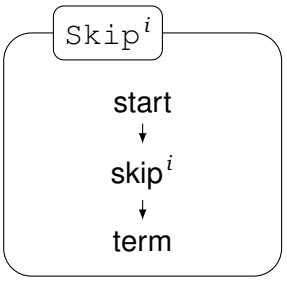


Figure 4.6: Control flow graph for the CSP termination operator `Skip`

naming convention of the associated `enable` and `effect` nodes, where the methods' names are extended by an ordinal, referring to their syntactical occurrence inside the CSP process definitions.

The i'th occurrence of event a in the CSP part is for instance associated with CFG nodes `enable_`a^i and `effect_`a^i. Analogously, the i'th occurrence of operator `Stop` in the CSP part is associated with a CFG node stop^i.

Next, we define the inductive construction of the CFG by devising individual graph transformations for each of the elements of the CSP part. The starting point of the construction is the definition of the `main` process.

A roughly comparable construction has been developed by Goltz [Gol88] for the representation of CCS terms by suitable Petri nets. However, the similarity is restricted only to the structure of the resulting graphs, while both structures serve completely different purposes.

Deadlock

The CSP `Stop` operator represents a non-terminating process. In the associated control flow graph depicted in Figure 4.5 this is reflected by the non-existence of an edge between the stop node and the term node:

$$G_{CFG}(\texttt{Stop}^i) := \begin{pmatrix} \{\mathsf{start}, \mathsf{stop}^i, \mathsf{term}\}, \\ \{(\mathsf{start}, \mathsf{stop}^i)\} \end{pmatrix}$$

Termination

The CSP `Skip` operator represents the termination of a process. In the associated control flow graph depicted in Figure 4.6 this is represented by two edges leading from the start node via the skip node to the term node:

$$G_{CFG}(\texttt{Skip}^i) := \begin{pmatrix} \{\mathsf{start}, \mathsf{skip}^i, \mathsf{term}\}, \\ \{(\mathsf{start}, \mathsf{skip}^i), (\mathsf{skip}^i, \mathsf{term})\} \end{pmatrix}$$

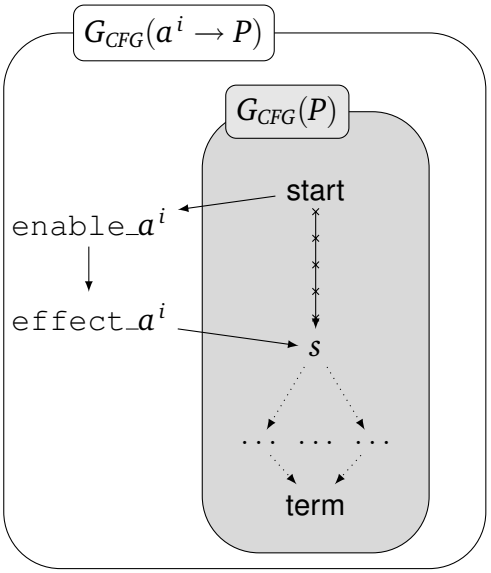


Figure 4.7: Control flow graph for the prefix operator $a^i \rightarrow P$

Prefix Operator

The control flow graph of a CSP process that starts with a prefix operator of the form $a^i \rightarrow P$ is obtained from the control flow graph of the subsequent process P by replacing its edge originating from the start node of P with an edge to the same target node, but originating from a newly introduced node $\texttt{effect\_}a^i$ which in turn has an arriving edge coming from another newly introduced node $\texttt{enable\_}a^i$. This pair of nodes represents the prefixed communication. Furthermore, an edge from the start node of P to the new node $\texttt{enable\_}a^i$ is added.

Since there is no new node introduced for the prefix operator, it is—in contrast to other operators—not necessary to distinguish it from its other textual occurrences, i.e., here it is not necessary to append an exponent to it. Nevertheless, the exponent is needed for the newly introduced pair of prefixed communication nodes in order to distinguish them from different occurrences of the same associated communication in the specification.

Given the previously computed control flow graph $G_{CFG}(P)$ of the involved process P, the control flow graph associated with the prefix operation $a^i \rightarrow P$ is depicted in Figure 4.7 and is defined as follows:

$$G_{CFG}(a^i \rightarrow P) := \left(\begin{array}{l} N_P \cup \left\{ \texttt{enable\_}a^i, \texttt{effect\_}a^i \right\}, \\ \left(\begin{array}{l} E_P \cup \left\{ (\mathsf{start}, \texttt{enable\_}a^i) \right\} \\ \quad \cup \left\{ (\texttt{enable\_}a^i, \texttt{effect\_}a^i) \right\} \\ \quad \cup \left\{ (\texttt{effect\_}a^i, s) \mid \exists (\mathsf{start}, s) \in E_P \right\} \end{array} \right) \\ \setminus \left\{ (\mathsf{start}, s) \in E_P \right\} \end{array} \right)$$

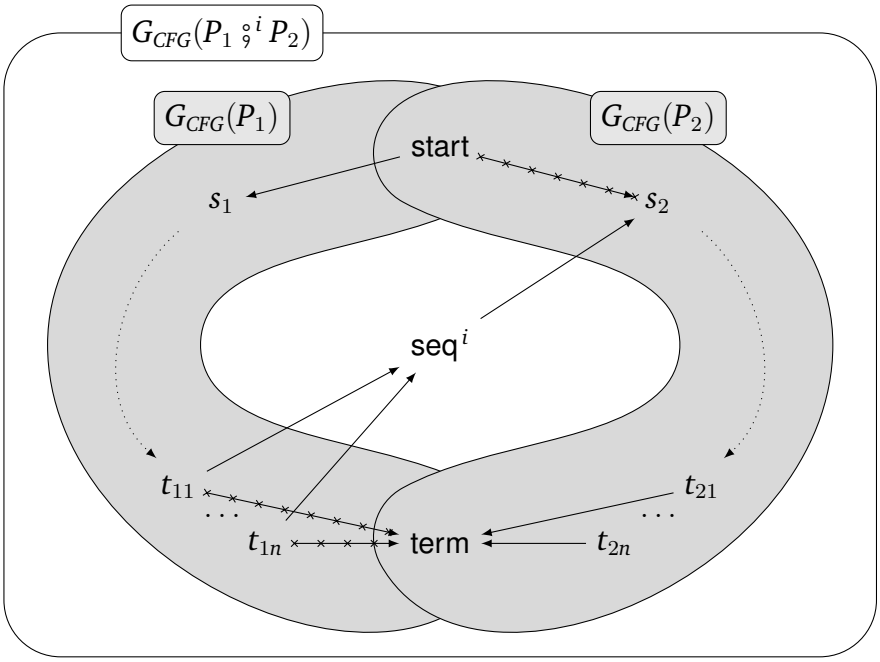


Figure 4.8: Control flow graph for the sequential composition operator $P_1 \,{}^{\circ}_{9}{}^{i}\, P_2$

Sequential Composition of Processes

Sequential composition of processes in the specification of the form

$$P_1 \;{}^{\circ}_{9}{}^{i}\; P_2$$

is represented in the resulting control flow graph by adding a newly introduced seq node to the control flow graphs of the associated subprocesses. Both subprocesses are connected via this node by replacing all edges that lead from nodes of the first process' control flow graph to its term node by edges that lead to the seq node. Additionally, any edge leading from the start node to a node of the second process' control flow graph is replaced by an edge with the same target, but originating in the seq node.

Given the control flow graphs of the involved processes P_1 and P_2, i.e., graphs $G_{CFG}(P_1)$ and $G_{CFG}(P_2)$, the control flow graph associated with their sequential composition $P_1 \;{}^{\circ}_{9}{}^{i}\; P_2$ is depicted in Figure 4.8 and is defined as follows:

$$G_{CFG}(P_1 \,{}^{\circ}_{9}{}^{i}\, P_2) := \left(\begin{array}{l} N_1 \cup N_2 \cup \left\{\mathsf{seq}^i\right\}, \\ \left(\begin{array}{l} E_1 \cup E_2 \\ \cup \left\{(t, \mathsf{seq}^i) \mid \exists (t, \mathsf{term}) \in E_1\right\} \\ \cup \left\{(\mathsf{seq}^i, s) \mid \exists (\mathsf{start}, s) \in E_2\right\} \end{array} \right) \\ \setminus (\{(t, \mathsf{term}) \in E_1\} \cup \{(\mathsf{start}, s) \in E_2\}) \end{array} \right)$$

Parallel Composition of Processes

From a control flow point of view, parallel compositions of processes of the form

$$P_1 \ |^i \ P_2 \quad \text{with the symbol } | \text{ representing an operator out of the set } \left\{ \underset{A}{\|}, \ {}_A\|_B, \ |||, \ \sqcap, \ \Box \right\}$$

can all be translated into the same operation on the associated subprocesses' control flow graphs. This operation consists in

- adding a new pair of op and unop nodes to the control flow graphs of the composed processes,
- replacing the edges originating from the start node by edges originating from the new op node that lead to the same target node as before,
- replacing the edges leading to the term node by edges leading to the new unop node that leave from the same source node as before, and
- finally adding edges from the start node to the new op node and from the new unop node to the term node.

In order to preserve the semantics of the specific operator in the control flow graph the original operator can be appended as an index to the op node, e.g., $\mathsf{par}_{\sqcap}$ or $\mathsf{par}_{\underset{A}{\|}}$.

Given the control flow graphs of the involved processes P_1 and P_2, i.e., graphs $G_{CFG}(P_1)$ and $G_{CFG}(P_2)$, the control flow graph associated with their parallel composition $P_1 \ |^i \ P_2$ is depicted in Figure 4.9 and is defined as follows:

$$G_{CFG}(P_1 \ |^i \ P_2) := \left(\begin{array}{l} N_1 \cup N_2 \cup \left\{ \mathsf{par}^i, \mathsf{unpar}^i \right\}, \\ \left(\begin{array}{l} E_1 \cup E_2 \\ \cup \left\{ (\mathsf{par}^i, s) \mid \exists (\mathsf{start}, s) \in (E_1 \cup E_2) \right\} \\ \cup \left\{ (t, \mathsf{unpar}^i) \mid \exists (t, \mathsf{term}) \in (E_1 \cup E_2) \right\} \\ \cup \left\{ (\mathsf{start}, \mathsf{par}^i) \right\} \cup \left\{ (\mathsf{unpar}^i, \mathsf{term}) \right\} \end{array} \right) \\ \setminus \left\{ (\mathsf{start}, s) \in (E_1 \cup E_2) \right\} \\ \setminus \left\{ (t, \mathsf{term}) \in (E_1 \cup E_2) \right\} \end{array} \right)$$

Process Definition

A process definition in the CSP part of CSP-OZ specifications is a unique line of the form

$$X \ = \ P$$

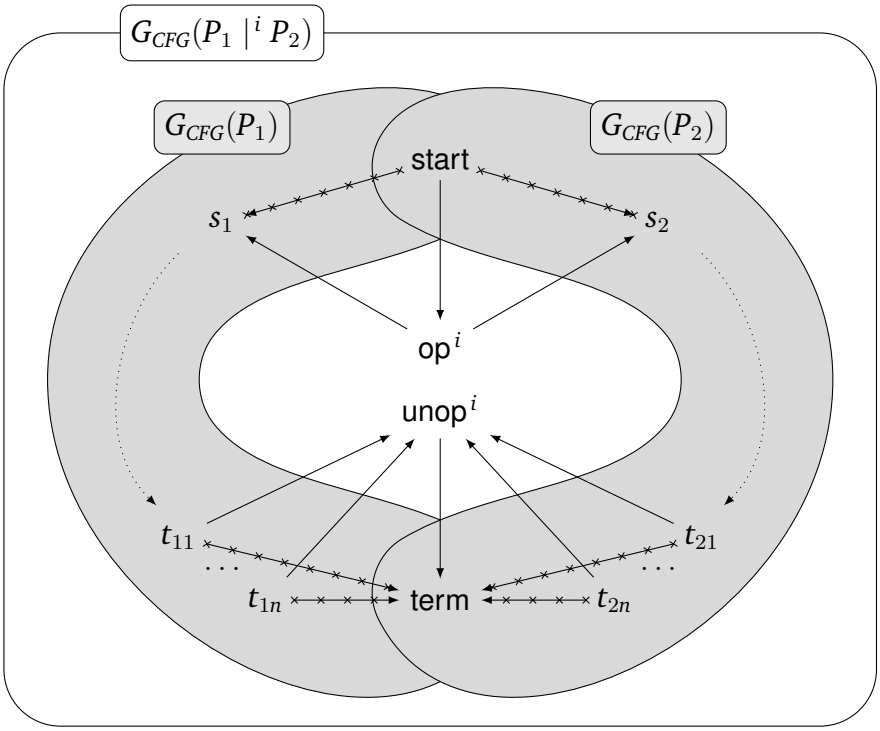


Figure 4.9: Control flow graph for parallel composition $P_1 \mid^i P_2$ with operator nodes $\mathsf{op} \in \{\mathsf{par}_A, \mathsf{par}_{A\cap B}, \mathsf{interleave}, \mathsf{intch}, \mathsf{extch}\}$ representing the involved CSP operator $\mid \in \left\{ \|_A, \|_B, |||, \sqcap, \Box \right\}$

which defines the process associated with the process identifier. The control flow graph for such a process definition is obtained from the control flow graph of the associated process by simply replacing the generic start and term nodes with nodes that are specific for the given process identifier. Additionally, edges are introduced that lead from recursive process calls to the newly created process start node. Edges from the newly created process term node to the process call return ret node can be omitted since it is obvious that they won't be reachable.

Given the control flow graph of process P, i.e., graph $G_{CFG}(P)$, the control flow graph associated with the process definition $X = P$ is depicted in Figure 4.10 and is defined as follows:

$$G_{CFG}(X = P) := \left(\begin{array}{l} \big(V_P \cup \{\mathsf{start}.X, \mathsf{term}.X\}\big) \setminus \{\mathsf{start}, \mathsf{term}\}, \\ \left(\begin{array}{l} E_P \cup \big\{(\mathsf{start}.X, s) \mid \exists (\mathsf{start}, s) \in E_P\big\} \\ \quad \cup \big\{(t, \mathsf{term}.X) \mid \exists (t, \mathsf{term}) \in E_P\big\} \\ \quad \cup \big\{(\mathsf{call}.X^i, \mathsf{start}.X) \mid \exists\, \mathsf{call}.X^i \in N_P\big\} \end{array} \right) \\ \setminus \{(\mathsf{start}, s) \in E_P\} \cup \{(t, \mathsf{term}) \in E_P\} \end{array} \right)$$

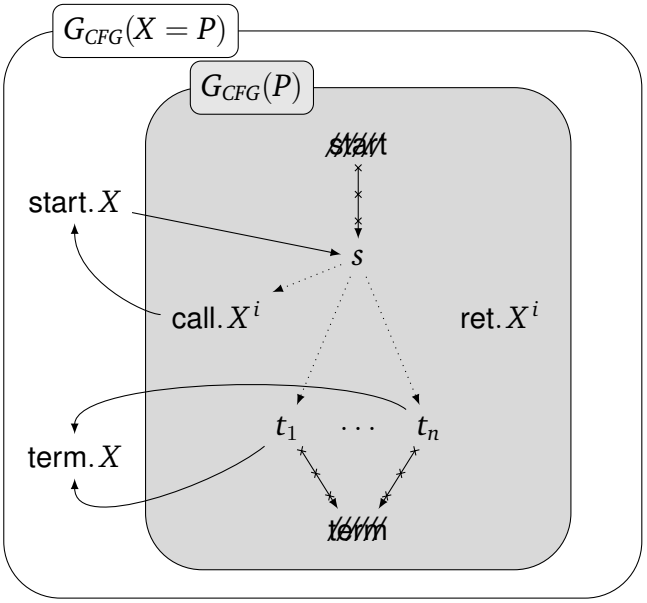


Figure 4.10: Control flow graph for a process definition $X = P$

Process Call

A process call via a process identifier of the form X^i is modelled in the control flow graph by introducing two new nodes, one (call. X^i) indicating the process call and the other (ret. X^i) indicating the return from the called process. Initially the control flow graph only contains edges from the start node to the call node and from the ret node to the term node, i.e., the termination of the represented process depends on the process definition: if this leads to a non-terminating loop, then the ret node and the term node of the process call won't be connected to the resulting control flow graph and since they won't be reachable, they will be removed from the final control flow graph together with the associated edge.

The control flow graph associated with a process call X^i is depicted in Figure 4.11 and is defined as follows:

$$G_{CFG}(X^i) := \begin{pmatrix} \{\mathsf{start}, \mathsf{call}.X^i, \mathsf{ret}.X^i, \mathsf{term}\}, \\ \{(\mathsf{start}, \mathsf{call}.X^i), (\mathsf{ret}.X^i, \mathsf{term})\} \end{pmatrix}$$

Composition of Separate Sections of the CSP Part

The CSP part of CSP-OZ specifications can be regarded as being composed out of several sections, each containing a number of lines with process definitions:

P_1 , P_2

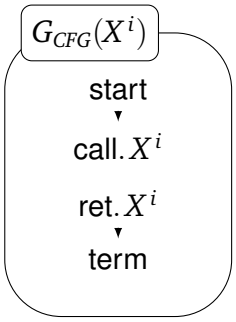


Figure 4.11: Control flow graph for process calls X^i

Here, the comma represents a line break separating such sections of the specification. The control flow graph of the composed specification is obtained from the control flow graphs of the individual sections by joining the set of nodes and edges and adding edges between process calls and process definitions resp. process termination and process call return points.

Given the individual control flow graphs $G_{CFG}(P_1)$ and $G_{CFG}(P_2)$, the control flow graph associated with their composition is depicted in Figure 4.12 and is defined as follows:

$$G_{CFG}(P_1 \ , \ P_2) := \left(\begin{array}{l} N_1 \cup N_2, \\ E_1 \cup E_2 \\ \quad \cup \left\{ (\mathsf{call}.X^i, \mathsf{start}.X) \;\middle|\; \begin{array}{l} \mathsf{call}.X^i \in N_j \\ \land\ \mathsf{start}.X \in N_k \\ \land\ j \neq k \end{array} \right\} \\ \quad \cup \left\{ (\mathsf{term}.X, \mathsf{ret}.X^i) \;\middle|\; \begin{array}{l} \mathsf{term}.X \in N_j \\ \land\ \mathsf{ret}.X^i \in N_k \\ \land\ j \neq k \end{array} \right\} \end{array} \right)$$

Pure Object-Z methods

For pure Object-Z methods, i.e., methods that do not occur inside the CSP process definitions, the control flow graph needs to be extended in the following way. Since such events can take place at any time without being restricted by the CSP part, they can be regarded as being interleaved with the CSP `main` process. Therefore, a fresh interleave node is inserted into the CFG between the start.`main` node and its immediate successor. This newly introduced interleave node is then connected with each `enable` node belonging to a pure Object-Z method. Finally, each of these `enable` nodes is connected to its associated `effect` node, analogously to the construction that we have defined in the previous section for the control flow graph for Object-Z classes.

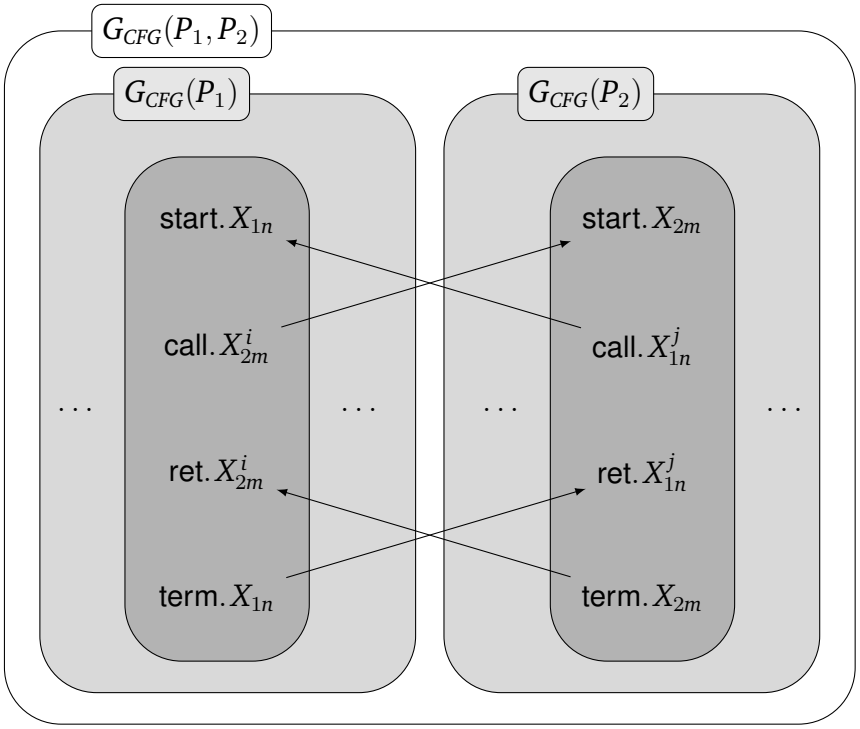


Figure 4.12: Control flow graph for the composition of separate sections of the CSP part P_1 , P_2

4.2.2 Dependence Graph

As for Object-Z specifications, the (program) dependence graph (PDG) for CSP-OZ specifications contains all nodes that have already been present within the control flow graph (CFG), supplemented by predicate nodes representing predicates from the specification's Object-Z schemas that have not been considered for the construction of the CFG.

Dependences between elements of CSP-OZ specifications are again defined in terms of suitable edges of the resulting dependence graph. The types of dependence edges include those of the dependence graph for Object-Z specifications as defined in the previous section, i.e., *predicate dependence*, *control* dependence, and *data* dependence. Some of these dependence definitions remain unchanged, while others need to be extended for CSP-OZ specifications.

Therefore, in the following we do not repeat those dependence definitions that remain unchanged in comparison to those given already for Object-Z specifications, but only refer to the previous section. However, we define several extensions of these definitions, resulting in additional subtypes of the previously defined dependences.

Moreover, one completely new type of dependence is introduced, i.e., *synchronisation dependence*, which represents mutual communication relations between

processes according to CSP's parallel composition operator with synchronisation.

Predicate Dependence

Predicate dependence for CSP-OZ specifications is defined exactly as for Object-Z specifications. No modifications or extensions are needed.

Control Dependence

In comparison to the definition of control dependence for Object-Z specifications, we need a slightly stronger definition of *control dependence due to nontrivial precondition*:

- *Control dependence due to nontrivial precondition* exists between an `enable` node and its `effect` node iff its `enable` schema is non-empty (i.e., not equivalent to true): for two nodes n and n' a control dependence edge due to nontrivial precondition $n \xrightarrow{cd} n'$ exists iff

$$\begin{aligned}\exists e \in E: \quad & n \equiv \texttt{enable\_}e \wedge n' \equiv \texttt{effect\_}e \\ & \wedge \langle n, n' \rangle \in path_{CFG}(n, n') \\ & \wedge \exists p_e : n \xleftrightarrow{pd} p_e.\end{aligned}$$

Therefore, only nodes representing events with non-trivial guards are sources of this kind of control dependence edges.

Note that the additional condition $\langle n, n' \rangle \in path_{CFG}(n, n')$ is trivially satisfied for any Object-Z specification, since for each method of an Object-Z there is only exactly one `enable` and one `effect` node present within the associated CFG, with the `effect` node being successor to the `enable` node. Thus, this requirement is only added due to the CFG of CSP-OZ specifications, which might contain several occurrences of the `enable` and `effect` nodes for each method, namely for representing each of the method's occurrence within the CSP part.

Furthermore, we define additional types of control dependence due to *external* and *internal* choice:

- *Control dependence due to external (resp. internal) choice* or *parallel composition with synchronisation* exists between an *extch* (resp. *intch*) or par_S node and its immediate CFG successors.

Moreover, the following new subtype of control dependence needs to be defined:

- *Control dependence due to synchronisation* exists between an `enable` node and its associated `effect` node iff both nodes are located inside a branch attached to a parallel composition node and their associated event belongs to the synchronisation alphabet of this parallel composition node.

Note that even an event with an empty `enable` schema can be source of a control dependence edge, since synchronisation determines whether control flow continues.

The concluding definitions of control dependence edges that have already been introduced for Object-Z specifications in order to achieve a well-formed graph are extended as follows in order to reflect the control flow structure resulting from the process definitions and process calls within the CSP part:

- *Call* edges exist between a *call* node and its associated *start* node.
- *Termination* edges exist between a *term* node and its associated *ret* node.
- *Start* edges exist between a *start* node and its immediate CFG successor.
- *Return* edges exist between a *ret* node and its immediate CFG successor.

Finally, *indirect control dependence* edges are defined in the very same way as for Object-Z specifications. In particular, these are again defined as extensions of all previously introduced *direct* control dependence edges. The only difference at this point is that direct control dependence edges of CSP-OZ specifications now also comprise all additional subtypes of control dependence that have been newly introduced within this section.

Data Dependence

The definitions of *direct data dependence* and *symmetric data dependence* as they were given for Object-Z specifications remain unchanged for CSP-OZ specifications.

However, two additional types of data dependence need to be defined that reflect two different kinds of parallel composition according to the CSP interleaving operator and the CSP operator of parallel composition with synchronisation:

- *Interference data dependence* exists between two nodes p_x and q_y iff the nodes of both associated schemas x and y are located in different CFG branches attached to the same interleaving or parallel composition operator, i.e., iff

$$\begin{aligned} & mod(p_x) \cap ref(q_y) \neq \varnothing \\ & \wedge\, \exists\, m : (m \equiv \mathsf{interleave} \vee m \equiv \mathsf{par}_S) \\ & \qquad \wedge\, \exists\, \pi_x \in path_{CFG}(m,x) \wedge \exists\, \pi_y \in path_{CFG}(m,y) : \\ & \qquad\quad \mathrm{ran}\ \pi_x \cap \mathrm{ran}\ \pi_y = \{m\} \end{aligned}$$

- *Synchronisation data dependence* exists between two predicate nodes p_x and q_y iff both are located inside `effect` schemas whose respective `enable` schemas are connected by a synchronisation dependence edge as defined below and one predicate has an output that the other predicate expects as input, i.e., iff

$$x = \texttt{effect\_}e \wedge y = \texttt{effect\_}e \wedge out(p_x) \cap in(q_y) \neq \varnothing$$

Synchronisation Dependence

The idea of *synchronisation dependence* edges

$$\overset{sd}{\longleftrightarrow} \subseteq N_{CFG} \times N_{CFG}$$

is to represent the influence that two `enable` schema nodes of the same event have on each other by being located inside two different branches of a parallel composition operator that has the schemas' associated event in its synchronisation alphabet. *Synchronisation dependence* exists between two nodes n and n' with $n \equiv n' \equiv \texttt{enable\_}e$ iff $\exists\, m \equiv \mathsf{par}_S$ with $e \in S$:

$$\exists\, \pi \in path_{CFG}(m,n) \land \exists\, \pi' \in path_{CFG}(m,n') :\ \mathrm{ran}\ \pi \cap \mathrm{ran}\ \pi' = \{m\}$$

This already concludes the definition of the program dependence graph for CSP-OZ specifications. Next, we will see the application to our example specification with several examples of the dependences that we have defined in this section.

4.2.3 Example: Untimed Air Conditioner Dependence Graph

The program dependence graph of the class *AirConditioner* that was introduced in the previous chapter can be found in Figure 4.13.

Note that we introduced several abbreviations in order to achieve better readability of the dependence graph:

- Predicate nodes are hidden within schema nodes, such that data dependence edges starting at or leading to one of those nodes actually represent an edge starting at or leading to one of these hidden predicate nodes.
- There are no separate schema nodes for `enable` and for `effect` schemas, but both are represented by a single node labelled with the associated method name. Therefore, a control dependence edge starting from such a node actually represents two control dependence edges: one starting at the associated `enable` node and leading to its `effect` node and another one starting at the associated `enable` node leading to its depicted target.
- Control dependence edges leading to the shaded boxes around groups of nodes actually represent a number of control dependence edges that all start at the same depicted source node and lead to each of the nodes contained within the box.

Except for synchronisation dependence and synchronisation data dependence, each previously defined type of dependence can be seen in this example dependence graph:

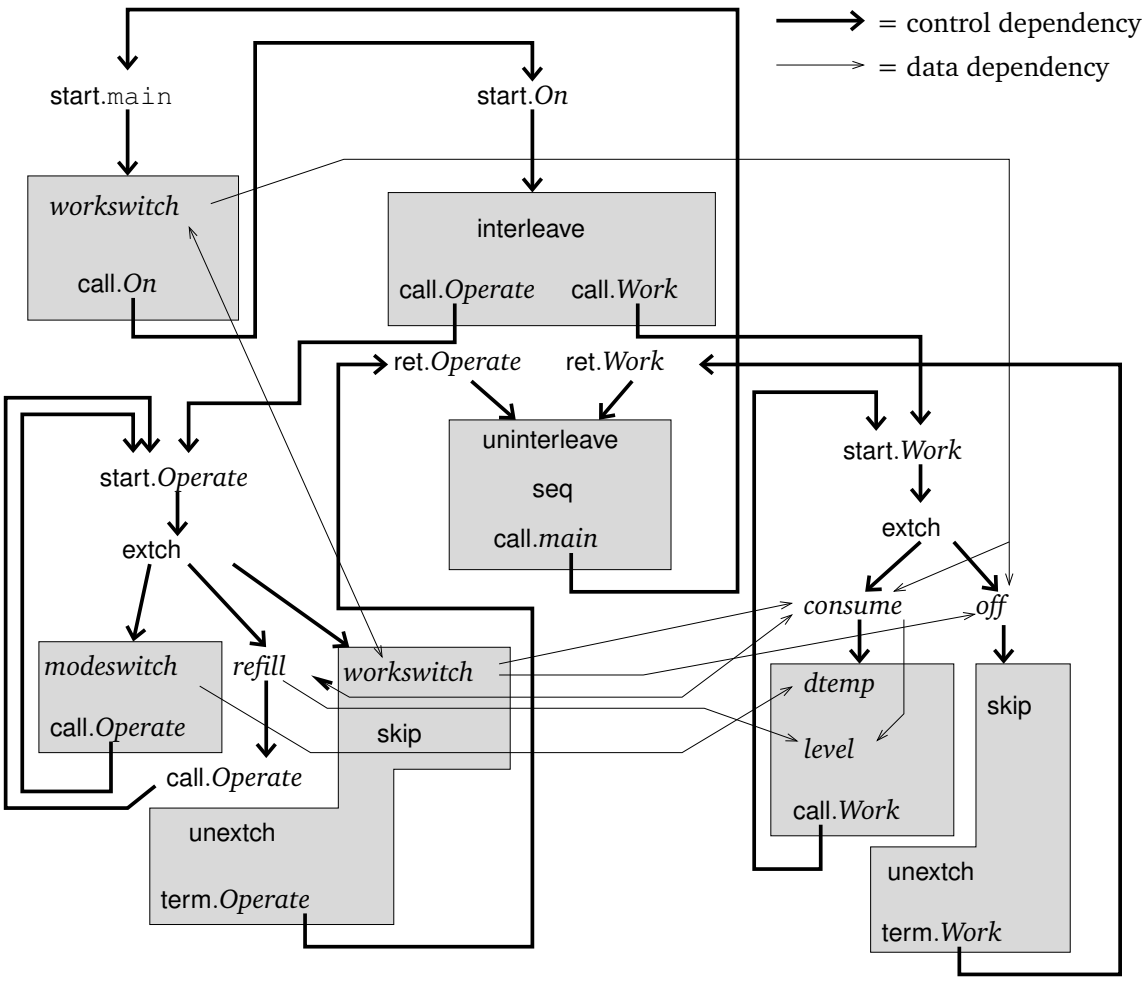


Figure 4.13: Dependence graph for the *AirConditioner* class of the untimed air conditioner specification

- *Control dependence due to external choice* appears within the *Operate* and the *Work* branch of the dependence graph, where extch and unextch nodes represent the CSP external choice operators.
- *Indirect control dependence* appears at each of the boxes around groups of nodes. Control dependence leading to the first node within the box is extended to all further nodes within the box, since they are not yet target of any other control dependence.
- *Interference* data dependence appears between nodes located in the *Operate* and the *Work* branch of the CSP interleaving operator, since each of the variables *fuel*, *work*, and *mode* is modified by a node in one branch and referenced by a corresponding node in the other branch.

Examples of synchronisation dependence and synchronisation data dependence will be appear in the next section, where they arise on the level of synchronisation between classes This, however, follows the same concept as synchronisation between processes within a single class.

4.3 CSP-OZ-DC Specifications

In this section on dependence analysis for CSP-OZ-DC specifications we essentially treat two extensions of the previous approaches: first, we define how to deal with specifications that comprise *several classes*; second, we extend the dependence analysis to cover the *DC part* of CSP-OZ-DC classes.

As previously defined for CSP-OZ specifications, the first step of the dependence analysis for CSP-OZ-DC specifications consists of the construction of the specification's CFG. The only new aspect that we add in this concluding section to the CFG construction is to cover parallel composition of *several classes*.

Note that this aspect is not specific to CSP-OZ-DC specifications, but applies in the same way to Object-Z and CSP-OZ specifications. However, since CSP-OZ-DC is the final stage of extension of our specification notation and represents thus a generalisation of the previous specifications, we deem this concluding section on dependence analysis as best suited for covering the general concern of parallel composition of classes.

The subsequent dependence analysis is then again based on the control flow graph and proceeds essentially according to the previous dependence definitions for CSP-OZ specifications. However, several further extensions of dependence definitions will be defined that mainly reflect the only aspect of CSP-OZ-DC specifications, which has not been present in Object-Z and CSP-OZ specifications, namely the *real-time requirements* formulated within the DC part.

4.3.1 Control Flow Graph

The construction of the CFG for CSP-OZ-DC specifications follows exactly the definition given in the previous section on CSP-OZ specifications. However, since we have previously not yet considered the parallel composition of several classes, we add this aspect here to the construction of the CFG. As stated before, the same approach applies to parallel composition of Object-Z and CSP-OZ classes.

Parallel Composition of Several Classes

When computing the dependence graph for the parallel composition of several classes, we start by constructing the CFGs for each individual class as usual. These individual CFGs are then combined into one single global CFG for the entire parallel composition according to the following steps:

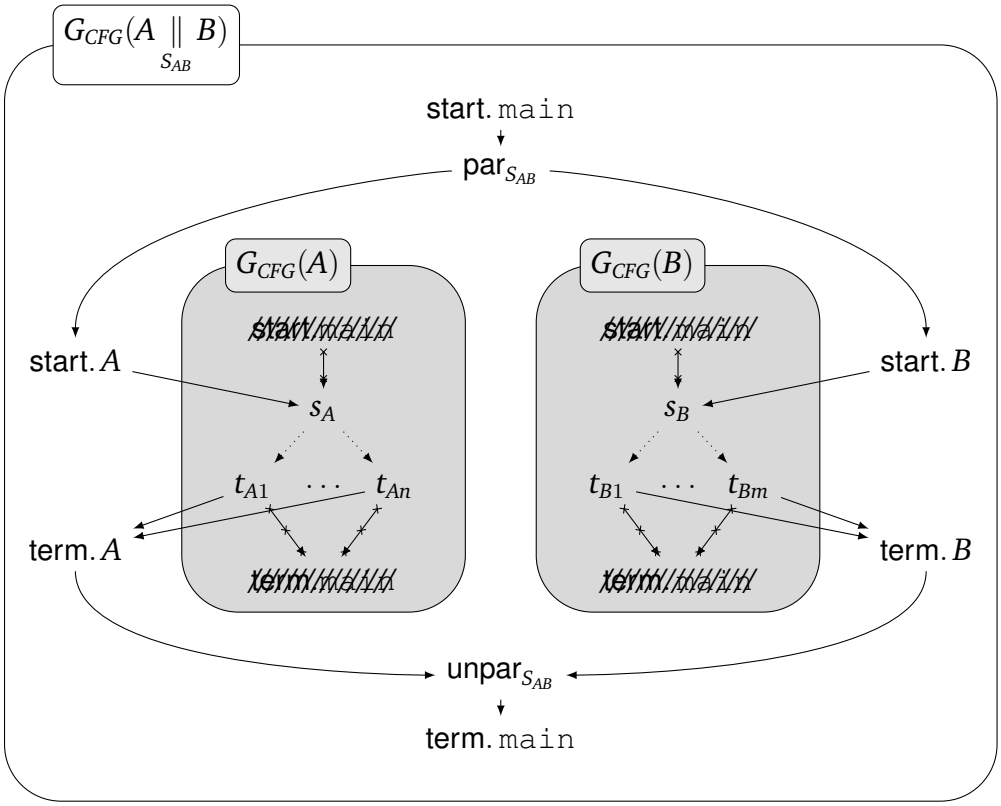


Figure 4.14: Control flow graph for the parallel composition of classes A and B with synchronisation on the set of common events S_{AB}, represented by CSP process $A \underset{S_{AB}}{\parallel} B$

1. The generic CFG entry and exit nodes *start*.`main` and *term*.`main` of each class C are renamed into class-specific nodes *start*.C and *term*.C, such that these nodes are unique in the final CFG.

2. For each pair of classes (C_1, C_2) that should run in parallel composition, parallel synchronisation nodes par_S and $unpar_S$ are created and linked via control flow edges to the respective *start* and *term* nodes of each CFG. The synchronisation alphabet S contains all events over which both classes need to synchronise.

3. Finally, a new pair of top level entry and exit nodes, *start*.`main` and *term*.`main`, is created. These nodes are connected via control flow edges to each of the newly created parallel synchronisation nodes.

For the parallel composition of two classes A and B, the parallel composition of their individual CFGs is illustrated in Figure 4.14. Instead of constructing one dependence graph for each individual class as it was defined in the previous sections and as it will also be explained in the following section, the construction

of the dependence graph for the parallel composition of all involved classes is then based on this previously constructed global CFG. Apart from this, the construction for parallel composition of classes proceeds as usual.

When computing the CFG for the air conditioner specification comprising the *AirConditioner* class and its *Environment* class as introduced in Section 3.3 of the previous chapter, we obtain the result depicted in Figure 4.15.

4.3.2 Dependence Graph

As for Object-Z and CSP-OZ specifications, the (program) dependence graph for CSP-OZ-DC specifications contains all nodes that have already been present within the control flow graph, supplemented by predicate nodes representing predicates from the specification's Object-Z schemas that have not been considered for the construction of the CFG.

The various types of dependence edges in the dependence graph for CSP-OZ-DC specifications are again based on the previous definitions for Object-Z and CSP-OZ specifications. The main extension that we introduce in this concluding section address the real-time requirements specified in the DC part of CSP-OZ-DC specifications, represented by a new type of dependence, i.e., *timing dependence* that also requires some modifications and extensions of the other previously defined types of dependence.

Predicate Dependence

Predicate dependence for CSP-OZ-DC specifications is defined exactly as for Object-Z and CSP-OZ specifications with the following extension that addresses the timing requirements coming from the DC part.

This extension applies to predicate nodes $n \equiv p_x$ of `effect` schemas that imply modifications of variables, which are mentioned in a counterexample formula *CE* within the DC part C_{DC} of the given class. Their so far unidirectional connection via the predicate dependence edge coming from their associated effect schema node $n' \equiv \texttt{effect\_}x$ needs to be complemented by another predicate dependence edge in the opposite direction. Therefore, in extension of the previous definition of predicate dependence, for two nodes n and n' a predicate dependence edge $n \xrightarrow{pred} n'$ exists between them iff

$$
\begin{aligned}
&(n = x \wedge n' = p_x) && \text{[\texttt{enable} and \texttt{effect} schema predicates]}\\
\vee\ &(n = p_x \wedge n' = x \\
&\qquad \wedge \exists e \in E : x \equiv \texttt{enable\_}e) \\
& && \text{[\texttt{enable} schema predicates only]}\\
\vee\ &(n = p_x \wedge n' = x \wedge \exists e \in E : x \equiv \texttt{effect\_}e \\
&\qquad \wedge \exists CE \in C_{DC} : mod(p_x) \cap vars(CE) \neq \varnothing). \\
& && \text{[\texttt{effect} schema predicates influenced by } CE\text{]}
\end{aligned}
$$

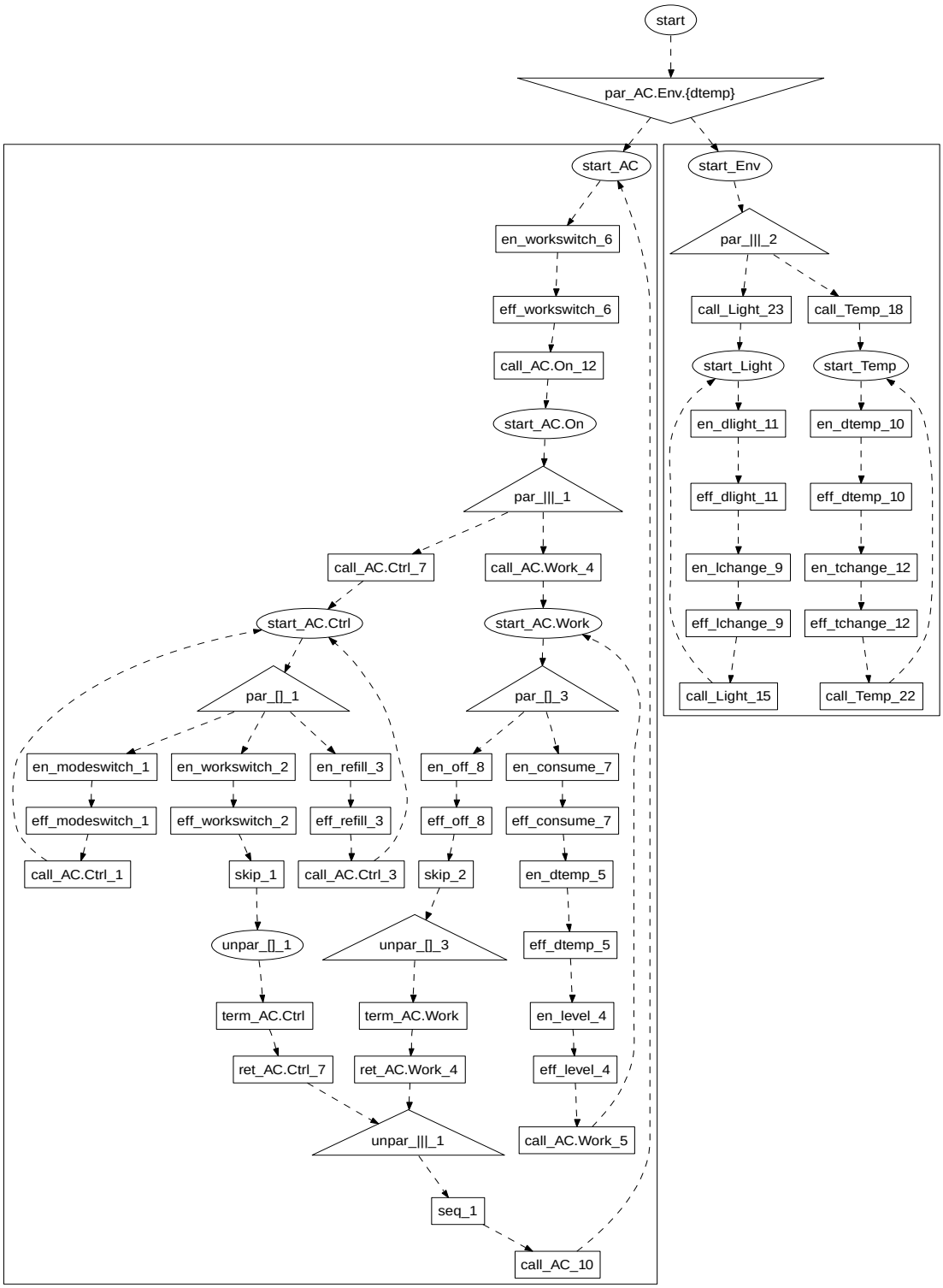


Figure 4.15: Control flow graph for the air conditioner specification comprising the *AirConditioner* class (AC) and its *Environment* class (Env).

This extension treats such predicate nodes in a similar way as predicate nodes of `enable` schemas, since they play—in conjunction with the DC formula—a similar role: They can be regarded as a guard for the associated event, since this event can only take place if the predicates in its `effect` schema comply with the restrictions given in the DC counterexample formula *CE*. Therefore, the DC part can inhibit the execution of events although their `enable` schema is satisfied. This extended form of precondition is reflected by introducing the additional predicate dependence edge for any involved `effect` schema predicate.

Control Dependence

The real-time requirements from the DC part also lead to an additional subtype of control dependence:

- *Control dependence due to timing* exists between an `enable` node and its associated `effect` node iff there exists a DC counterexample formula *CE* in the DC part C_{DC} of the given class that mentions the given event or variables that are modified by it, i.e., for two nodes n and n' a control dependence edge due to timing $n \xrightarrow{cd} n'$ exists iff

$$\begin{aligned} \exists e \in E: \quad & n \equiv \texttt{enable\_}e \wedge n' \equiv \texttt{effect\_}e \\ & \wedge \exists CE \in C_{DC}: e \in \mathit{events}(CE) \vee \mathit{vars}(CE) \cap \mathit{mod}(e) \neq \varnothing. \end{aligned}$$

Again, even events with an empty `enable` schema can be source of a control dependence edge, since the DC part may restrict, whether control flow continues with a given event, in spite of the `enable` schema of the event being satisfied.

Apart from this extension, control dependence for CSP-OZ-DC specifications is defined exactly as for CSP-OZ specifications.

Data and Synchronisation Dependence

The definitions of data dependence and synchronisation dependence for CSP-OZ specifications are directly applicable also to CSP-OZ-DC specifications, since the DC part does not require any modifications or extensions of these types of dependence.

Timing Dependence

Intuitively, a timing dependence exists between two nodes in the PDG if their associated events or variables are constrained by one of the formulae of the DC part. Since DC formulae of CSP-OZ-DC specifications define real-time requirements in terms of counterexamples of arbitrary length, the number of events and variables referenced by a single DC formula can be arbitrarily large. Therefore, the same holds for the number of associated PDG nodes.

CE	$::=$	$\neg(Ph \frown (Ph \mid Ev) \frown \ldots \frown (Ph \mid Ev) \frown true)$	counterexample traces, consisting of phase expressions Ph and event expressions Ev
Ph	$::=$	$(true \mid \lceil p \rceil)$	phase invariant, optionally with...
		$[\wedge\ \ell \sim t]$	...time bound ($\wedge\ \ell \sim t$) and ...
		$(\wedge \boxminus ev)^*$	...definition of forbidden events ($\boxminus ev$)
$\sim$	$::=$	$\leq \mid < \mid > \mid \geq$	time bound operator
Ev	$::=$	$\updownarrow ev$	required event: a zero-time (point) interval where event ev takes place
	$\mid$	$\not\updownarrow ev$	forbidden event: event ev must not take place at the given point interval
	$\mid$	$Ev \vee Ev$	disjunction of required/forbidden events
	$\mid$	$Ev \wedge Ev$	conjunction of required/forbidden events

Figure 4.16: Syntax of counterexample formulae CE with $ev \in Events$ being an event and p being a predicate over a set of state variables V.

The number of PDG edges required for representing all of the mutual influences between these nodes even grows with the factorial of the number of involved nodes. In order to limit the number of necessary edges, we will in the following define an ordering of the involved PDG nodes according to the syntactical ordering of the associated elements of the given DC formula.

Thus, the basic idea of *timing dependence edges*

$$\overset{td}{\longleftrightarrow} \subseteq N_{PDG} \times N_{PDG}$$

is to represent the mutual influence between PDG nodes that are associated to neighbouring elements of a DC counterexample formula.

The representation of dependences arising from the DC part needs some additional preparation. In Figure 4.16 we first remind the syntax of DC counterexample formulae used within the DC part that has already been introduced in the previous Chapter 3.

Next, we introduce a data structure that will be useful for easily referencing variables and events that are constrained by counterexample formulae of the DC part. As defined by its syntax, any counterexample formula CE consists of a finite series of phase expressions Ph and event expressions Ev with the initial element of the series being a phase expression. According to [Hoe06], we can represent each phase expression Ph together with all its preceding event expressions Ev by using a

data structure *PhaseSpec* of the following form, where V is the set of state variables of the underlying formula:

$$\mathit{TimeOp} ::= \mathit{none} \mid \mathit{less} \mid \mathit{lessequal} \mid \mathit{greater} \mid \mathit{greaterequal}$$

PhaseSpec

$$\begin{array}{l} \mathit{inv} : \mathcal{L}(V) \\ \mathit{allowEmpty} : \mathbb{B} \\ \mathit{timeop} : \mathit{TimeOp} \\ \mathit{bound} : \mathsf{Time} \\ \mathit{forbidden} : \mathbb{F}\mathit{Events} \\ \mathit{entryEvents} : \mathcal{L}(\mathit{Events}) \\ \hline \mathit{allowEmpty} \Rightarrow (\mathit{inv} = \mathit{true} \wedge \mathit{timeop} \notin \{\mathit{greater}, \mathit{greaterequal}\}) \\ \mathit{timeop} = \mathit{none} \Leftrightarrow \mathit{bound} = 0 \end{array}$$

In order to represent all information of a given phase *Ph* together with all its preceding events *Ev*, the following conditions must be satisfied by *PhaseSpec*:

- a phase invariant of the form $\lceil p \rceil$ (where p is a predicate, referencing only variables in V) is represented by $\mathit{inv} \equiv p$,
- a phase *true* is represented by $\mathit{inv} \equiv \mathit{true}$, together with $\mathit{allowEmpty} \equiv \mathit{true}$, in order to distinguish *true* phases from $\lceil \mathit{true} \rceil$ phases, since the latter require non-empty time-intervals,
- time bounds of the form $\ell \sim t$ are represented by a value of *timeop* according to the time bound operator $\sim$ and a value of *bound* according to the time bound t,
- the absence of time bounds is represented by $\mathit{timeop} \equiv \mathit{none}$ and $\mathit{bound} \equiv 0$,
- forbidden events of the form $\boxminus \mathit{ev}$ are represented by the set *forbidden* containing the mentioned event, i.e., $\mathit{ev} \in \mathit{forbidden}$, and
- preceding entry event formulae containing expressions of the form $\updownarrow \mathit{ev}$ and $\not\updownarrow \mathit{ev}$ are captured as a whole within *entryEvents*, which either is the conjunction of all such formulae, or which has the value *true* if there are no such formulae.

With $\mathit{events}(\mathit{PhaseSpec})$ we refer to the set of events being mentioned within *PhaseSpec.forbidden* and *PhaseSpec.entryEvents*. Similarly, we let $\mathit{vars}(\mathit{PhaseSpec})$ denote the set of variables being mentioned within *PhaseSpec.inv*.

A complete counterexample formula *CE* can then be represented as a finite sequence of suitable *PhaseSpec* data structures:

$$CE \mathrel{\widehat{=}} PhaseSpec_0^{CE};\ PhaseSpec_1^{CE};\ PhaseSpec_2^{CE};\ \ldots$$

Based on this representation of DC counterexample formulae *CE*, we now define an associated *timing node sequence*

$$TNS_{CE} : \operatorname{seq} n$$

consisting of dependence graph nodes n with relevance to the given formula which are defined as follows:

$$\begin{aligned} n \in \operatorname{ran} TNS_{CE} \ \Leftrightarrow\ & \\ & n \equiv \mathsf{start}.\texttt{main} \wedge PhaseSpec_0^{CE}.timeop \neq none \\ \vee\ & \exists PhaseSpec_i^{CE} : mod(n) \cap vars(PhaseSpec_i^{CE}) \neq \varnothing \\ & \qquad \vee\ n \equiv \texttt{enable\_}ev \wedge ev \in events(PhaseSpec_i^{CE}) \end{aligned}$$

Therefore, the relevant nodes that are gathered within a timing node sequence include

- the $\mathsf{start}.\texttt{main}$ node of the given class, if the initial phase of *CE* has a time bound different from 0, i.e.,

  $$PhaseSpec_0^{CE}.timeop \neq none,$$

- predicate nodes n implying modifications of variables being mentioned in some phase of *CE*, i.e.,

  $$\exists PhaseSpec_i^{CE} : mod(n) \cap vars(PhaseSpec_i^{CE}) \neq \varnothing,$$

  and

- nodes $n \equiv \texttt{enable\_}ev$ representing the `enable` schema of some event *ev* that is mentioned in some phase of *CE*, i.e.,

  $$\exists PhaseSpec_i^{CE} : ev \in events(PhaseSpec_i^{CE}).$$

Note that the $\mathsf{start}.\texttt{main}$ node is never mentioned explicitly inside a DC formula, but is rather indirectly used as a reference point for the length of the first phase: If *CE* restricts the length of the initial phase in some way, i.e., if $PhaseSpec_0^{CE}.timeop \neq none$, the $\mathsf{start}.\texttt{main}$ node will be relevant, since *CE* can then be regarded as being anchored to the point of initialisation, which is represented by the $\mathsf{start}.\texttt{main}$ node.

The nodes inside each TNS_{CE} are ordered according to the syntactical occurrence of their associated specification elements within *CE*. The only exceptions are the following:

- the start.`main` node: this node lacks an associated specification element and will thus—presumed it is included at all—appear as the first element of the timing node sequence, since it is associated with the point of initialisation.
- nodes that modify the same variable referenced by *CE* are ordered alphabetically according to their names.
- nodes that correspond to different occurrences of an event in the CSP part are ordered according to their syntactical occurrence in the CSP part.

Based on these timing node sequences, a bidirectional *timing dependence edge* $n \overset{td}{\longleftrightarrow} n'$ is defined to exist between any two dependence graph nodes n and n' iff there is a counterexample formula *CE* with an associated timing node sequence TNS_{CE} that contains two neighbouring timing nodes n and n', i.e., iff

$$\exists\, TNS_{CE} : \{n, n'\} \subseteq \operatorname{ran} TNS_{CE} \land \operatorname{dom} n + 1 = \operatorname{dom} n'.$$

Therefore, all dependence graph nodes related to elements that appear in a given DC counterexample formula are connected via a chain of bidirectional timing dependence edges in the order of their syntactical occurrence within the DC formula.

4.3.3 Example: Timed Air Conditioner Dependence Graph

An example of each type of the previously defined types of dependence can be found in the dependence graph of our example specification of the timed air conditioner system, depicted in Figure 4.17:

- *Control dependence due to synchronisation* appears at the edge between nodes en_dtemp_10 and eff_dtemp_10. Without event *dtemp* being in the synchronisation alphabet of class *AirConditioner* and class *Environment*, the `enable_`*dtemp* schema would not be source of a control dependence edge, since it does not contain any predicate, such that the guard of event *dtemp* is equivalent to *true* and thus always trivially satisfied.
- *Indirect control dependence* appears at the edge between nodes start_Light and en_lchange_9. Supposed, the predecessor of event *lchange* in the CSP part of class *Environment*, i.e., event *dlight*, would have had a non-empty `enable` schema, then the associated node would have been source of a control dependence leading to node `enable_`*lchange*. Since this is not the case, such an edge does not exist, but instead both nodes `enable_`*dlight* and `enable_`*lchange* are directly attached to the start node of process *Light*, the latter one only due to the definition of indirect control dependence.

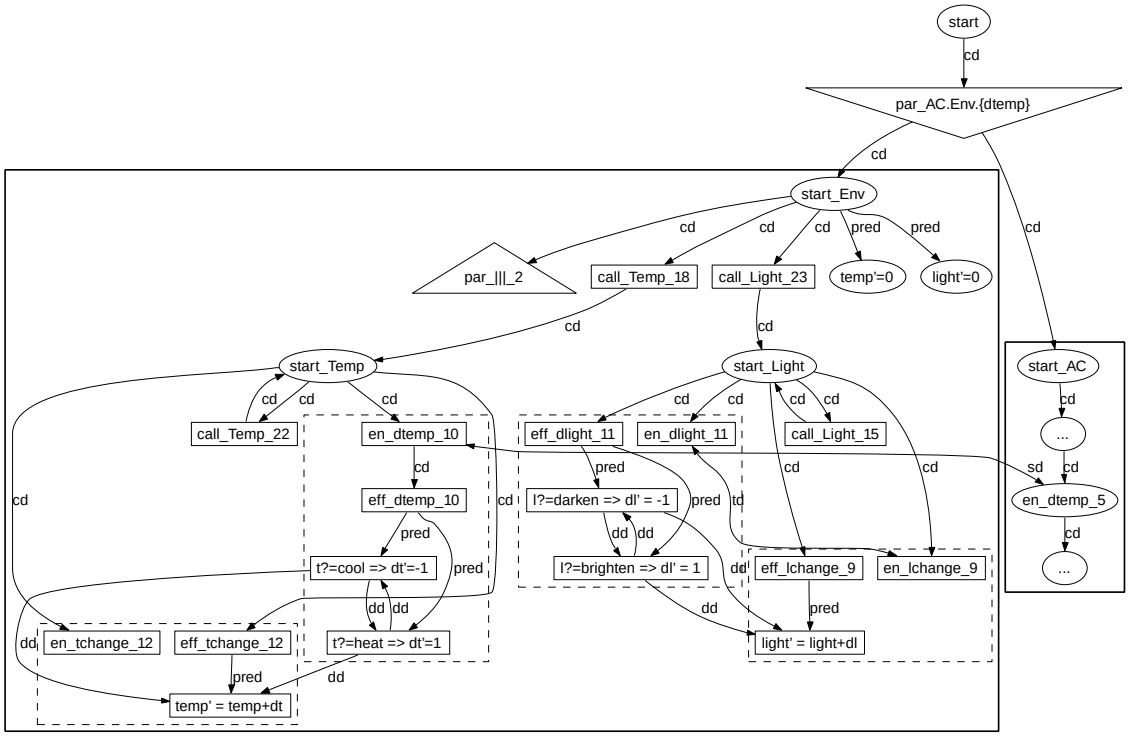


Figure 4.17: Program dependence graph for the air conditioner system. Nodes inside bold bounding rectangles belong to the same class, nodes inside dashed bounding rectangles to the same event. Note that most of the *AirConditioner* part is hidden, indicated by ". . ." nodes.

- *Direct data dependence* appears at the edge between nodes t?=cool => dt'=1 and temp'=temp+dt, where the modification of variable *dt* at the source node may directly reach the reference of this variable at the target node.

- *Synchronisation data dependence* is not visible within the example, but appears in the full dependence graph between nodes associated with the `effect_dtemp` schemas within class *Environment* and class *AirConditioner*, since *AirConditioner* communicates a change in temperature on channel *dtemp* via variable *t*! (that is restricted within the *AirConditioner* schema `effect_dtemp`) to the receiving variable *t*? (that is referenced within the *Environment* schema `effect_dtemp`).

- *Synchronisation dependence* appears at the edge between node en_dtemp_5 and node en_dtemp_10, which both belong to the synchronisation alphabet of *AirConditioner* and *Environment*. If one of both events is relevant, this also applies to the other one, since both need to agree in order to occur.

- *Timing dependence* appears at the edge between nodes en_dlight_11 and en_lchange_9. This timing dependence is derived from the DC formula

  $$CE \equiv \neg(true \frown \updownarrow dlight \frown \boxminus lchange \wedge \ell > 1 \frown true)$$

  which appears in the DC part of the environment specification, and which relates both involved events *dlight* and *lchange* as follows.

  The resulting sequence of *PhaseSpec* data structures

  $$CE \mathrel{\widehat{=}} \langle PhaseSpec_0^{CE}, PhaseSpec_1^{CE}, PhaseSpec_2^{CE} \rangle$$

  consists of three *PhaseSpec* elements, representing each of the three phases that *CE* contains. The initial and concluding phases of *CE* refer to intervals of arbitrary length (*true*), represented by

  $$PhaseSpec_i^{CE}.inv = true,$$
  $$PhaseSpec_i^{CE}.allowEmpty = true, \text{ and}$$
  $$PhaseSpec_i^{CE}.timeop = none$$

  with $i \in \{0, 2\}$.

  In between, the formula contains the phase $\boxminus lchange \wedge \ell > 1$ that refers to a non-empty interval, preceded by the event expression $\updownarrow dlight$ that refers to a point interval. Both of these intervals are represented by the single data structure $PhaseSpec_1^{CE}$, which contains information about the required initial events, namely $\updownarrow dlight$, represented by

  $$PhaseSpec_1^{CE}.entryEvents = \updownarrow dlight,$$

  information about events that are forbidden throughout the particular interval, namely $\boxminus lchange$, represented by

  $$PhaseSpec_1^{CE}.forbidden = \{lchange\},$$

  and information about a required minimal bound over the length of the particular interval, namely $\ell > 1$, represented by

  $$PhaseSpec_1^{CE}.allowEmpty = false,$$
  $$PhaseSpec_1^{CE}.timeop = greater, \text{ and}$$
  $$PhaseSpec_1^{CE}.bound = 1.$$

  Therefore, the resulting timing node sequence contains only the nodes associated with the `enable` schemas of events *dlight* and *lchange*, which are ordered according to their references within *CE*:

  $$TNS_{CE} = \langle en\_dlight\_11, en\_lchange\_9 \rangle.$$

  Finally, these nodes are connected via a bidirectional timing edge.

5 Specification Slices

Contents

The construction of the program dependence graph for formal specifications as defined in the previous chapter has been completely independent of the actual verification property, given as a formula of a stuttering invariant logic. The formula comes only now into play when the slicing is carried out.

In ordinary program slicing, the slicing criterion is the value of a variable at a certain program statement. In order to construct the slice of a program with respect to this criterion, the dependence graph node representing the statement of interest is first determined. Afterwards, all dependence graph nodes are included in the slice, which are backwards reachable (via dependence edges) from the initially determined node. In this way, that part of the program is obtained, which might influence the slicing criterion. Conversely, this implies that all remaining parts of the program are irrelevant with respect to the slicing criterion. Thus, they may finally be removed from the program in order to obtain the desired slice.

When slicing formal specifications with respect to logical formulae it is less easy to determine the actual slicing criterion. We first have to find out what the "start nodes" for slicing are, i.e., which nodes represent the slicing criterion. This question will be addressed in the next section.

Afterwards, we define how to compute the backwards slice on the program dependence graph and how to obtain the actual result of slicing the program dependence graph, i.e., a set of nodes that is not reachable for the given slicing criterion and can thus be safely removed from the specification without changing the verification result.

Finally, we will define how to obtain slices of specifications, given the outcome of the backwards slice on the program dependence graph.

A concluding section discusses the classification of the presented slicing approach with respect to the possible criteria that have been introduced in Chapter 2.

5.1 Slicing Criterion

The slicing criterion that we use for slicing formal specifications in the context of verification will be formulae in the notation of some stuttering-invariant logic. Our goal is then to show that a given specification satisfies such formulae. In the case of Object-Z and CSP-OZ the employed logic might be some temporal logics such as the next-free projection of LTL, LTL_{-X}, or the untimed projection of Duration Calculus formulae, SE-IL, that we consider in this thesis, while in the case of CSP-OZ-DC specifications the logic should also be able to express real-time properties as it is the case for test formulae, the subclass of DC, which we will consider in this thesis.

However, regardless of which concrete logic we actually use to formulate the verification property, the basic building blocks of any formula will be references to events and variables defined in the specification. Thus, from a given formula φ it is straightforward to derive a set of relevant events E_φ and a set of relevant variables V_φ, namely those appearing directly in the formula. From these sets we can determine the set of dependence graph nodes N_φ, which directly manipulate these variables or influence the execution of these events:

$$N_\varphi = \{p_x \in N_{pred} \mid mod(p_x) \cap V_\varphi \neq \varnothing\}$$
$$\cup \{\texttt{enable\_}e \in N_{CFG} \mid e \in E_\varphi\}$$

Thus, this set of nodes represents all dependence graph nodes that have *direct relevance* for the verification property. Altogether we have now obtained three different characterisations of the slicing criterion on three different levels, with each characterisation being derived from the previous level:

1. On the level of logical formulae the *primary* slicing criterion is given as a formula φ in some stuttering-invariant logic.

2. On the abstract level of events and variables that are referred to within the specification, the *intermediate* slicing criterion is given as the pair (E_φ, V_φ) of sets of events and variables, which are derived from the primary slicing criterion on the level of logical formulae, i.e., directly from formula φ which mentions the events E_φ and variables V_φ.

3. On the level of the dependence graph, the *final* slicing criterion is given as the set N_φ of nodes with direct relevance for the secondary and thus also for the primary slicing criterion.

Next, we will use the slicing criterion on the dependence graph level to determine further nodes that might affect our primary slicing criterion, the verification property φ.

5.2 Dependence Graph Backwards Slice

Starting from the initial set of nodes that we obtained in the previous section as the final representation of the slicing criterion, we compute the backward slice by a reachability analysis of the dependence graph. The resulting set of backwards reachable nodes contains all nodes that lead via an arbitrary number and arbitrary combination of predicate, control, data, synchronisation or timing dependence edges to one of the nodes that already are in N_φ.

In addition to all nodes from N_φ, the backward slice contains therefore also all dependence graph nodes with indirect influence on the given property, i.e., it is the set of all *relevant nodes* for the specification slice:

$$N' = \{n' \in N_{PDG} \mid \exists\, n \in N_\varphi : n' \ (\xrightarrow{pred} \cup \xrightarrow{cd} \cup \xrightarrow{dd} \cup \xrightarrow{sd} \cup \xrightarrow{td})^*\ n\}$$

Thus, the backward slice contains the set of all nodes which have some kind of influence on the truth value of our primary slicing criterion φ according to the dependences comprising the dependence graph. Simultaneously, these relevant dependence graph nodes give us the events, predicates and variables which have to remain in the reduced specification, namely those associated with one or more of the relevant nodes as follows.

Relevant events are those associated with nodes from N' that represent relevant `enable` or `effect` schemas

$$E' = \{e \mid \exists\, n \in N' : n \equiv \texttt{enable\_}e \vee n \equiv \texttt{effect\_}e\}$$

and *relevant variables* are those associated with nodes from N' that represent relevant predicates:

$$V' = \bigcup_{p_x \in N'} vars(p_x).$$

5.3 Object-Z Specification Slices

Given the set of relevant variables V' and the set of relevant events E' from the previous backwards slice computation on the program dependence graph, it is then straightforward to construct a reduced version of a given specification class C with respect to the primary slicing criterion, verification property φ. The reduced class C' contains

- a state schema with variables from V' only (with the same type as in the original class C),
- an *Init* schema restricting only variables in V',
- `enable` and `effect` schemas only for events in E', and
- within these schemas only predicates that refer to (primed or unprimed) variables from V'.

We refer to the resulting reduced schemas that belong to this sliced class specification as

$$State', Init', \texttt{enable\_}e', \texttt{effect\_}e'$$

in order to properly distinguish them from their unprimed counterparts in the original specification C.

5.3.1 Example: Tic-Tac-Toe Specification

We will now use the previously introduced Tic-Tac-Toe specification as an example to demonstrate the effect that can be achieved by slicing Object-Z specifications with respect to two different temporal logic verification properties.

Example Slicing Criterion: Number of Moves and Free Positions

The first verification property and slicing criterion that we use for slicing the class *TicTacToe* is the state/event interval logic formula

$$\varphi_1 := \boxed{L}\lceil moves = 9 - \#free \rceil,$$

expressing an invariant of the Tic-Tac-Toe specification that requires a linear relation between the number of moves and the number of free fields on the board. We restrict the explanation of this formula at this point to its informal description, since this contains already everything that matters for the illustration of the slicing approach, namely, the events and variables that the formula mentions. A formal definition of state/event interval logic formulae will follow in Chapter 6 on slicing correctness.

The sets of relevant events and variables derived from the formula are thus

$$E_{\varphi_1} = \{\} \text{ and } V_{\varphi_1} = \{moves, free\},$$

leading to the following set of relevant nodes that we obtain initially:

$$\begin{aligned} N_{\varphi_1} = \{&(moves' = moves + 1)_{\texttt{effect\_}white}, \\ &(moves' = moves + 1)_{\texttt{effect\_}black}, \\ &(free' = Posn \setminus (bposn' \cup wposn'))_{\texttt{effect\_}white}, \\ &(free' = Posn \setminus (bposn' \cup wposn'))_{\texttt{effect\_}black}\} \end{aligned}$$

Note that both of the latter predicates are actually part of the state invariant which became part of the effect schemas due to our normalisation. The final results of the subsequent backwards slice computation on the dependence graph are the following sets of relevant nodes

$$N'_{\varphi_1} = N \setminus \left\{ \begin{array}{l} \texttt{effect\_}\mathit{result}, \\ \left(\begin{array}{l} \mathit{lines}(\mathit{bposn}) > \mathit{lines}(\mathit{wposn}) \\ \Rightarrow \mathit{result}! = \mathit{black\_wins} \end{array} \right)_{\texttt{effect\_}\mathit{result}}, \\ \left(\begin{array}{l} \mathit{lines}(\mathit{wposn}) > \mathit{lines}(\mathit{bposn}) \\ \Rightarrow \mathit{result}! = \mathit{white\_wins} \end{array} \right)_{\texttt{effect\_}\mathit{result}}, \\ \left(\begin{array}{l} \mathit{lines}(\mathit{wposn}) = \mathit{lines}(\mathit{bposn}) \\ \Rightarrow \mathit{result}! = \mathit{draw} \end{array} \right)_{\texttt{effect\_}\mathit{result}} \end{array} \right\}$$

and the following sets of relevant variables and events:

$$V'_{\varphi_1} = V$$
$$E'_{\varphi_1} = E$$

Thus, no variables and no complete methods can be removed from the specification; however, a number of predicates are irrelevant and can indeed be removed. Based on this result we compute the specification slice with respect to φ_2 that is depicted in Figure 5.1.

To summarise, the slice with respect to φ_1 exhibits only one difference in comparison to the original specification, namely only the `effect_`*result* schema is removed along with its contained predicates. This schema determines the final result that is communicated by event *result* to the environment. This outcome is sensible, of course, since the communicated result does not have any influence on the given property.

Example Slicing Criterion: Alternation of Moves

Our second example of slicing the Tic-Tac-Toe specification uses the following state/event interval logic formula as the verification property and slicing criterion:

$$\begin{aligned} \varphi_2 := {} & \neg \diamondsuit_{\mathrm{L}}(\mathit{black} \frown (\lceil \mathit{true} \rceil \wedge \neg(\lceil \mathit{true} \rceil \frown \mathit{white} \frown \lceil \mathit{true} \rceil)) \frown \mathit{black}) \\ & \wedge \neg \diamondsuit_{\mathrm{L}}(\mathit{white} \frown (\lceil \mathit{true} \rceil \wedge \neg(\lceil \mathit{true} \rceil \frown \mathit{black} \frown \lceil \mathit{true} \rceil)) \frown \mathit{white}) \end{aligned}$$

This formula describes the way of how both Tic-Tac-Toe players are expected to perform their moves, namely in an alternating fashion. The formula specifies this requirement by negating two suitable counterexamples: it should never happen that between two moves of one player, no move of its opponent takes place.

When slicing the class *TicTacToe* with respect to the formula, we first determine the sets of relevant events and variables appearing directly within the slicing

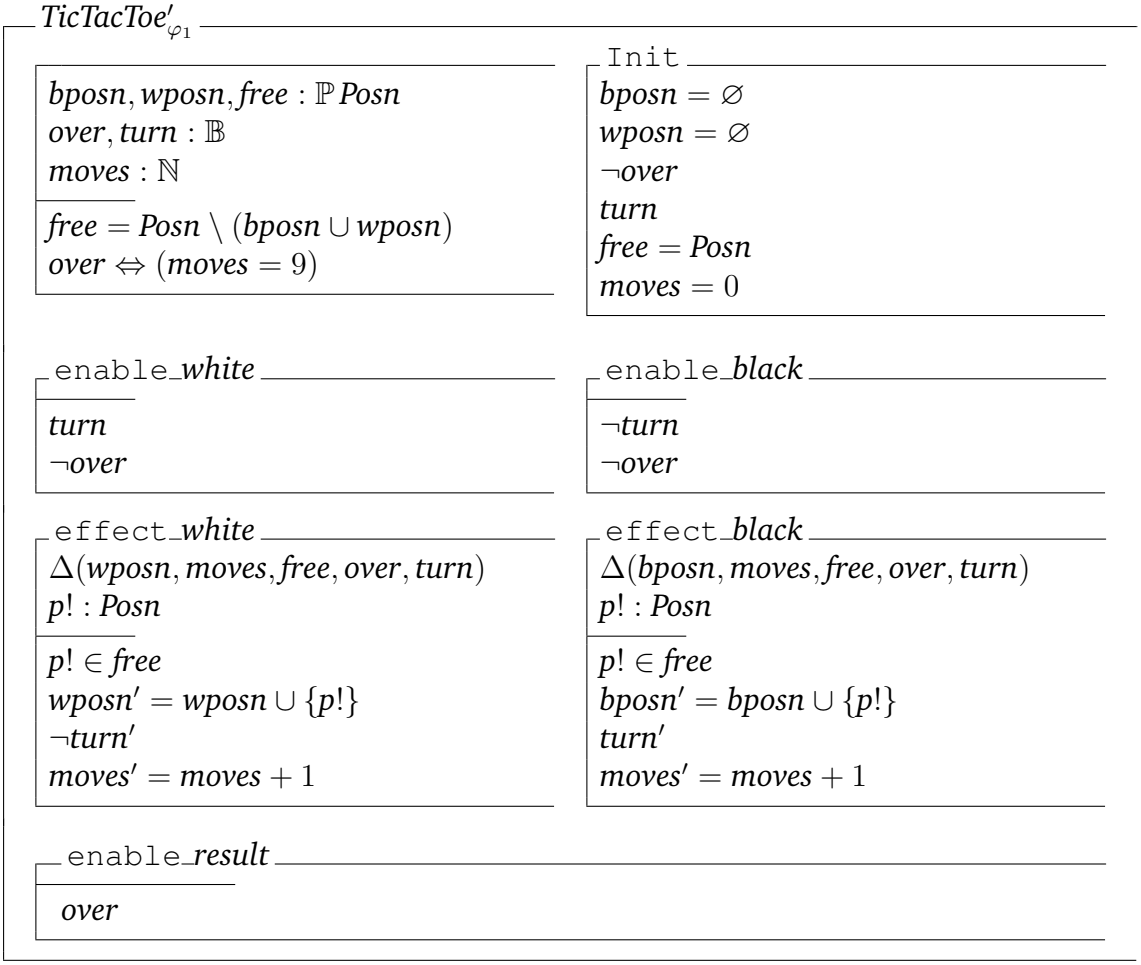


Figure 5.1: Slice of the Tic-Tac-Toe specification with respect to φ_1

criterion formula, namely $E_{\varphi_2} = \{black, white\}$ and $V_{\varphi_2} = \{\}$. From these we obtain the following initial set of relevant nodes

$$N_{\varphi_2} = \{\texttt{enable\_}white, \texttt{enable\_}black\}.$$

When starting at these nodes and subsequently computing the backwards slice on the dependence graph, the final results are the following sets of relevant dependence graph nodes

$$\begin{aligned} N'_{\varphi_2} = N \setminus \{&(bposn = \varnothing)_{Init}, (wposn = \varnothing)_{Init}, (free = Posn)_{Init}, \\ &(p! \in free)_{\texttt{effect\_}white}, (wposn' = wposn \cup \{p!\})_{\texttt{effect\_}white}, \\ &(free' = Posn \setminus (bposn' \cup wposn')_{\texttt{effect\_}white}, \\ &(p! \in free)_{\texttt{effect\_}black}, (bposn' = bposn \cup \{p!\})_{\texttt{effect\_}black}, \end{aligned}$$

$(free' = Posn \setminus (bposn' \cup wposn')_{\texttt{effect\_}black},$
$\texttt{effect\_}result,$
$(lines(bposn) > lines(wposn) \Rightarrow r! = black\_wins)_{\texttt{effect\_}result},$
$(lines(wposn) > lines(bposn) \Rightarrow r! = white\_wins)_{\texttt{effect\_}result},$
$(lines(wposn) = lines(bposn) \Rightarrow r! = draw)_{\texttt{effect\_}result},$
$(free' = Posn \setminus (bposn' \cup wposn')_{\texttt{effect\_}result}\}$

and the following sets of relevant variables and events:

$V'_{\varphi_2} = V \setminus \{bposn, wposn, free\}$
$E'_{\varphi_2} = E$

Based on this result we compute the specification slice with respect to φ_2 that is depicted in Figure 5.2.

Thus, in addition to the difference that we saw in the previous example, this slice yields an additional reduction in comparison to the original specification: All predicates have been removed that determine the sets of free and occupied fields, along with the variables that record the occupation of fields.

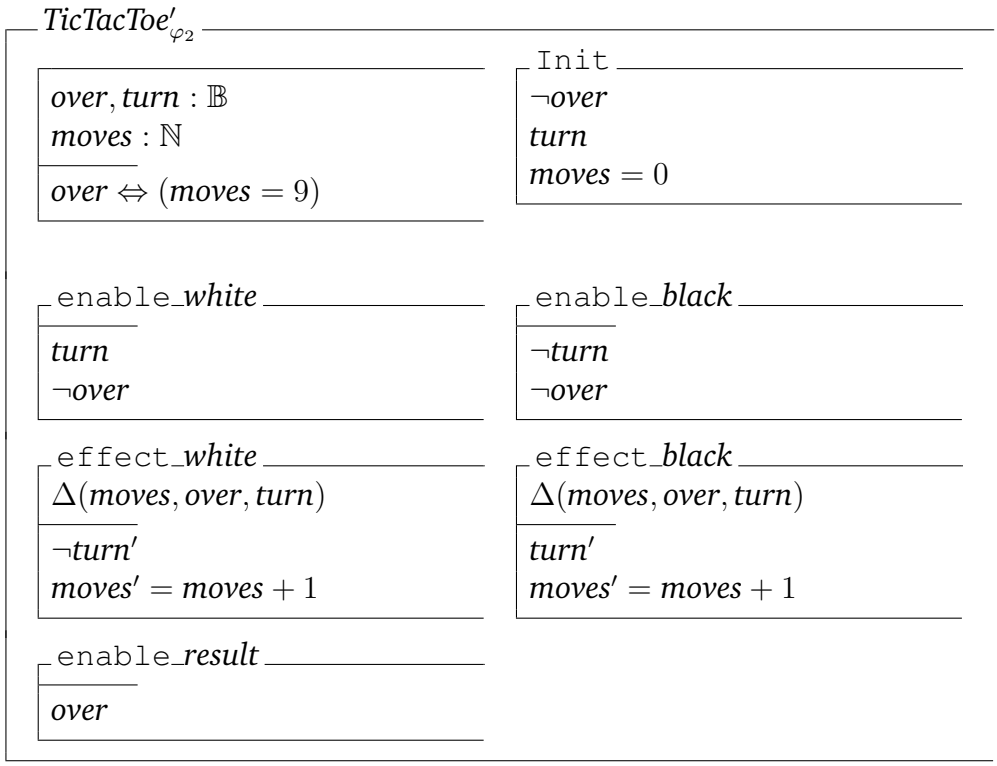


Figure 5.2: Slice of the Tic-Tac-Toe specification with respect to φ_2

This difference is sensible, since the given property expresses only that there is a strict alternation between the moves of the players. In order to analyse the sequence of moves the players can perform, the exact occupation of fields during the course of the game is irrelevant and all related predicates can safely be removed together with the variables that store the associated information about free and occupied fields.

All in all, the examples of slicing Object-Z specification with respect to verification properties that we presented in this section clearly suggest that slicing can substantially reduce the size of the specification and hence the underlying state space, such that verification of temporal logic properties will be facilitated.

5.4 CSP-OZ Specification Slices

For CSP-OZ specifications, we can compute a reduced specification in a similar way as for Object-Z specifications in that we use the same starting point, namely the result from the previously performed backwards slice computation on the program dependence graph, which are essentially the set of relevant nodes N', the set of relevant variables V' and the set of relevant events E'.

However, in addition to computing reduced versions of state space, `Init`, `enable`, and `effect` schemas, we now also have to compute a reduced version of the CSP part in order to obtain a reduced CSP-OZ specification. To this end we define the following notion of projection of CSP process definitions onto a given set of relevant events:

Definition 5.4.1 (Projection of CSP processes)**.** *Let P be the right-hand side of a process definition from the CSP part of a specification and E be the set of events that appear in the specification. The projection of P with respect to a set of events $E' \subseteq E$, denoted by $P|_{E'}$, is inductively defined:*

1. $\texttt{Skip}|_{E'} := \texttt{Skip}$ *and* $\texttt{Stop}|_{E'} := \texttt{Stop}$
2. $(e \rightarrow P)|_{E'} := \begin{cases} P|_{E'} & \text{if } e \notin E' \\ e \rightarrow P|_{E'} & \text{else} \end{cases}$
3. $(P \circ Q)|_{E'} := P|_{E'} \circ Q|_{E'}$ *with* $\circ \in \{\fatsemi, |||, \sqcap, \Box\}$
4. $(P \underset{S}{\parallel} Q)|_{E'} := P|_{E'} \underset{S \cap E'}{\parallel} Q|_{E'}$

The projection of the complete CSP part is then defined by applying the above definition to the right side of each process definition.

The application of this notion of projection can lead to CSP process definitions of the form $P = P$ which mean that a call of a process P introduces divergence at the

point of the process call, i.e., the process call P can then be regarded as equivalent to the divergence process `Div` which does nothing but diverge [Ros97].

From an operational semantics point of view, the divergence can in turn be regarded as being equivalent to process `Stop`, since a process' operational semantics is defined in terms of the sequence of events that the process communicates, and both `Div` and `Stop` never communicate. This gives rise to further simplifications of the CSP part that remove void process definitions of the form $P = P$ and suitably replace references to process P:

Definition 5.4.2 (Pruning of void process definitions)**.** *Let C_{CSP} be the CSP part of a CSP-OZ specification. A version of the CSP part that is* pruned from void process definitions of the form $P = P$*, denoted by* $\mathsf{PruneVoid}(C_{CSP})$*, is then obtained by applying the following transformation, until a fix point is reached, i.e., until the CSP part does not contain any more process definitions of the form $P = P$. For any process definition of the form $P = P$ the transformation steps are the following:*

1. *The process definition $P = P$ is removed from C_{CSP}.*
2. *Any reference to process P is replaced with a reference to the divergence process* `Div`*.*
3. *The following simplifications are applied to the right-hand side of each process definition of C_{CSP}:*
   a) $e \rightarrow \texttt{Div} := e \rightarrow \texttt{Stop}$
   b) $Q \fatsemi \texttt{Div} := Q \fatsemi \texttt{Stop}$
   c) $\texttt{Div} \fatsemi Q := \texttt{Stop}$
   d) $Q \,|||\, \texttt{Div} := Q$
   e) $Q \underset{A}{\parallel} \texttt{Div} := Q \underset{A}{\parallel} \texttt{Stop}$
   f) $Q \,\Box\, \texttt{Div} := Q$
   g) $Q \sqcap \texttt{Div} := Q \sqcap \texttt{Stop}$

Having previously identified the sets of relevant events E' and relevant variables V', which might influence the property (formula) under interest, the slice of a CSP-OZ specification can next be determined. In contrast to the original specification, the sliced specification class contains

- only channels associated with the set of relevant events E',
- only CSP process definitions that are projections of CSP process definitions from the original specification onto the set of relevant events E',

- inside the state schema only variables from V' (same type as in C),
- inside the Init schema only predicates restricting variables from V',
- only Object-Z schemas associated with events from E', and
- inside these schemas only predicates associated with nodes in N'.

Furthermore, the CSP part is pruned from void process definitions according to definition 5.4.2.

Thus, the only additional point in comparison to the previous slice computation for pure Object-Z specifications addresses the specification's CSP part and the computation of projections of CSP process definitions with respect to relevant events.

5.4.1 Example: Air Conditioner Slice

For illustrating the slicing approach for CSP-OZ specifications we now use the untimed air conditioner specification consisting of only the *AirConditioner* class that we have introduced in the previous chapters. Our verification property and slicing criterion will be the following state/event interval logic formula:

$$\varphi \equiv \boxed{\iota}(\mathit{work} \Rightarrow \mathit{fuel} > 5),$$

This formula expresses an invariant of the air conditioner requiring that, whenever the system is activated (*work*), the remaining fuel supply must stay above a certain critical level ($\mathit{fuel} > 5$).

Note that for the time being we only give this informal explanation of the meaning of this formula, since for the purpose of illustrating the slice computation it only matters, which variables and events are mentioned by the formula. For a formal definition of state/event interval logic formulae we refer again to the next Chapter 6 on slicing correctness.

For slicing the air conditioner with respect to φ we obtain the set of relevant events $E_\varphi = \{\}$ and the set of relevant variables $V_\varphi = \{\mathit{work}, \mathit{fuel}\}$. This, in turn, leads to the following initial set of relevant nodes:

$$\begin{aligned} N_\varphi = \{&(\mathit{work}' = \neg \mathit{work})_{\texttt{effect\_}\mathit{workswitch}}, \\ &(\mathit{fuel}' = \mathit{fuel} - 1)_{\texttt{effect\_}\mathit{consume}}, \\ &(\mathit{fuel}' = \mathit{min}(\mathit{fuel} + f?, 100))_{\texttt{effect\_}\mathit{refill}}\}. \end{aligned}$$

The result of the backwards slice on the dependence graph is the following:

$$\begin{aligned} N' &= N \setminus \{\texttt{effect\_}\mathit{modeswitch}, (\mathit{mode}' = m?)_{\texttt{effect\_}\mathit{modeswitch}}, \\ &\qquad \texttt{effect\_}\mathit{dtemp}, (t! = \mathit{mode})_{\texttt{effect\_}\mathit{dtemp}}, \\ &\qquad \texttt{effect\_}\mathit{level}, (f! = \mathit{fuel})_{\texttt{effect\_}\mathit{level}}\}, \\ E' &= E \setminus \{\mathit{modeswitch}, \mathit{dtemp}, \mathit{level}\}, \\ V' &= V \setminus \{\mathit{mode}\} \end{aligned}$$

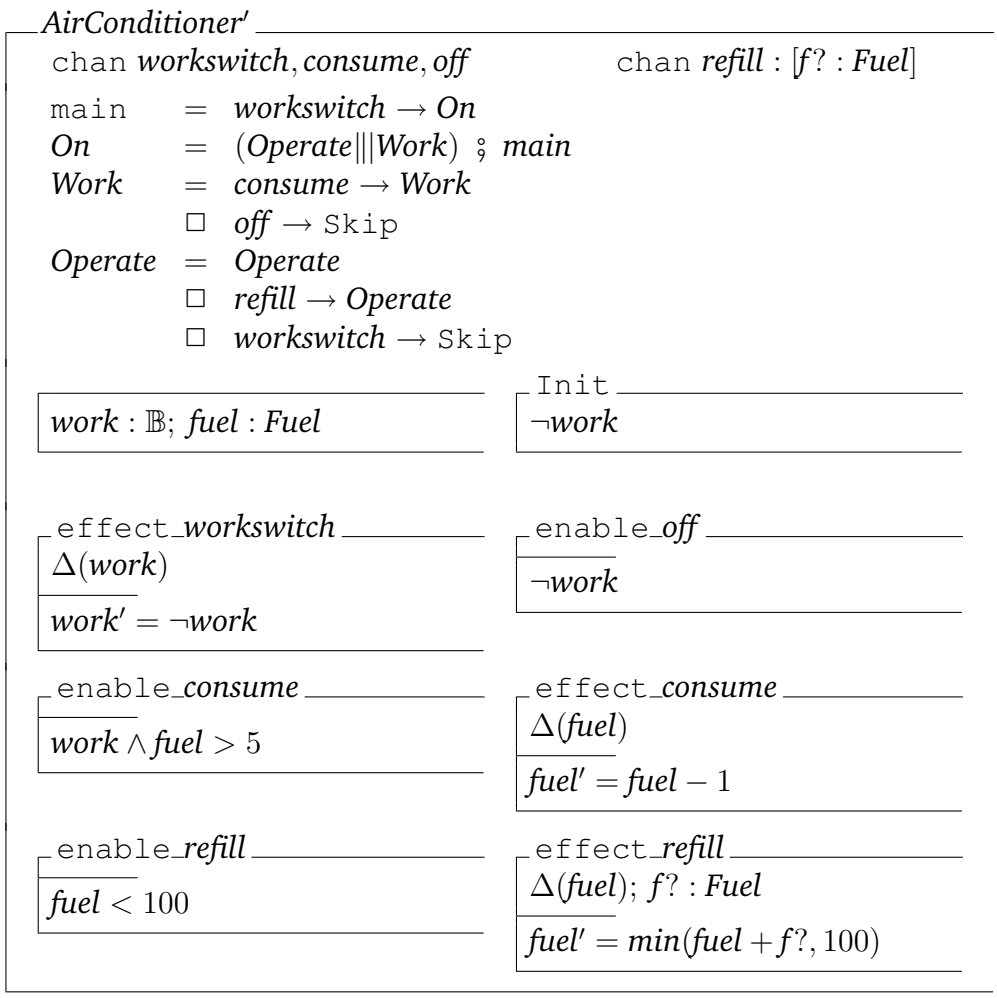


Figure 5.3: Slice of the untimed air conditioner class with respect to φ

Thus, the only irrelevant variable with respect to our given formula is variable *mode*, while three events are completely irrelevant for our formula and can therefore be removed from the specification. This leads to the specification slice depicted in Figure 5.3. All in all, the reductions achieved by applying our slicing algorithm to this example are:

1. Event *modeswitch* and thus a complete branch of the interleaving operator in the CSP Operate process has been removed together with variable *mode*, which is sensible, since the mode of operating of the air conditioner (heating or cooling) does not have any influence on the slicing criterion, property $\boxed{L}(work \Rightarrow fuel > 5)$, since the mode does neither have any kind of influence

on the activation status (*work*) nor on the remaining fuel supply ($fuel > 5$) of the air conditioner.

2. Events *dtemp* and *level* have been removed, which is also sensible, since neither modelling the effect that the air conditioner induces on the environment (*dtemp*) nor communicating the current amount of fuel to the environment (*level*) have any influence on the given property.

To summarise, the specification's state space has not only been reduced with respect to its control flow space (events *dtemp*, *modeswitch* and *level*), but also with respect to its data state space (variable *mode*).

Note that neither the original nor the sliced *AirConditioner* specification satisfies the given property, so the verification result will be negative in both cases. Nevertheless, this is exactly what we wanted to achieve: A specification slice must satisfy a slicing criterion if and only if the original specification does so which will be shown in the next Chapter 6 on slicing correctness.

5.5 CSP-OZ-DC Specification Slices

Also for the slicing approach for CSP-OZ-DC specifications, the final step consists in the computation of a reduced version of the original specification, i.e., a reduced specification without all details which are not relevant for the property that served as the primary slicing criterion. Verification with respect to this property can afterwards be performed on this reduced specification while the verification result will be the same.

Given the previously computed sets N', V' and E', it is then straightforward to construct the reduced specification. For each class C its sliced version C' contains

- only channels from the set of relevant events E',
- the projection of the original specification's CSP part onto E',
- a state schema with variables from V' only (same type as in C),
- an *Init* schema with only predicates restricting variables from V',
- only Object-Z schemas associated with events from E',
- inside these schemas only predicates associated with nodes in N', and
- a DC part with only counterexample formulae that mention variables from V' and events from E'.

Thus, the only additional point in comparison to the slice computation for CSP-OZ specifications addresses the specification's DC part and the possible exclusion of irrelevant counterexample formulae.

Note that due to the construction of timing dependence edges, either all or none of the variables and events of any given counterexample formula will be part of the slice.

5.5.1 Example: Timed Air Conditioner System Slice

For illustrating the slicing approach for CSP-OZ-DC specifications we now use the timed air conditioner system introduced in the previous chapters, i.e., the class *AirConditioner* in parallel composition with the class *Environment*. As our verification property and slicing criterion consider the following test formula:

$$\varphi \equiv \neg(\mathit{true} \frown \lceil \mathit{work} \wedge \mathit{fuel} < 5 \rceil \frown \mathit{true}).$$

This formula uses the pattern of a counterexample specification to express the following requirement: the air conditioner system must never reach a state where it is activated and where simultaneously its fuel supply has dropped below some critical value.

Note that for the time being we only give this informal explanation of this test formula's meaning, since for the purpose of illustrating the slice computation it only matters, which variables and events are mentioned by the formula. The formal definition of test formulae will be given in the next Chapter 6 on slicing correctness.

The sets of relevant events and variables that we obtain for φ are $E_\varphi = \{\}$ and $V_\varphi = \{\mathit{work}, \mathit{fuel}\}$. For slicing the air conditioner with respect to φ we then first compute the following initial set of relevant nodes:

$$\begin{aligned} N_\varphi = \{&(\mathit{work}' = \neg \mathit{work})_{\texttt{effect\_}\mathit{workswitch}}, \\ &(\mathit{fuel}' = \mathit{fuel} - 1)_{\texttt{effect\_}\mathit{consume}}, \\ &(\mathit{fuel}' = \mathit{min}(\mathit{fuel} + f?, 100))_{\texttt{effect\_}\mathit{refill}}\}. \end{aligned}$$

The result of the backwards slice on the dependence graph is the set of relevant nodes

$$\begin{aligned} N' = N \setminus \{&\texttt{effect\_}\mathit{level}, (f! = \mathit{fuel})_{\texttt{effect\_}\mathit{level}}, \\ &\texttt{effect\_}\mathit{tchange}, (\mathit{temp}' = \mathit{temp} + \mathit{dt})_{\texttt{effect\_}\mathit{tchange}}, \\ &\texttt{effect\_}\mathit{dlight}, \\ &\quad (l? = \mathit{darken} \Rightarrow \mathit{dl}' = -1)_{\texttt{effect\_}\mathit{dlight}}, \\ &\quad (l? = \mathit{brighten} \Rightarrow \mathit{dl}' = 1)_{\texttt{effect\_}\mathit{dlight}}, \\ &\texttt{effect\_}\mathit{lchange}, (\mathit{light}' = \mathit{light} + \mathit{dl})_{\texttt{effect\_}\mathit{lchange}}\}. \end{aligned}$$

This set of relevant dependence graph nodes leads to the following sets of relevant events for the *AirConditioner* class:

$$E'_{AirConditioner} = E_{AirConditioner} \setminus \{level\},$$
$$V'_{AirConditioner} = V_{AirConditioner}$$

And these are the results for the *Environment* class:

$$E'_{Environment} = E_{Environment} \setminus \{tchange, dlight, lchange\},$$
$$V'_{Environment} = V_{Environment} \setminus \{light, temp, dl\}.$$

The resulting slices of class *AirConditioner* and class *Environment* are depicted in Figures 5.4 and 5.5. Altogether we have thus achieved the following reductions:

AirConditioner: Method *level* has been removed, which is sensible, since all this method does is to communicate the current amount of fuel to the environment (not the class *Environment*), which does not influence the verification property φ.

Note that methods *modeswitch*, *dtemp* as well as variable *mode* have not been removed. The reason is that method *dtemp* belongs to the synchronisation alphabet and might therefore block, depending on the behaviour of its synchronisation partner. Therefore, both need to remain in the slice.

Environment: Methods *tchange*, *dlight* and *lchange* have been removed as well as variables *light*, *temp*, and *dl* and DC formula

$$\neg\Diamond(\updownarrow dlight \frown \boxminus lchange \wedge \ell > 1).$$

This result is also sensible, since the actual effect imposed on the environment's temperature (*tchange* and *temp*) does not influence the verification property and the modelling of the environment's lighting behaviour (*dlight*, *lchange*, *light* and *dl*) is not related to the verification property φ at all.

To summarise, the specification's state space has not only been reduced with respect to its control flow space (events *level*, *tchange*, *dlight*, and *lchange*), but also with respect to its data state space (variables *light*, *temp*, and *dl*) and its timing requirements (the DC part of *Environment*).

Note that in both cases neither the original nor the sliced specification satisfies the given property, so the verification result will be negative in both cases. Nevertheless, this is exactly what we wanted to achieve: A specification slice must satisfy a slicing criterion if and only if the original specification does so.

In the next Chapter 6 on slicing correctness we will show that our slicing algorithm guarantees this outcome for any specification and any slicing criterion formulated in a stuttering invariant logic.

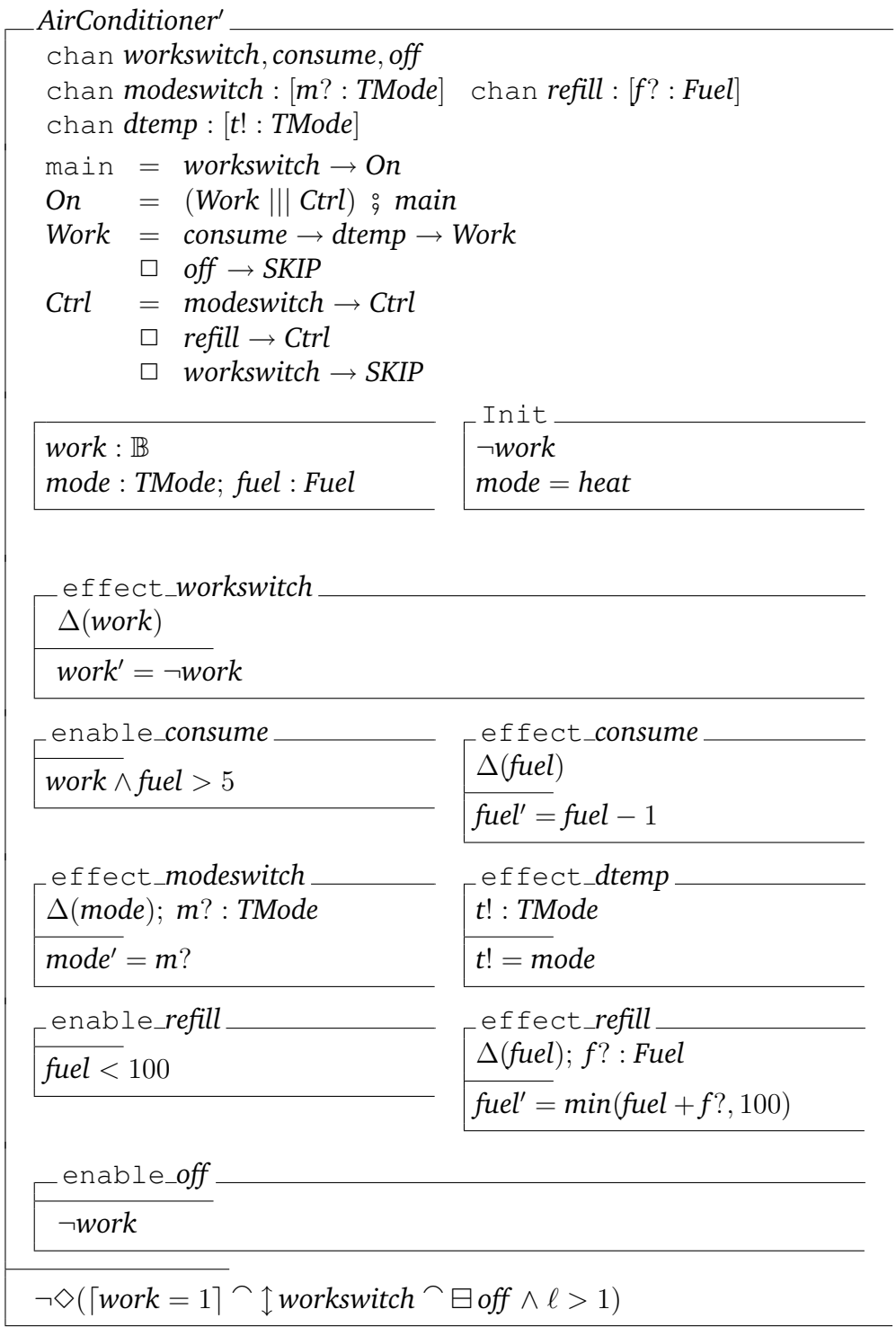


Figure 5.4: Slice of the timed air conditioner class with respect to φ

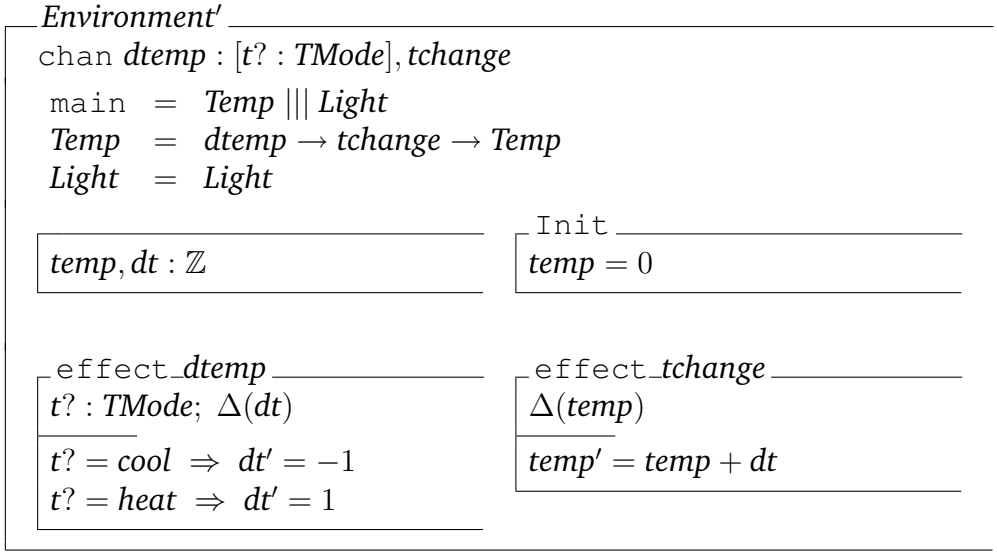


Figure 5.5: Slice of the environment class with respect to φ

5.6 Classification of the Slicing Approach

With respect to the possible classifications of slicing approaches presented in Chapter 2, the slicing approach developed in this thesis belongs to the following categories:

- The slice computation is obviously *based on dependence graphs* instead of data flow equations.
- The slices are computed *statically*, since no assumptions are imposed on the input during run-time of the specified systems.
- The direction of slicing is *backward*, since we use slicing to identify influences on a given slicing criterion instead of computing what specification elements are influenced by the slicing criterion.
- Of course, the attribute of executability does not really make sense with respect to formal specifications. However, the specifications obtained from our slicing approach share the following concepts with conventionally computed executable slices: they are syntactically valid and they have a well-defined semantics that can be related with the semantics of the original slicing target. In this sense, the slices computed by our slicing approach can be regarded as being *executable*.

- The slicing criterion used is *static*, since no assumption on the environment is made. However, it does not name a particular location of the specification. Instead, these are given *implicitly* by the verification property and have to be determined in a previous analysis of the slicing criterion.
- The *target language* of the slicing approach obviously is a formal specification notation that comprises concepts of *concurrency* and *real-time* aspects.
- The primary *application area* of our slicing approach is *verification*, since our main motivation is to develop a method for mitigating the problem of state space explosion in model checking. However, additional applications are of course possible, such as those for comprehension or measurement purposes that have been introduced in Chapter 2.

6 Slicing Correctness

Contents

The task of the slicing approach presented in the previous chapters is to compute a reduced specification with respect to a verification property, such that the given property can be verified on the reduced instead of the full specification. To this end, the reduction obtained by slicing needs to be exact in the sense that with respect to the verification property, both the full and the reduced system are equivalent. This is the notion of *correctness* of the slicing algorithm that we will show in this chapter, i.e., we show that the property (and slicing criterion) φ holds for the full specification if and only if it holds for the reduced specification.

In Section 6.1 we first define the notion of a *projection* relationship between interpretations that will be the basis of the subsequent correctness proof. Moreover, Section 6.1 contains several lemmas that establish the relation between specification elements of the full and the reduced specification with artefacts of the slicing algorithm such as associated control flow graph elements and according types of dependences. These lemmas will be needed in the actual correctness proof.

Next, we show in Section 6.2 that an interpretation of the reduced specification is within the projection of an interpretation of the full specification onto some relevant subset of the variables and events (obtained from the slicing algorithm), i.e., both specifications only differ on variables and events that are not relevant with respect to the given formula.

Having established such a projection relationship and having shown that this relationship is guaranteed by our slicing approach is the main result of the correctness proof. However, to complete the proof, we still have to show that the particular logics we use are *stuttering invariant* with respect to the projection relation, i.e., they cannot distinguish between interpretations of the full and the reduced version of the specification. We show this in Section 6.3 for the logic that we use for expressing real-time properties of CSP-OZ-DC specifications, i.e., for *test formulae*, and in Section 6.4 for the logic that we use for expressing temporal logic properties of Object-Z and CSP-OZ specifications, i.e., for *state/event interval logic formulae*.

6.1 Relating Slicing Results with Specification Elements

This section first defines the central concept of the correctness proof, namely the notion of projection relation between interpretations. This kind of projection relationship will later be shown to exist between interpretations of the full specification and those of the reduced specification, provided that we apply the slicing approach developed in the previous chapters.

Furthermore, this section prepares the actual correctness proof with several lemmas showing the relationships between objects of the slicing approach on the one hand such as paths in the control flow graph or events and variables identified by slicing as being relevant and specification elements on the other hand such as CSP process definitions of the full and the reduced specifications or relevant and irrelevant DC counterexample formulae. Each of the lemmas will be needed in the next section that contains the actual correctness theorem, i.e., the claimed existence of a projection relation between specifications and their associated slices.

In the following, we assume E' to be the set of relevant events and V' to be the set of relevant variables obtained from applying the slicing algorithm to a specification C with respect to a slicing criterion φ, resulting in a specification slice C'.

6.1.1 Projection Relation between Interpretations

Intuitively, when computing the projection of a given interpretation onto a set of relevant variables and a set of relevant events, one divides the interpretation into blocks formed by time intervals beginning at one relevant event and ending at the next relevant event. The corresponding block of an interpretation in the projection refers to the same time interval, but does not contain any of the irrelevant events that may appear inside the block of the original interpretation. Moreover, the interpretation and its projection coincide throughout both blocks on the valuation of all relevant variables.

Definition 6.1.1 (Projection of interpretation)**.** *Let O' be a set of CSP-OZ-DC-observables, $E' = O' \cap$ Events the set of events within O' and $\mathcal{I}, \mathcal{I}'$ be two E'-fair*

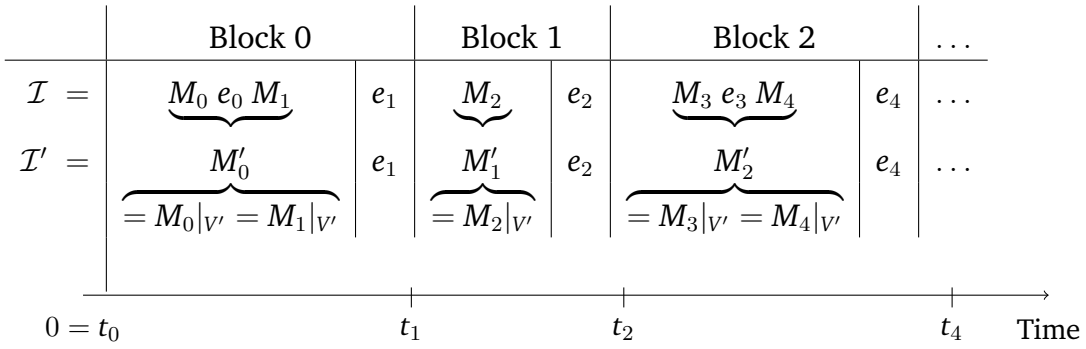


Figure 6.1: Exemplary interpretation $\mathcal{I}$ and a corresponding interpretation $\mathcal{I}' \in \mathsf{Projection}_{O'}(\mathcal{I})$, within its projection with respect to a set of CSP-OZ-DC-observables O', with $O' \supseteq (E' \cup V')$ and $E' \supseteq \{e_1, e_2, e_4\}$, but $e_0, e_3 \notin E'$.

interpretations with $0 = t_0 < t_1 < t_2 < \ldots$ *and* $0 = t'_0 < t'_1 < t'_2 < \ldots$ *being the points in time where* $\mathcal{I}$ *and* $\mathcal{I}'$ *change, respectively.* $\mathcal{I}'$ *is in the* projection of $\mathcal{I}$ with respect to O'*, denoted by* $\mathcal{I}' \in \mathsf{Projection}_{O'}(\mathcal{I})$*, iff*

1. $\forall\, t\colon \mathcal{I}(t)|_{O' \setminus E'} = \mathcal{I}'(t)|_{O' \setminus E'}$
2. $\forall\, i \geq 0\colon\ \exists j\colon \quad (t_i = t'_j \wedge \mathsf{TakesPlace}(\mathcal{I}, t_i) = \mathsf{TakesPlace}(\mathcal{I}', t'_j))$
   $\vee\ (t'_j < t_i < t'_{j+1} \wedge \mathsf{TakesPlace}(\mathcal{I}, t_i) \cap E' = \varnothing)$

A graphical illustration of the effect of the projection definition is depicted in Figure 6.1 where also the intuitive notion of *projection blocks* is illustrated. All states inside a block are projection-equivalent (i.e., they coincide on the given set of variable valuations) and all events inside a block are irrelevant events (i.e., events not from the given set of events) except for the last event in a block which is a relevant event (i.e., an event from the given set of events).

The projection of the original interpretation contains then any interpretation such that for each of the blocks of the original interpretation all states and irrelevant events are mapped onto one single state of the new interpretation, while the relevant event remains in the new interpretation without changing the point in time of its occurrence.

Given a logic which is invariant under projections, such a projection relationship between any two interpretations then guarantees that formulae which only mention observables from O', i.e., variables from V' and events from E' as obtained from slicing, hold for either both or none of the interpretations.

Lamport [Lam02] introduced the following notion of *stuttering invariance* for the setting of discrete, non-continuous systems:

> "[...] A stuttering step represents a change only to some part of the system not described by the formula; adding it to the behavior should not affect the truth of the formula. We say that a formula F is *invariant*

under stuttering iff adding or deleting a stuttering step to a behavior σ does not affect whether σ satisfies F."

Another related notion of stuttering is introduced by Apt and Olderog [AO97] in the context of verification of parallel programs. There, discrete stuttering steps arise from replacing assignments to auxiliary variables with *skip* statements. These stuttering steps are then shown to be safely removable without changing the semantics of the given programs.

Note that projection can be regarded as a particular form of stuttering, since each of the intermediate irrelevant events that are present in the original interpretation resembles a stuttering step that does not have any relevant effect on the state space.

In the following, we will therefore also apply a very similar notion of stuttering invariance in the continuous setting of interpretations. We will characterise formulae as being *stuttering invariant* iff they cannot distinguish between interpretations and their projections.

6.1.2 Transitions of CSP Process Projections

Our first lemma considers the case of a single CSP transition: Either this transition is associated with a relevant event $e \in E'$ or with an irrelevant event $e \notin E'$. In the former case it is easy to see that the associated projection also can perform this relevant event e, while in the latter case some further considerations lead to the conclusion that there is some subsequent relevant event f that the original process will eventually perform (after e and possibly further irrelevant events have occurred), which is the same event that will immediately take place within the associated projection.

Lemma 6.1.2 (Single transitions of CSP process projections)**.** *Let P and Q be CSP processes with $P \xrightarrow{e} Q$, and E' a set of relevant events. Projections of P and Q with respect to E' are related in one of the following ways:*

1. $e \in E' \quad \Rightarrow \quad P|_{E'} \xrightarrow{e} Q|_{E'}$
2. $e \notin E' \quad \Rightarrow \quad \forall f \in E', \forall R\colon \left(Q\ |_{E'} \xrightarrow{f} R|_{E'} \quad \Rightarrow \quad P\ |_{E'} \xrightarrow{f} R|_{E'}\right)$

Proof: We show both cases by induction over the structure of P. Since we know that P can perform event e, we only have to consider a limited set of CSP constructs.

1. e is a relevant event:
   a) $P \equiv e \longrightarrow Q$. ✓
   b) $P \equiv P_1 \,\square\, P_2$ with $P_i \xrightarrow{e} Q$ for $i \in \{1, 2\}$.
   Then $P|_{E'} \equiv P_1|_{E'} \,\square\, P_2|_{E'}$ and, since $P_i \xrightarrow{e} Q$, the induction assumption leads us to $P|_{E'} \xrightarrow{e} Q|_{E'}$. ✓

c) $P \equiv P_1 ||| P_2$ with $P_i \xrightarrow{e} P_i'$ for $i \in \{1,2\}$ and $Q \equiv \begin{cases} P_1' ||| P_2 & \text{if } i = 1 \\ P_1 ||| P_2' & \text{else} \end{cases}$.
Then $P|_{E'} \equiv P_1|_{E'} ||| P_2|_{E'}$ and, since $P_i \xrightarrow{e} P_i'$, we have $P|_{E'} \xrightarrow{e} Q|_{E'}$ with $Q|_{E'} \equiv \begin{cases} P_1'|_{E'} ||| P_2|_{E'} & \text{if } i = 1 \\ P_1|_{E'} ||| P_2'|_{E'} & \text{else} \end{cases}$. ✓

d) $P \equiv P_1 \underset{S}{\|} P_2$ with two cases for e:

i. $e \in S$, i.e., $P_1 \xrightarrow{e} P_1'$, $P_2 \xrightarrow{e} P_2'$ and $Q \equiv P_1' \underset{S}{\|} P_2'$.
Then $P|_{E'} \equiv P_1|_{E'} \underset{S \cap E'}{\|} P_2|_{E'}$ and, since $P_i \xrightarrow{e} P_i'$, the induction assumption yields $P_i|_{E'} \xrightarrow{e} P_i'|_{E'}$
and thus also $Q|_{E'} \equiv P_1'|_{E'} \underset{S \cap E'}{\|} P_2'|_{E'}$. ✓

ii. $e \notin S$, i.e., $P_i \xrightarrow{e} P_i'$ for $i \in \{1,2\}$ and $Q \equiv \begin{cases} P_1' ||| P_2 & \text{if } i = 1 \\ P_1 ||| P_2' & \text{else} \end{cases}$
which is analogous to case (c). ✓

2. e is no relevant event:

a) $P \equiv e \longrightarrow Q$.
Then $P|_{E'} \equiv Q|_{E'}$. ✓

b) $P \equiv P_1 \,\square\, P_2$ with $P_i \xrightarrow{e} Q$ for $i \in \{1,2\}$.
Then $P|_{E'} \equiv P_1|_{E'} \,\square\, P_2|_{E'}$ and, since $P_i \xrightarrow{e} Q$, it follows from the induction assumption that

$$\forall f \in E' : Q|_{E'} \xrightarrow{f} R|_{E'} \quad \Rightarrow \quad P_i|_{E'} \xrightarrow{f} R|_{E'}$$

and thus the further implication $P|_{E'} \xrightarrow{f} R|_{E'}$. ✓

c) $P \equiv P_1 ||| P_2$ with $P_i \xrightarrow{e} P_i'$ for $i \in \{1,2\}$ and $Q \equiv \begin{cases} P_1' ||| P_2 & \text{if } i = 1 \\ P_1 ||| P_2' & \text{else} \end{cases}$.
From the induction assumption it follows that

$$\forall f \in E' : P_i'|_{E'} \xrightarrow{f} R|_{E'} \quad \Rightarrow \quad P_i|_{E'} \xrightarrow{f} R|_{E'}$$

and, since the other component of P remains unchanged during the transition from P to Q, it is obvious that

$$\forall f \in E' : Q|_{E'} \xrightarrow{f} R|_{E'} \quad \Rightarrow \quad P|_{E'} \xrightarrow{f} R|_{E'}$$

holds. ✓.

d) $P \equiv P_1 \underset{S}{\|} P_2$ with two cases for e:

i. $e \in S$, i.e., $P_1 \xrightarrow{e} P_1'$, $P_2 \xrightarrow{e} P_2'$ and $Q \equiv P_1' \underset{S}{\parallel} P_2'$.
Then $P|_{E'} \equiv P_1|_{E'} \underset{S \cap E'}{\parallel} P_2|_{E'}$ and $Q|_{E'} \equiv P_1'|_{E'} \underset{S \cap E'}{\parallel} P_2'|_{E'}$. The induction assumption yields

$$\forall f_i \in E', \forall R_i : P_i'|_{E'} \xrightarrow{f_i} R_i|_{E'} \Rightarrow P_i|_{E'} \xrightarrow{f_i} R_i|_{E'}.$$

Therefore, also $Q|_{E'} \xrightarrow{f} R|_{E'}$ implies $P|_{E'} \xrightarrow{f} R|_{E'}$ with either (1) $f = f_1 = f_2 \in S \cap E' : P_1'|_{E'} \xrightarrow{f} R_1|_{E'} \wedge P_2'|_{E'} \xrightarrow{f} R_2|_{E'}$ or otherwise (2) $\exists i \in \{1,2\} : f = f_i \in S \cap E' : P_i'|_{E'} \xrightarrow{f} R_i|_{E'}$. ✓

ii. $e \notin S$, i.e., $P_i \xrightarrow{e} P_i'$, and $Q \equiv \begin{cases} P_1' \underset{S}{\parallel} P_2 & \text{if } i = 1 \\ P_1 \underset{S}{\parallel} P_2' & \text{else} \end{cases}$
which is analogous to case (c). ✓ □

The next lemma extends the previous one from single transitions to transition sequences associated with complete projection blocks. It states that the projection of each residual CSP process associated with a projection interval without relevant events, as defined in Definition 6.1.1, can mimic the behaviour of the residual CSP process associated with the last state of the projection block, i.e., the relevant event at the end of the block is enabled at any previous step inside the block when computing the CSP projection.

Lemma 6.1.3 (Transitions of CSP process projections)**.** *Let $P_j, \ldots, P_{j+k+1}$ be CSP processes, E' a set of relevant events, $e_{j+1}, \ldots, e_{j+k-2} \notin E'$ irrelevant events, and $e_{j+k} \in E'$ a relevant event, such that*

$$P_j \xrightarrow{e_{j+1}} P_{j+2} \xrightarrow{e_{j+3}} \ldots \xrightarrow{e_{j+k-2}} P_{j+k-1} \xrightarrow{e_{j+k}} P_{j+k+1}$$

is a valid transition sequence. Then the following holds:

$$P \xrightarrow{e_{j+k}} P_{j+k+1}|_{E'} \text{ with } P \in \{P_j|_{E'}, \ldots, P_{j+k-1}|_{E'}\}$$

Note that $P_j|_{E'} = \ldots = P_{j+k-1}|_{E'}$ does not necessarily hold.

Proof: To prove this we apply the two clauses of Lemma 6.1.2 backwards, starting with the last step of the transition sequence:

1. For $P \equiv P_{j+k-1}|_{E'}$ this is obvious due to clause 1 of Lemma 6.1.2.
2. For the projections of the remaining processes
$$P \equiv P_{j+k-3}|_{E'}, \ldots, P \equiv P_j|_{E'}$$
we can repeatedly apply clause 2 of Lemma 6.1.2 to the respective previous case. □

6.1.3 CSP Transition Sequences

Next, we bridge the gap between transition sequences that we can observe for CSP processes and paths that are present in the associated control flow graph.

Lemma 6.1.4 (CSP transition sequences and CFG paths)**.** *Let C be a class specification, CFG its control flow graph, $\mathcal{I}$ an interpretation satisfying C with $0 = t_0 < t_1 < t_2 < \ldots$ being the points in time where $\mathcal{I}$ changes, t_i with $i > 0$ one of these points with $e \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$ and $f \in \mathsf{TakesPlace}(\mathcal{I}, t_{i+1})$. Then the two corresponding nodes* `enable_`e *and* `enable_`f *of CFG are related in either one of the following ways:*

1. *There exists a path in CFG which leads from* `enable_`e *to* `enable_`f*:*

$$path_{CFG}(\texttt{enable\_}e, \texttt{enable\_}f) \neq \varnothing$$

2. *There exists a CFG node $interleave^i$ or par^i_S with $S \cap \{e,f\} = \varnothing$ which has* `enable_`e *and* `enable_`f *as successors in different branches:*

$$\begin{aligned}&\exists\, n \in CFG\colon n \equiv interleave^i \vee (n \equiv par^i_S \wedge S \cap \{e,f\} = \varnothing) :\\ &\quad \exists\, \pi_e \in path_{CFG}(n, \texttt{enable\_}e) \wedge \exists\, \pi_f \in path_{CFG}(n, \texttt{enable\_}f) :\\ &\qquad \pi_e \cap \pi_f = \{n\}\end{aligned}$$

Proof: Let $P \equiv \mathsf{CSP}_C(\mathcal{I}, t_i)$ and $Q \equiv \mathsf{CSP}_C(\mathcal{I}, t_{i+1})$ such that $P \xrightarrow{e} Q$ is a valid CSP transition. We show both cases by considering the structure of P and Q, respectively. Since we know that P can perform event e, and Q can perform event f, we again only have to consider a limited set of CSP constructs for the structure of P and Q. For the structure of P we can distinguish the following cases:

1. $P \equiv e \longrightarrow Q$.
   Then the following cases apply for the structure of Q:
   a) $Q \equiv f \longrightarrow R$. ✓ (Path exists)
   b) $Q \equiv Q_1 \,\Box\, Q_2$ with $Q_i \xrightarrow{f} R$ for $i \in \{1,2\}$. ✓ (Path exists)
   c) $Q \equiv Q_1 \sqcap Q_2$: analogous. ✓ (Path exists)
   d) $Q \equiv Q_1 \,|||\, Q_2$ with $Q_i \xrightarrow{f} Q_i'$ for $i \in \{1,2\}$
   and $R \equiv \begin{cases} Q_1' \,|||\, Q_2 & \text{if } i = 1 \\ Q_1 \,|||\, Q_2' & \text{else} \end{cases}$. ✓ (Path exists)
   e) $Q \equiv Q_1 \underset{S}{\parallel} Q_2$ with two cases for f:
      i. Either $f \in S$: $Q_1 \xrightarrow{f} Q_1'$, $Q_2 \xrightarrow{f} Q_2'$
      and $R \equiv Q_1' \,|||\, Q_2'$. ✓ (Path exists)

ii. Otherwise $f \notin S$: analogous to 1.(d). ✓ (Path exists)

f) $Q \equiv X$ with X as in one of the previous cases for Q:
analogous to the applicable case. ✓ (Path exists)

2. $P \equiv P_1 \,\square\, P_2$ with $P_i \xrightarrow{e} Q$ for $i \in \{1,2\}$.
Structure of Q: as in the previous case for P. ✓ (Path exists)

3. $P \equiv P_1 \sqcap P_2$ with $P_i \xrightarrow{e} Q$ for $i \in \{1,2\}$.
Structure of Q: as in the previous cases for P. ✓ (Path exists)

4. $P \equiv P_1 \| \| P_2$ with $P_i \xrightarrow{e} P_i'$ for $i \in \{1,2\}$ and $Q \equiv \begin{cases} P_1' \| \| P_2 & \text{if } i = 1 \\ P_1 \| \| P_2' & \text{else} \end{cases}$.
Then the following cases apply for the structure of Q:

a) Either it has the same structure as in one of the previous cases for P. ✓ (Path exists)

b) Or the other branch of the interleaving operator comes into play, such that we have e and f in different branches of the interleaving operator of P. ✓ (Different Branches)

5. $P \equiv P_1 \underset{S}{\|} P_2$ with two cases for e:

a) Either $e \in S$: $P_1 \xrightarrow{f} P_1'$, $P_2 \xrightarrow{f} P_2'$ and $Q \equiv P_1' \underset{S}{\|} P_2'$ which leads to another case distinction with respect to the structure of Q, analogous to case 1.

b) Otherwise $e \notin S$: which is analogous to 4. ✓

6. $P \equiv X$ with X as in one of the previous cases.
Then one of the respective previous cases applies, according to the structure of Q. ✓ (Either path exists or different branches.) □

6.1.4 Irrelevant Events

The following lemma states that the set of irrelevant events appearing inside a projection block does not have any influence on the relevant variables associated with the states inside the block.

Lemma 6.1.5 (No influence of irrelevant events on relevant variables)**.** *Let C be a class specification, $\mathcal{I}$ an interpretation satisfying C with*

$$0 = t_0 < t_1 < t_2 < \dots$$

being the points in time where $\mathcal{I}$ changes, each of these t_i associated with an event $e_i \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$ for $i > 0$ such that $e_{j+1}, \dots, e_{j+k-1} \notin E'$ are irrelevant events and

$e_j, e_{j+k} \in E'$ are relevant events with E' being the set of relevant events obtained from the slicing algorithm for slicing C with respect to some formula φ (with an associated set of variables V_φ).
Then the following holds:

$$\mathcal{I}(t_j)|_{\overline{V}_j} = \ldots = \mathcal{I}(t_{j+k-1})|_{\overline{V}_j} \quad \textit{with } \overline{V}_j = V_\varphi \cup \bigcup_{e \in \{e_i \in E' \mid i \geq j\}} \textit{ref}(e).$$

Proof: Suppose, the equality does not hold. Then there is some irrelevant event e_l ($j+1 \leq l \leq j+k-1$) that modifies some variable $v \in \overline{V}_j$. One of the following cases applies:

- $v \in V_\varphi$: According to the definition of the slice (which gathers all nodes modifying a relevant variable within the initial set of directly relevant nodes) this leads to $e_l \in E'$, which is a contradiction.
- $v \notin V_\varphi$: According to the definition of $\overline{V}_j$, there is a subsequent relevant event $e_i \in E'$ ($i \geq j+k$) that refers to v. According to Lemma 6.1.4 the control flow graph nodes associated with e_l and e_i are related in either one of the following ways:
  1. There is a path in the control flow graph that connects both nodes directly.
  2. Both nodes are located in separate branches of an interleaving operator or parallel composition node.

  Both cases imply the existence of a data dependence (either a direct or an interference data dependence) between e_l and e_i. Therefore, according to the construction of the slice (which follows the dependence edges in a backwards-oriented direction), $e_i \in E'$ implies that also $e_l \in E'$, which is a contradiction to our initial assumption. □

6.1.5 Irrelevant DC Formulae

Our last preparatory lemma states that DC formulae which the slicing algorithm identified to be irrelevant with respect to a property to be verified do not impose restrictions on any relevant event.

Lemma 6.1.6 (No influence of irrelevant DC formulae on relevant events)**.** *Let C be a class specification, E' the set of relevant events obtained from slicing C with respect to some slicing criterion φ, and CE a counterexample formula from the DC part of C which is irrelevant with respect to φ. Let*

$$E_{CE} = \textit{events}(CE) \cup \{e \in \textit{Events} \mid \textit{mod}(e) \cap \textit{vars}(CE) \neq \varnothing\}$$

be the set of events that CE refers to either directly or indirectly by referring to some variable that is modified by the respective event. Then the following holds:

1. *There exists no CFG path connecting events from E_{CE} with events from E'.*

2. *The points in time where events from E_{CE} take place do not depend on the points in time where events from E' take place.*

Proof:

1. Suppose, there exists a CFG path from some node `enable_`$e \in E_{CE}$ to an event `enable_`$f \in E'$. We have to consider two cases for e:

   a) Either this event is directly mentioned inside *CE*. Then its `enable` schema node is source of a control dependence edge in the dependence graph (control dependence due to timing) and therefore backwards reachable from the `enable_`f node via control dependence edges which is a contradiction to e being irrelevant.

   b) The other case is that $e \in E_{CE}$, since *CE* mentions some variable that is modified by e. Nevertheless, `enable_`e will then be source of a control dependence edge in the dependence graph (again control dependence due to timing), such that the same argument holds, which contradicts e being irrelevant.

2. Suppose, there is an event $e \in E_{CE}$ that needs to occur at some point in time before another event $f \in E'$. This can only be the case if e and f are related in one of the following ways:

   a) There exists a CFG path from `enable_`e to `enable_`f. Since there will also be a control dependence edge due to timing leaving from `enable_`e, this leads to a contradiction according to the first part of this lemma.

   b) Both events are directly related by being mentioned in (or by modifying variables mentioned in) one and the same DC formula. This gives rise to an associated timing dependence edge and thus is a contradiction to e being irrelevant.

   c) Both events are indirectly related by being mentioned in (or by modifying variables mentioned in) two different DC formulae, which are anchored to the same point in time, i.e., which either both refer to the same anchor event, common to both DC formulae, or which are both anchored to the point of initialisation by imposing a time bound on their initial phase. In both cases we obtain a timing dependence edge connecting both events and thus a contradiction to e being irrelevant. □

6.2 Projection Relation Established by Slicing

Now we come to the main result of the correctness proof, namely the following theorem that states the existence of a projection relationship between any two interpretations associated with the original specification on the one hand and with the reduced specification that we obtained from our slicing approach on the other hand.

Theorem 6.2.1 (Slicing guarantees projection relation)**.** *Let C be a class specification and C' the class obtained when slicing C with respect to a formula φ with E_φ being the set of events and V_φ being the set of variables that φ contains. Let E' and V' be the set of events and variables, respectively, which the slicing algorithm delivers as those of interest (in particular, $E_\varphi \subseteq E'$ and $V_\varphi \subseteq V'$). Then for any E'-fair interpretation $\mathcal{I}$ satisfying C there is a corresponding E'-fair interpretation $\mathcal{I}'$ satisfying C' (and vice versa) such that the following holds:*

$$\mathcal{I}' \in \mathsf{Projection}_{V' \cup E'}(\mathcal{I})$$

Proof:
We need to consider two directions: (1) We have to show that for any interpretation of C we can construct a corresponding interpretation of C' such that the projection relation holds and (2) vice versa.

Direction (1): $C \rightsquigarrow C'$

Let $\mathcal{I}$ be an interpretation satisfying C with $0 = t_0 < t_1 < t_2 < \ldots$ being the points in time where $\mathcal{I}$ changes and let $\overline{V}_i$ be the set of variables that are associated with the slicing criterion and for each point in time t_i all relevant variables that are referenced by relevant events $e \in E'$ taking place at t_i or later points in time:

$$\overline{V}_i = V_\varphi \cup \{ref(e) \cap V' \mid \exists\, t_j \geq t_i \colon e \in \mathsf{TakesPlace}(\mathcal{I}, t_j)\}$$

In comparison to V', the set $\overline{V}_i$ contains only those variables of V' whose values can indeed influence the holding of the formula, since they are referenced at t_i or later, i.e., they occur in unprimed form in predicates of subsequent relevant events. Variables out of $V' \setminus \overline{V}_i$ are still present and needed in the reduced specification, since there might be predicates referring to their primed versions.

As an example, consider an effect schema with predicates $u' = v'$ and $v' = 5$ where $u \in V(\varphi)$. Since the value of u in the post-state is constrained by that of v, both predicates and variables are needed in the reduced specification. Consequently, v' is in the set of relevant variables due to a symmetric data dependence between predicate $u' = v'$ and $v' = 5$. However, the value of v in some state is only used

at points in time, where the associated event takes place. In particular, it is never referenced anywhere else and it thus cannot influence any different variable or event. Thus v would be in V', but not in any of the $\overline{V}_i$.

Using these sets of referenced variables, we then inductively construct an interpretation $\mathcal{I}'$ such that for all $i > 0$

$$\begin{aligned}
&\mathsf{TakesPlace}(\mathcal{I}, t_i) \cap E' = \varnothing \\
&\quad \Rightarrow \mathsf{TakesPlace}(\mathcal{I}', t_i) \cap E' = \varnothing \\
&\qquad \wedge\, \mathcal{I}'([t_{i-1}, t_{i+1}))|_{\overline{V}_i} = \mathcal{I}([t_{i-1}, t_{i+1}))|_{\overline{V}_i}
\end{aligned}$$

and

$$\begin{aligned}
&\mathsf{TakesPlace}(\mathcal{I}, t_i) \cap E' = \{e\} \\
&\quad \Rightarrow \mathsf{TakesPlace}(\mathcal{I}', t_i) \cap E' = \{e\} \\
&\qquad \wedge\, V_{mod} = \overline{V}_i \cap mod(e) \\
&\qquad\quad \Rightarrow \mathcal{I}'([t_{i-1}, t_i))|_{V_{mod}} = \mathcal{I}([t_{i-1}, t_i))|_{V_{mod}} \\
&\qquad\qquad \wedge\, \mathcal{I}'([t_i, t_{i+1}))|_{V_{mod}} = \mathcal{I}([t_i, t_{i+1}))|_{V_{mod}} \\
&\qquad \wedge\, V_{const} = \overline{V}_i \setminus mod(e) \\
&\qquad\quad \Rightarrow \mathcal{I}'([t_{i-1}, t_{i+1}))|_{V_{const}} = \mathcal{I}([t_{i-1}, t_{i+1}))|_{V_{const}}.
\end{aligned}$$

$\mathcal{I}'$ satisfies both conditions (1) and (2) of the projection definition. We now have to show that $\mathcal{I}'$ satisfies C'. This is done by induction over the length of the corresponding interpretations with the initial interval serving as induction base.

Induction base: Interval $[0, t_1)$
$\mathcal{I}$ satisfies C on the interval $[0, t_1)$, i.e., $\mathcal{I}$ satisfies the CSP part of C, C_{CSP}, on this interval, as well as its OZ part, C_{OZ}, and its DC part, C_{DC}.

- CSP part: $\mathcal{I}([0, t_1)) \models C_{CSP}$ implies $\mathcal{I}'([0, t_1)) \models C'_{CSP}$, since no event takes place on this interval.
- OZ part: $\mathcal{I}([0, t_1)) \models C_{OZ}$ implies $\mathcal{I}'([0, t_1)) \models C'_{OZ}$, since $\mathcal{I}'$ satisfies the `Init` schema of C_{OZ} and the `Init` schema of C'_{OZ} contains the same or fewer predicates than that of C_{OZ} and thus has less restrictions for the state space.
- DC part: $\mathcal{I}([0, t_1)) \models C_{DC}$ implies $\mathcal{I}'([0, t_1)) \models C'_{DC}$, since $\mathcal{I}'$ satisfies all formulae of C_{DC} and C'_{DC} contains the same or fewer formulae than C_{DC}.

Induction step: Interval $[0, t_i) \rightsquigarrow$ Interval $[0, t_{i+1})$
Since t_i is a point in time where $\mathcal{I}$ changes, there must be exactly one associated event $e \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$. We distinguish the following two cases for this event, depending on whether it belongs to the set E' of relevant events, or not:

- $e \notin E'$: In this case $\mathsf{TakesPlace}(\mathcal{I}', t_i) = \varnothing$ holds, i.e., no event takes place in $\mathcal{I}'$ at time t_i. This is in accordance with each part of C':
  - CSP part: Since $e \notin E'$, and C'_{CSP} is computed as the projection of C_{CSP} onto the set E' of relevant events, e does not appear in C'_{CSP}. Therefore, the omission of e in $\mathcal{I}'$ fits to C'_{CSP}.
  - OZ part: Since $\mathcal{I}'$ is in accordance with C'_{OZ} up to t_i and no event occurs at t_i, $\mathcal{I}'$ does not change at t_i and remains in accordance with C'_{OZ} on the subsequent interval.
  - DC part: Since $e \notin E'$, its occurrence and the modifications it imposes on the state space can not be constrained by any formula of C_{DC} and consequently not by any formula of C'_{DC}. Thus, the absence of e does not contradict C'_{DC}.
- $e \in E'$: In this case, the same event e takes place in $\mathcal{I}'$ at time t_i, i.e., $\mathsf{TakesPlace}(\mathcal{I}', t_i) = \{e\}$. This is in accordance with each part of C':
  - CSP part: Since C_{CSP} allows event e, and $e \in E'$ holds, also C'_{CSP} allows event e according to Lemma 6.1.4.
  - OZ part: We have to show that (1) e is enabled in $\mathcal{I}'$ at time t_i and that (2) the change of $\mathcal{I}'$ at time t_i fits to the modification that is induced by e on the state space.
    1. $\mathcal{I}'([t_{i-1}, t_i)) \models \texttt{enable\_}e'$: This follows directly from

       $$\mathcal{I}([t_{i-1}, t_i)) \models \texttt{enable\_}e,$$

       since $\texttt{enable\_}e'$ only mentions variables from $\overline{V}_i$ and $\mathcal{I}'([t_{i-1}, t_i))$ coincides with $\mathcal{I}([t_{i-1}, t_i))$ on variables from $\overline{V}_i$.
    2. $\mathcal{I}'([t_{i-1}, t_i)), \mathcal{I}'([t_i, t_{i+1})) \models \texttt{effect\_}e'$: This follows from

       $$\mathcal{I}([t_{i-1}, t_i)), \mathcal{I}([t_i, t_{i+1})) \models \texttt{effect\_}e,$$

       since $\texttt{effect\_}e'$ has the same or fewer predicates than $\texttt{effect\_}e$, i.e., it imposes the same or less restrictions on variables from $\overline{V}_i$, and $\mathcal{I}'([t_{i-1}, t_i))$ coincides with $\mathcal{I}([t_{i-1}, t_i))$ on variables from $\overline{V}_i$.
  - DC part: From $\mathcal{I}([0, t_{i+1})) \models C_{DC}$ we can infer that $\mathcal{I}'([0, t_{i+1})) \models C'_{DC}$ holds, since C'_{DC} contains the same or fewer restrictions than C_{DC} and $\mathcal{I}([0, t_{i+1}))$ coincides with $\mathcal{I}'([0, t_{i+1}))$ on all variables and events from $\overline{V}_i$ and E'.

Direction (2): $C' \rightsquigarrow C$

Let $\mathcal{I}'$ be an interpretation satisfying C' with $0 = t_0 < t_1 < t_2 < \ldots$ being the points in time where $\mathcal{I}'$ changes with $\mathsf{TakesPlace}(\mathcal{I}', t_i) \cap E' \neq \varnothing$. Associated with these points in time, we let again

$$\overline{V}_i = V_\varphi \cup \{ref(e) \cap V' \mid \exists\, t_j \geq t_i \colon e \in \mathsf{TakesPlace}(\mathcal{I}', t_j)\}$$

be the set of relevant variables referenced by a relevant event occurring within $\mathcal{I}'$ at t_i or later. Furthermore, we let $P'_i \equiv \mathsf{CSP}_{C'}(\mathcal{I}', t_i)$ represent the remaining CSP processes according to the operational semantics of the CSP part of C'.

We then inductively construct an interpretation $\mathcal{I}$ satisfying the original specification class C with

$$0 = t_0 < t_0^1 < t_0^2 < \ldots < t_0^{n_0} < t_1 < t_1^1 < t_1^2 < \ldots < t_1^{n_1} < t_2 < t_2^1 < \ldots$$

being the points in time where $\mathcal{I}$ changes,

$$\mathsf{TakesPlace}(\mathcal{I}, t_i) \cap E' = \mathsf{TakesPlace}(\mathcal{I}', t_i) \cap E' \text{ for all } i \geq 0,$$

and

$$\mathsf{TakesPlace}(\mathcal{I}, t_i^{j_i}) \cap E' = \varnothing \text{ for all } i \geq 0 \text{ and for all } 1 \leq j_i \leq n_i.$$

Associated with these points in time, we construct a sequence of CSP processes, consisting of $P_i \equiv \mathsf{CSP}_C(\mathcal{I}, t_i)$ and $P_i^j \equiv \mathsf{CSP}_C(\mathcal{I}, t_i^j)$, representing the remaining CSP processes according to the operational semantics of the CSP part of C.

As in the previous direction, the induction is carried out over the length of the corresponding interpretations with the initial point interval being the induction base:

Induction base: Point interval $[0, t_0]$
$\mathcal{I}'$ satisfies C' on the interval $[0, 0]$, i.e., $\mathcal{I}'$ satisfies the CSP part of C', C'_{CSP}, on this interval, as well as its OZ part, C'_{OZ}, and its DC part, C'_{DC}.

The same must hold for our newly constructed $\mathcal{I}$ with respect to C.

- CSP part: $\mathcal{I}'([0, t_0]) \models C'_{CSP}$.
  Out of the main process $\texttt{main} = P$ of C the slicing algorithm computes a reduced main process $\texttt{main} = P|_{E'}$ of C'. Since $P|_{E'}$ exists, we know that an associated process P exists, which can be constructed by applying the CSP process projection rules backwards.

- OZ and DC part: $\mathcal{I}'([0, t_0]) \models C'_{OZ}$ and $\mathcal{I}'([0, t_0]) \models C'_{DC}$.
  We know that $\mathcal{I}'([0, t_0])$ satisfies *Init*′ and all DC formulae from C'. *Init* has

at least all predicates that *Init*′ has, possibly plus some additional predicates that restrict some further variables which are not present in C'. Moreover, C contains the same DC formulae, possibly plus some additional ones that restrict some further variables or events which are not present in C'. Since no events occur at the initial point interval, only the initial variable valuations are of interest here. Therefore, we can choose $\mathcal{I}$ such that it coincides with $\mathcal{I}'([0,t_0])$ on all valuations of variables from C' and is modified in the remaining variables in order to satisfy the additional predicates from *Init* and the additional DC formulae from C.

Induction step: Interval $[0,t_i] \rightsquigarrow$ Interval $[0,t_{i+1}]$
Assume, we have constructed $\mathcal{I}$ up to some point in time t_i with $\mathcal{I}(t)|_{\overline{V}_i} = \mathcal{I}'(t)|_{\overline{V}_i}$ and an associated $P_i|_{E'} = P'_i$.

From $\mathsf{TakesPlace}(\mathcal{I}, t_{i+1}) \cap E' = \mathsf{TakesPlace}(\mathcal{I}', t_{i+1}) \cap E' \neq \varnothing$ we can derive that there is some event $e_{i+1} \in \mathsf{TakesPlace}(\mathcal{I}', t_{i+1})$. Therefore, we know that e_i is enabled in $\mathcal{I}([t_i, t_{i+1}))$ and its execution leads to a valuation $\mathcal{I}(t_{i+1})$. Furthermore, we know that `effect_`e in C has at least all predicates that `effect_`e in C' has. In consequence we have $\mathcal{I}(t_i)|_{\overline{V}_i} = \mathcal{I}'(t_i)|_{\overline{V}_i}$.

Nevertheless, e_{i+1} might not yet be enabled in P_i, but some intermediate irrelevant events $e_i^j \notin E'$ might be necessary to reach a P_i^j such that the relevant event e_{i+1} is enabled in P_i^j and leads to P_{i+1} with $P_{i+1}|_{E'} = P'_{i+1}$. We now have to show that these intermediate irrelevant events e_i^j are possible, that they do not change $\mathcal{I}$ on $\overline{V}_i$, such that the relevant event e_{i+1} is enabled in $\mathcal{I}$ over $[t_i, t_{i+1})$, that they lead to some P_i^j with $P_i^j \xrightarrow{e_{i+1}} P_{i+1}$ and $P_{i+1}|_{E'} = P'_{i+1}$, and that the additional DC formulae of C are satisfied.

We show this inductively by considering the structure of P_i from which the slicing algorithm computed P'_i.

1. $P_i \equiv e \rightarrow P$:
   - $e \in E'$: $P'_i \equiv e \rightarrow P|_{E'}$ (no intermediate steps are necessary)
   - $e \notin E'$: $P'_i \equiv P|_{E'}$
     In this case, intermediate steps are necessary to get from $\mathcal{I}(t_i)$ and P_i to $\mathcal{I}(t_{i+1})$ and P_{i+1}. We now have to show that
     a) these steps are possible, i.e., the associated events are enabled in $\mathcal{I}(t_i)$ and P_i. Suppose, one of the intermediate steps e_i^j is not enabled in $\mathcal{I}(t_i)$. This would be due to an unsatisfied predicate in its `enable` schema. According to Lemma 6.1.4 (first case) we would therefore have either a control dependence between e_i^j and e_{i+1} that leads us (according to the construction of the slice) to $e_i^j \in E'$ which is a contradiction.

The other possibility according to Lemma 6.1.4 (second case) is that e_i^j and e_{i+1} are located in different branches of the same interleaving node. In this case we do not need to make a transition in this blocked branch of the interleaving operator, but can safely proceed with a transition on the other branch.

b) these steps do not change variables in $\overline{V}_i$. This is a direct consequence of Lemma 6.1.5. Supposed, there would be some $e_i^j \notin E'$ that changes some $v \in \overline{V}_i$. Since $\overline{V}_i \subseteq V'$, there would again exist a data dependence between that $e_i^j \notin E'$ and some subsequent $e \in E'$ such that, due to the construction of the slice, e_i^j had to be in E'. (contradiction!)

c) these steps lead to some $\mathcal{I}(t_i^j)$ and P_i^j with $P_i^j \xrightarrow{e_{i+1}} P_{i+1}$ and $\mathcal{I}(t_i^j) \models \texttt{enable\_}e_{i+1}$ and $\mathcal{I}(t_i^j), \mathcal{I}(t_{i+1}) \models \texttt{effect\_}e_{i+1}$.

- $\mathcal{I}(t_i^j) \models \texttt{enable\_}e_{i+1}$: Since e_{i+1} is already enabled in $\mathcal{I}(t_i)$ and none of the e_i^j changes any $v \in V'$, e_{i+1} is still enabled in $\mathcal{I}(t_i^j)$. The `effect` schema is satisfied, since $\mathcal{I}(t_i), \mathcal{I}(t_{i+1}) \models \texttt{effect\_}e_{i+1}$ holds and $\mathcal{I}(t_i)|_{\overline{V}_i} = \mathcal{I}(t_i^j)|_{\overline{V}_i}$.
- P_i^j: Since e_i^j are the steps that are removed from P_i in order to get $P_i' \equiv P_i|_{E'}$ and e_{i+1} is enabled in P_i', the execution of e_i^j will lead to some P_i^j such that e_{i+1} is enabled as well.

2. $P_i \equiv P_{i,1} \,\square\, P_{i,2}$:
   - Either $\exists j \in \{1,2\} : P_{i,j} \xrightarrow{e} Q$ with $e \in E'$. Then $e_{i+1} \equiv e$ and $P'_{i+1} \equiv Q|_{E'}$. ✓
   - Otherwise $\exists j \in \{1,2\} : P_{i,j} \xrightarrow{e} Q$ with $e \notin E'$. Then e is one of the intermediate events and we have to start another case analysis for the intermediate process Q.
3. $P_i \equiv P_{i,1} \sqcap P_{i,2}$: analogous to the previous case. ✓
4. $P_i \equiv P_{i,1} ||| P_{i,2}$:
   - Either $\exists j \in \{1,2\} : P_{i,j} \xrightarrow{e} Q$ with $e \in E'$. Then $e_{i+1} \equiv e$ and $P'_{i+1} \equiv \begin{cases} Q|_{E'} \;|||\; P_{i,2}|_{E'} & \text{if } j = 1 \\ P_{i,1}|_{E'} \;|||\; Q|_{E'} & \text{else} \end{cases}$ ✓
   - Otherwise $\exists j \in \{1,2\} : P_{i,j} \xrightarrow{e} Q$ with $e \notin E'$. Then e is one of the intermediate events and we have to start another case analysis for the intermediate process $\begin{cases} Q ||| P_{i,2} & \text{if } j = 1 \\ P_{i,1} ||| Q & \text{else} \end{cases}$
5. $P_i \equiv P_{i,1} \underset{S}{\|} P_{i,2}$

- Asynchronous case, i.e., $\exists\, e \notin S\colon\ \exists j \in \{1,2\}\colon P_{i,j} \xrightarrow{e} Q$
  This is analogous to the previous case (interleaving). ✓
- Synchronous case, i.e., $\exists\, e \in S\colon P_{i,1} \xrightarrow{e} Q_1 \wedge P_{i,2} \xrightarrow{e} Q_2$
  - Either $e \in E'$: Then $e_{i+1} \equiv e$ and $P'_{i+1} \equiv Q_1|_{E'} \underset{S}{\parallel} Q_2|_{E'}$ ✓
  - Otherwise $e \notin E'$: Then e is one of the intermediate events and we have to start another case analysis for the intermediate process $Q_1 \underset{S}{\parallel} Q_2$.

Finally, we have to show that the insertion of e_i^j complies with the additional DC formulae of C. This, however, is a direct consequence of Lemma 6.1.6. Since, according to its first clause, there is no CFG path between irrelevant and relevant events, the additional DC formulae do not affect any relevant events $e_i \in E'$ or variables $v \in \overline{V}_i$, but only irrelevant events $e_i^j \notin E'$ or variables $v \in \overline{V}_i$. According to the second clause of Lemma 6.1.6, we are therefore free to choose the points in time t_i^j for the occurrence of each irrelevant event e_i^j in $\mathcal{I}$, such that the timing restrictions of the additional DC formulae are satisfied by $\mathcal{I}$.

Similarly, we are free to choose the valuations of irrelevant variables $v \notin \overline{V}_i$ in $\mathcal{I}$, such that the restrictions given in the additional DC formulae are satisfied by the full interpretation $\mathcal{I}$. □

6.3 Stuttering Invariance of Test Formulae

For describing real-time properties of CSP-OZ-DC classes we use the DC fragment of test formulae [Mey05, MFR06], which is a superset of counterexample formulae as they are used in the DC part of CSP-OZ-DC specifications. Satisfaction of test formulae can be evaluated on the set of interpretations defined by the semantics of a given CSP-OZ-DC class. Thus, a CSP-OZ-DC class satisfies a given test formula iff all interpretations within the semantics of the class satisfy the test formula.

The full syntax of test formulae is depicted in Figure 6.2, where $ev \in \mathit{Events}$ is an event and p a predicate over a set of variables V. As an example of test formulae consider again the following property of the air conditioner specification that we already introduced informally in Chapter 5 on slicing CSP-OZ-DC specifications:

$$\varphi \equiv \neg(\mathit{true} \frown \lceil \mathit{work} \wedge \mathit{fuel} < 5 \rceil \frown \mathit{true})$$

Property φ states an invariant over two variables of the air conditioner system, formulated as a counterexample: there should not be an interval in which the Boolean variable *work* is true, while simultaneously the fuel supply drops below some critical value. The counterexample is defined in terms of a single negated trace that starts and ends with a *true* phase, allowing the middle interval containing

TF	$::=$	F	basic formula
	$\mid$	$TF_1 \frown TF_2$	chop operator on the level of test formulae: divides the given interval into two parts where TF_1 holds on the first and TF_2 holds on the second part
	$\mid$	$TF_1 \wedge TF_2$	conjunction of test formulae
	$\mid$	$TF_1 \vee TF_2$	disjunction of test formulae
F	$::=$	Tr	trace
	$\mid$	$\neg F$	negation
	$\mid$	$F_1 \wedge F_2$	conjunction of basic formulae
Tr	$::=$	Ph	phase
	$\mid$	$\updownarrow ev$	event: ev occurs at the given point interval
	$\mid$	$\not\updownarrow ev$	not-event: ev does not occur within the given point interval
	$\mid$	$Tr_1 \frown Tr_2$	chop operator on the level of traces: divides the given interval into two parts where Tr_1 holds on the first and Tr_2 holds on the second part
Ph	$::=$	$\ell > 0 \wedge \ell \sim k$	time bound: the given interval has a length according to time bound $\ell \sim k$ with $k \in \mathbb{R}_{>0}$ and $\sim \in \{<, \leq, \geq, >\}$
	$\mid$	$Ph \wedge \lceil p \rceil$	phase with state invariant: predicate p holds almost everywhere on the given interval
	$\mid$	$Ph \wedge \boxminus ev$	forbidden event: the given interval contains no occurrence of event ev

Figure 6.2: Syntax of test formulae TF with $ev \in Events$ being an event and p being a predicate over a set of variables V. Additionally, a test formula must satisfy the side condition that its first element is a phase.

the invariant predicate to be located at an arbitrary point in time at or after system initialisation.

Whether a test formula holds for a given specification is evaluated with respect to interpretations defined over dense real-time. An interpretation satisfies a formula iff the formula holds on all intervals $[0, e]$, $e \in \mathbb{R}_{>0}$, or more formally:

Definition 6.3.1 (Satisfaction of Test Formulae)**.** *Let $\mathcal{I}$ be an interpretation with $0 = t_0 < t_1 < t_2 < \ldots$ being the points in time where $\mathcal{I}$ changes, and $[b, e]$ an interval over* Time*. The satisfaction of test formulae on intervals of $\mathcal{I}$ is then defined inductively as follows:*

$$
\begin{array}{ll}
\mathcal{I},[b,e] \models TF_1 \frown TF_2 & \textit{iff } \exists\, m, b \leq m \leq e : \mathcal{I},[b,m] \models TF_1 \wedge \mathcal{I},[m,e] \models TF_2 \\
\mathcal{I},[b,e] \models TF_1 \wedge TF_2 & \textit{iff } \mathcal{I},[b,e] \models TF_1 \textit{ and } \mathcal{I},[b,e] \models TF_2 \\
\mathcal{I},[b,e] \models TF_1 \vee TF_2 & \textit{iff } \mathcal{I},[b,e] \models TF_1 \textit{ or } \mathcal{I},[b,e] \models TF_2 \\
\mathcal{I},[b,e] \models \neg F & \textit{iff not } \mathcal{I},[b,e] \models F \\
\mathcal{I},[b,e] \models F_1 \wedge F_2 & \textit{iff } \mathcal{I},[b,e] \models F_1 \textit{ and } \mathcal{I},[b,e] \models F_2 \\
\mathcal{I},[b,e] \models \updownarrow ev & \textit{iff } b = e \wedge \exists\, i \in \mathbb{N}\colon t_i = b \wedge ev \in \mathsf{TakesPlace}(\mathcal{I}, t_i) \\
\mathcal{I},[b,e] \models \not\updownarrow ev & \textit{iff } b = e \wedge \forall\, i \in \mathbb{N}\colon t_i \neq b \vee ev \notin \mathsf{TakesPlace}(\mathcal{I}, t_i) \\
\mathcal{I},[b,e] \models Tr_1 \frown Tr_2 & \textit{iff } \exists\, m, b \leq m \leq e : [b,m] \models Tr_1 \wedge [m,e] \models Tr_2 \\
\mathcal{I},[b,e] \models \ell > 0 \wedge \ell \sim k & \textit{iff } e - b > 0 \wedge e - b \sim k \\
\mathcal{I},[b,e] \models Ph \wedge \lceil p \rceil & \textit{iff } e - b > 0 \wedge \mathcal{I},[b,e] \models Ph, \textit{and } \int_b^e P_{\mathcal{I}}(t)\,dt = e - b \\
 & \qquad \textit{with } P_{\mathcal{I}}(t) := \begin{cases} 1 & \mathcal{I}(t) \models p \\ 0 & \textit{otherwise} \end{cases} \\
\mathcal{I},[b,e] \models Ph \wedge \boxminus ev & \textit{iff } \mathcal{I},[b,e] \models Ph \\
 & \quad \textit{and } \forall\, i \in \mathbb{N}\colon t_i \in [b,e] \Rightarrow ev \notin \mathsf{TakesPlace}(\mathcal{I}, t_i)
\end{array}
$$

An interpretation $\mathcal{I}$ satisfies a test formula φ iff for all intervals $[0,t]$ starting at time zero, $\mathcal{I},[0,t] \models \varphi$ holds.

Based on this definition, we proceed next by showing stuttering invariance of test formulae via induction over their structure. To this end, we show that a given test formula can not distinguish between two interpretations, provided that one of them is in the projection of the other with respect to a set of events and variables that are mentioned within the test formula.

This notion of projection is the one that we introduced in the previous Section 6.2 and that our slicing approach guarantees to exist between interpretations of CSP-OZ-DC specifications and slices thereof with respect to a set of relevant events and variables, i.e., those that potentially affect the given test formula.

Theorem 6.3.2 (Stuttering invariance of test formulae)**.** *Let φ be a test formula over $O' \supseteq (V' \cup E')$, with V' variables and E' events of interest, let $\mathcal{I}$ and $\mathcal{I}'$ be E'-fair interpretations defining valuations for a set of variables $V \supseteq V'$ and a set of events $E \supseteq E'$. If $\mathcal{I}' \in \mathsf{Projection}_{O'}(\mathcal{I})$ then the following holds:*

$$\mathcal{I} \models \varphi \quad \textit{iff} \quad \mathcal{I}' \models \varphi\ .$$

Proof: A test formula is satisfied by an interpretation iff the formula holds on all intervals $[0,e]$ with $e \in \mathbb{R}_{>0}$. We proof more general that for all test formulae φ and all intervals $[b,e]$

$$\mathcal{I},[b,e] \models \varphi \quad \text{iff} \quad \mathcal{I}',[b,e] \models \varphi$$

holds according to Definition 6.3.1.

We do this by induction over the structure of φ. According to clause (2) of Definition 6.1.1 of the projection relation between interpretations, both interpretations $\mathcal{I}$ and $\mathcal{I}'$ need to agree on the occurrence of relevant events. Let therefore $0 = t_0 < t_1 < t_2 < \ldots$ be the points in time where both $\mathcal{I}$ and $\mathcal{I}'$ change, i.e., the points in time where relevant events $ev \in E'$ take place. Note that there might be further points in time where any of the involved interpretations change, but according to the definition of projection not with relevant events taking place.

Induction Base:
The induction starts with the most basic building blocks of test formulae, which are the following:

Phases with time bound: $\varphi \Leftrightarrow \ell > 0 \wedge l \sim k$
Starting with the definition of satisfaction of phase expressions, we obtain

$$\begin{aligned} & \mathcal{I}, [b,e] \models \ell > 0 \wedge \ell \sim k \\ \text{iff} \quad & b < e \wedge e - b \sim k \\ \text{iff} \quad & \mathcal{I}', [b,e] \models \ell > 0 \wedge \ell \sim k. \quad \checkmark \end{aligned}$$

The equivalence is obvious, since the definition is not based on the interpretation, but only on the actual interval lengths.

Events: $\varphi \Leftrightarrow \updownarrow ev$ with $ev \in E'$
Starting with the definition of satisfaction of event expressions, we obtain

$$\begin{aligned} & \mathcal{I}, [b,e] \models \updownarrow ev \\ \text{iff} \quad & b = e \wedge \exists i \in \mathbb{N}\colon t_i = b \wedge ev \in \mathsf{TakesPlace}(\mathcal{I}, t_i) \\ \text{iff} \quad & b = e \wedge \exists i \in \mathbb{N}\colon t_i = b \wedge ev \in \mathsf{TakesPlace}(\mathcal{I}', t_i) \\ \text{iff} \quad & \mathcal{I}', [b,e] \models \updownarrow ev. \quad \checkmark \end{aligned}$$

The intermediate equivalence holds due to the second clause of Definition 6.1.1 of the projection relation between $\mathcal{I}$ and $\mathcal{I}'$: Since ev is mentioned in φ, it is element of $E' \subseteq O'$ and thus

$$ev \in \mathsf{TakesPlace}(\mathcal{I}, t_i) \Leftrightarrow ev \in \mathsf{TakesPlace}(\mathcal{I}', t_i).$$

Not-Events: $\varphi \Leftrightarrow \not\updownarrow ev$ with $ev \in E'$
Starting with the definition of satisfaction of not-event expressions, we obtain

$$\begin{aligned} & \mathcal{I}, [b,e] \models \not\updownarrow ev \\ \text{iff} \quad & b = e \wedge \forall i \in \mathbb{N}\colon t_i \neq b \vee ev \notin \mathsf{TakesPlace}(\mathcal{I}, t_i) \\ \text{iff} \quad & b = e \wedge \forall i \in \mathbb{N}\colon t_i \neq b \vee ev \notin \mathsf{TakesPlace}(\mathcal{I}', t_i) \\ \text{iff} \quad & \mathcal{I}, [b,e] \models \not\updownarrow ev. \quad \checkmark \end{aligned}$$

The argument for the validity of the intermediate equivalence is analogous to the previous case: The projection relation between $\mathcal{I}$ and $\mathcal{I}'$ with respect to O' and $ev \in E' \subseteq O'$ (due to *ev* being mentioned within the formula) leads us to the conclusion that either *ev* takes place in both interpretations at the same given point in time or it does not take place at this point in time in any of both interpretations.

Induction Step:
Suppose, the lemma is proven for phases *Ph*, for traces Tr_1 and Tr_2, for basic formulae *F*, F_1, and F_2, and for test formulae TF_1, TF_2. The remaining proofs for chop ($TF_1 \frown TF_2$ and $Tr_1 \frown Tr_2$), for negation ($\neg F$), for conjunction ($TF_1 \wedge TF_2$ and $F_1 \wedge F_2$), for disjunction ($TF_1 \vee TF_2$), and for phases with forbidden events ($Ph \wedge \boxminus ev$) are straightforward and follow the same pattern, relying on both parts of Definition 6.3.1 and the existence of the projection relation between the involved interpretations; therefore, we only cover the following example of these cases:

Phase with state invariant: $\varphi \Leftrightarrow Ph \wedge \lceil p \rceil$ with *p* being a predicate over a set V' of variables with $V' \subseteq O'$ and $V' \cap E' = \varnothing$.
Starting with the definition of satisfaction of phases with state invariants, we obtain

$$
\begin{aligned}
& \mathcal{I}, [b,e] \models Ph \wedge \lceil p \rceil \\
\text{iff} \quad & e - b > 0 \wedge \mathcal{I}, [b,e] \models Ph \text{ and } \int_b^e P_{\mathcal{I}}(t)\, dt = \ell \\
& \qquad \text{with } P_{\mathcal{I}}(t) := \begin{cases} 1 & \mathcal{I}(t) \models p \\ 0 & \textit{otherwise} \end{cases} \\
\text{iff} \quad & e - b > 0 \wedge \mathcal{I}', [b,e] \models Ph \text{ and } \int_b^e P_{\mathcal{I}'}(t)\, dt = \ell \\
& \qquad \text{with } P_{\mathcal{I}'}(t) := \begin{cases} 1 & \mathcal{I}'(t) \models p \\ 0 & \textit{otherwise} \end{cases} \\
\text{iff} \quad & \mathcal{I}', [b,e] \models Ph \wedge \lceil p \rceil. \quad \checkmark
\end{aligned}
$$

For the validity of the intermediate equivalence, we apply again the definition of the projection relation between $\mathcal{I}$ and $\mathcal{I}'$: Its first clause requires $\forall t\colon \mathcal{I}(t)|_{O' \setminus E'} = \mathcal{I}'(t)|_{O' \setminus E'}$. Since all variables that appear within predicate *p* are part of O', but not part of E', both interpretations need to agree on them, such that *p* either holds almost everywhere within the given interval on both interpretations or on none of them. □

In conjunction with the result of the previous Section 6.2, namely the existence of the projection relation between a specification and its slice, the stuttering (viz.

projection) invariance of test formulae then directly yields the intended result of slicing correctness: Our slicing approach for the verification of CSP-OZ-DC specifications with respect to test formulae is correct, i.e., a test formula holds on a specification if and only if it holds on the slice of the specification with respect to the given test formula.

6.4 Stuttering Invariance of State/Event Interval Logic

This section examines formulae of the state/event interval logic (SE-IL), the logic that we use to specify temporal properties for Object-Z and CSP-OZ specifications. In Chapter 3 we defined the semantics of Object-Z and CSP-OZ specifications in terms of event-labelled Kripke structures (LKS), thus SE-IL formulae are interpreted over paths of alternating states and events of LKS.

Untimed projections of interpretations, which we defined for evaluating the CSP and the Object-Z part of CSP-OZ-DC specifications, can be regarded as such paths. Therefore, the results of the previous Section 6.2 carry over to the discrete setting of alternating paths of state and events defined over labelled Kripke structures.

Thus we obtain the existence of the projection relation between Object-Z or CSP-OZ specifications and slices thereof along the same lines as for CSP-OZ-DC specifications and only need to show stuttering invariance for the temporal logic that we use in the untimed setting, namely SE-IL.

6.4.1 State/Event Interval Logic

The state/event interval logic SE-IL that we use for expressing temporal properties of Object-Z and CSP-OZ specifications is inspired by the Duration Calculus (DC), and allows us to reason about events and states but (for suiting our purposes) not about time. It can therefore be regarded as an untimed projection of DC.

There are two reasons for considering this logic as a notation for properties of Object-Z and CSP-OZ specifications: first of all, our ultimate goal has been to apply slicing to integrated specifications, which in addition to Object-Z contain parts specifying the dynamic behaviour (in CSP) and timing constraints (in DC). The logic for expressing properties of this type of specifications is a subset of the full DC, namely test formulae, so SE-IL seems to be an adequate solution when staying in the untimed setting but still remaining compatible to the logic used in the timed setting.

As a second reason, we are interested in a logic which can precisely express orderings between events and state propositions (e.g., like “when event e happens then immediately afterwards variable x has the value 5”). However, since we are interested in reducing the specification, it should, on the other hand, not be able to precisely speak about *steps* of the system (e.g., like “the 10th operation of the system is event e”). The paths of the reduced specification will be projections of

$$
\begin{array}{rcll}
\varphi & ::= & \lceil p \rceil & \text{– phase } (p \text{ holds in all states of the given interval}) \\
& | & ev & \text{– event } (ev \text{ occurs in the given interval}) \\
& | & \neg\varphi & \text{– negation} \\
& | & \varphi \wedge \psi & \text{– conjunction} \\
& | & \Diamond^{L}\varphi & \text{– eventually operator with liveness} \\
& & & (\varphi \text{ holds inside or beyond the given interval}) \\
& | & \varphi \frown \psi & \text{– chop operator (divides the given interval into two} \\
& & & \text{parts where } \varphi \text{ holds on the first and } \psi \text{ holds on the} \\
& & & \text{second part)}
\end{array}
$$

Figure 6.3: Syntax of state/event interval logic (SE-IL) formulae

the paths of the full specification (omitting some irrelevant events), and thus a preservation of properties under slicing does only make sense for logics which do not talk about particular steps.

When using (state-based) LTL for expressing properties, this phenomenon is taken into account by restricting formulae to the next-free part of the logic (e.g., [HDZ00]). The bottom line of this observation is that any logic used for slicing has to be invariant under stuttering, where the notion of projection that we use is a particular kind of stuttering.

The grammar depicted in Figure 6.3 describes formulae of the state/event interval logic SE-IL (where $ev \in E$ is an event and $p \in AP$ an atomic proposition).

We use the abbreviation $\Box^{L}\varphi$ to stand for $\neg\Diamond^{L}\neg\varphi$. For a formula φ we let E_φ denote the set of events occurring in it and V_φ the set of variables of atomic propositions in it. The chop operator requires that there is some position in the underlying path (or a section of a path) such that before this position the formula on its left hand side holds and after it the formulae on its right hand side.

In order to define when a Kripke structure satisfies an interval logic formula, we first define satisfaction on paths. Duration Calculus is used to reason about continuous time models, and the validity of formulae is defined via quantification over all time intervals: a formula holds iff it is true in all intervals (starting at time 0). This definition is now transferred to the discrete setting of paths:

Definition 6.4.1 (Satisfaction of SE-IL formulae)**.** *A path satisfies a formula iff the formula holds on all intervals* $[0, e]$*,* $e \in \mathbb{N}$*. Let* $\pi = s_0 e_1 s_2 e_3 s_4 \ldots$ *be a path and* $\pi[i]$ *the i-th component of* π*:* $\pi[i]$ *can either be an event or a state.*

1. $\pi, [b, e] \models \lceil p \rceil$ *iff* $\exists\, m, b \leq m \leq e : \pi[m] \in S$
   and $\forall\, m, b \leq m \leq e : \pi[m] \in S \Rightarrow p \in L(\pi[m])$*,*

2. $\pi, [b, e] \models ev$ *iff* $b = e$ *and* $\pi[b] = ev$*,*

3. $\pi, [b,e] \models \neg\varphi$ *iff not* $\pi, [b,e] \models \varphi$,

4. $\pi, [b,e] \models \varphi \wedge \psi$ *iff* $\pi, [b,e] \models \varphi$ *and* $\pi, [b,e] \models \psi$,

5. $\pi, [b,e] \models \diamondsuit\varphi$ *iff* $\exists\, m_1, m_2 \geq b : \pi, [m_1, m_2] \models \varphi$,

6. $\pi, [b,e] \models \varphi \frown \psi$ *iff* $(\exists\, m, b \leq m \leq e : \pi, [b,m] \models \varphi$ *and* $\pi, [m,e] \models \psi)$
$\vee\ (\pi[b] \in S$ *and* $\pi[b, b-1] \models \varphi$ *and* $\pi, [b,e] \models \psi)$
$\vee\ (\pi[e] \in S$ *and* $\pi[e, e-1] \models \psi$ *and* $\pi, [b,e] \models \varphi)$

Some explanations for this unusual definition are at place.

- Item 1: the decision taken here is that during execution of an event we do not know what atomic propositions hold, thus the formula $\lceil p \rceil$ evaluates to false on an interval with an event only. This reflects the fact that events may invalidate atomic propositions, which hold in the state before their execution and make others become true in the state after their execution. In order to be able to say that an event causes a state change, we can neither assume that atomic propositions in pre-states still hold while the event takes place nor that those of post-states already hold.

- Item 2, 3 and 4 should be as expected.

- Item 5: The eventually operator has to reason about positions outside the current interval, since we want to achieve real liveness, not just bounded liveness. This operator is taken from the DC with liveness [Ska94]; the standard DC does not allow to reason about unbounded liveness.

- Item 6: The first part of the disjunction captures the case where the interval is divided into two parts, such that φ holds on the first part and ψ on the second. The second and third part of the disjunction mimic the phenomenon that in continuous time one can chop off an empty interval from every interval. The empty interval is denoted by $[b, b-1]$ (or $[e, e-1]$). In an empty interval neither $\lceil p \rceil$ nor *ev* holds.

Note that for instance $ev \frown ev \equiv ev$ but $\neg\lceil p \rceil \not\equiv \lceil \neg p \rceil$. In the former case the formula *ev* only holds on a zero interval $[b,e]$ with $b = e$ and the chop operator can divide this interval into two zero intervals that both satisfy *ev*. In the latter case, from the fact that $\lceil p \rceil$ does not hold on an interval one cannot conclude that $\lceil \neg p \rceil$ holds on this interval.

A Kripke structure then satisfies a formula if all of its paths do so. Likewise, an Object-Z or a CSP-OZ class satisfies a property when its Kripke structure does.

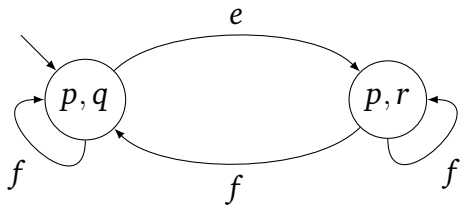


Figure 6.4: Exemplary labelled Kripke structure K

Definition 6.4.2 (Satisfaction of formulae on labelled Kripke structures)**.** *Let $K = (S, S_0, \rightarrow, L)$ be a Kripke structure and φ an SE-IL formula. A path π satisfies φ if $\pi, [0, e] \models \varphi$ holds for all $e \in \mathbb{N}$. K* satisfies *φ ($K \models \varphi$) iff $\pi \models \varphi$ holds for all paths of K. K* fairly satisfies *φ with respect to a set of events $E' \subseteq E$ ($K \models_{E'} \varphi$) iff $\pi \models \varphi$ holds for all E'-fair paths of K.*

As an example consider the Kripke structure K depicted in Figure 6.4. For K we for instance have $K \models \boxdot_L p$ (p always holds), $K \models \neg\diamondsuit_L(e \frown \lceil\neg r\rceil)$ (r holds after e, formulated as a counterexample: there is no interval in which $\neg r$ holds immediately after e) but $K \not\models \diamondsuit_L e$ (event e will eventually happen is not true, since there are paths with event f only) and $K \not\models \boxdot_L \lceil q\rceil$.

As a further example of SE-IL formulae we consider again the following two properties of class *TicTacToe* that we already introduced informally in Chapter 5 on slicing Object-Z specifications:

$$\varphi_1 \equiv \boxdot_L \lceil moves = 9 - \#free\rceil$$
$$\varphi_2 \equiv \neg\diamondsuit_L(black \frown (\lceil true\rceil \wedge \neg(\lceil true\rceil \frown white \frown \lceil true\rceil)) \frown black)$$
$$\wedge\ \neg\diamondsuit_L(white \frown (\lceil true\rceil \wedge \neg(\lceil true\rceil \frown black \frown \lceil true\rceil)) \frown white)$$

Property φ_1 states an invariant between two variables of the class and φ_2 states that moves are taken in turn. The second property is again formulated as a counterexample: there should not be an interval in which an event *black* is followed by a nonempty interval in which no *white* happens, which is then followed by another *black* (and similar for *white*). Non-emptiness of the middle interval is achieved by conjunction with $\lceil true\rceil$.

6.4.2 Projection of Event-Labelled Kripke Structures

For the discrete setting of SE-IL formulae, defined over paths of event-labelled Kripke structures, the notion of projection of interpretations that we introduced in the beginning of this chapter can be reduced to an untimed notion of projection.

Therefore, the projection relation is now redefined on paths instead of interpretations. The intuition remains the same as for projections of interpretations: When

	Block 0		Block 1		Block 2		
π =	$\underbrace{s_0\ e_0\ s_1}$	e_1	$\underbrace{s_2}$	e_2	$\underbrace{s_3\ e_3\ s_4}$	e_4	...
π' =	$\underbrace{r_0}_{= s_0\|_{AP'} = s_1\|_{AP'}}$	e_1	$\underbrace{r_1}_{= s_2\|_{AP'}}$	e_2	$\underbrace{r_2}_{= s_3\|_{AP'} = s_4\|_{AP'}}$	e_4	...

Figure 6.5: Exemplary path π and a corresponding path $\pi' \in Pr_{AP',E'}(\pi)$ in its projection with respect to a set of relevant atomic propositions AP' and events $E' \supseteq \{e_1, e_2, e_4\}$.

computing the projection of a given path onto a set of atomic propositions and a set of events, one divides the path into blocks, such that all states inside a block are "projection-equivalent" (i.e., they coincide on the given set of atomic propositions) and all events inside a block are "irrelevant" events (i.e., events not from the given set of events), except for the last event in the block, which is a "relevant" event (i.e., an event from the given set of events). The projection of the original path contains then any path such that for each of the blocks of the original path all states and irrelevant events are mapped onto one single state of the new path, while the "relevant" event remains in the new path as illustrated in the sketch of a projection of a path depicted in Figure 6.5.

More formally, the notion of projection of labelled Kripke structure paths is defined as follows:

Definition 6.4.3 (Projection of labelled Kripke structure paths)**.** *Let the sequence $\pi = s_0e_0s_1e_1s_2e_2s_3\ldots$ be an E'-fair path over a set of atomic propositions AP and a set of events $E \supseteq E'$. The* projection of π onto a set of atomic propositions AP' and a set of events E' ($Pr_{AP',E'}(\pi)$) *contains any E'-fair path $\rho = r_0f_0r_1f_1r_2f_2r_3\ldots$ such that there is a sequence of indices $0 = i_0 < i_1 < i_2 < \ldots$ (that divides π into blocks) with*

- $\forall\, k \geq 0$: $L(s_{i_k}) \cap AP' = L(s_{i_k+1}) \cap AP' = \cdots = L(s_{i_{k+1}-1}) \cap AP' = L(r_k) \cap AP'$
  (relevant atomic propositions do not change within a block and are the same in the corresponding state of ρ),
- $\forall\, l \in \mathbb{N}, \forall\, k : i_l \leq k < i_{l+1} - 1 : e_k \in E \setminus E'$
  (no relevant events occur inside a block),
- $\forall\, l \geq 1 : e_{i_l-1} = f_{l-1} \in E'$
  (transitions between blocks are labelled with the same relevant event as the corresponding transition of ρ).

For comparing the Kripke structures we restrict the definition to *fair* paths, since we are only considering satisfaction of formulae on fair paths.

Definition 6.4.4. *Let $K_i = (S_i, S_{0,i}, \rightarrow_i, L_i)$, $i \in \{1, 2\}$, be labelled Kripke structures over a set of atomic propositions AP and a set of events E, $AP' \subseteq AP$ a subset of the atomic propositions and $E' \subseteq E$ a subset of the events.*

K_2 is in the projection *of K_1 onto AP' and E' ($K_2 \in Pr_{AP',E'}(K_1)$) iff the following holds:*

1. *For each E'-fair path π in K_1 there exists an E'-fair path π' in K_2 such that $\pi' \in Pr_{AP',E'}(\pi)$,*
2. *and vice versa, for each E'-fair path π' in K_2 there exists an E'-fair path π in K_1 such that $\pi' \in Pr_{AP',E'}(\pi)$.*

Such a projection relation between two Kripke structures guarantees that formulae which only mention propositions from AP' and events from E' hold for either both or none of the Kripke structures.

Theorem 6.4.5 (Stuttering invariance of state/event interval logic formulae)**.** *Let φ be an SE-IL formula over AP' and E', and K_1, K_2 labelled Kripke structures over a set of atomic propositions AP and a set of events E with $AP' \subseteq AP$ and $E' \subseteq E$. If $K_2 \in Pr_{AP',E'}(K_1)$ then the following holds:*

$$K_1 \models_{E'} \varphi \quad \textit{iff} \quad K_2 \models_{E'} \varphi \,.$$

The notion of projection implies a correspondence between elements of a path and elements of paths from its projection that we define as follows.

Definition 6.4.6. *Let $\pi_1 = s_0 e_0 s_1 e_1 \ldots$ and $\pi_2 = r_0 f_0 r_1 f_1 \ldots$ be E'-fair paths with $\pi_2 \in Pr_{AP',E'}(\pi_1)$ and $0 = i_0 < i_1 < i_2 < \ldots$ the associated sequence of indices as in Definition 6.4.3.*

Then any path component s_i or e_i of π_1 is projected *onto exactly one path component of π_2 and any path component r_i or f_i of π_2 is* back-projected *onto a set of path components of π_1 as follows.*

Any state s_m with $i_k \leq m < i_{k+1}$ and any event e_n with $i_k \leq n < i_{k+1} - 1$ are projected onto the state r_k.

Any event e_m with $m = i_{k+1} - 1$ is projected onto the event f_k.

Any state r_m is back-projected onto the set of states $\{s_n \mid i_m \leq n < i_{m+1}\}$.

Any event f_m is back-projected onto the event e_n with $n = i_{m+1} - 1$.

Note that sub-paths consisting of projection-equivalent states and projection-irrelevant events are projected onto one single state of the projection, while projection-relevant events remain in the path.

To prove Theorem 6.4.5 we will apply the following lemma.

Lemma 6.4.7 (Corresponding sections of projection paths)**.** *Let φ be a state/event interval logic formula over AP′ and E′ and let $\pi_1 = s_0 e_0 s_1 e_1 \ldots$ and $\pi_2 = r_0 f_0 r_1 f_1 \ldots$ be E′-fair paths with $\pi_2 \in Pr_{AP',E'}(\pi_1)$. Then the following holds:*

$$\forall\, b_1, e_1\, \exists\, b_2, e_2 \colon\ (\pi_1, [b_1, e_1] \models \varphi \Leftrightarrow \pi_2, [b_2, e_2] \models \varphi)$$

Proof: For every b_1, e_1 with $\pi_1, [b_1, e_1] \models \varphi$ we choose some corresponding b_2, e_2 such that $\pi_2, [b_2, e_2] \models \varphi$ and vice versa. We do this inductively over the structure of φ.

Induction Base: The most basic elements of SE-IL formulae are phases ($\varphi \equiv \lceil p \rceil$) and events ($\varphi \equiv ev$), which form the induction base.

Direction $\pi_1, [b_1, e_1] \models \varphi \Rightarrow \pi_2, [b_2, e_2] \models \varphi$**:**
According to Definition 6.4.6 there are indices m and n such that $\pi_1[b_1]$ is projected onto $\pi_2[m]$ and $\pi_1[e_1]$ is projected onto $\pi_2[n]$. Choose $b_2 = m$ and

$$e_2 = \begin{cases} b_2 - 1 & \text{if } e_1 = b_1 - 1 \qquad \text{(empty interval)} \\ & \text{or } e_1 = b_1 \wedge \pi_1[b_1] \in E \setminus E' \qquad \text{(zero interval with one event)} \\ n & \text{otherwise} \end{cases}$$

and hence $\pi_2, [b_2, e_2] \models \varphi$ holds.

Direction $\pi_1, [b_1, e_1] \models \varphi \Leftarrow \pi_2, [b_2, e_2] \models \varphi$**:**
According to Definition 6.4.6 there are sets of indices $A = \{m_1, \ldots, m_p\}$ and $B = \{n_1, \ldots, n_q\}$ such that $\pi_2[b_2]$ is back-projected onto $\{\pi_1[m] \mid m \in A\}$ and $\pi_2[e_2]$ is back-projected onto $\{\pi_1[n] \mid n \in B\}$. Note that A and B might overlap if b_2 and e_2 both belong to the same projection-block. Choose $b_1 \in A$ and

$$e_1 = \begin{cases} b_1 - 1 & \text{if } e_2 = b_2 - 1 \qquad \text{(empty interval)} \\ n \in B \wedge n \geq b_1 & \text{otherwise} \end{cases}$$

and hence $\pi_1, [b_1, e_1] \models \varphi$ holds.

Induction Step: Suppose, the lemma is proven for $\varphi \equiv \psi$ and $\varphi \equiv \chi$. Since for negation ($\varphi \equiv \neg\psi$), conjunction ($\varphi \equiv \psi \wedge \chi$) and eventually operator ($\varphi \equiv \Diamond\psi$) the proof is straightforward, we omit these cases and show only how to deal with the chop operator ($\varphi \equiv \psi \frown \chi$):
Due to the induction hypothesis the following holds:

$$\forall\, b_1, m_1^1\, \exists\, b_2, m_2^1 \colon\ (\pi_1, [b_1, m_1^1] \models \psi \Leftrightarrow \pi_2, [b_2, m_2^1] \models \psi)$$

and

$$\forall\, m_1^2, e_1\, \exists\, m_2^2, e_2 \colon\ (\pi_1, [m_1^2, e_1] \models \chi \Leftrightarrow \pi_2, [m_2^2, e_2] \models \chi)\ .$$

Direction $\pi_1, [b_1, e_1] \models \varphi \Rightarrow \pi_2, [b_2, e_2] \models \varphi$**:**
In order to show the lemma for $\varphi \equiv \psi \frown \chi$ we assume that $\pi_1, [b_1, e_1] \models \psi \frown \chi$ holds and show that there is a corresponding interval $[b_2, e_2]$ on π_2, such that $\pi_2, [b_2, e_2] \models \psi \frown \chi$ holds. According to the definition of the chop operator we have to distinguish the following three cases:

1. $\exists\, m_1, b_1 \leq m_1 \leq e_1$:
   (1) $\pi_1, [b_1, m_1^1] \models \psi$ and (2) $\pi_1, [m_1^2, e_1] \models \chi$ and (3) $m_1^1 = m_1^2 = m_1$:
   From clauses (1) and (2) and the induction hypothesis we can directly derive that there are m_2^1 and m_2^2 such that $\pi_2, [b_2, m_2^1] \models \psi$ and $\pi_2, [m_2^2, e_2] \models \chi$. So we only have to show that there is an m_2, such that $m_2^1 = m_2^2 = m_2$ holds. Since $m_1^1 = m_1^2$ holds, the only potential for a conflict between m_2^1 and m_2^2 is caused by the choice of m_2^1 in the induction base when $m_1^1 = b_1 - 1$ or if $m_1^1 = b_1 \wedge \pi_1[b_1] \in E \setminus E'$. In these cases we map the original pair of intervals onto a pair of one empty and another interval such that $\pi_2, [b_2, e_2] \models \psi \frown \chi$ holds again according to the second clause of the definition of the chop operator.

2. $\pi_1[b_1] \in S$ and $\pi_1, [b_1, b_1 - 1] \models \psi$ and $\pi_1, [b_1, e_1] \models \chi$:
   $\pi_1[b_1] \in S$ implies that also $\pi_2[b_2] \in S$ holds, since states are projected onto states. From the second and third clause we can—based on the induction hypothesis—directly derive that $\pi_2, [b_2, b_2 - 1] \models \psi$ and $\pi_2, [b_2, e_2] \models \chi$ hold, i.e., the second clause of the definition of the chop operator.

3. $\pi_1[e_1] \in S$ and $\pi_1, [b_1, e_1] \models \psi$ and $\pi_1, [e_1, e_1 - 1] \models \chi$:
   This case is completely symmetric to the previous one.

Direction $\pi_1, [b_1, e_1] \models \varphi \Leftarrow \pi_2, [b_2, e_2] \models \varphi$**:**
We omit the proof of the reverse implication, since it is mostly symmetric to the opposite direction, except for the slightly different choice of the right interval limit in the induction base (which even has the positive effect to slightly facilitate the argumentation in this direction) and the application of the back-projection instead of the projection. □

The stuttering invariance of SE-IL formulae that we have claimed in Theorem 6.4.5 is then a direct consequence of the Definition 6.4.4 of the presumed projection relation between labelled Kripke structures in conjunction with Lemma 6.4.7.

Together with the result of the previous Section 6.2, namely the existence of the projection relation between a specification and its slice (that carries over to the

untimed version of projection that we defined in this section), the stuttering (viz. projection) invariance of SE-IL formulae then yields the concluding correctness result: our slicing approach for the verification of Object-Z and CSP-OZ specifications with respect to SE-IL formulae is correct, i.e., an SE-IL formula holds on a specification if and only if it holds on the slice of the specification with respect to the given SE-IL formula.

7 Tool Support and Experimental Evaluation

Contents

In the previous chapters we have developed a slicing approach for CSP-OZ-DC specifications. Our main motivation for computing slices of such integrated formal specifications has been to mitigate the problem of state space explosion during their automatic verification. In order to evaluate the effectiveness of slicing with respect to this purpose, the slicing approach has been implemented and several experiments with suitable case studies have been carried out.

Basis for our slicing implementation and all slicing experiments is *Syspect*, the *graphical modelling environment for CSP-OZ-DC* specifications. The first Section 7.1 of this chapter introduces Syspect with details of the types of diagrams and some of its further features, in particular those with respect to automatic verification.

The following Section 7.2 introduces the actual *slicing plug-in* of Syspect, which is the straightforward realisation of the slicing approach that has been devised in the previous chapters of this thesis.

This slicing plug-in is the key for carrying out experiments in order to evaluate the effectiveness of slicing. Such experiments have been conducted with a number of specifications that have been modelled within Syspect, ranging from the running examples presented in previous chapters up to a larger specification motivated by an industrially relevant context.

The concluding Section 7.3 documents these specifications together with the verification properties that have been used as slicing criteria; furthermore it gives an account on the *experimental results* that have been achieved by applying the slicing plug-in during specification export and automatic verification.

7.1 Syspect — Modelling Environment for CSP-OZ-DC

The platform at the heart of all slicing experiments is Syspect, the graphical modelling environment for CSP-OZ-DC specifications. The diagram depicted in Figure 7.1 shows Syspect together with the associated chain of verification tools that have been developed in the context of subproject R1 “Beyond Timed Automata” of the DFG SFB/TR 14 AVACS [AVA07].

Syspect is the result of a student project of the same name [Sys06]. The project has been carried out at the Correct System Design Group [Cor07a] of the University of Oldenburg during a period of two semesters from October 2005 through September 2006. A team of eleven students participated in the Syspect project group, which was supervised by Ernst-Rüdiger Olderog, Michael Möller and Andreas Schäfer. The overall goal of the project was to assemble a graphical specification environment for CSP-OZ-DC specifications. This goal has successfully been achieved by implementing Syspect as a rich client platform (RCP) based on the integrated development environment (IDE) Eclipse [Ecl07].

The resulting software application allows us to generate CSP-OZ-DC specifications by using a suitable UML profile for CSP-OZ [MORW07] that has previously been researched within the DFG project *ForMooS* [Cor07b]. For developing models that correspond to CSP-OZ-DC specifications, the CSP-OZ UML profile is extended with suitable annotations for representing DC counterexample formulae.

The CSP-OZ-DC UML profile utilises a number of UML diagrams that will be introduced in the following sections. These diagrams are augmented by several dedicated annotations in order to allow the graphical definition of system specifications

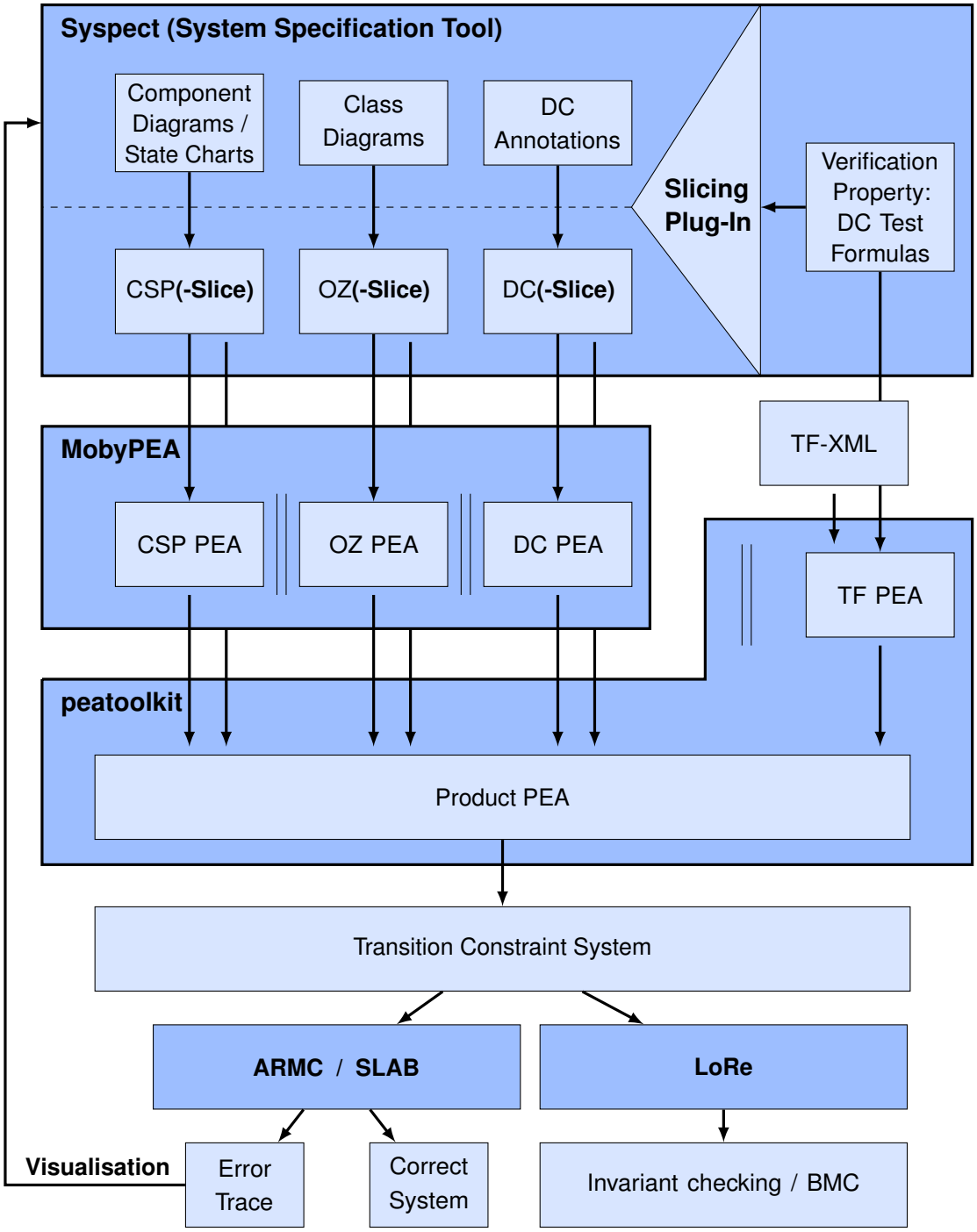


Figure 7.1: The Syspect modelling environment for CSP-OZ-DC and the associated tool chain, developed in DFG SFB/TR 14 AVACS, subproject R1 “Beyond Timed Automata”.

corresponding to a consistent subclass of CSP-OZ-DC.

Once a specification has been modelled within Syspect, several export facilities are offered, as also sketched within Figure 7.1. The associated export plug-ins allow the fully automatic export of specifications into *CSP-OZ-DC LaTeX mark-up* as well as into its semantical representation of *phase event automata* (PEA). To a large extent, the PEA representation generated by Syspect can be imported by Moby/PEA [Cor07c], a graphical modelling tool for PEA, which has been developed by Johannes Faber within the AVACS R1. A small but important limitation of this import facility are Object-Z expressions, which cannot be imported into Moby/PEA at the moment due to an incompatibility with the CZT XML format.

Furthermore, Syspect also allows for the direct translation of models together with associated verification properties represented by test formulae into the format of *transition constraint systems* (TCS), as defined by Hoenicke and Maier [HM05a]. For automatic verification, the resulting TCS can finally be fed into the model checkers ARMC [Ryb07] and SLAB [BDFW07], which have both been developed within AVACS R1. Internally, this translation is carried out via an intermediate representation of the model and its associated verification properties as corresponding PEA. Subsequently, the product automaton defined by parallel composition of all these comprised PEAs is computed. Finally, the resulting product automaton is translated into one single flat TCS.

Within his diploma thesis [Hob07], Ulrich Hobelmann pushed the integration of ARMC into Syspect so far that property-violating traces discovered by ARMC can automatically be translated back to the level of Syspect. Within Syspect such error traces are presented to the user in a tabular form, such that the low-level format of TCS and associated TCS traces remain completely hidden. Thus, problematic system behaviour can be analysed directly on the level of UML diagrams, where the specification has been constructed. This visualisation allows easier understanding of system behaviour and is a much more user-friendly way than having to cope with low level error traces.

Additional tool support for Syspect specifications comes with “LoRe”, a verification tool that is developed in Saarbrücken by Swen Jacobs and Viorica Sofronie-Stokkermanns within the AVACS subproject R1; currently, this tool is being linked to the format of transition control systems. LoRe employs methods of local and hierarchical reasoning to allow invariant checking and bounded model checking for classes of systems with unbounded parameters, which have previously not been amenable to automatic analysis.

7.1.1 Class Diagrams

The basic idea of the class diagrams of Syspect is to model the overall structure of the system specification by defining its constituting components in terms of class definitions. According to the UML profile for CSP-OZ [MORW07], these classes can

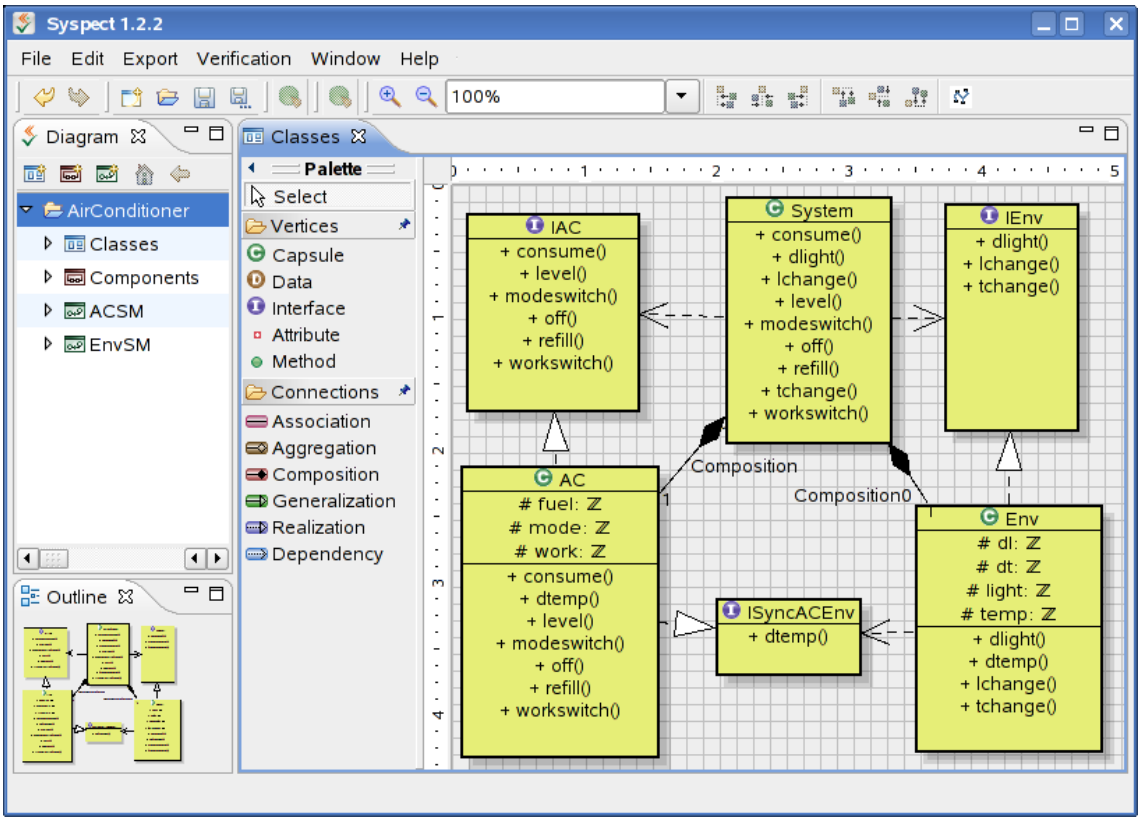


Figure 7.2: Syspect class diagram editor, containing a class diagram of the timed air conditioner system.

either be

- *capsules*, corresponding to CSP-OZ-DC classes on the specification level,
- *interfaces*, corresponding to interfaces of CSP-OZ-DC classes, or
- pure *data classes*, corresponding to complex data types, i.e., CSP-OZ-DC classes that serve as data containers and thus neither contain a CSP, nor an Object-Z part, nor have any methods defined within the class.

An example of a class diagram as it can be modelled with Syspect is depicted in Figure 7.2. The classes correspond to the CSP-OZ-DC classes of the timed air conditioner system that has been introduced in the previous chapters.

Methods defined within a capsule are always local to the respective capsule. If a method should be offered to the environment, it needs to be defined within an interface class, which is either *implemented* by the capsule (*Realisation* association) or which the capsule *depends* on (*Dependency* association). In addition to such Realisation/Dependency associations between capsules and interfaces, several further

types of association between classes can be defined, ranging from simple association over aggregation and composition up to generalisation defining inheritance relationships between classes.

According to the UML profile for CSP-OZ [MORW07], attributes can only be defined within capsules and data classes, but not within interfaces. The types of attributes can be freely chosen from several standard Z types (like integer $\mathbb{Z}$ or Boolean $\mathbb{B}$) up to arbitrary custom-defined types. However, at the time of writing, the verification back-ends consider any variable within a CSP-OZ-DC specification as being of type real. Therefore, the definition of types within the Syspect model is currently ignored completely during verification export.

The property tab associated with capsules allows for the definition of further details of the associated classes, ranging from the predicates contained within the CSP-OZ-DC `Init` schema over invariants defined within the CSP-OZ-DC state schema up to type definitions local to the respective capsule and the DC part of the capsule in terms of a list of counterexample formulae.

7.1.2 State Machines

The behaviour of Syspect capsules can be defined by simple UML state machines, which are subsequently translated into a subset of CSP, forming the CSP part of the respective CSP-OZ-DC classes.

An example of a state machine as modelled within Syspect is depicted in Figure 7.3. The state machine corresponds to the CSP part of the *AirConditioner* class that has been introduced in the previous chapters.

The ingredients of the state machines of Syspect are basically states and transitions between states, possibly labelled with events defined within the associated capsule. Next to ordinary states, there are three special types of states:

- *initial states*, denoting the position within the state machine at initialisation, corresponding to the CSP `main` process,
- *final states*, denoting termination of the state machine, corresponding to a concluding CSP `Skip` operator, and
- *hierarchical states*, containing a number of regions with further state machines running in parallel, corresponding to CSP interleaving operators with each of the interleaved processes defined within a separate region.

Except for transitions starting at initial states, each transition edge can be decorated with a trigger event that takes place, when the associated transition edge is taken, supposed the event's precondition is satisfied. Furthermore, it is also possible to explicitly omit a trigger. Omission corresponds to a nondeterministic decision of whether the transition is taken or not, whenever the state machine visits the transition's source state.

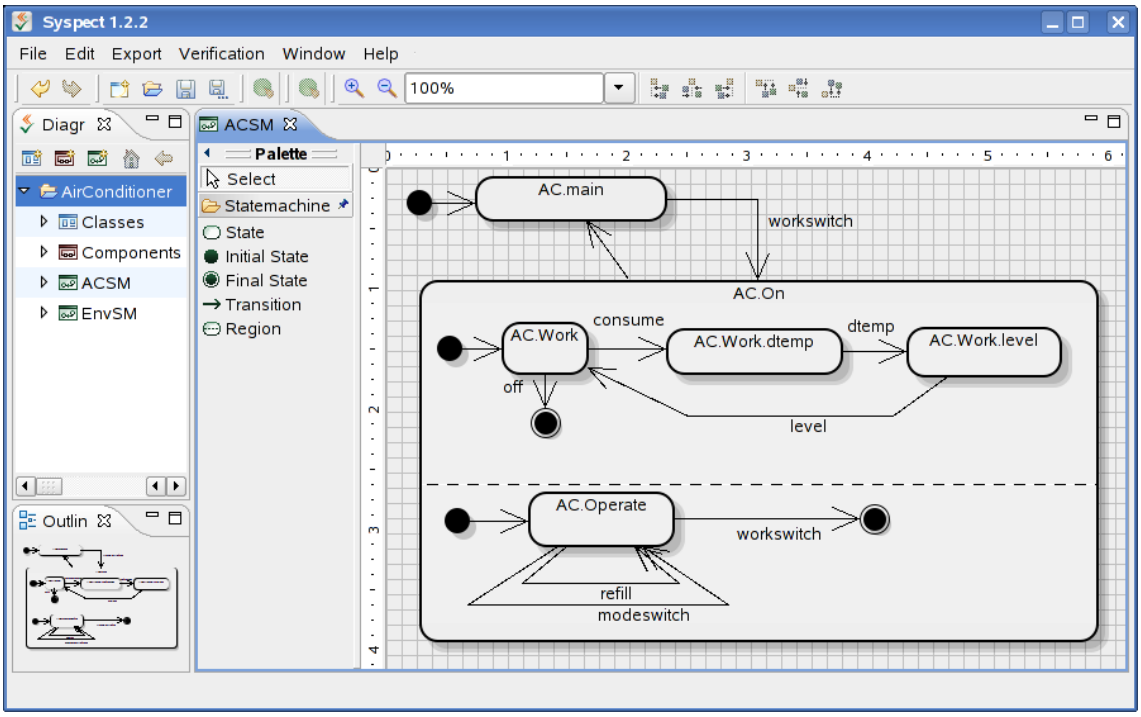


Figure 7.3: Syspect state machine editor, containing a state machine of the air conditioner class.

7.1.3 Component Diagrams

The role of component diagrams with respect to CSP-OZ and CSP-OZ-DC specifications [FOW01, MORW04] and thus also within Syspect is complementary to those of class diagrams. Where class diagrams only contain the *static view* onto the classes comprised by the specified system and their mutual associations, component diagrams define the *dynamic view* onto associations between instantiations of classes. In particular, component diagrams allow us to define multiplicities of instances of classes that assemble the system.

Another aspect modelled within component diagrams is the *communication* between associated components. Suppose, a previously constructed class diagram defines that one class A realizes an interface I that a different class B depends on. When dragged into a component diagram, class A then obtains a *base port*, denoted by a circle annotated with the interface class I, representing the fact that class A *provides* methods as defined within interface I. Class B, on the other hand, obtains a *conjugated port*, denoted by a semi-circle annotated with the interface class I, representing the fact that class B *depends* on the methods defined within interface I. Both ports can then be connected in order to define that the interface I provided by class A is used by class B who, in turn, depends on exactly that interface I.

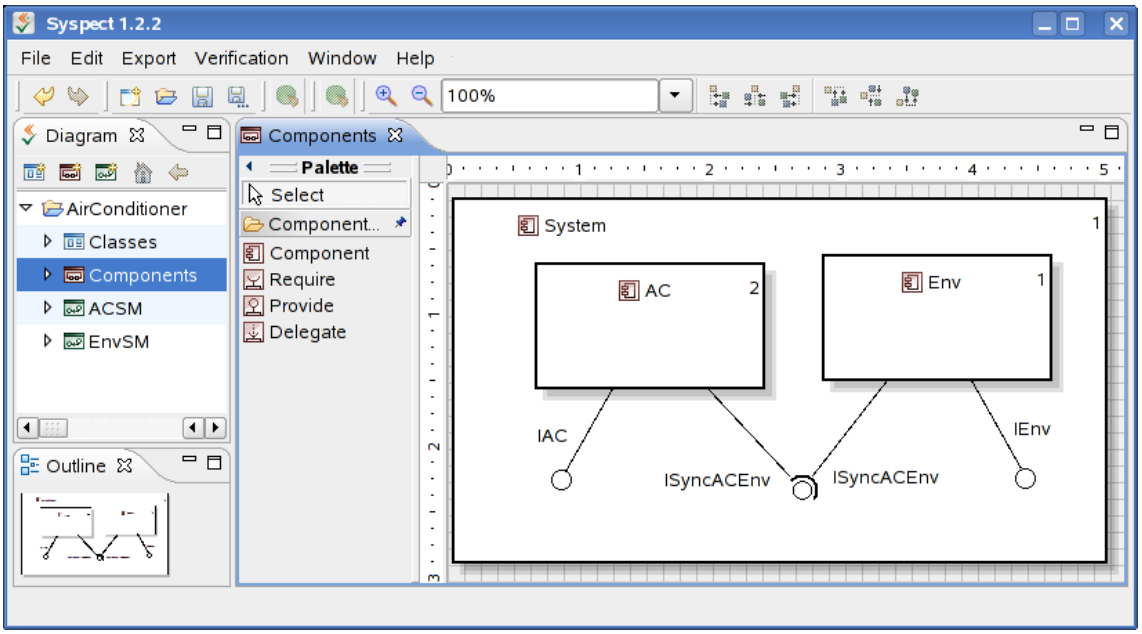


Figure 7.4: Syspect component diagram editor, containing a component diagram of the timed air conditioner system.

An example of a component diagram as modelled within Syspect is depicted in Figure 7.4. This example represents the air conditioner specification of the previous chapters where parallel composition of class *AC* with class *Env* yields the system class *System*. Communication between both involved classes is defined by the interface *ISyncACEnv* that is implemented by class *AC* and that class *Env* depends on.

Thus, the main purpose of component diagrams with respect to system specifications within Syspect is the top-level assembly of the system out of previously defined smaller components. Furthermore, component diagrams also add the aspect of expressing multiplicities during dynamic instantiation and the aspect of relations between system components in terms of inter-component communication.

During the CSP-OZ-DC export, each component diagram is translated into a separate CSP-OZ-DC class containing only a CSP part. The resulting CSP equations define the parallel composition of the classes that are composed within the component diagram. Furthermore, this class contains an interface defining the channels that are left unbound within the component diagram, such that the associated ports are delegated to the environment and thus allow outbound or inbound communication.

Communication between components is modelled by the CSP parallel operator with synchronisation on the alphabet of the methods contained within the linking interface, while components that do not directly communicate with each other are

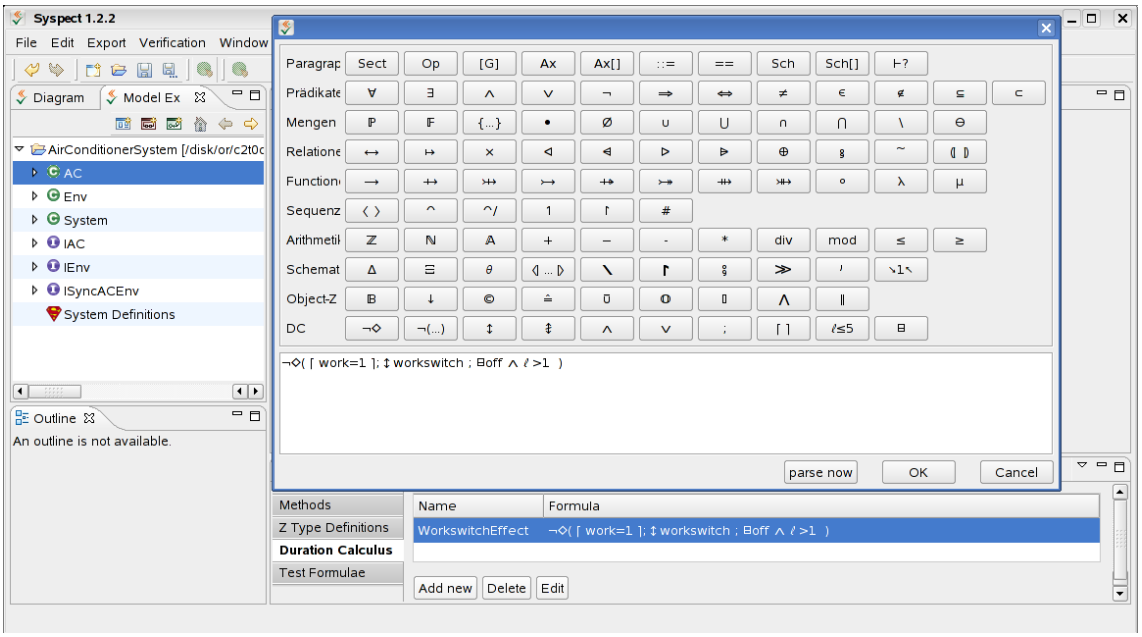


Figure 7.5: Syspect counterexample formula editor, containing a counterexample formula of the timed air conditioner class.

connected by an interleaving operator, i.e., without the obligation to synchronise.

7.1.4 DC Counterexample Formulae

The DC part of CSP-OZ-DC classes in Syspect is defined in terms of a list of the counterexample formulae as they have been introduced in Section 3.3 on CSP-OZ-DC specifications of this thesis. These counterexample formulae can be defined within the according Syspect editor that is depicted within Figure 7.5.

For handling the constituting elements of counterexample formulae, namely Z expressions, augmented by several symbols dedicated to DC formulae, the Community Z Tools (CZT, [MU05]) have been integrated into Syspect. The resulting formula editor offers a convenient way of designing and editing counterexample formulae, including all of the special symbols that they may contain. Moreover, each counterexample formula can be given a name to facilitate references to them during subsequent verification, especially when an error trace needs to be traced back to a certain DC formula.

Counterexample formulae defined within the Syspect counterexample editor can be translated directly into corresponding LaTeX mark-up. However, with respect to the export via the verification backend, some restrictions exist currently. Due to the power set construction involved in the translation of counterexample formulae into corresponding phase event automata and current limitations of the underlying

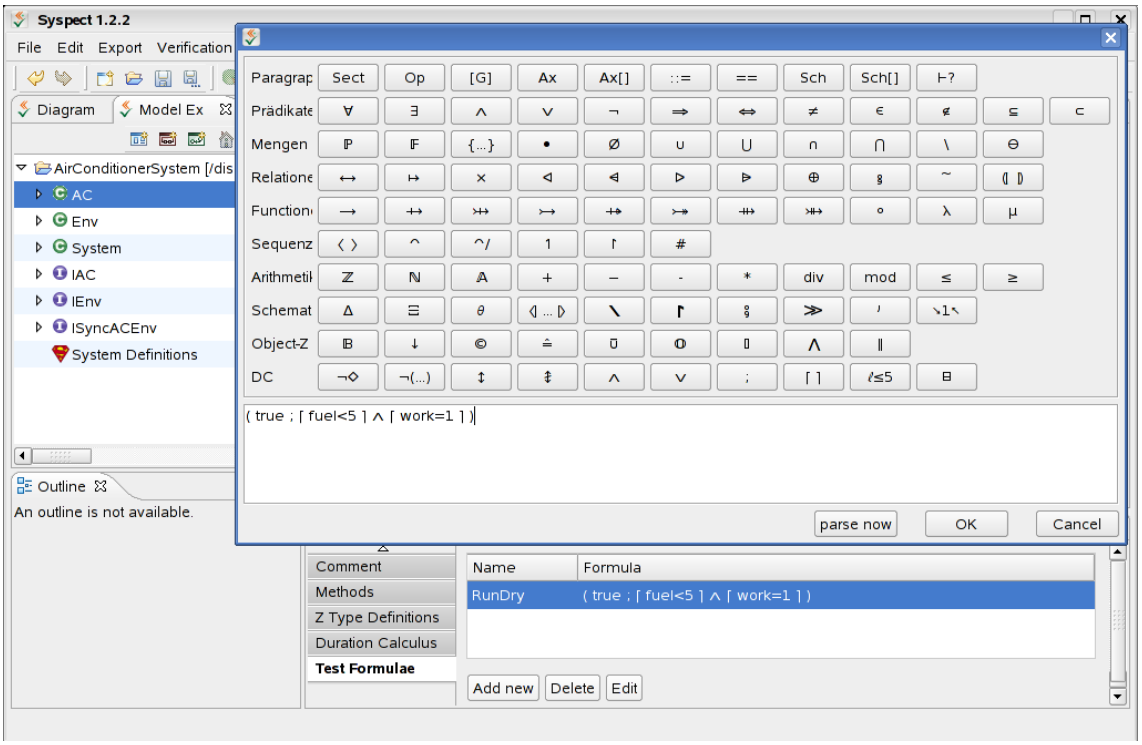


Figure 7.6: Syspect test formula editor, containing a test formula that serves as a verification property for the timed air conditioner system.

constraint decision diagrams (CDDs) (see [Hoe06] and [Cor07d]), it is not possible to define symbolic constraints on the lengths of intervals, i.e., only concrete numeric bounds such as $\ell < 3$ can be given.

7.1.5 DC Test Formulae and Syspect Verification

Once a specification has been constructed within Syspect, another essential feature of the modelling environment is its verification back-end. In order to perform verification runs, we first of all need facilities for formulating and managing requirements, i.e., verification properties that the specification should satisfy.

To this end, Syspect allows us to enter test formulae as defined in Section 6.3 of this thesis. Test formulae can be designed and manipulated within the according Syspect editor, which is designed very similar to the Syspect counterexample formulae editor as shown in Figure 7.6.

These test formulae define the properties which are expected to hold for the specified system. Consequently, they are used as the verification properties during model checking runs. As stated for counterexample formulae, also test formulae are managed within lists associated with specification classes. Furthermore, they can also be given names for more convenient reference during subsequent verification.

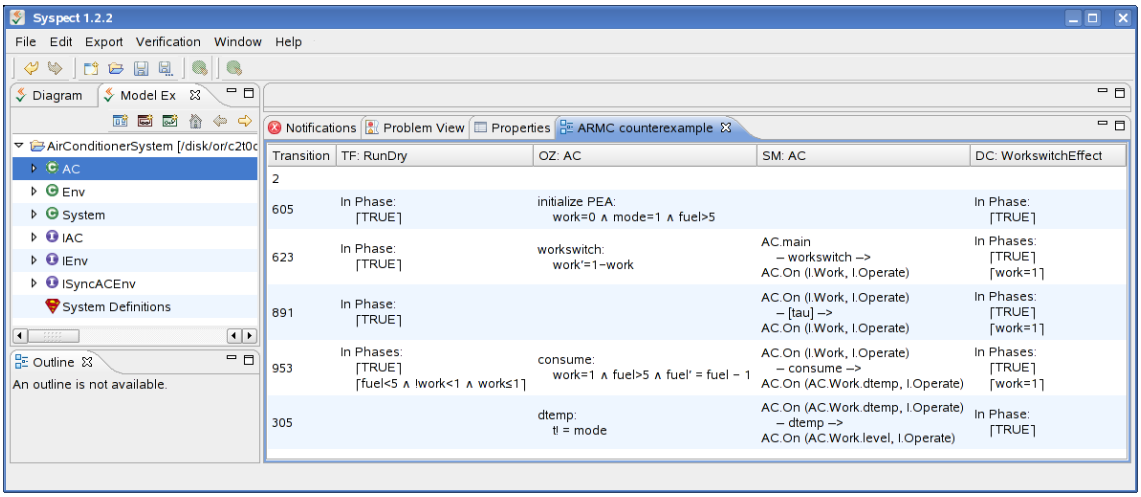


Figure 7.7: Syspect error trace visualisation, containing an error trace of the timed air conditioner system violating the verification property depicted in Figure 7.6.

During each verification export of Syspect, one of the previously defined test formulae can be exported along with the actual specification, such that the subsequently operating model checking tools can then be used for analysing whether the required property is indeed satisfied.

When such a model checking run discovers system behaviour that violates the given property, an appropriate error trace is generated which can—in the case of ARMC-based verification—be transformed back into the Syspect modelling environment, where it is displayed to the user as depicted within Figure 7.7.

The error trace is presented in a tabular format with each step of the trace contained within a separate line, to be read from the top line (defining the initial step of the trace) to the bottom line (defining the last step of the trace where the undesired behaviour has taken place).

Within each line, there are several columns, containing detailed information about the associated step of the error trace, such as

- the transition number associated with the trace step (originating from the low-level transition constraint system),
- the current state of the test formula (the verification property),
- the Object-Z method associated with the trace step, together with the `effect` schema predicate of the method,
- the CSP event associated with the trace step in terms of the associated state machine transition, and
- the current states of the involved DC formulae.

The current states of test and counterexample formulae are represented in terms of the prefixes of phases that have been observed so far. When phases have not yet been observed in a given step, they are not displayed in the associated line of the error trace.

The design and implementation of this back-transformation and visualisation has been subject of the diploma thesis of Ulrich Hobelmann [Hob07], who, moreover, completed the ARMC verification export of Syspect, since this feature had not been fully finished during the work of the Syspect project group.

7.1.6 Specification Export

The specifications that have been constructed within Syspect can—as well as individual CSP-OZ-DC classes or combinations thereof—subsequently be exported into various formats:

CSP-OZ-DC LaTeX mark-up: This export format corresponds to the LaTeX mark-up as it has been used within this thesis, including line breaks for overlong lines of LaTeX code and page breaks for specifications comprising several CSP-OZ-DC classes that possibly run over several pages. The LaTeX mark-up needs to be processed by `latex` or `pdflatex` in order to produce DVI or PDF files as output. For certain commands, the `czt.sty` style file might be necessary in addition to the generated LaTeX file.

CSP-OZ-DC XML representation: This export format allows the integration of further tools by offering a clean interface for subsequent import of CSP-OZ-DC specifications. Tools that are capable of importing these XML files must of course implement the same CSP-OZ-DC XML schema. Currently, there exist no such tools.

Phase Event Automata (PEA) XML: This export format corresponds to the XML representation of the automata-theoretic semantic model of CSP-OZ-DC in terms of phase event automata (PEA) that can be generated for each part of each class of a Syspect model, i.e., for all CSP parts of classes (corresponding to the state machines and component diagrams of Syspect), all Object-Z parts (corresponding to the class diagrams, component diagrams and associated Object-Z annotations of Syspect), and all DC formulae (corresponding to a formula out of the lists of DC counterexamples within a Syspect class). The XML format of the Community Z Tools (CZT, [UTS+03]) is used for the representation of Z expressions contained within the exported files.

Transition Constraint Systems (TCS): This export format also results from the generation of the specification's PEA semantics, which is used to compute a large parallel product representing the complete system in conjunction

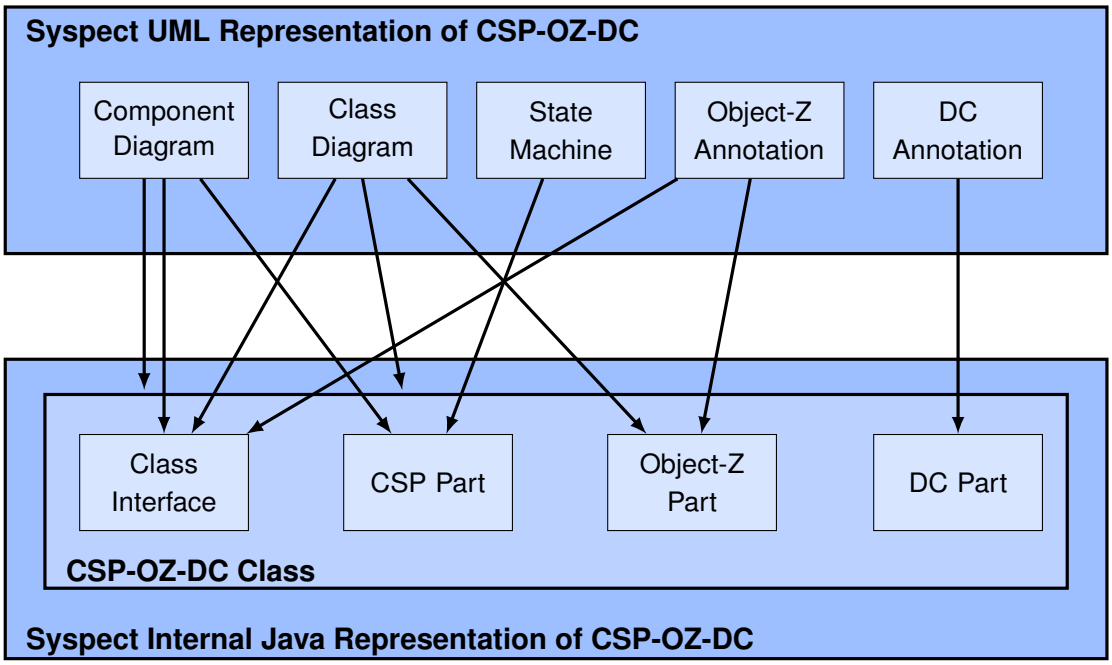


Figure 7.8: Syspect translation from UML to CSP-OZ-DC.

with the verification property represented by a special PEA. The verification property is the requirement that the specification should meet, which is derived from one of the test formulae described in the previous section.

The resulting product automaton of all involved PEAs together with the special test formula PEA is then translated into the format of transition constraint systems (TCS), which can finally be fed into model checkers such as ARMC or SLAB.

Diagram exports: The various types of diagrams offered by Syspect such as component diagrams, class diagrams, and state machines can be exported into a number of graphic formats such as pixel-oriented formats like JPEG and BMP or vector-oriented formats like EPS and PDF.

Each of the above mentioned types of export—except for the diagram exports into graphic formats—proceeds initially as depicted in Figure 7.8.

The CSP-OZ-DC UML representation of Syspect as defined according to the UML profile for CSP-OZ [MORW07] is translated into an internal Java representation of the defined specification:

CSP-OZ-DC classes are generated according to their definition within component diagrams and class diagrams as CSP-OZ-DC capsules or CSP-OZ-DC data classes.

Interfaces of CSP-OZ-DC classes are generated according to their definition within component diagrams, class diagrams and Object-Z annotations (the latter, e.g., for the definition of channel types).

CSP parts of CSP-OZ-DC classes are generated according to their definition within component diagrams and state machines.

OZ parts of CSP-OZ-DC classes are generated according to their definition within class diagrams (defining attributes and methods) and Object-Z annotations (defining predicates that form invariants, guards and transformations on the state space).

DC parts of CSP-OZ-DC classes are directly generated from a class' DC annotations, i.e., the list of DC counterexample formulae.

After this initial step, each type of export is then based on the same internal Java representation of CSP-OZ-DC classes. Consequently, this representation also serves as the starting point for the slicing plug-in of Syspect, as we will see in the next section.

7.2 Slicing Implementation within Syspect

The component of Syspect with the highest relevance to this thesis is the slicing plug-in within Syspect. This plug-in has been developed by Sven Linker during his work as a student assistant within subproject R1 of AVACS. It is the straightforward implementation of the slicing algorithms defined in the previous chapters of this thesis, such that the previously introduced modelling and verification facilities of Syspect are enhanced by a completely automatic slice computation.

This section introduces the slicing plug-in with respect to its positioning within the architecture of Syspect and its role in the workflow associated with the modelling and analysis process that comes along with Syspect. To this end, this section presents the general functionality offered by the slicing plug-in as well as each of the output documents resulting from its application. These documents include graphical representations of the associated data structures like the *control flow graph* and the *program dependence graph*, on which the slicing algorithm is based, and, furthermore, the *slicing report*, which summarises the reduction achieved by slicing, and, last but not least, the actual *slicing result*, namely a reduced version of the respective Syspect export.

7.2.1 Syspect Slicing Plug-In

Slicing for Syspect is implemented as a plug-in that can optionally be added to the functionality of Syspect. If included, it offers additional *slicing* variants of

each of the export and verification facilities of Syspect models. This is achieved by an on-the-fly computation of slices when performing the respective exports into CSP-OZ-DC LaTeX mark-up, into phase event automata, or into ARMC/SLAB transition constraint systems as well as when performing ARMC-based verification.

Each of these additional export and verification variants work in a similar way by extending the ordinary form of export: the first step always consists of converting the given Syspect model from the level of its UML representation into an internal representation within a resulting CSP-OZ-DC Java model as depicted in Figure 7.8.

However, while the ordinary lines of export and verification immediately use this internal representation for further computations, their slicing variants first transform this CSP-OZ-DC Java model into a reduced version of it, i.e., its slice with respect to a user-specified test formula as the slicing criterion.

Once this reduced CSP-OZ-DC Java model has been computed, the slicing variants of export and verification proceed exactly as their ordinary counterparts, except for the fact that they then work on the slice instead of the original model.

Thus, the slicing variants of the export and verification facilities of Syspect essentially add only one additional step into the export and verification chain as depicted within Figure 7.9.

In addition to the actual slice of the internal Java representation of the specification, the slicing plug-in also generates output files containing representations of the main data structures of the slicing algorithm, i.e., the control flow graph and the dependence graph for the specification that will next be presented.

7.2.2 Control Flow Graph

The control flow graphs generated by the Syspect slicing plug-in correspond directly to the control flow graph as defined in Chapter 4 of this thesis. Their main purpose within the development of the slicing plug-in has been to facilitate debugging of the slicing implementation, since the subsequent construction of the dependence graph depends on the correct construction of the control flow graph. The definition of many of the dependence types that comprise the dependence graph rely on a previous analysis of the control flow graph. For instance, a necessary condition for the existence of a direct data dependence edge between two nodes is the existence of a direct control flow graph path between both nodes.

The Syspect slicing plug-in does not directly produce graphical files containing the control flow graph, but rather generates a textual file describing the control flow graph structure in terms of nodes and edges, defined in the simple input language of the `dot` tool from the *Graph Visualization Software* (Graphviz, [Gra07]).

Therefore, another step is necessary to obtain the actual graphical representation of the graph structure, i.e., a run of the Graphviz `dot` tool that automatically computes a layout for the graph structure and renders an associated graphical representation.

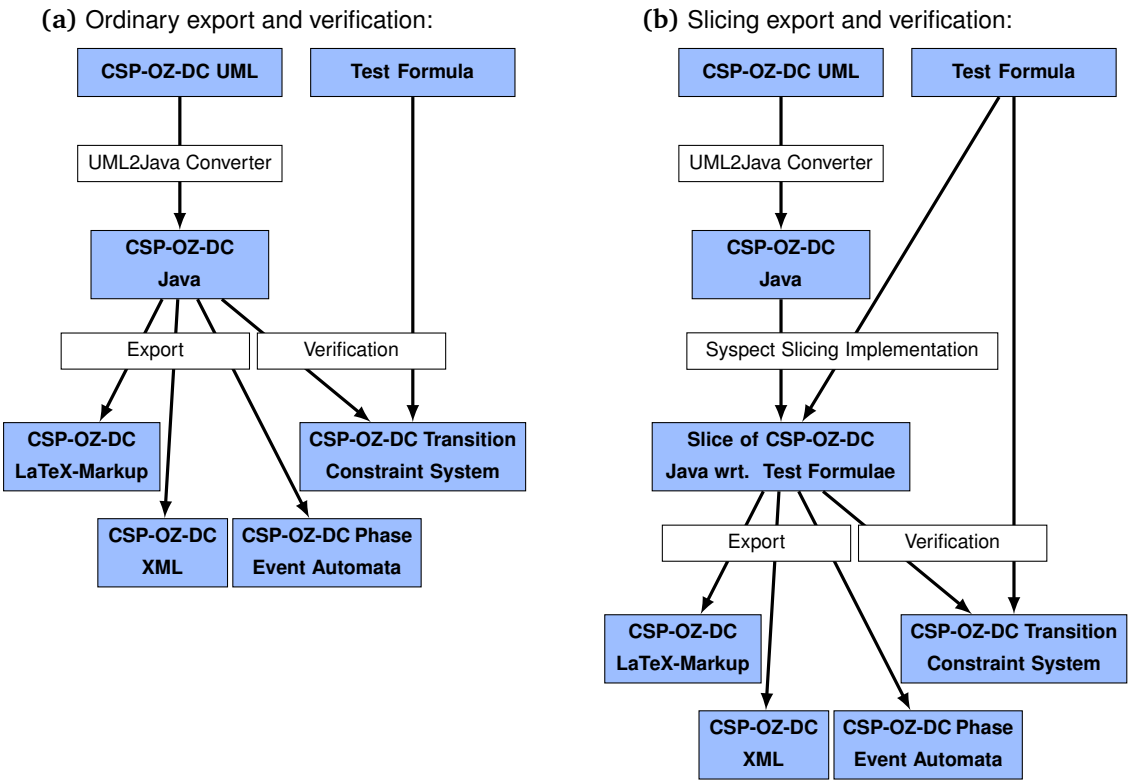


Figure 7.9: Comparison of the ordinary line of export and verification (a) with its slicing line of export and verification (b), where the slicing implementation is embedded as one additional intermediate step.

An example of a control flow graph as generated by the Syspect slicing plug-in (after fully automatic layout by the `dot` tool) is depicted in Figure 7.10.

Each node of the control flow graph is labelled with a description that identifies its source, i.e., the associated specification element following the definition of the control flow graph in Chapter 4. In addition to these basic annotations, the `dot` input language offers several further possibilities for defining specific layouts for different types of nodes and edges, which are heavily used by the slicing plug-in in order to enhance the readability of the resulting graph layout. The following control flow graph nodes can be distinguished according to their shape and their colour:

Structural nodes are displayed as rectangles with white background. They represent CSP operators such as sequential composition, `Skip`, or `Stop` operators.

Class parallel and class unparallel nodes are displayed as house shapes with white background. They represent regions in the graph that belong to classes put in parallel composition. The synchronisation alphabet is given as an

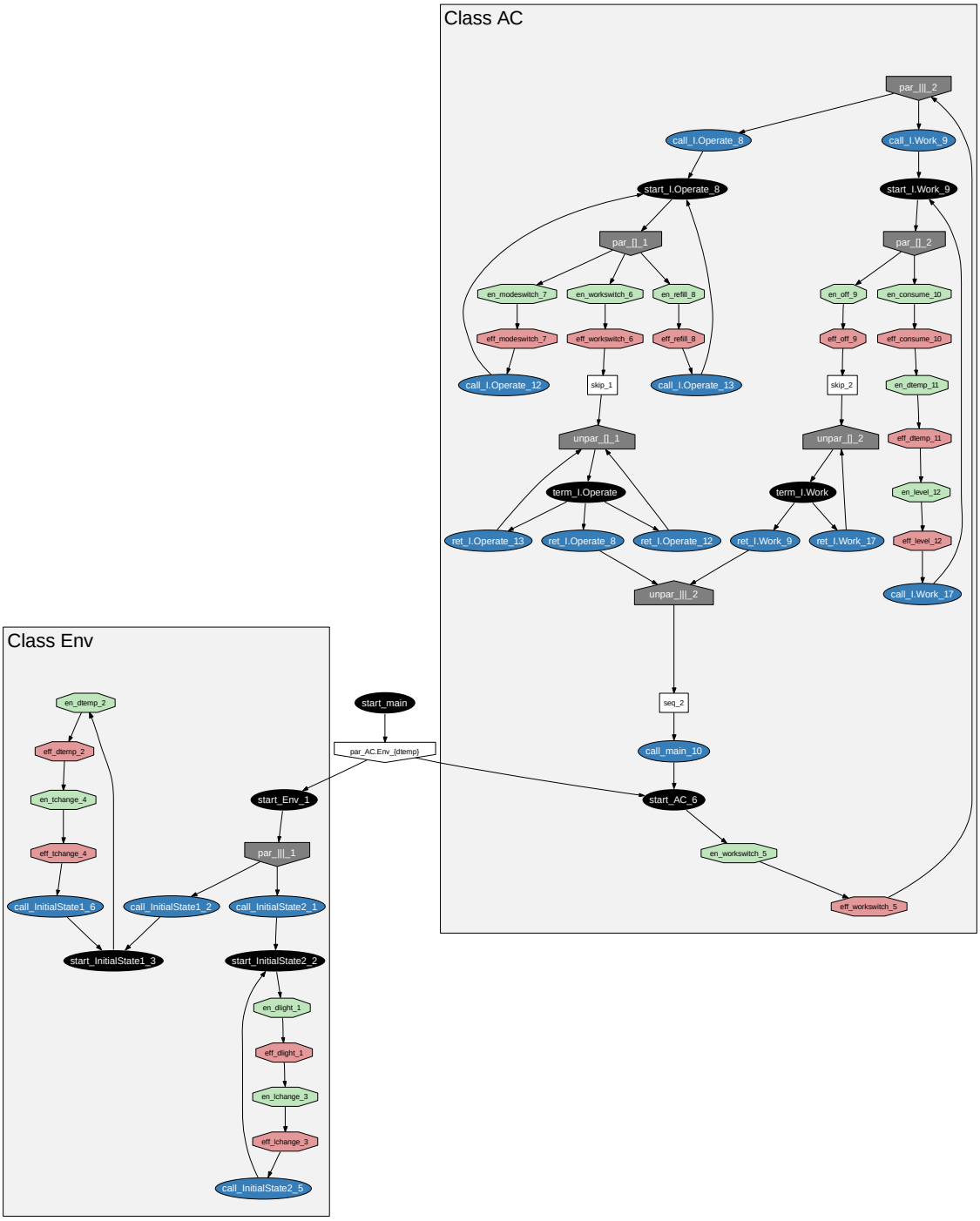


Figure 7.10: Control flow graph as automatically generated by the Syspect slicing plug-in for the specification of the timed air conditioner system after automatic layout by the dot tool.

extension of the node label within curly braces.

Ordinary parallel and unparallel nodes are displayed as house shapes with grey background. They represent regions in the graph that belong to branches of CSP parallel composition operators. The synchronisation alphabet is again given as an extension of the node label within curly braces.

Schema nodes are displayed as octagons with green and red backgrounds. Green versions represent `enable` schemas and red versions represent `effect` schemas.

Process start/term nodes are displayed as ovals with black background. They represent unique entry and optional exit points of control flow into/out of processes.

Process call/return nodes are displayed as ovals with blue background. They represent process calls and return points for control flow when a CSP process is referenced or when a referenced CSP process terminates and control flow returns to the point of its previous reference.

For a legend summarising these different types of nodes see Figure 7.12a in the following Section 7.2.3 on dependence graphs generated by the slicing plug-in.

Another feature of the `dot` input language is the definition of clusters, i.e., groups of nodes that are combined within rectangular areas and laid out separately. In the `dot` description of the control flow graph this feature is used to distinguish different classes of the underlying specification: for each involved class, a cluster is defined that comprises all control flow graph nodes related to the respective class. Furthermore, the cluster is labelled with the name of the associated class.

Finally, all edges of the control flow graph are represented by the same type of directed line, since all of them have the same type and thus do not need to be further distinguished.

7.2.3 Dependence Graph

As stated for the control flow graphs, the Syspect slicing plug-in generates dependence graphs that correspond directly to the dependence definitions of Chapter 4 of this thesis. Moreover, the main motivation for implementing a facility for rendering graphical representations of the resulting graph structures has been the same as for control flow graphs, namely to facilitate debugging of the slicing implementation.

Furthermore, it turned out that dependence graphs can be used as a nice visual instrument for analysing a given specification with respect to the identification of relations between specification components. They were especially useful for finding out, between which components the strongest mutual dependences exist,

and which components are less closely coupled and might thus have less relevance to certain analysis properties during compositional verification.

Like control flow graphs, the slicing plug-in does not directly render dependence graphs, but rather generates descriptions of their graph structure in terms of `dot` input files. These are laid out and rendered by the graph layout tool `dot` of the Graph Visualization Software Graphviz [Gra07].

An example of a dependence graph as automatically generated by the Syspect slicing plug-in (after fully automatic layout by the `dot` tool) is depicted in Figure 7.11.

Again, several facilities of the `dot` input language are used in order to distinguish different types of nodes and edges of the dependence graph. A legend for all these types of graphical elements of the fully rendered program dependence graph is depicted in Figure 7.12.

Except for predicate nodes, all types of dependence graph nodes have already been present in the control flow graph and are displayed in the same way in both graphs. The newly introduced predicate nodes are displayed in the same way as structural nodes, namely as rectangles with white background. They represent predicates or conjuncts of predicates as they appear within Object-Z schemas.

The different types of dependence edges are distinguished according to their colour as follows.

- *Control dependence* is represented by black edges, without any further differentiation between the different types of control dependence.

- The different types of *data dependence* are represented by
  - brown edges for *direct data dependence* between predicate nodes of different schemas,
  - orange edges for *interference data dependence* between predicate nodes of schemas within different branches of a parallel node, and
  - yellow edges for *symmetric data dependence* between predicate nodes of the same schema.

- *Synchronisation dependence* is represented by blue edges for synchronisation dependence between nodes of two different classes over which both classes synchronise; furthermore, violet edges represent the associated *synchronisation data dependence*, resulting from transmission of data via an output variable on the sending side and an input variable on the receiving side.

- *Timing dependence* is represented by green edges, connecting all nodes that are directly or indirectly referred to by one of the DC counterexample formulae of the underlying specification.

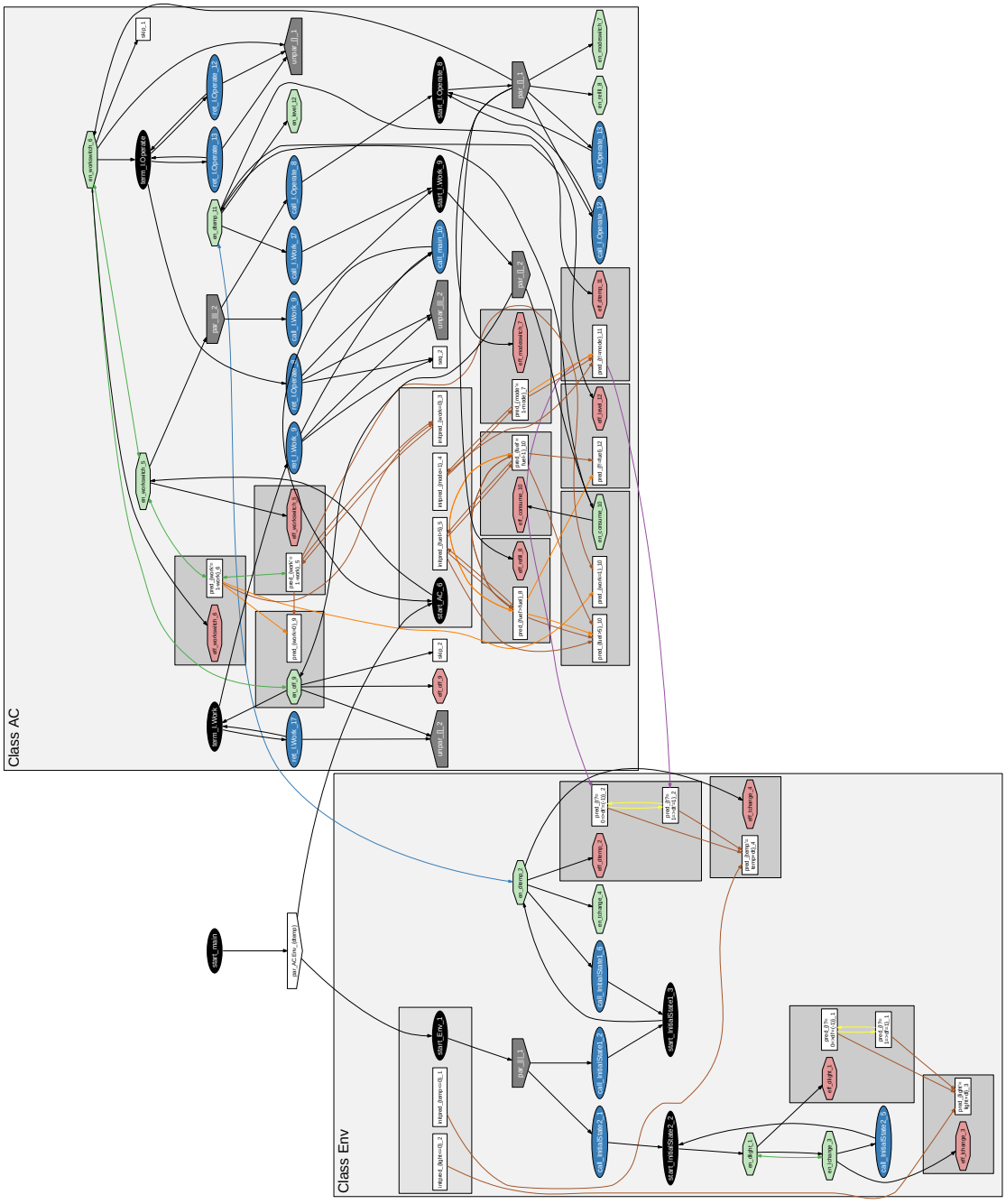


Figure 7.11: Dependence graph as automatically generated by the Syspect slicing plug-in for the specification of the air conditioner system after automatic layout by the dot tool.

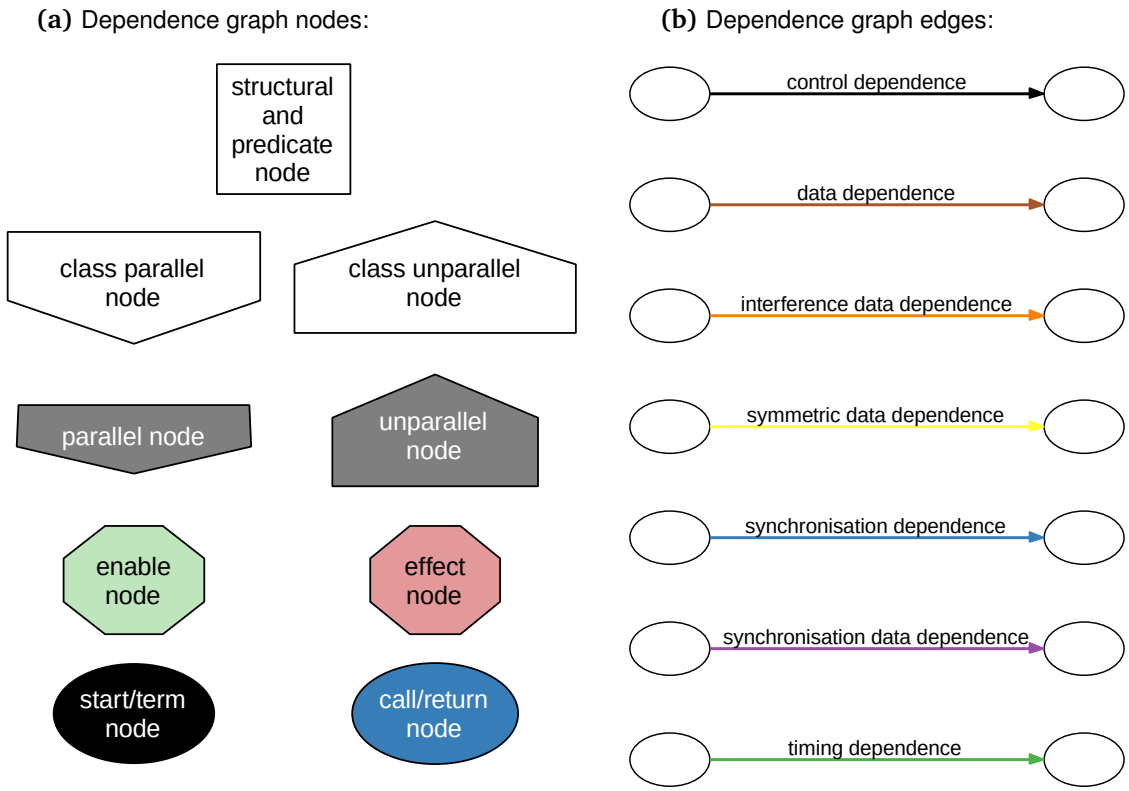


Figure 7.12: Legend for the graphical elements comprised by dependence graphs generated by the Syspect slicing plug-in.

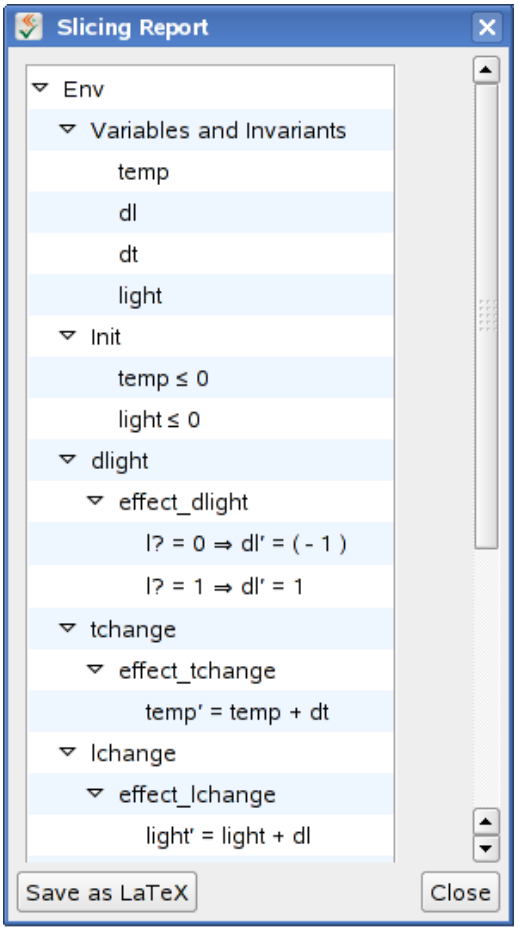
7.2.4 Slicing Report

Once a slice computation has been successfully completed, the slicing plug-in presents a concluding slicing report dialog window as depicted in Figure 7.13a. The slicing report summarises the reductions that have been achieved by the slice computation, such that the effect of the slice computation can easily be assessed by the user.

To this end, the report is organised according to the specification structure into sections for each of the involved classes of the specification. Each section lists all of the elements that have been removed from the original specification in order to obtain the slice of the respective class. The specification elements, which are possibly contained in the slicing report, range from variables and invariants contained within the state schema, over predicates of the `Init`, `enable` and `effect` schemas up to counterexample formulae of the DC part of each class.

In addition to the tree-like organised summary presented in the concluding dialog of the slicing plug-in, the slicing report can also be exported as LaTeX mark-up as depicted in Figure 7.13b, containing a list of all specification elements that have

(a) Slicing report window:



(b) Slicing report LaTeX mark-up:

- Env
  - Variables and Invariants
    - $temp : \mathbb{Z}$
    - $dl : \mathbb{Z}$
    - $dt : \mathbb{Z}$
    - $light : \mathbb{Z}$
  - Init
    - $temp \leq 0$
    - $light \leq 0$
  - dlight
    - effect
      - $l? = 0 \Rightarrow dl' = (-1)$
      - $l? = 1 \Rightarrow dl' = 1$
  - tchange
    - effect
      - $temp' = temp + dt$
  - lchange
    - effect
      - $light' = light + dl$
  - $[\ldots]$

Figure 7.13: Slicing report window (a) and LaTeX mark-up (b), both listing all specification elements of the timed air conditioner system that are irrelevant with respect to the verification property depicted in Figure 7.6.

been removed by slicing.

7.3 Benchmarks and Case Studies

This section gives an account on the results that have been achieved by applying slicing to a selection of several specifications. The specifications range from simple toy examples as presented in the previous chapter to illustrate the slicing approach up to two larger case studies that have been developed within AVACS R1 and within a student project, respectively.

All specifications have been constructed within the Syspect modelling environment for CSP-OZ-DC and all of these Syspect models are available within the public subversion repository of Syspect [Cor07e], where they reside in the directory `src/SyspectExample/projects`.

The following versions of software tools have been used:

- Syspect 1.2.2 in subversion revision r5569.
- Syspect slicing plug-in in subversion revision r256.
- Syspect PEA-Tool in subversion revision r302.
- dot (Graph Visualization Software) version 2.6 (Tue May 2 07:58:05 UTC 2006) [Gra07].
- ZGRViewer v0.7.2 ("A Visualization Tool for Graphviz based on ZVTM") [Pie07].
- ARMC in version 3.20.03 (12-Apr-2007) [Ryb07].
- SLAB in subversion revision r147.
- Java 2 Runtime Environment, Standard Edition (build 1.5.0_13-b05).
- CLP-prover in version 0.2 (Mar-29-2007) [Ryb06].

All of the experiments have been carried out on an AMD Sempron 2600+ with 1 GB of memory under SuSE Linux 10.1.

7.3.1 Tic-Tac-Toe

When using Syspect to model the Tic-Tac-Toe specification presented in Chapter 3, a CSP-OZ-DC export produces exactly the intended form of CSP-OZ-DC LaTeX mark-up as previously depicted in a manually generated version.

Furthermore, when using the slicing plug-in of Syspect to compute reduced versions of this specification, we obtain exactly the same results as from the previous manual computations presented in Chapter 5. Slices have been computed with respect to both of the following previously introduced verification properties:

- $\varphi_1 \equiv \Box\lceil \mathit{moves} = 9 - \#\mathit{free}\rceil$
  This formula defines the number of moves in terms of free fields on the Tic-Tac-Toe board.

- $\varphi_2 \equiv \neg\Diamond(\mathit{black} \frown (\lceil \mathit{true}\rceil \wedge \neg(\lceil \mathit{true}\rceil \frown \mathit{white} \frown \lceil \mathit{true}\rceil)) \frown \mathit{black})$
  $\wedge\, \neg\Diamond(\mathit{white} \frown (\lceil \mathit{true}\rceil \wedge \neg(\lceil \mathit{true}\rceil \frown \mathit{black} \frown \lceil \mathit{true}\rceil)) \frown \mathit{white})$
  This formula characterises the alternations of moves between the black and the white player.

The obtained reductions are summarised in Table 7.1 that lists the numbers of specification elements that have been manually counted for the full specification and for the slices. Additionally, percentage figures are given for the slices relative to the full specification.

Specification Elements	**Full**	φ_1**-Slice**	**%**	φ_2**-Slice**	**%**
Interface	3	3	100	3	100
Variables	6	6	100	3	50
State invariants	2	2	100	1	50
Init predicates	6	6	100	3	50
`enable` schemas	3	3	100	3	100
`effect` schemas	3	3	100	3	100
Total # predicates	24	21	88	13	54
Total # spec. elements	47	44	94	29	62

Table 7.1: Slicing results for the Tic-Tac-Toe specification. The size of the slice for each property is given in terms of the absolute number of remaining specification elements and as percentage figures relative to the full specification.

These results confirm the observation that the reduction that can be achieved by slicing depends very much on the kind of property used as a slicing criterion: With respect to property φ_1 only the communication of the final outcome of the game can be removed from the specification, since all remaining information is relevant to the property, such that the slice still contains 94% of all elements of the full specification, i.e., only 6% have been removable.

For property φ_2, on the other hand, only the aspect of alternation of moves between both players is important, while the actual result of the game including the markings on the board fields are not relevant. Thus, a lot more specification

elements of the original specification can be removed and only 62% of them remain in the slice.

Note that although the specification's CSP-OZ-DC LaTeX mark-up can be exported without a problem and thus the merely syntax-based computation of its slice is completely unproblematic, we can unfortunately not obtain any verification results for this specification, since it involves set calculations such as $wposn' = wposn \cup \{p!\}$, which are not yet covered by the Syspect translation into phase event automata and can hence currently not be handled by the verification back-end of Syspect.

Next, we will introduce another Object-Z specification for which we can in fact successfully carry out verification runs.

7.3.2 Cash Register

In order to complement our collection of slicing examples with an Object-Z specification that is indeed amenable to verification, we now introduce a completely new specification of a cash register.

This *CashRegister* specification defines a very simplistic version of a supermarket cash register containing only two storage entities: one representing the current amount of money inside the cash register (variable *cash*) and the other one counting the number of items sold so far (variable *item*). Initially, both variables have a value of zero. Furthermore, there are two events that this cash register can perform. The *pay* event represents the process of an item being sold. A payment is received and the cash variable and the item counter are updated accordingly.

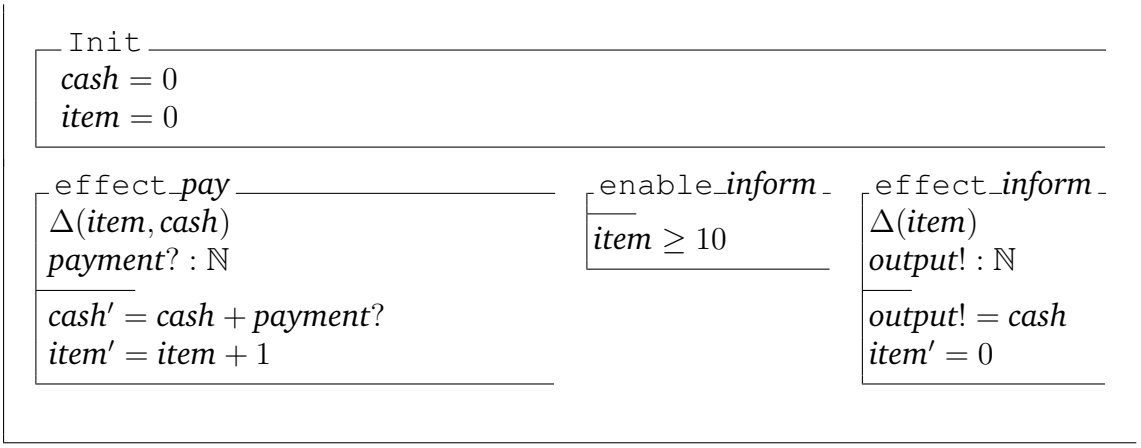
The motivation of the other event *inform* could be thought of as a means of the supermarket manager to be informed from time to time about the current amount of money inside the cash register. Since he doesn't want to be distracted from his other important work, he only wants to receive this information when, for instance, at least ten items have been sold since the last information was sent. Therefore, the *inform* event is only enabled, when the item counter is greater than or equal to ten. In this case, the current amount of cash can be communicated to the supermarket manager and the item counter is reset.

A specification of such a cash register has been constructed within Syspect, leading to the following export as CSP-OZ-DC LaTeX mark-up:

CashRegister

$\texttt{chan}\ inform : [output! : \mathbb{N}]$
$\texttt{chan}\ pay : [payment? : \mathbb{N}]$

$cash : \mathbb{N}$
$item : \mathbb{N}$

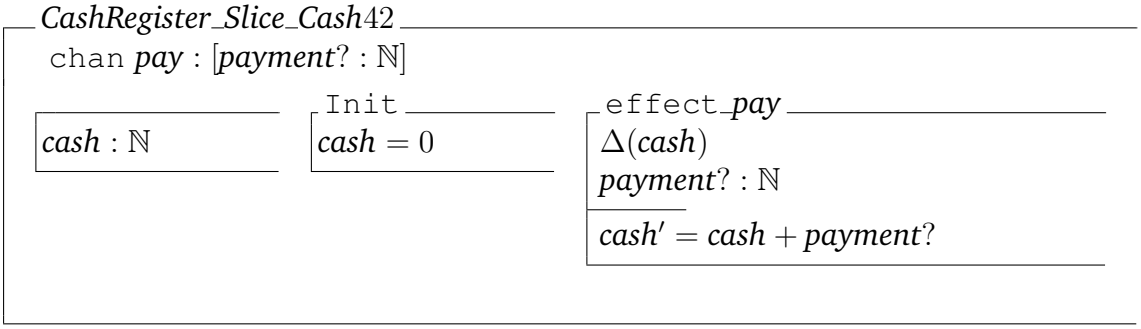


As the slicing criterion we consider the verification property

$$\varphi \equiv \Diamond\lceil \mathit{cash} = 42 \rceil,$$

defining that the cash register can eventually reach a state where its variable *cash* has a value of 42.

When computing the slice of the *CashRegister* specification with respect to this property using the Syspect slicing plug-in, we obtain the following specification slice:



In comparison to the original specification, the *inform* event has been removed from the interface along with its associated `enable` and `effect` schema as well as the associated variable *item*. This result is sensible, since the property of reaching a state with a certain valuation of variable *cash* is only influenced by the *pay* event, but not at all by event *inform* that only has the role of sending information on the current state of the cash register to the environment of the class.

When comparing the verification runs of ARMC and SLAB as depicted in Table 7.2, we can observe that slicing leads to a reduction in terms of model size down to 80%

of that of the original system, resulting in an even stronger reduction of verification runtime down to 53% in the case of ARMC and down to 33% in the case of SLAB, each time relative to the verification runtime needed for the full system.

Dimension	**Full**	**Slice**	**%**
model size [TCS transitions]	58	47	81
ARMC runtime [s]	0.15	0.08	53
SLAB runtime [s]	31.58	10.50	33

Table 7.2: Model sizes and verification runtimes for the *CashRegister* specification and its slice with respect to φ. Additionally, percentage figures relative to the full system are given for its slice.

As expected, both model checkers identify a run of the model violating the property φ in the case of the full specification as well as in the case of the specification slice.

7.3.3 Untimed Air Conditioner

In Chapter 3 of this thesis we have introduced the CSP-OZ specification of an untimed air conditioner system as our running example of CSP-OZ specifications. This specification has been modelled within Syspect as a project containing only one single capsule. The class corresponds to the capsule *AC* of the class diagram that has been shown in Figure 7.2 in the beginning of this chapter.

The remaining capsules and interfaces of that class diagram have not been included in the Syspect model, since the untimed air conditioner only consists of the class *AC*. The state machine of the *AC* class is the same as previously shown in Figure 7.3.

With respect to verification, one main difference between the specification and the resulting Syspect model are the data types of the involved variables. As mentioned, Syspect allows us to define variables of arbitrary non-real types, but the verification export plug-ins currently ignore any such types. Instead, any variable is regarded as being of type real. This difference needs to be kept in mind already during modelling, if subsequent verification is planned.

The ARMC verification export plug-in of Syspect has then been used to perform verification runs for the full specification with respect to verification properties

$$\varphi_1 \equiv \neg\Diamond[\mathit{fuel} < 5] \quad \text{and} \quad \varphi_2 \equiv \neg\Diamond[\mathit{fuel} < 5 \wedge \mathit{work} = 1].$$

As expected, the verification result is in each case an error trace like that depicted in the screenshot of Figure 7.14a.

(a) Full specification error trace:

ARMC counterexample

Transition	TF: RunDry	OZ: AC	SM: AC
2			
168	In Phase: ⌈TRUE⌉	initialize PEA: work=0 ∧ mode=1 ∧ fuel>5	
172	In Phase: ⌈TRUE⌉	workswitch: work'=1−work	AC.main – workswitch –> AC.On (I.Operate, I.Work)
177	In Phase: ⌈TRUE⌉		AC.On (I.Operate, I.Work) – [tau] –> AC.On (I.Operate, I.Work)
206	In Phases: ⌈TRUE⌉ ⌈fuel<5⌉	consume: work=1 ∧ fuel>5 ∧ fuel' = fuel − 1	AC.On (I.Operate, I.Work) – consume –> AC.On (I.Operate, AC.Work.dtemp)
78			AC.On (I.Operate, AC.Work.dtemp) – [tau] –> AC.On (I.Operate, AC.Work.dtemp)

(b) Sliced specification error trace:

ARMC counterexample

Transition	TF:	OZ: AC	SM: AC
2			
168	In Phase: ⌈TRUE⌉	initialize PEA: work = 0 ∧ fuel > 5	
172	In Phase: ⌈TRUE⌉	workswitch: work'=1−work	AC.main – workswitch –> AC.On (I.Work, I.Operate)
176	In Phase: ⌈TRUE⌉		AC.On (I.Work, I.Operate) – [tau] –> AC.On (I.Work, I.Operate)
224	In Phases: ⌈TRUE⌉ ⌈fuel<5⌉	consume: work=1 ∧ fuel>5 ∧ fuel' = fuel − 1	AC.On (I.Work, I.Operate) – consume –> AC.On (AC.Work.dtemp, I.Operate)
123			AC.On (AC.Work.dtemp, I.Operate) – dtemp –> AC.On (AC.Work.level, I.Operate)

Figure 7.14: Error traces obtained when verifying the full (7.14a) and the sliced (7.14b) untimed air conditioner specification with respect to property φ_1.

The error trace shows that the specification violates the verification properties due to the too weak precondition of method *consume*. The associated `enable` schema allows the method to take place even if the remaining fuel supply has dropped down to a value just over five fuel units (*fuel* > 5). Schema `effect_`*consume* in turn decreases the fuel supply by one unit, such that afterwards it might be below the safety critical limit of five fuel units required by the verification properties.

The number of transitions contained within the resulting TCS models, along with the runtimes needed to complete the verification tasks with the model checkers ARMC and SLAB are depicted within the column "Full" of Table 7.3.

Property	**Dimension**	**Full**	**Slice**	**%**
φ_1	model size [TCS transitions]	317	197	62
	ARMC runtime [s]	0.28	0.16	57
	SLAB runtime [s]	540.57	247.94	46
φ_2	model size [TCS transitions]	571	355	62
	ARMC runtime [s]	0.52	0.30	58
	SLAB runtime [s]	1483.62	675.85	46

Table 7.3: Model sizes and verification runtimes for the untimed air conditioner system specification and its slices with respect to verification properties φ_1 and φ_2. Additionally, percentage figures relative to the full system are given for each of the slices.

When comparing the sizes of the TCS models generated for properties φ_1 and φ_2, we see that the additional restriction *work* $= 1$, imposed by verification property φ_2, leads to an additional overhead of the resulting model size. This enlargement is sensible, since due to the additional restriction the test formula is translated into a resulting larger phase event automaton, such that also the final transition constraint system must have additional transitions.

The same verification runs have then been carried out after computing the slices of the specification with respect to φ_1 and φ_2. Apart from the partially different transition numbers, the only difference in the error trace depicted in Figure 7.14b is the additional restriction *mode* $= 1$ of the `Init` predicate of the full specification that is not present in its slice, since variable *mode* has been removed from the specification.

The resulting model sizes and runtimes for ARMC and SLAB verification are given in column "Slice" of Table 7.3. Additionally, column "%" lists percentage figures relative to the model sizes and runtimes obtained for the full version of the

specification.

When comparing the figures of the full specification with those of their slices, we see that, regardless of the verification property, slicing leads to a considerable reduction down to about 60% in terms of model size and down to around 50% in terms of required verification time.

7.3.4 Timed Air Conditioner System

The CSP-OZ-DC specification of the timed air conditioner system that we have introduced in Chapter 3 of this thesis as a running example of CSP-OZ-DC specifications has been modelled within Syspect. Parts of the resulting Syspect model have already been presented in illustrating screenshots contained within the previous sections of this chapter: Figure 7.2 shows its class diagram, Figure 7.3 shows the state machine of the air conditioner class *AC*, Figure 7.5 shows the DC part of *AC*, Figure 7.4 shows the component diagram of the system, and Figure 7.7 shows the error trace of a verification run of the full specification with respect to verification property $\varphi_2 \equiv \neg\Diamond\lceil \mathit{fuel} < 5 \wedge \mathit{work} = 1\rceil$.

Table 7.4 lists the results of verification runs on the transition constraint systems resulting from verification exports of only the air conditioner class as well as the complete air conditioner system and slices thereof with respect to the verification properties

$$\varphi_1 \equiv \neg\Diamond\lceil \mathit{fuel} < 5\rceil \quad \text{and} \quad \varphi_2 \equiv \neg\Diamond\lceil \mathit{fuel} < 5 \wedge \mathit{work} = 1\rceil.$$

For comparison, the figures for the untimed air conditioner class ("Unt. *AC*") of the previous section are included in the table. All verification runs lead to essentially the same error trace that violates the verification property with an occurrence of method *consume* leading to a decrement of variable *fuel* below the required minimum level due to the too weak precondition of the `enable_`*consume* schema.

When comparing the model sizes generated for verification of the timed and the untimed air conditioner class with respect to verification properties φ_1 and φ_2, the following anomaly stands out: For the untimed class, the stronger property φ_2 leads to a considerably larger model, while for the timed class, using the stronger property for verification leads to a transition constraint system with less transitions. Obviously, the larger test automaton resulting from the additional requirement within φ_2 causes a considerable reduction in conjunction with the phase event automaton resulting from the DC part of the timed air conditioner class when computing the global product automaton of all involved classes.

The effect of slicing, however, is consistent for all instances of the single air conditioner class: in each case, the model size as well as the ARMC verification runtime is reduced down to little more than half of that of the full class. The benefit achieved for the SLAB verification runtime is even slightly larger with a reduction down to little less than half of that of the full class.

Specification	Prop.	Model Size		ARMC Runtime		SLAB Runtime	
		# trans.	%	sec	%	sec	%
Unt. *AC*	φ_1	317	100	0.28	100	541	100
Unt. *AC*-Slice	φ_1	197	62	0.16	57	248	46
Unt. *AC*	φ_2	571	100	0.52	100	1484	100
Unt. *AC*-Slice	φ_2	355	62	0.30	58	676	46
Timed *AC*	φ_1	3198	100	4.10	100	7219	100
Timed *AC*-Slice	φ_1	1854	58	2.30	56	3237	45
Timed *AC*	φ_2	1600	100	2.08	100	3195	100
Timed *AC*-Slice	φ_2	928	58	1.18	57	1501	47
System	φ_1	228206	100	699.19	100	oom	n/a
System-Slice	φ_1	22686	10	27.51	4	oom	n/a
System	φ_2	114104	100	304.79	100	oom	n/a
System-Slice	φ_2	11344	10	12.28	4	oom	n/a

Table 7.4: Experimental results for the untimed and the timed air conditioner class (*AC*), the full timed air conditioner system (*System*), and their slices with respect to verification properties φ_1 and φ_2. For each model, the size is given in terms of transitions of the transition constraint system. For each verification task, the verification runtime is given in seconds needed to discover an error trace. The effect achieved by slicing is given in terms of percentage relative to the model size and verification runtime of the full models. The notation "oom" indicates that the model checker ran out of memory. In these cases no reduction factor can be given (n/a).

An even more impressive effect is achieved for the complete timed air conditioner system, comprising the air conditioner class *AC* and its environment class *Env*: there slicing reduces the model size down to only 10% of the original specification. As expected, this reduction is mainly achieved by the removal of large parts of the environment class such as the aspect of modelling the lighting situation that is not relevant with respect to the given properties.

This reduction in model size results in an even larger acceleration in ARMC verification where the error trace is found 25 times faster than before. For the model checker SLAB, however, all of the resulting transition constraint systems of the complete timed model, including the sliced versions, are too large, such that no successful verification can be carried out.

7.3.5 Elevator

As another example of a CSP-OZ-DC specification, an elevator has been modelled within Syspect. We use the associated state machine that is depicted in Figure 7.15 to introduce the functionality of the underlying system.

The elevator's state machine corresponds to the CSP part of the elevator class

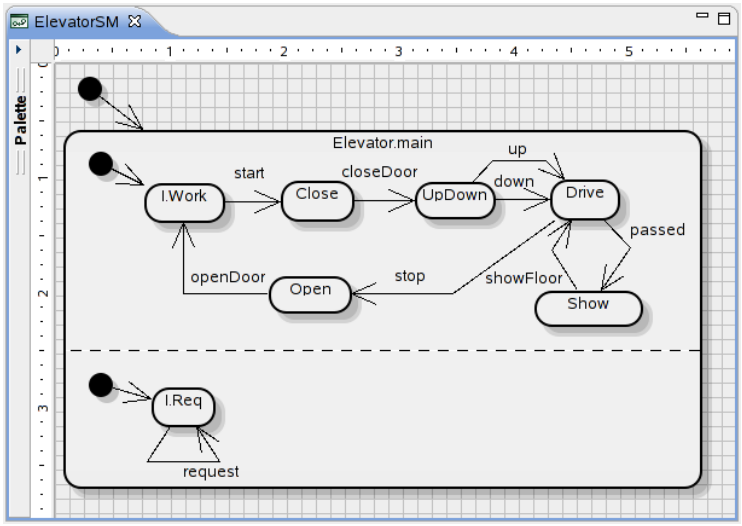


Figure 7.15: State machine of the elevator class.

and thus defines the elevator's behaviour in terms of the admissible ordering of events.

Upon its activation with event *start*, the elevator closes its door (*closeDoor*) and decides afterwards, whether it starts moving upwards (*up*) or downwards (*down*). Whenever the elevator passes a floor (*passed*), the elevator's display shows the current floor number (*showFloor*) by turning on an associated light. When the elevator has finally reached its goal, it stops (*stop*), opens its door (*openDoor*), and starts again with a new working cycle.

Interleaved with the main working cycle is another cyclic process, which models the non-deterministic choice of a destination floor (*request*). Such a request can happen at any time, whereupon the desired floor number is added to the sequence of requests that the elevator processes in the order of their insertion.

Initially, the elevator is at the lowest floor (which is simultaneously the initial target floor), there are not yet any requests, and the doors are open.

Furthermore, the elevator contains the following two real-time requirements:

- $\mathit{DoorOpen} \equiv \neg\Diamond(\updownarrow \mathit{openDoor} \frown \ell \leq 30 \frown \updownarrow \mathit{start})$
  requires that the door must not remain open for less than 30 time units.

- $\mathit{Restart} \equiv \neg\Diamond(\updownarrow \mathit{stop} \frown \lceil \mathit{request} \neq \varnothing \rceil \wedge \boxminus \mathit{up})$
  requires that, after stopping, the elevator must not move upwards while there are no pending requests.

With respect to verification of the elevator model, a similar problem as for the previously introduced Tic-Tac-Toe specification arises: the elevator specification

contains operations on sequences such as the predicate $requests' = requests \frown f?$ that models the extension of the requests queue by a new request $f?$ within schema `effect_`*requests*. Due to these operations that are not yet supported by the verification back-end of Syspect, we can currently not carry out any verification tasks for the elevator specification.

The computation of slices, however, is no problem, such that slices have been computed for the following verification properties:

- $NoReach \equiv \neg\Diamond(\lceil status = closed \rceil \land \ell > 42)$
  This formula describes that the elevator must not keep its door closed for more than 42 time units, i.e., reach its target within at most 42 time units.

- $NoStart \equiv \neg\Diamond(\updownarrow request \frown \lceil pos = tar \rceil \land \ell > 42)$
  This formulae describes that, upon the occurrence of a request, the elevator must not stay at the current floor for more than 42 time units.

Note that the constant 42 is used as a workaround for a parametric value, which is not implemented in the test formula parser of Syspect. With respect to slicing, however, it does not matter, whether the time bound is given as a numeric constant or as a symbolic variable.

The results obtained from the slice computations are summarised in Table 7.5.

Specification Elements	**Full**	*NoReach*- **Slice**	**%**	*NoStart*- **Slice**	**%**
Interface	9	8	89	6	67
Variables	5	4	80	3	60
Init predicates	6	4	67	3	50
`enable` schemas	5	5	100	5	100
`effect` schemas	7	6	86	4	57
Method predicates	16	14	88	12	75
DC formulae	2	2	100	2	100
Total # spec. elements	50	43	86	35	70

Table 7.5: Slicing results for the elevator specification. The absolute numbers of specification elements have been manually counted. For each of the slices, percentage figures relative to the numbers of the full specification are given.

For both properties, slicing is able to remove all specification elements that are related to displaying the current floor. This result is sensible, since the removed

specification parts purely address output functionality and thus do not have any impact on the given properties.

In addition to this base reduction, slicing with respect to property *NoStart* reduces the specification in another aspect, namely by eliminating all components that model opening and closing of the elevator door. Also this outcome is sensible, since the handling of the door is a feature of the elevator that is actually independent from its main functionality. This holds in particular with respect to the verification property *NoStart*, which relates the occurrence of requests with the subsequent states of the current and the destination floor.

When slicing with respect to property *NoMove*, the situation is different, since this property contains an explicit reference to the status of the door. Consequently, the concerned specification elements then need to remain in the resulting specification slice.

To summarise the results obtained from the elevator specification, slicing achieves a considerable reduction of the specification. However, both DC formulae as well as all `enable` schemas need to remain in the slice due to the close coupling of all involved parts of the specification.

7.3.6 ETCS-EM Case Study

The European Train Control System (ETCS) is an international standard [ERT02], which has been jointly developed by national European rail services in order to replace national train control systems by one harmonised framework. Thus, one of the primary aims of the ETCS standard is to enable trains to move between different European countries without the need to provide different versions of train-borne equipment for communication with country-specific infrastructure.

Currently, train localisation, speed detection, and integrity checks still mainly rely on sensors installed along railway tracks. These are not required anymore in the final ETCS implementation level. Instead, radio block centres (RBC) are associated to dedicated railway segments, on which they have to supervise trains via wireless communication.

One of the most safety-critical aspects of the communication between an RBC and its supervised trains is the treatment of emergency messages. If a leading train detects an emergency situation and needs to perform an emergency brake, it is essential that its following trains receive emergency information in time, such that collisions are impossible to occur.

This aspect of the ETCS standard has been modelled as a CSP-OZ-DC specification by Johannes Faber within AVACS R1 [Fab07]. The primary intention of this development was to conduct a realistic case study of an industrially relevant system, which could be used on the one hand for the evaluation of CSP-OZ-DC as a specification notation. On the other hand, the case study was the basis for the assessment and further development of automatic verification techniques within

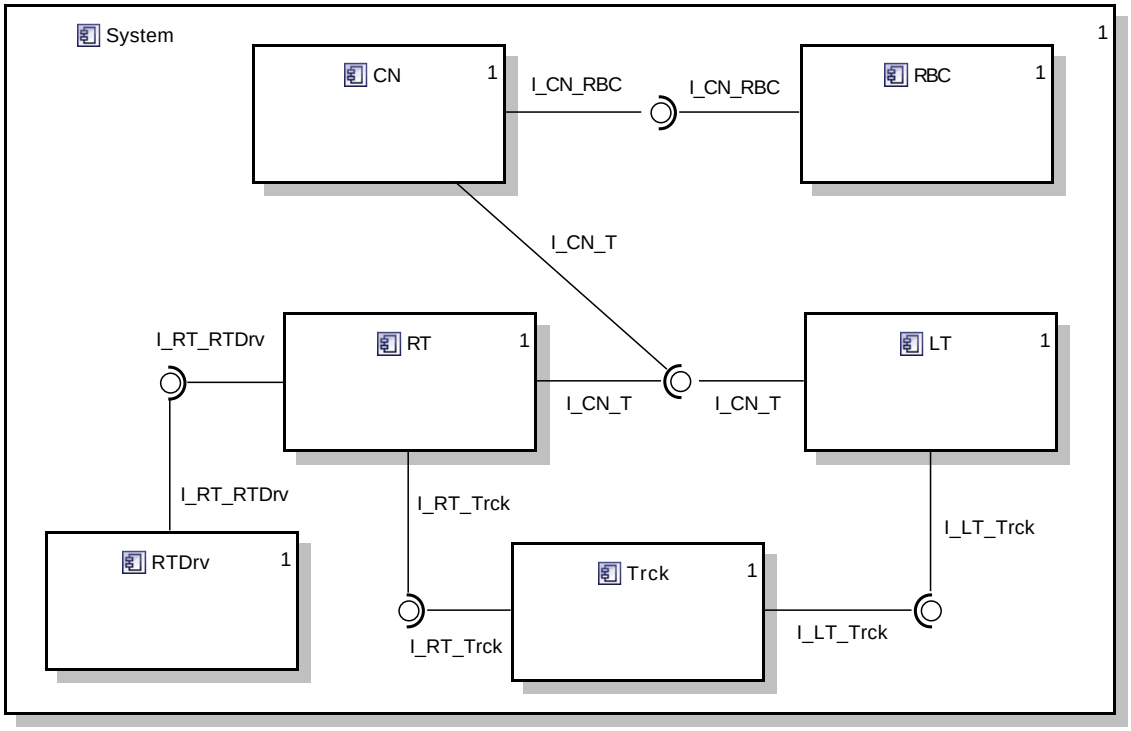


Figure 7.16: The ETCS-EM system composition as defined within Syspect in the main component diagram of this specification.

AVACS R1.

The resulting CSP-OZ-DC specification ETCS-EM has successfully been verified with respect to the top-level property of collision freedom between trains [FM06, MFR06]. However, the CSP-OZ-DC specification has not been fully automatically translated into transition constraint systems. Instead, the phase event automata semantics of the specification has first been manually constructed within Moby-PEA [Cor07c], the graphical development environment for phase event automata. Afterwards, the resulting phase event automata have been automatically translated into transition constraint systems.

Furthermore, the complete specification was not tractable by direct automatic verification. Instead, a compositional approach has been applied: several sub-properties were verified for suitably decomposed parts of the system. Additional arguments then lead to the conclusion that also the global property of collision freedom is satisfied.

In order to assess the application of slicing to the ETCS-EM specification, it has been modelled within Syspect, such that the slicing plug-in can be used to compute slices with respect to some of the sub-properties.

Figure 7.16 shows the global system composition as defined by the component

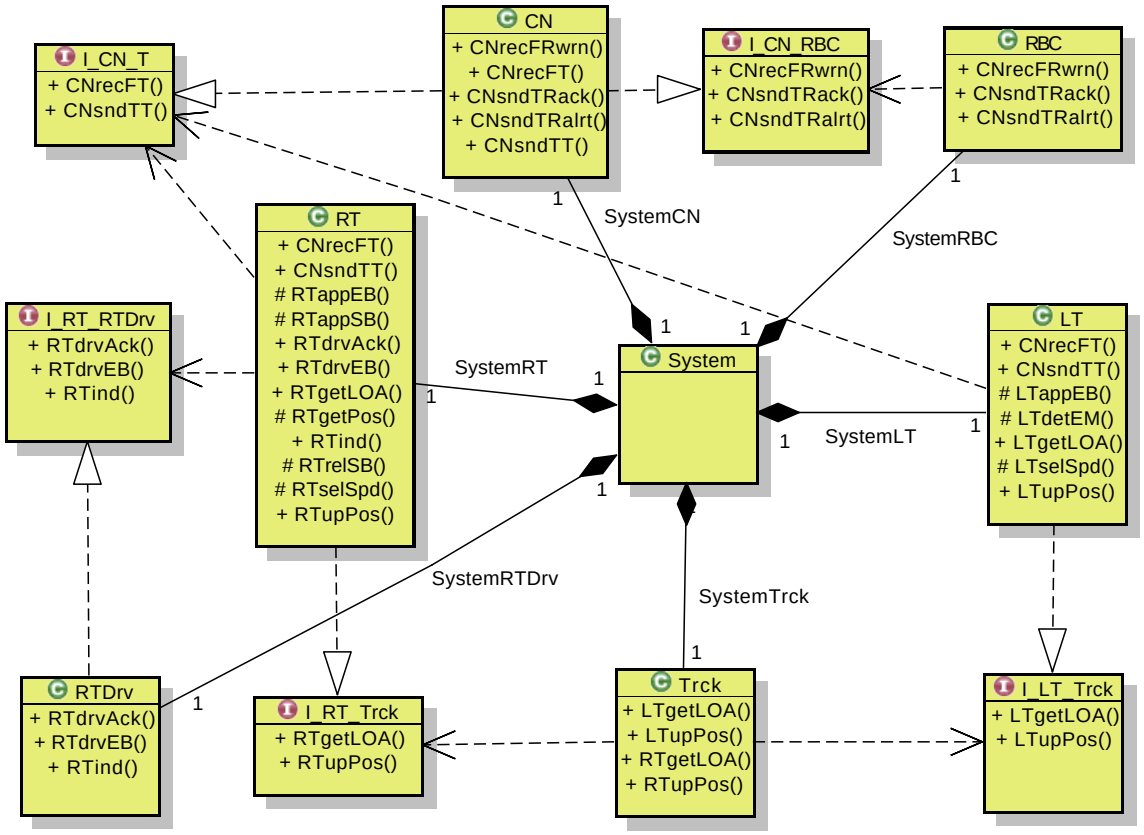


Figure 7.17: The classes and their mutual associations comprised by the ETCS-EM specification as defined within Syspect.

diagram of the ETCS-EM specification. In the centre of the specification are both involved trains, the leading train (class *LT*) and the following rear train (class *RT*). Both of these communicate with the railway track (class *Trck*) and the communication network (class *CN*) that models the wireless communication with the RBC (class *RBC*). Only the rear train has a driver (class *RTDrv*), since the leading train is only of interest with respect to its detection of an emergency situation, which is independent from its driver.

All classes involved in the component diagram are also shown in the specification's class diagram depicted in Figure 7.17. Moreover, the class diagram contains details of the interface of each of the classes, including the methods they offer or require, respectively.

The original CSP-OZ-DC specification of the ETCS-EM system as developed by Johannes Faber contains several sophisticated constructs of CSP-OZ-DC that are not directly expressible within Syspect. However, workarounds have been found for representing all of the intended features of the specification. In detail, the following problems occurred, when modelling the ETCS-EM specification within Syspect:

- Within the CSP part of the classes of the original CSP-OZ-DC specification communication between CSP events is modelled via parameter variables such as variable *id* in the following process definition within class *RBC*:

  $$HandleEM \stackrel{c}{=} send.Warning.id \rightarrow receive.Ack.id$$

  The state machines that are employed within Syspect for modelling the CSP part do not offer a direct possibility to express the link between both references to *id*. Therefore, auxiliary class variables are introduced in order to express this local form of communication between CSP events.

- Another feature of CSP-OZ-DC that is used by the ETCS-EM specification are conditional statements within the CSP part such as in the following process definition that appears in class *RT*:

  $$\begin{aligned} Running \stackrel{c}{=}\ & updatePosition.ID?pos \rightarrow getLOA.ID?loa \\ & \rightarrow computeSBI!loa?sbi \rightarrow \\ & \textbf{if } sbi \leq pos \\ & \textbf{then } applySB \rightarrow selectSpeed \rightarrow Running \\ & \textbf{else } releaseSB \rightarrow selectSpeed \rightarrow Running \end{aligned}$$

  Again, the state machines of Syspect do not contain any equivalent construct. As a workaround, the initial CSP events *applySB* and *releaseSB* of both branches are enriched with suitable `enable` schemas containing the branching condition and its negation, respectively, as a predicate. Instead of the conditional construct, the branching is then modelled as an external choice between both processes.

- Variables used within a Syspect project have global scope. Therefore, variables of different classes that have the same name, such as for instance *currentPosition* within classes *LT* and *RT* need to be renamed such that no interference is possible. The extension consists of a prefix according to the parent class, i.e., instead of *currentPosition*, the variables *LTcurrentPosition* and *RTcurrentPosition* are used.

- As already mentioned, the DC formula editor of Syspect does currently not allow symbolic variables as time bounds, such as the time bound *updateBound* that appears in the following DC counterexample formula of class *LT*:

  $$\neg(true \frown \updownarrow updatePosition \frown \ell > updateBound \frown updatePosition)$$

  Therefore, instead of *updateBound*, an arbitrary but unique numerical value is used within the concerned DC formula in the Syspect model.

Slices have been computed with respect to the six test formulae that were used by Faber [Fab07] to establish local properties of the complete specification. The results of these slice computations are summarised in Table 7.6 that shows the absolute numbers of syntactical specification elements such as interface elements (channels, methods), state variables, predicates of the `Init` schema, `enable` and `effect` schemas, predicates (or rather conjuncts) within the latter, and DC formulae. These numbers have been obtained by counting the particular specification elements and summarising them over all involved classes of the specifications. Additionally, percentage figures relative to the full specification are given for each of its slices.

Specification Elements	**Full**	TF_1- **Slice**	**%**	TF_2- **Slice**	**%**	TF_3- **Slice**	**%**	TF_4- **Slice**	**%**	TF_5- **Slice**	**%**	TF_6- **Slice**	**%**
Interface	50	36	72	36	72	27	54	36	72	37	74	37	74
CSP lines	35	29	83	29	83	23	66	29	83	30	86	30	86
Variables	29	18	62	13	45	2	7	14	48	13	45	15	52
State Invariants	5	5	100	5	100	0	0	5	100	5	100	5	100
Init predicates	15	12	80	10	67	0	0	9	60	10	67	10	67
`enable` schemas	3	1	33	1	33	0	0	1	33	1	33	1	33
`effect` schemas	26	10	38	7	27	9	35	8	31	7	27	8	31
Method predicates	55	21	38	17	31	10	18	18	33	17	31	19	35
DC formulae	9	9	100	9	100	7	78	9	100	9	100	9	100
Total # elements	227	141	62	127	56	78	34	129	57	129	57	134	59

Table 7.6: Slicing results for the ETCS-EM specification. The size of the slice for each test formula is given in terms of the absolute number of remaining specification elements and as percentage figures relative to the numbers of the full specification.

In most of the cases, slicing with respect to the sub-properties achieves a reduction down to about 60% of all considered specification elements, i.e., about 40% of the original specification is removable. This result is impressive, but still considerably larger than the decomposed models that Faber successfully used for verification. However, the same decomposition can of course also be applied for the specifications generated by Syspect.

Unfortunately, direct automatic verification of the ETCS-EM specification from within Syspect is currently not feasible, since even for decomposed versions of the sliced specification the generated transition constraint systems all have a size of several Gigabytes, containing millions of TCS transitions which is simply too large

to be tractable by the model checkers. Therefore, we currently cannot comment on verification runtimes for this specification and its slices.

7.3.7 Airport Specification

Modelling of CSP-OZ specifications with an early version of Syspect has been evaluated within a minor thesis at the University of Paderborn [Jak07]. Within his thesis, the author, Marcel Jakoblew, developed a very detailed specification of the flight control of a fictitious airport flight control system.

The resulting Syspect model and the generated CSP-OZ specification consists of 14 classes, ranging from airport infrastructure such as runway and control tower over involved personnel such as control tower operators, customs and police officers up to airport and aircraft equipment such as instrument landing systems and fuel hydrant dispenser.

The top-level structure of the complete airport flight control system is depicted in the component diagram in Figure 7.18. The resulting LaTeX mark-up of the CSP-OZ specification generated from the Syspect model comprises 25 pages.

Before slicing could be applied to the specification, the following modifications of the original specification were necessary:

- Due to the global scope of variable names within Syspect projects, variables of the same name that were present in different classes had to be renamed in order to be unique within the Syspect airport project. A renaming convention was used that attached an abbreviated name of the containing class as a prefix to the existing variable names.

- Several state machines of the original model contained transitions leaving hierarchical states that were labelled with trigger events. Due to the implemented UML profile, such transitions are translated into sequential composition of the process representing the hierarchical state with a CSP `Stop` operator. Since this was certainly not intended, these transitions were replaced by unlabelled transitions leading to an additional simple state, from where transition edges with the previously removed triggers then leave to the targets of the original transitions.

The original specification as well as its modified version are available within the Syspect repository [Cor07e].

Slices of the airport specification have been computed with respect to the following properties:

RunwayCollision:

$$TF_{RC} \equiv \neg\Diamond(\updownarrow \mathit{RunwaySperren} \frown \boxminus \mathit{RunwayFreigeben} \frown \updownarrow \mathit{RunwaySperren})$$

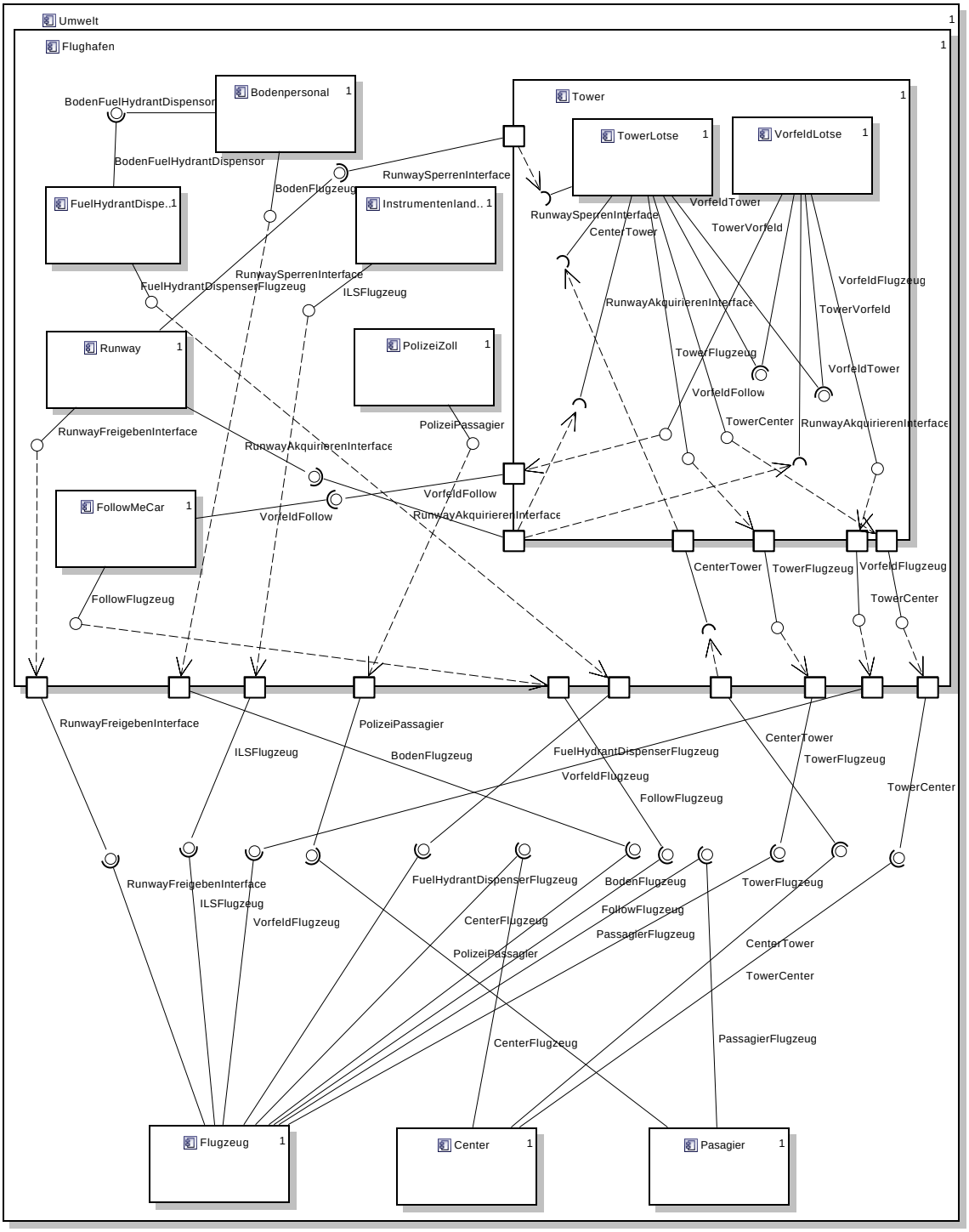


Figure 7.18: The airport flight control system composition as defined within Syspect in the main component diagram of this specification.

This property describes an undesired situation where the runway is blocked (*RunwaySperren*) at two consecutive points in time without an intermediate clearance (*RunwayFreigeben*). Since blocking of the runway is required before it may be used by an aircraft, this scenario possibly leads to the dangerous scenario of two aircrafts simultaneously using the same runway, giving rise to the risk of collision between them.

OverFull:

$$TF_{OF} \equiv \neg\Diamond(\lceil Tankinhalt = Tankkapazitaet \rceil \frown \updownarrow BetankungAnfordern)$$

This property describes an undesired situation where the fuel tank of the aircraft is already filled up to its maximum capacity ($\lceil Tankinhalt = Tankkapazitaet \rceil$) and, in spite of that, an additional fuelling is initiated ($\updownarrow BetankungAnfordern$).

UnsafeBoarding:

$$TF_{UB} \equiv \neg\Diamond(\lceil PersonSicher = false \rceil \frown \updownarrow Boarding)$$

This property describes an undesired situation where a person that has not yet passed the security checks and that is thus possibly dangerous ($PersonSicher = false$) is allowed to board the aircraft ($\updownarrow Boarding$).

The results of these slice computations are summarised in Table 7.7 that contains the numbers of specification elements of the full specification and each of its slices, given as absolute figures and, for the slices, also as percentage figures relative to the full specification. The specification elements that have been counted are interface elements (channels and methods), lines of CSP process definitions, state variables, predicates (or rather conjuncts) of `Init` schemas, `enable` and `effect` schemas, and predicates of these. In each case, these specification elements have been summarised for all of the 14 involved classes of the specification.

Slicing with respect to each of the considered test formulae achieves a reduction of the specification down to about 75% of the size of the original specification in terms of the number of counted specification elements. In comparison to the previously presented case study, this is a relatively high number, obviously due to the close coupling between the involved classes that becomes obvious when inspecting the generated dependence graph of the specification: most of the methods offered by the classes need to synchronise with other classes, leading to synchronisation control dependence between them that prevents the concerned methods from being removable.

Specification Elements	**Full**	TF_{RC}- **Slice**	**%**	TF_{OF}- **Slice**	**%**	TF_{UB}- **Slice**	**%**
Interface	188	138	73	141	75	146	78
CSP lines	133	123	92	125	94	129	97
Variables	39	19	49	21	54	21	54
Init predicates	7	5	71	5	71	6	86
`enable` schemas	47	42	89	44	94	43	91
`effect` schemas	51	21	41	26	51	23	45
Method predicates	131	84	64	91	69	88	67
Total # elements	596	432	72	453	76	456	77

Table 7.7: Slicing results for the airport specification. The size of the slice for each test formula is given in terms of the absolute number of remaining specification elements and as percentage figures relative to the numbers of the full specification.

The greatest reduction is obtained by slicing with respect to property "Runway-Collision". The resulting slice contains only 72% of the specification elements of the full specification, i.e., nearly 30% of the specification were removable. In contrast to the other test formulae, this property allows reductions within the classes modelling ground personal (Bodenpersonal), passenger (Passagier), fuel hydrant dispenser (FuelHydrantDispenser), and environment (Umwelt), since these contain elements such as the details of the different required checks of the aircraft that do not affect the verification property.

Property "OverFull", on the other hand, is affected by the class FuelHydrant-Dispenser that, in turn, needs to synchronise with class Bodenpersonal, resulting in additional specification elements remaining in the slice. Similarly, property "UnsafeBoarding" is affected by the class Passagier, which again leads to additional specification elements remaining in the slice. Both of these slices are supersets of the slice with respect to property "RunwayCollision". This can again be explained with the close coupling of the involved classes. Everything that affects property "RunwayCollision" will probably be part of any arbitrary slice due to the tight synchronisation between all involved specification elements.

Further reductions might be achieved by additional arguments showing that some of the involved synchronisations are always possible such that the resulting dependences might be removed. Similar research with respect to the removal of control dependences has been carried out [BMW06]. This approach, however, has currently not yet been automated, such that we leave it open as future work, as also discussed in the concluding Chapter 8.

As stated for the previous case study of the ETCS-EM system, verification runs for

the airport specification with respect to the presented properties were not possible, mainly due to the massive size of the project. Even the smallest slice still comprises 22 pages of LaTeX mark-up, such that Syspect even fails to generate the specification's semantics in terms of their phase event automata product, not to mention the resulting transition constraint system. Another important problem is the hierarchical structure of the airport specification that employs several component diagrams with several hierarchy levels of nesting (Umwelt → Flughafen → Tower), which is not yet supported by the PEA and ARMC export of Syspect. This latter problem can be circumvented by exporting only base classes, leaving out all component diagram classes (Umwelt, Flughafen, Tower). However, even this "flattened" export fails with insufficient memory due to the massive size of the specification.

7.4 Summary of Experimental Results

The presented experiments document the validity of the initial hypothesis of this thesis, claiming that slicing can be used as a valuable tool in mitigating the problem of state space explosion in the context of automatic verification of integrated formal specifications. This observation becomes most obvious at the examples where verification runs are indeed possible, i.e., for the cash register specification and for the untimed and timed air conditioner specifications that contain several system aspects which are partly independent from each other and which are, furthermore, not all relevant with respect to the considered verification properties. This situation can obviously not yet be recognised and exploited by the model checkers, such that slicing as a pre-processing step achieves indeed a considerable reduction as well in verification runtime as in the space required for representing the resulting verification models on the level of transition control systems.

The case studies of the ETCS-EM system and the airport specification, however, clearly illustrate that slicing alone will not be sufficient to tackle larger specifications, but must be complemented by additional approaches, for instance those offered by compositional verification.

However, the experiments also illustrate that, even if specifications are currently too large for successful automatic verification, the slicing approach along with the artefacts produced during slicing, such as the program dependence graph, are nevertheless useful with respect to different purposes, such as human analysis and comprehension of the specification.

8 Conclusion

This chapter summarises the results of this thesis and discusses possible perspectives of future work.

8.1 Summary

Within this thesis we have developed an approach of computing slices of integrated formal specifications. Our slicing approach is dedicated to the context of automatic verification. The main motivation of applying slicing in this context is the problem of state space explosion that frequently arises during model checking. Slicing is used as one of several possible countermeasures against this problem, allowing us to obtain reduced versions of specifications that are custom-tailored to the verification property at hand. This reduction allows verification methods to focus on the part of the specification that is essential for the given property and to simultaneously ignore any specification element without influence on the verification result.

Since the technique of slicing has its origin in the field of program analysis, we started in Chapter 2 with providing some background on the extensive related work on slicing. Furthermore, we discussed alternative techniques for lessening the problem of state space explosion along with their relation to slicing.

In Chapter 3, we then incrementally introduced the *specification notations* Object-Z, CSP-OZ, and CSP-OZ-DC that form our slicing targets. At the core of the slicing approach is the analysis of such specifications with respect to various types of dependences that can be identified between specification elements, such as *predicate*, *control*, *data*, *synchronisation*, and *timing dependence*. All of these along with several sub-types have been defined in Chapter 4.

The resulting *dependence graph* is the basis for the actual slice computation, based on a graph reachability analysis that uses the verification property as its starting point. From the obtained sets of reachable and unreachable dependence graph nodes we can then compute *specification slices* as we showed in Chapter 5.

In our application context of verification, the *correctness* of the presented slicing approach is essential. Therefore, we showed correctness of slicing in Chapter 6 in the sense that we obtain the same verification result, regardless whether verification is carried out on the original specification or on its slice with respect to the given verification property.

The correctness proof consists of two main steps: First, we showed slicing to guarantee the existence of a *projection relation* between specifications and its slices.

Second, we showed the particular logic that is used to express verification properties to be *stuttering invariant*, i.e., to be unable to distinguish between specifications and their slices. We showed the latter for two different logics, namely on the one hand for *test formulae* for expressing real-time properties of specifications, and on the other hand for *state/event interval logic formulae* for expressing properties in the discrete setting of Object-Z and CSP-OZ specifications.

Finally, we presented in Chapter 7 the graphical modelling environment for CSP-OZ-DC specifications, *Syspect*, as the platform for experimental evaluation of our slicing approach by means of its implementation as the Syspect *slicing plug-in*. Furthermore, we gave an account on the application of slicing to several case studies and the results obtained from slicing. These experiments show that in most cases slicing yields a considerable reduction of the target specification. Where verification runs are not prevented by unsupported data types or the massive size of the specifications and even its slices, the application of slicing also leads to remarkable speed-ups in verification runs. With respect to verification of the larger case studies, however, we had to conclude that slicing alone is not sufficient for successfully carrying out automatic verification, but additional methods will be necessary such as compositional approaches to verification.

8.2 Perspectives

Although the slicing approach presented in this thesis is already applicable to a large set of specifications, there are several topics offering themselves as starting points for future work.

Enhanced Dependence Analysis As proposed by [BMW06], additional calculations during dependence analysis allow us to *remove* certain control dependence edges from the dependence graph, possibly leading to smaller and thus more precise slices. These enhancements can be further improved by automation of the required auxiliary calculations by using theorem proving techniques. Furthermore, it might be possible to remove additional types of dependences in a similar way, for instance synchronisation dependence edges, provided it can be shown that the involved methods always successfully synchronise.

Weakening Slicing Exactness The main constraint for the development of the slicing approach presented in this thesis is its *exactness* in the sense that a specification satisfies a verification property if and only if its slice with respect to the verification property satisfies it.

The approach of [Weh06] proposes a slicing variant for the verification of CSP-OZ specifications that drops this requirement of exactness. Instead, it allows the computation of *imprecise slices*, which are *incrementally* refined in a similar way as

in the framework of counterexample guided abstraction refinement. The iteration proceeds until a slice has been obtained that either satisfies the verification property or leads to a valid error trace. It is certainly worthwhile to extend this approach to CSP-OZ-DC specifications. Furthermore, suitable tool support would be needed for evaluating the efficiency.

Additional Application Contexts The application of slicing within this thesis has mainly been dedicated to lessening the problem of state space explosion within automatic verification. However, different application contexts are, of course, also possible.

One such example is to apply slicing for *verification re-use* [Weh05, OW05]. This application is still situated in the context of automatic verification. However, instead of directly aiming at the state space explosion problem, its goal is to omit complete verification runs for modified specifications, provided that slicing techniques lead to the conclusion that the result of a previous verification run still holds for the modified specification.

The application of slicing for *compositional verification* has already been mentioned within Chapter 7 of this thesis at the example of verification of the ETCS-EM case study, where slicing was used to compute decompositions of the original specification with respect to a set of sub-properties, derived from the top-level verification property. However, modifications of the slicing approach, such as the imprecise slicing proposed by [Met07] or the integration with methods of program conditioning, might further improve the results, when using slicing as a decomposition technique.

Another completely different example of the application of slicing formal specifications is proposed by Bollin [Bol04], who uses slicing for supporting *human comprehension* of complex specifications, which certain also applies to CSP-OZ-DC specifications.

Slicing at Different Stages The application of slicing at the lower level of *transition constraint systems* has been shown to be extremely useful in combination with techniques of automatic deductive verification [BDFW07, Bri07].

The experiments documented within this thesis show that this result even holds when slicing has previously been applied at a higher level of the verification work flow. Thus, an interesting direction of research is that into the application of slicing at some *intermediate level*. For instance, an additional slice computation at the semantic level of phase event automata before computation of their product automaton might result in further reductions on the way down to transition constraint systems.

Slicing Implementation The experiments presented in this thesis have been obtained from applying the slicing approach in its implementation as a Syspect plug-in. Although being very usable, the plug-in is of course a prototype and thus some limitations apply, which offer several very straightforward possibilities for future extensions:

- *Variable Scope*: Although this problem is mainly due to the current implementation of Syspect, it can also be solved by an extension of its slicing plug-in. Any variable used within a Syspect specification has a scope that is global to the specification. Thus, variables with class scope need to be made unique by applying a suitable naming convention. A more comfortable solution would require the slicing plug-in to automatically identify the correct variable scope and to reflect this correctly within the dependence graph.

- *Dependence Graphs*: Currently, control flow and dependence graphs are statically computed without offering any automated connection to their source specification elements. A desirable feature for easier analysis of specifications would be to establish a link between nodes of the graphs and their associated specification elements. To this end, the currently external graph visualisation would need to be implemented within Syspect.

- *Slicing Report*: The figures presented as experimental results within Chapter 7 of this thesis have been manually extracted from specifications by counting the specification elements of interest. An obvious enhancement to the slicing report would be to not only list the removable specification elements, but also to summarise and compare such absolute and relative figures for the specification and its slice.

- *Re-Import of Slices*: Currently, reduced versions of specifications can only be computed during specification exports. A desirable extension of the slicing plug-in or rather of Syspect itself would be the possibility to re-import sliced specifications into the graphical modelling environment of Syspect. To this end, an extension of Syspect with suitable import facilities would be required.

- *Slices of Syspect Projects*: Another possibility to reach the same goal would be to compute slices directly on the Syspect model without the need to carry out an intermediate export. To this end, however, slicing techniques for the involved UML profile need to be developed.

Finally, the current version of the CSP-OZ-DC specification language is planned to be extended by further DC operators such as the *integration* operator. Thus it will be necessary to adapt the presented slicing approach by carefully extending the current dependence analysis and the correctness proof in order to cover such additional specification elements of future extensions.

Bibliography

[ABGS+01] Árpád Beszedes, T. Gergely, Z. M. Szabó, J. Csirik, and T. Gyimothy. Dynamic Slicing Method for Maintenance of Large C Programs. In *CSMR '01*, pages 105–113. IEEE, 2001.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr06] J.-R. Abrial. Formal methods in industry: achievements, problems, future. In *ICSE '06*, pages 761–768. ACM, 2006.

[ADS93] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[Agr91] H. Agrawal. *Towards automatic debugging of computer programs*. PhD thesis, Purdue University, 1991.

[AH90] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*, pages 246–256. ACM, 1990.

[AH00] J. Ahn and T. Han. Static Slicing of a First-Order Functional Language based on Operational Semantics. Technical Report CS/TR-99-144, Korea Advanced Institute of Science and Technology (KAIST), 2000.

[AO97] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 1997.

[ASU97] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1997.

[AVA07] AVACS. Transregional Collaborative Research Center SFB TR 14 "Automatic Verification and Analysis of Complex Systems" (AVACS). `http://www.avacs.org/`, 2007.

[BA05] F. Badeau and A. Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *ZB '05*, volume 3455 of *LNCS*, pages 334–354. Springer, 2005.

[Bal93] T. J. Ball. *The use of control-flow and control dependence in software tools*. PhD thesis, University of Wisconsin-Madison, 1993.

[BCRS01] D. Binkley, R. Capellini, L. R. Raszewski, and C. Smith. An Implementation of and Experiment with Semantic Differencing. In *ICSM '01*, pages 82–91, 2001.

[BDFW07] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing Abstractions. In *International Symposium on Fundamentals of Software Engineering*, volume 4767 of *LNCS*, pages 17–32. Springer, 2007.

[BDG+04] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing Executable Dynamic and Forward Slicing. In *SCAM '04*, pages 43–52. IEEE, 2004.

[BE93] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93*, pages 509–518. IEEE, 1993.

[Bec93] J. A. Beck. *Interface slicing: a static program analysis tool for software engineering*. PhD thesis, West Virginia University, 1993.

[BFG00] M. Bozga, J.-C. Fernandez, and L. Ghirvu. Using Static Analysis to Improve Automatic Test Generation. In *TACAS '00*, number 1785 in LNCS, pages 235–250. Springer, 2000.

[BG96] D. Binkley and K. B. Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

[BG97] R. Bodík and R. Gupta. Partial Dead Code Elimination using Slicing Transformations. In *PLDI '97*, pages 159–170. ACM, 1997.

[BGL+00] S. Bensalem, V. Ganesh, Y. Lakhnech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An Overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196. NASA Langley Research Center, 2000.

[BGM01] M. Bozga, S. Graf, and L. Mounier. Automated validation of distributed software using the IF environment. In *NCA '01: IEEE International Symposium on Network Computing and Applications*, pages 268–274. IEEE, 2001.

[BGO02] V. A. Braberman, D. Garbervetsky, and A. Olivero. Improving the Verification of Timed Systems Using Influence Information. In *TACAS '02*, pages 21–36. Springer, 2002.

[BGO+04] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and Applications: the IF toolset. In *SFM-04:RT: 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, volume 3185 of *LNCS*, pages 237–267. Springer, 2004.

[BH93a] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *AADEBUG '93*, pages 206–222. Springer, 1993.

[BH93b] S. Bates and S. Horwitz. Incremental Program Testing Using Program Dependence Graphs. In *POPL '93*, pages 384–396. ACM, 1993.

[BH04] D. Binkley and M. Harman. A Survey of Empirical Results on Program Slicing. *Advances in Computers*, 62:105–178, 2004.

[BHL+96] J. P. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog, and A. P. Ravn. A ProCoS II Project Final Report: ESPRIT Basic Research Project 7071. *Bulletin of the European Association for Theoretical Computer Science*, 59:76–99, 1996.

[BHR95] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[Bin92] D. Binkley. Using Semantic Differencing to Reduce the Cost of Regression Testing. In *ICSM '92*, pages 41–50, 1992.

[Bin93] D. Binkley. Precise executable interprocedural slices. *ACM Lett. Program. Lang. Syst.*, 2(1-4):31–45, 1993.

[Bin95] D. Binkley. Reducing the Cost of Regression Testing by Semantics Guided Test Case Selection. In *ICSM '95*, pages 251–260. IEEE, 1995.

[Bin97] D. Binkley. Semantics Guided Regression Test Cost Reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997.

[Bin98] D. Binkley. The Application of Program Slicing to Regression Testing. *Journal of Information and Software Technology*, 40(11 and 12):583–594, 1998.

[Bin02] D. Binkley. An Empirical Study of the Effect of Semantic Differences on Programmer Comprehension. In *IWPC '02*, pages 97–106. IEEE, 2002.

[BMW06] I. Brückner, B. Metzler, and H. Wehrheim. Optimizing Slicing of Formal Specifications by Deductive Verification. *Nordic Journal of Computing*, 13(1–2):22–45, 2006.

[BO94] J. M. Bieman and L. M. Ott. Measuring Functional Cohesion. *IEEE Trans. Softw. Eng.*, 20(8):644–657, 1994.

[Bol04] A. Bollin. *Specification Comprehension — Reducing the Complexity of Specifications*. PhD thesis, University of Klagenfurt, 2004.

[Bri07] D. Brill. Deductive Model Checking with Transition Constraint Systems. Master's thesis, Saarland University, 2007.

[BRSH00] D. Binkley, L. R. Raszewski, C. Smith, and M. Harman. An Empirical Study of Amorphous Slicing as a Program Comprehension Support Tool. In *IWPC '00*, pages 161–170. IEEE, 2000.

[Brü07] I. Brückner. Slicing Concurrent Real-Time System Specifications for Verification. In *IFM '07*, volume 4591 of *LNCS*, pages 54–74. Springer, 2007.

[BS03] E. Börger and R. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer, 2003.

[BSV93] F. Balarin and A. L. Sangiovanni-Vincentelli. An Iterative Approach to Language Containment. In *CAV '93*, pages 29–40. Springer, 1993.

[BW05a] I. Brückner and H. Wehrheim. Slicing an Integrated Formal Method for Verification. In *ICFEM 2005*, volume 3785 of *LNCS*, pages 360–374. Springer, 2005.

[BW05b] I. Brückner and H. Wehrheim. Slicing Object-Z Specifications for Verification. In *ZB '05*, volume 3455 of *LNCS*, pages 414–433. Springer, 2005.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77*, pages 238–252. ACM, 1977.

[CC99] P. Cousot and R. Cousot. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[CCL98] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11–12):595–607, 1998.

[CDHR02] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera Specification Language. *STTT*, 4(1):34–56, 2002.

[CF94] J.-D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, 1994.

[CFR+99] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[CFR+02] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing for VHDL. *STTT*, 4(1):125–137, 2002.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV '00*, pages 154–169. Springer, 2000.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[CH96] J. J. Comuzzi and J. M. Hart. Program Slicing Using Weakest Preconditions. In *FME '96*, pages 557–575. Springer, 1996.

[Che93] J. Cheng. Slicing Concurrent Programs – A Graph-Theoretical Approach. In *Automated and Algorithmic Debugging*, pages 223–240. ACM, 1993.

[CLM89] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *Fourth Annual Symposium on Logic in computer science*, pages 353–362. IEEE, 1989.

[CLYK01] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01*, pages 605–609. ACM, 2001.

[Cor07a] Correct System Design Group. Correct System Design Group. `http://csd.informatik.uni-oldenburg.de/`, 2007.

[Cor07b] Correct System Design Group. ForMooS. `http://csd.informatik.uni-oldenburg.de/projects/formoos.en.html`, 2007.

[Cor07c] Correct System Design Group. Moby/PEA. `http://csd.informatik.uni-oldenburg.de/~moby/`, 2007.

[Cor07d] Correct System Design Group. PEA Toolkit. `http://csd.informatik.uni-oldenburg.de/projects/epea.html`, 2007.

[Cor07e] Correct System Design Group. Syspect Subversion Repository. `http://homer.informatik.uni-oldenburg.de/svn/syspect`, 2007.

[CR94] J. Chang and D. Richardson. Static and Dynamic Specification Slicing. In *Fourth Irvine Software Symposium*, pages 25–37, 1994.

[CX01] Z. Chen and B. Xu. Slicing Concurrent Java Programs. *ACM SIGPLAN Notices*, 36(4):41–47, 2001.

[Dav93] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[DBG+06] S. Danicic, D. Binkley, T. Gyimothy, M. Harman, A. Kiss, and B. Korel. A Formalisation of the Relationship between Forms of Program Slicing. *Science of Computer Programming*, 62(3):228–252, 2006.

[DDF+05] S. Danicic, D. Daoudi, C. Fox, M. Harman, R. Hierons, J. Howroyd, L. Ouarbya, and M. Ward. ConSUS: A Light-Weight Program Conditioner. *Journal of Systems and Software*, 77(3):241–262, 2005.

[DHH+06] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, R. Wallentine, and T. Wallentine. Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In *TACAS '06*, volume 3920 of *LNCS*, pages 73–89. Springer, 2006.

[dLFM96] A. de Lucia, A. R. Fasolino, and M. Munro. Understanding Function Behaviors through Program Slicing. In *IWPC '96*, pages 9–18. IEEE, 1996.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *CAV '04*, volume 3114 of *LNCS*, pages 496–500. Springer, 2004.

[DR00] R. Duke and G. Rose. *Formal object-oriented specification using Object-Z*. Macmillan, 2000.

[DS94] J. Davies and S. Schneider. *Theories and experiences for real-time system development*, chapter Real-time CSP, pages 31–82. World Scientific Publishing, 1994.

[Ecl07] Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2007.

[ELLL04] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed Explicit-State Model-Checking in the Validation of Communication Protocols. *STTT*, 5(2–3):247–267, 2004.

[ERT02] ERTMS User Group, UNISIG. ERTMS/ETCS System requirements specification. `http://www.aeif.org/ccm/default.asp/`, 2002.

[Fab07] J. Faber. R1 Benchmark: Emergency Messages in ETCS. `http://www.avacs.org/`, 2007.

[FDHH04] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Softw., Pract. Exper.*, 34(1):15–46, 2004.

[Fis97] C. Fischer. CSP-OZ: A Combination of Object-Z and CSP. In *FMOODS '97*, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2000.

[FM06] J. Faber and R. Meyer. Model Checking Data-Dependent Real-Time Properties of the European Train Control System. In *FMCAD '06*, pages 76–77. IEEE, 2006.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[FOW01] C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In *FASE '01*, volume 2029 of *LNCS*, pages 91–108. Springer, 2001.

[FRT95] J. Field, G. Ramalingam, and F. Tip. Parametric Program Slicing. In *POPL '95*, pages 379–392. ACM, 1995.

[GH97] R. Gerber and S. Hong. Slicing real-time programs for enhanced schedulability. *ACM TOPLAS*, 19(3):525–555, 1997.

[GHD98] W. Grieskamp, M. Heisel, and H. Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In *FASE '98*, volume 1382 of *LNCS*, pages 88–106. Springer, 1998.

[GHS92] R. Gupta, M. Harrold, and M. Soffa. An Approach to Regression Testing using Slicing. In *ICSE '92*, pages 299–308. IEEE, 1992.

[GL91] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE TSE*, 17(8):751–761, 1991.

[GO01] K. B. Gallagher and L. O'Brien. Analyzing Programs via Decomposition Slicing: Initial Data and Observations. In *WESS '01: 7th International Workshop on Empirical Studies of Software Maintenance*, 2001.

[God95] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. PhD thesis, Universite de Liege, 1995.

[Gol88] U. Goltz. On Representing CCS Programs by Finite Petri Nets. In *MFCS '88*, pages 339–350. Springer, 1988.

[Gra07] Graphviz. Graphviz — Graph Visualization Software. `http://www.graphviz.org/`, 2007.

[GS97] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV '97*, pages 72–83. Springer, 1997.

[GSS99] V. Ganesh, H. Saidi, and N. Shankar. Slicing SAL. Technical report, SRI International, Menlo Park, 1999.

[Gup97] B. S. Gupta. A Critique of Cohesion Measures in the Object-Oriented Paradigm. Master's thesis, Michigan Technological University, 1997.

[Har07a] M. Harman. List of Program Slicing References. `http://www.brunel.ac.uk/~csstmmh2/slice.html`, 2007.

[Har07b] M. Harman. List of Program Slicing Researchers. `http://www.brunel.ac.uk/~csstmmh2/slicing.html`, 2007.

[HBD03] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *J. Syst. Softw.*, 68(1):45–64, 2003.

[HCD+99] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *SAS '99*, volume 1694 of *LNCS*, pages 1–18. Springer, 1999.

[HD97] M. Harman and S. Danicic. Amorphous Program Slicing. In *IWPC '97*, pages 70–79. IEEE, 1997.

[HDS95a] M. Harman, S. Danicic, and Y. Sivagurunathan. Program Comprehension Assisted by Slicing and Transformation. In *WPC '95: First UK workshop on program comprehension*, 1995.

[HDS+95b] M. Harman, S. Danicic, Y. Sivagurunathan, B. Sivagurunathan, and B. Jones. Cohesion metrics. In *8th International Software Quality Week*, 1995.

[HDZ00] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[HEH+98] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, and J.-L. Hainaut. Program Understanding in Databases Reverse Engineering. In *DEXA '98: 9th International Conference on Database and Expert Systems Applications*, pages 70–79. Springer, 1998.

[HFH+99] M. Harman, C. Fox, R. Hierons, D. Binkley, and S. Danicic. Program Simplification as a Means of Approximating Undecidable Propositions. In *IWPC '99*, pages 208–217. IEEE, 1999.

[HG98] M. Harman and K. B. Gallagher. Program slicing. *Journal of Information and Software Technology*, 40(11 and 12):557–581, 1998.

[HH00] R. M. Hierons and M. Harman. Program Analysis and Test Hypotheses Complement. In *ICSE International Workshop on Automated Program Analysis, Testing and Verification*, pages 32–39. IEEE, 2000.

[HH01] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[HHD99] R. M. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification & Reliability*, 9(4):233–262, 1999.

[HHD00] M. Harman, R. Hierons, and S. Danicic. The Relationship Between Program Dependence and Mutation Analysis. In *Mutation 2000*, pages 15–23. Kluwer, 2000.

[HHF+01] M. Harman, R. M. Hierons, C. Fox, S. Danicic, and J. Howroyd. Pre/Post Conditioned Slicing. In *ICSM '01*, pages 138–147. IEEE, 2001.

[HHF+02] R. M. Hierons, M. Harman, C. Fox, M. Daoudi, and L. Ouarbya. Conditioned Slicing Supports Partition Testing. *Softw. Test., Verif. Reliab.*, 12(1):23–28, 2002.

[Hie04] R. Hierons. Program Slicing References Collection. `http://people.brunel.ac.uk/~csstrmh/research/slicing.html`, 2004.

[HLS05] H. S. Hong, I. Lee, and O. Sokolsky. Abstract Slicing: A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking. In *SCAM '05*, pages 25–34. IEEE, 2005.

[HM05a] J. Hoenicke and P. Maier. Model-checking specifications integrating processes, data and time. In *FM '05*, volume 3582 of *LNCS*, pages 465–480. Springer, 2005.

[HM05b] J. Hoenicke and P. Maier. Model-checking specifications integrating processes, data and time. Technical Report 5, SFB/TR 14 AVACS, `http://www.avacs.org/`, 2005.

[HMR93] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. In *ISSTA '93*, pages 160–170. ACM, 1993.

[HO02a] J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes Data and Time. In *IFM '02*, volume 2335 of *LNCS*, pages 245–266. Springer, 2002.

[HO02b] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A Combination of Specification Techniques for Processes, Data and Time. *Nordic Journal of Computing*, 9(4):301–334, 2002.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hob07] U. Hobelmann. Verifying Properties of Processes, Data, and Time: Linking Counterexamples to High-Level Specifications. Master's thesis, Carl von Ossietzky University of Oldenburg, 2007.

[Hoe01] J. Hoenicke. Specification of Radio Based Railway Crossings with the Combination of CSP, OZ, and DC. FBT 2001, 2001.

[Hoe06] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2006.

[Hor90] S. Horwitz. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In *PLDI '90*, pages 234–245. ACM, 1990.

[HOSD97] M. Harman, M. Okulawon, B. Sivagurunathan, and S. Danicic. Slice-Based Measurement of Function Coupling. In *PMESSE '97: IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution*, pages 26–32, 1997.

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[HQR98] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *CAV '98*, volume 1427 of *LNCS*, pages 440–451. Springer, 1998.

[HR92] S. Horwitz and T. Reps. The Use of Program Dependence Graphs in Software Engineering. In *ICSE '92*, pages 392–411. ACM, 1992.

[HR05] S. Horwitz and T. Reps. Wisconsin Program-Slicing Project. `http://www.cs.wisc.edu/wpis/html/`, 2005.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM SIGPLAN Notices*, 23(7):35–46, 1988.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[HSD98] M. Harman, Y. Sivagurunathan, and S. Danicic. Analysis of Dynamic Memory Access using Amorphous Slicing. In *ICSM '98*, pages 336–345. IEEE, 1998.

[HW97] M. P. E. Heimdahl and M. W. Whalen. Reduction and Slicing of Hierarchical State Machines. In *ESEC '97/FSE-5: 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 450–467. Springer, 1997.

[HZ97] M. R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9:283–330, 1997.

[IKI03] T. Ishio, S. Kusumoto, and K. Inoue. Application of Aspect-Oriented Programming to Calculation of Program Slice. Technical Report 392, Osaka University, 2003.

[INIY96] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program Slicing on VHDL Descriptions and Its Applications. In *Asian Pacific Conference on Hardware Description Languages (APCHDL)*, pages 132–139, 1996.

[ISO01] ISO – International Standard Organization. *(ISO/IEC 15437) Standard for Information Technology—Enhancements to LOTOS (E-LOTOS)*, 2001.

[ISO02] ISO – International Standard Organization. *(ISO/IEC 13568) Standard for Information Technology—Z formal specification notation — Syntax, type system and semantics*, 2002.

[Jak07] M. Jakoblew. Studienarbeit "Erstellung einer CSP-OZ Spezifikation der Flugkontrolle eines Flughafens mittels Syspect". Universität Paderborn, 2007.

[JG01] G. Jia and S. Graf. Verification Experiments on the MASCARA Protocol. In *SPIN '01*, volume 2057 of *LNCS*, pages 123–142. Springer, 2001.

[JJ03] A. Janowska and P. Janowski. Slicing Timed Systems. In *CS&P '03: Proc. of the Int. Workshop on Concurrency, Specification and Programming*, volume 1, pages 235–250. Warsaw University, 2003.

[JJ04] A. Janowska and P. Janowski. Slicing Timed Systems. *Fundamenta Informaticae*, 60(1–4):187–210, 2004.

[JM05] R. Jhala and R. Majumdar. Path Slicing. In *PLDI '05*, pages 38–47. ACM, 2005.

[JR94] D. Jackson and E. J. Rollins. Chopping: A Generalization of Slicing. Technical Report CMU-CS-94-169, Carnegie Mellon University, 1994.

[Kam95] M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, 1995.

[KB96] B.-K. Kang and J. M. Bieman. Design-Level Cohesion Measures: Derivation, Comparison, and Applications. In *COMPSAC '96*, pages 92–97. IEEE, 1996.

[KFG04] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying Aspect Advice Modularly. *SIGSOFT Softw. Eng. Notes*, 29(6):137–146, 2004.

[KH01] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS '01*, pages 40–56. Springer, 2001.

[KK95] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *ICSM '95*, pages 222–231. IEEE, 1995.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81*, pages 207–218. ACM, 1981.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP '97*, pages 220–242, 1997.

[KNNI02] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental Evaluation of Program Slicing for Fault Localization. *Empirical Softw. Engg.*, 7(1):49–76, 2002.

[KR97] B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In *Automated and Algorithmic Debugging*, pages 43–58, 1997.

[Kri98] J. Krinke. Static Slicing of Threaded Programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '98)*, pages 35–42. ACM, 1998. ACM SIGPLAN Notices 33(7).

[Kri03] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2003.

[KS98] J. Krinke and G. Snelting. Validation of Measurement Software as an Application of Slicing and Constraint Solving. *Information and Software Technology*, 40(11–12):661–675, 1998.

[KS01] G. Kassel and G. P. Smith. Model Checking Object-Z Classes: Some Experiments with FDR. In *APSEC '01: Eighth Asia-Pacific on Software Engineering Conference*, pages 445–452. IEEE, 2001.

[KSCH99a] T. Kim, Y.-T. Song, L. Chung, and D. T. Huynh. Dynamic Software Architecture Slicing. In *COMPSAC '99*, pages 61–66. IEEE, 1999.

[KSCH99b] T. Kim, Y.-T. Song, L. Chung, and D. T. Huynh. Software Architecture Analysis Using Dynamic Slicing. In *AoM/IAoM 17th International Conference on Computer Science*, pages 242–247, 1999.

[KSCH00] T. Kim, Y.-T. Song, L. Chung, and D. T. Huynh. Software Architecture Analysis: A Dynamic Slicing Approach. *ACIS Int. J Comp. Inf. Sci.*, 1(2):91–103, 2000.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[KY94] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *ISSTA '94*, pages 66–79. ACM, 1994.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman, 2002.

[LCYK01] W. K. Lee, I. S. Chung, G. S. Yoon, and Y. R. Kwon. Specification-based program slicing and its applications. *Journal of Systems Architecture*, 47(5):427–443, 2001.

[LD98] A. Lakhotia and J.-C. Deprez. Restructuring Programs by Tucking Statements into Functions. *Information and Software Technology*, 40(11–12):677–690, 1998.

[Lem94] J. A. Leminen. Slicing and slice based measures for the assessment of functional cohesion of Z operation schemas. Master's thesis, Michigan Technological University, 1994.

[Li01] B. Li. A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement. Technical Report 415, Turku Centre for Computer Science (TUCS), 2001.

[LKCK00] W. J. Lee, H. N. Kim, S. D. Cha, and Y. R. Kwon. A slicing-based approach to enhance Petri net reachability analysis. *Journal of Research Practices and Information Technology*, 32(2):131–143, 2000.

[LKR05] P. Lam, V. Kuncak, and M. Rinard. Crosscutting Techniques in Program Specification and Analysis. In *AOSD '05*, pages 169–180. ACM, 2005.

[LR87] H. K. N. Leung and H. K. Reghbati. Comments on program slicing. *IEEE Trans. Softw. Eng.*, 13(12):1370–1371, 1987.

[Luc01] A. D. Lucia. Program slicing: Methods and applications. In *SCAM '01*, pages 142–149. IEEE, 2001.

[LV97] F. Lanubile and G. Visaggio. Extracting Reusable Functions by Flow Graph-Based Program Slicing. *IEEE Trans. Softw. Eng.*, 23(4):246–259, 1997.

[Lyl95] J. R. Lyle. Program Slicing References Collection. `http://hissa.nist.gov/~jimmy/refs.html`, 1995.

[MBPRR01] R. T. Mittermeir, A. Bollin, H. Pozewaunig, and D. Rauner-Reithmayer. Goal-driven combination of software comprehension approaches for component based development. In *SSR '01: Symposium on Software reusability*, pages 95–102. ACM, 2001.

[McM92] K. L. McMillan. *Symbolic Model Checking — An approach to the state explosion problem*. PhD thesis, Carnegy Mellon University, 1992.

[MD98] B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *ICSE '98*, pages 95–104. IEEE, 1998.

[MD99] B. P. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In *FM '99*, pages 1166–1185. Springer, 1999.

[MD00] B. Mahony and J. S. Dong. Timed communicating Object-Z. *IEEE TSE*, 26(2):150–177, 2000.

[Met07] B. Metzler. Decomposing Integrated Specifications for Verification. In *IFM '07*, volume 4591 of *LNCS*, pages 459–479. Springer, 2007.

[Mey05] R. Meyer. Model-Checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten. Master's thesis, Carl von Ossietzky University of Oldenburg, 2005.

[MFR06] R. Meyer, J. Faber, and A. Rybalchenko. Model Checking Duration Calculus: A Practical Approach. In *ICTAC '06*, volume 4281 of *LNCS*, pages 332–346. Springer, 2006.

[MMK06] D. P. Mohapatra, R. Mall, and R. Kumar. An Overview of Slicing Techniques for Object-Oriented Programs. *Informatica*, 30(2):253–278, 2006.

[MORW04] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *IFM '04*, number 2999 in LNCS, pages 267–286. Springer, 2004.

[MORW07] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a Formal Method into a Software Engineering Process with UML and Java. *Formal Aspects of Computing*, 2007. To appear.

[MT98] L. I. Millett and T. Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *SPIN '98*, pages 75–83, 1998.

[MT00] L. I. Millett and T. Teitelbaum. Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation. *STTT*, 2(4):343–349, 2000.

[MU05] P. Malik and M. Utting. CZT: A Framework for Z Tools. In *ZB '05*, volume 3455 of *LNCS*, pages 65–84. Springer, 2005.

[Muc00] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 2000.

[Nan01] M. G. Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, Bombay, 2001.

[NNH99] H. R. Nielson, F. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NR00] M. G. Nanda and S. Ramesh. Slicing Concurrent Programs. In *ISSTA '00*, pages 180–190. ACM, 2000.

[OA93] T. Oda and K. Araki. Specification Slicing in Formal Methods of Software Development. In *COMPSAC '93*, pages 313–319. IEEE, 1993.

[OB98] L. M. Ott and J. M. Bieman. Program Slices as an Abstraction for Cohesion Measurement. *Information and Software Technology*, 40(11–12):691–700, 1998.

[OBKM95] L. M. Ott, J. M. Bieman, B.-K. Kang, and B. Mehra. Developing Measures of Class Cohesion for Object-Oriented Software. In *AOWSM '95: Annual Oregon Workshop on Software Metrics*, 1995.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *First ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM, 1984.

[OT89] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *ICSE '89*, pages 198–204. ACM, 1989.

[OT93] L. M. Ott and J. Thuss. Slice Based Metrics for Estimating Cohesion. In *IEEE-CS International Metrics Symposium*, pages 78–81, 1993.

[Ott92] L. M. Ott. Using Slice Profiles and Metrics during Software Maintenance. In *10th Annual Software Reliability Symp.*, pages 16–23, 1992.

[OW05] E.-R. Olderog and H. Wehrheim. Specification and (property) inheritance in CSP-OZ. *Science of Computer Programming*, 55:227–257, 2005.

[Pel98] D. Peled. Ten Years of Partial Order Reduction. In *CAV '98*, pages 17–28. Springer, 1998.

[PF01] M. Plakal and C. N. Fischer. Concurrent garbage collection using program slices on multithreaded processors. *ACM SIGPLAN Notices*, 36(1):94–100, 2001.

[Pie07] E. Pietriga. ZGRViewer. `http://zvtm.sourceforge.net/zgrviewer.html`, 2007.

[PR06] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL '07*, volume 4354 of *LNCS*, pages 245–259. Springer, 2006.

[Rak07] A. Rakow. Slicing Petri Nets. Technical report, Carl von Ossietzky University of Oldenburg, 2007.

[Rep98a] T. Reps. Program analysis via graph reachability. Technical Report TR-1386, University of Wisconsin, Madison, 1998.

[Rep98b] T. Reps. Program Analysis via Graph Reachability. *Information and Software Technology*, 40(11–12):701–726, 1998.

[RH96] G. Rothermel and M. J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, 1996.

[RH04] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependences for slicing concurrent Java programs. In *CC '04: Compiler Construction*, volume 2985 of *LNCS*, pages 39–56. Springer, 2004.

[RLG02] J. Rilling, H. F. Li, and D. Goswami. Predicate-Based Dynamic Slicing of Message Passing Programs. In *SCAM '02*, pages 133–144. IEEE, 2002.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[RR95] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT '95*, pages 41–52. ACM, 1995.

[RY89] T. W. Reps and W. Yang. The Semantics of Program Slicing and Program Integration. In *TAPSOFT '89*, pages 360–374. Springer, 1989.

[Ryb06] A. Rybalchenko. CLP-Prover. `http://www.mpi-sws.mpg.de/~rybal/clp-prover/`, 2006.

[Ryb07] A. Rybalchenko. ARMC. `http://www.mpi-sws.mpg.de/~rybal/armc/`, 2007.

[Sü02] C. Sühl. *An Integration of Z and Timed CSP for Specifying Real-Time Embedded Systems*. PhD thesis, Technische Universität Berlin, 2002.

[Sch90] S. Schneider. *Correctness and Communication in Real-time Systems*. PhD thesis, Oxford University, 1990.

[Sch99] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 1999.

[SD96] S. Schoenig and M. Ducassé. A Backward Slicing Algorithm for Prolog. In *SAS '96*, pages 317–331. Springer, 1996.

[SD01] G. P. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems — An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[SGM02] G. Szilágyi, T. Gyimóthy, and J. Małuszyński. Static and Dynamic Slicing of Constraint Logic Programs. *Automated Software Engg.*, 9(1):41–65, 2002.

[SH96] A. M. Sloane and J. Holdsworth. Beyond Traditional Program Slicing. In *ISSTA '96*, pages 180–186. ACM, 1996.

[SH02] G. P. Smith and I. Hayes. An introduction to Real-Time Object-Z. *Formal Aspects of Computing*, 13(2):128–141, 2002.

[She06] A. Sherif. *A Framework for Specification and Validation of Real Time Systems Using Circus Action*. PhD thesis, Universidade Federal de Pernambuco, 2006.

[SHS02] Y. Sivagurunathan, M. Harman, and B. Sivagurunathan. Slice-Based Dynamic Memory Modelling — A Case Study. In *COMPSAC '02*, pages 351–356. IEEE, 2002.

[SJCS05] A. Sherif, H. Jifeng, A. Cavalcanti, and A. Sampaio. A Framework for Specification and Validation of Real Time Systems Using Circus Action. In *ICTAC '04*, number 3407 in LNCS, pages 478–493. Springer, 2005.

[Ska94] J. U. Skakkebæk. Liveness and Fairness in Duration Calculus. In *CONCUR '94*, volume 836 of LNCS, pages 283–298. Springer, 1994.

[Smi92] G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992.

[Smi95] G. P. Smith. A Fully Abstract Semantics of Classes for Object-Z. *Formal Asp. Comput.*, 7(3):289–313, 1995.

[Smi97] G. P. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *FME '97*, volume 1313 of *LNCS*, pages 62–81. Springer, 1997.

[Smi00] G. P. Smith. *The Object-Z Specification Language*. Kluwer, 2000.

[Sne96] G. Snelting. Combining Slicing and Constraint Solving for Validation of Measurement Software. In *SAS '96*, pages 332–348, 1996.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, 1989.

[SRK06] G. Snelting, T. Robschink, and J. Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM TOSEM*, 15(4):410–457, 2006.

[SS94] V. Sarkar and B. Simons. Parallel Program Graphs and their Classification. In *6th International Workshop on Languages and Compilers for Parallel Computing*, pages 633–655. Springer, 1994.

[ST02] S. Schneider and H. Treharne. Communicating B Machines. In *ZB '02*, pages 416–435. Springer, 2002.

[ST03] S. Schneider and H. Treharne. CSP Theorems for Communicating B Machines. Technical Report CSD-TR-02-12, Royal Holloway University of London, 2003.

[Sta00] J. Stafford. *A Formal, Language-Independent, and Compositional Approach to Control Dependence Analysis*. PhD thesis, University of Colorado, 2000.

[Sto00] S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN '00*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

[SW03] G. P. Smith and K. Winter. Proving temporal properties of Z specifications using abstraction. In *ZB '03*, volume 2561 of *LNCS*, pages 280–299. Springer, 2003.

[Sys06] Syspect. Endbericht der Projektgruppe Syspect. Technical report, Carl von Ossietzky University of Oldenburg, 2006.

[TA97] K. Taguchi and K. Araki. Specifying Concurrent Systems by Z + CCS. In *International Symposium on Future Software Technology*, pages 101–108, 1997.

[TCFR96] F. Tip, J.-D. Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In *OOPSLA '96*, pages 179–197. ACM, 1996.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[Ton03] P. Tonella. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Trans. Software Eng.*, 29(6):495–509, 2003.

[TS99] H. Treharne and S. Schneider. Using a Process Algebra to control B OPERATIONS. Technical Report CSD-TR-99-01, Royal Holloway University of London, 1999.

[TS00] H. Treharne and S. Schneider. How to Drive a B Machine. In *ZB '00*, pages 188–208. Springer, 2000.

[TWC01] J. Tretmans, K. Wijbrans, and M. R. V. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System — Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[Upc97] R. Upchurch. Program Slicing References Collection of the University of Massachusetts in Dartmouth. `http://www2.umassd.edu/SWPI/slicing/slicing.html`, 1997.

[UTS+03] M. Utting, I. Toyn, J. Sun, A. Martin, J. S. Dong, N. Daley, and D. W. Currie. ZML: XML Support for Standard Z. In *ZB '03*, volume 2651 of *LNCS*, pages 437–456. Springer, 2003.

[VABT03] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A Hierarchical Test Generation Approach Using Program Slicing Techniques on Hardware Description Languages. *Journal of Electronic Testing*, 19(2):149–160, 2003.

[Vas98] W. W. Vasconcelos. A Flexible Framework for Dynamic and Static Slicing of Logic Programs. In *PADL '99*, pages 259–274. Springer, 1998.

[VEA07] S. Vasudevan, E. A. Emerson, and J. A. Abraham. Improved Verification of Hardware Designs through Antecedent Conditioned Slicing. *STTT*, 9(1):89–101, 2007.

[Ven91] G. A. Venkatesh. The Semantic Approach to Program Slicing. In *PLDI '91*, pages 107–119. ACM, 1991.

[WA98] M. Woodward and S. Allen. Slicing Algebraic Specifications. *Information and Software Technology*, 40(2):105–118, 1998.

[War89] M. P. Ward. *Proving Program Refinements and Transformations*. PhD thesis, St. Annes College Oxford, 1989.

[War02] M. P. Ward. Program Slicing via FermaT Transformations. In *COMPSAC '02*, pages 357–362. IEEE, 2002.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of Circus. In *ZB '02*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.

[Weh04] H. Wehrheim. Preserving properties under change. In *FMCO '03*, volume 3188 of *LNCS*, pages 330–343. Springer, 2004.

[Weh05] H. Wehrheim. Slicing techniques for verification re-use. *Theor. Comput. Sci.*, 343(3):509–528, 2005.

[Weh06] H. Wehrheim. Incremental Slicing. In *ICFEM '06*, volume 4260 of *LNCS*, pages 514–528. Springer, 2006.

[Wei79] M. Weiser. *Program slices: formal psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[Wei81] M. Weiser. Program Slicing. In *ICSE '81*, pages 439–449. IEEE, 1981.

[Wei82] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[Wei84] M. Weiser. Program Slicing. *IEEE TSE*, 10(4):352–357, 1984.

[WG84] W. M. Waite and G. Goos. *Compiler Construction*. Springer, 1984.

[WM97] R. Wilhelm and D. Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer, 1997.

[WS02] K. Winter and G. P. Smith. Compositional Verification for Object-Z. Technical Report 02–42, University of Queensland, 2002.

[WS03] K. Winter and G. P. Smith. Compositional Verification for Object-Z. In *ZB '03*, volume 2651 of *LNCS*, pages 280–299. Springer, 2003.

[WY04] F. Wu and T. Yi. Slicing Z specifications. *ACM SIGPLAN Notices*, 39(8):39–48, 2004.

[WZH05] M. P. Ward, H. Zedan, and T. Hardcastle. Conditioned Semantic Slicing via Abstraction and Refinement in FermaT. In *CSMR '05*, pages 178–187. IEEE, 2005.

[XQZ+05] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[ZCU96] J. Zhao, J. Cheng, and K. Ushijim. Static Slicing of Concurrent Object-Oriented Programs. In *COMPSAC '96*, pages 312–320. IEEE, 1996.

[ZGZ04] X. Zhang, R. Gupta, and Y. Zhang. Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. In *ICSE '04*, pages 502–511. IEEE, 2004.

[Zha98] J. Zhao. Applying Slicing Technique to Software Architectures. In *4th IEEE International Conference on Engineering of Complex Computer Systems*, pages 87–98, 1998.

[Zha99] J. Zhao. Slicing Concurrent Java Programs. In *International Workshop on Program Comprehension*, pages 126–133. IEEE, 1999.

List of Figures

List of Tables

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

P

Q

R

S

T

U

V

W

Z