Towards Automated Service Composition Under Quality Constraints

Felix Mohr

Dissertation

Faculty of Electrical Engineering,
Computer Science and Mathematics

Paderborn University - Germany

Paderborn, 2016

Zusammenfassung

Services sind plattformunabhängige Software-Komponenten. Automatisierte Servicekomposition wird in dieser Arbeit als die Aufgabe verstanden, ein neues Programm aus bestehenden Services zu synthetisieren ohne dass die Struktur der Lösung bekannt ist. Die Vision von automatisierter Servicekomposition ist es dem Entwickler zu gestatten, Teile des Programms deklarativ zu beschreiben und den dazu passenden ausführbaren Code automatisch ableiten zu lassen. Dieses Problem wird typischerweise als Planungsproblem verstanden und gelöst.

Automatisiertes Planen und Komposition sind seit Jahren etablierte Forschungsfelder, aber die meisten dort bekannten Ansätze können mit wichtigen Aspekten des Servicekompositionsproblems nicht oder nur begrenzt umgehen. Erstens arbeiten die meisten Ansätze auf der Basis von aussagenlogischen Beschreibungen, was mit der Kernanforderung der Beschreibung von Beziehungen zwischen Inputs und Outputs von Operationen in Konflikt steht. Auch werden die wichtigen Aspekte Hintergrundwissen und Servicequalität (QoS) nur selten betrachtet. Drittens sind fast alle Ansätze auf das Finden von Programmen ohne alternative Pfade oder Schleifen beschränkt. Mir ist kein Ansatz bekannt, der all diese Anforderungen gleichzeitig berücksichtigt.

In dieser Arbeit schlage ich eine Auswahl von Ansätzen vor, die diese Probleme lösen. Der originäre Beitrag besteht aus drei Teilen:

1. Ich stelle zwei Planungsalgorithmen vor, die das sequenzielle Kompositionsproblem lösen. Das erste basiert auf Rückwärtssuche und das zweite auf Planen mit Halbordnungen. Die Algorithmen gehören zu einigen der sehr wenigen, die Planungsprobleme lösen können, in denen Operationen neue Objekte herstellen können. Sie sind die ersten, die gleichzeitig nicht-skalare und nicht-additive Kosten, insb. QoS, berücksichtigen.

2. Meines Wissens stellt diese Arbeit als erste ein Verfahren zur Ableitung von Kompositionen nicht nur mit alternativen Pfaden sondern mit Schleifen vor. Schleifen werden nicht beliebig sondern in Form von Instanziierungen von Schablonen erzeugt. Schablonen sind vorgedachte und generische (d.h. domänenunabhängige) Schleifenmuster, die in vielen Anwendungen auftreten. Mit Ausnahme von Try-Until-Schleifen, die Aktionen solange ausprobieren bis sie gelingen, haben vorherige Ansätze keine Schleifen berücksichtigt.

3. Im Rahmen dieser Arbeit habe ich eine umfassende Evaluation durchgeführt, die nicht nur die grundsätzliche praktische Anwendbarkeit der vorgestellten Ansätze nahelegt, sondern auch Vergleiche zwischen ihnen erlaubt. Wegen des Fehlens standardisierter Tests haben wir dazu eine Testumgebung entwickelt, die automatisch Kompositionsprobleme mit verschiedenen Schwierigkeitsgraden erzeugen kann. Schleifenbasierte Komposition stellt sich für einfache Probleme als gut lösbar heraus, bedarf aber noch einer Optimierung, um in der Praxis anwendbar zu sein. Sequenzielle Kompositionsprobleme hingegen können bereits für leichte und mittelschwierige Probleme, die denen aus realen Anwendungen nahekommen, gelöst werden. Das Verfahren basierend auf Planen mit Halbordnungen stellt sich hierbei der Rückwärtssuche als deutlich überlegen heraus.

Der Kontext dieser Arbeit ist der Sonderforschungsbereich 901 für On-The-Fly Computing an der Universität Paderborn mit dem Ziel “Techniken und Prozesse für automatische on-the-fly Konfiguration und Bereitstellung von individualisierten IT-Services anhand von Basis-Services, die auf weltweiten Märkten angeboten werden, zu entwickeln”. Der Beitrag dieser Arbeit in dem Kontext des SFB 901 ist die Entwicklung von Techniken, die die *Konfiguration* von Services erlauben.

Abstract

Services are self-contained and platform independent software components. Automated service composition as understood in this thesis is the task of automatically assembling new software artifacts from existing services *without* structural knowledge about the solution. The vision of automated service composition is to allow software developers to specify parts of the program code declaratively and to automatically find compilable pieces of code that satisfy the declarative specifications and may replace them. The service composition problem is typically seen and addressed as a planning problem.

Automated planning and composition have been heavily studied research fields for many years, but the majority of available approaches cannot cope with important aspects of the service composition problem. First, most approaches only work for propositional logical service descriptions, which is incompatible with the core requirement to express relations between inputs and outputs of their operations. Second, background knowledge and service qualities (QoS) are highly important but only considered rarely. Third, most approaches are limited to find sequential compositions only, i.e. without alternative branches, not to mention loops. I am not aware of any approach that considers all of these requirements simultaneously.

This thesis presents a selection of approaches that address the above shortcomings. My original contributions are as follows:

1. I present two planning algorithms that solve the sequential composition problem. One is based on backward search, and one is based on ideas from partial-order planning. The algorithms are in a line with very few existing methods and techniques that can solve planning problems in which operations create new constants. The ones presented here are the first to additionally consider non-scalar and non-additive costs, which represent the service qualities like price, runtime, trust, etc.

2. To the best of my knowledge, this is the first work that presents a technique to find compositions not only with alternative branches but also with loops. Loops are integrated not arbitrarily but in form of template instantiations, i.e. the loop ideas are pre-thought patterns that are captured in generic templates. Previous approaches have no loop support or only insert loops that reflect try-until behaviors but not a typical loop behavior as found in programming, e.g. to iterate over a set of items and perform actions on them.

3. I conducted an exhaustive evaluation that not only suggests practical solvability of the problem but also compares the algorithms regarding runtime and solution quality. Due to the absence of benchmarks, we have developed an extensive benchmark environment to conduct the evaluation. The technique for composition with loops is feasible in simple setups but needs to undergo some optimizing revisions before being usable in practice. Sequential composition, in contrast, seems to be already solvable for simple and intermediate real world problems using partial-order based composition, which I show to outperform backward search based composition in these setups.

The context of this work is the Collaborative Research Center 901 for On-The-Fly Computing at the University of Paderborn whose goal "is to develop techniques and processes for automatic on-the-fly configuration and provision of individual IT services out of base services that are available on world-wide markets". The contribution of this thesis in the context of the CRC is to develop techniques that enable the *configuration* of such services.

CONTENTS

PREFACE

This manuscript and all the publications I have been involved so far are the result of delighting teamwork I was permitted to enjoy over the last couple of years—in the office as well as at home. This is the time and the place for thanking the fellows that contributed to this work in one way or the other.

My first thanks go to my supervisors and reviewers Hans Kleine Büning, Jörg, and Friedhelm. Thank you for taking your valuable time for challenging discussions and for so (too) many hints on related work. It helped a lot to put the work into the right context.

Secondly, thanks go to the people in Heike's, Eyke's, and Bernd's groups for discussions and suggestions. First, I highlight Alex and Sven as my primary co-authors. I have appreciated a lot the collaboration with Alex who was always pushing our work towards runnable code, which would make the approaches more usable and visible. Working with Sven has always been a pleasure, not so much because he was always ready to provide a question that would jeopardize my paper ideas but rather due to his high demands for new contributions. Third, I thank Marcel and David for thousands of lines of code and conceptual colaboration for the evaluation, which enabled all the nice plots. Fourth, I thank Eyke for his patience in finishing this thesis before we could start off with new topics in machine learning.

Yet, the person who deserves to be highlighted most for his contribution to this thesis is Theo Lettmann. Hours over hours of delving into the conceptual details, taking different perspectives, checking also the disgustingly boring parts of the proofs over and over was more than just sporadic support. Your fingerprint is on all aspects of the thesis, and it was a pleasure to still learn so much more about planning and search. Thank you so much for not only holding out every new idea I came up with but even to accompany me in developing them in more detail than I would have been able on my own—even though (or precisely because) this eventually meant to dump many of them. Thank you for everything I learned from you during this time; not only the things that made it into this file.

For the professional part, I finally want to thank the organizers and the still unmentioned coworkers within the CRC 901. Realizing this thesis in the context of the CRC 901 brought a lot of practical benefits for which I am very thankful. On one hand, working with so many interesting people with so different points of view on a common project has been a challenging and valuable experience I would not want to miss. In these regards, I want to thank Marie for her great coordination work and for forgiving me that I wouldn't answer her emails most of the time. On the other hand, I am very thankful for the possibility to participate in conferences at rather remote locations, which enabled great networking. Also, working in the CRC gave me the possibilty to completely focus on research and develop and elaborate much more ideas than it would have been possible otherwise.

But it is also clear that this project could not have been finished neither with great support from home. Thank you Moni for keeping our home a coasy and tidy place, listening to my desperates attempts of explaining you weird compute science stuff, but, above all, for accompanying Isabel with so much love and dedication in spite of all the challenges we experienced; she coldn't ever have been in better hands.

1. Introduction

This thesis deals with a problem called *automated service composition*. There are many interpretations of what automated service composition means, but, in this thesis, I see this as the task to automatically find the *implementation* of a program whose[1] intended functionality is described logically, and no knowledge about the solution structure is available.

A service is a software component that offers a set of *operations* (cf. Figure 1.1). Like functions in programming languages, the operations have input and output parameters. In addition, they have (i) logical preconditions and postconditions to describe their behavior on an abstract level and (ii) quality properties like price or response time, which occur on usage.

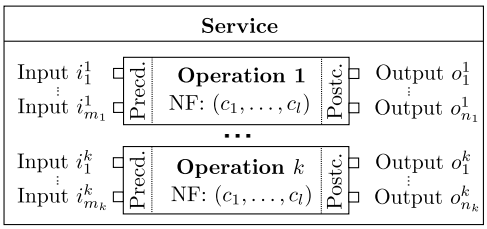


Figure 1.1: Scheme of a Service Description

Given the description of a desired service and existing services, the task is to find correct implementations for the operations of the desired service. The non-functional quality properties of each implementation must satisfy the bounds for the respective operation in the specification of the desired service.

The overall research question tackled in this thesis is as follows:

> *How can automatic service composition be achieved in principle, and can we expect it to be feasible in practical applications?*

Based on this general question, this thesis has three key objectives:

1. *Formalizing* the automated service composition problem
   The objective is to give a formal definition of a problem whose solutions are executable compositions that are correct w.r.t. a meaningful description.

2. *Solving* the automated service composition problem
   The objective is to develop correct and complete algorithms that solve the problem.

3. *Analyzing* the practicability of the solutions
   The objective is to make assertions about the performance of the developed algorithms that allow to judge their suitability for today's software development.

[1] Some people feel that *whose* cannot be used as a possessive with inanimate objects. But it *can* [41, p.887], is more elegant than "of which", and it has been used a lot in standard literature, e.g. the King James Bible.

1.1 The Vision: Augmenting Classical Programming

A key vision of automatic programming is to augment classical (i.e. imperative or functional) programming by the possibility to express goals or tasks without saying how these are achieved [71]. In classical programming, the developer must exactly write down what the process that executes the program shall do. This is often a good way to encode the developers' intentions, but sometimes it is easier to simply write down properties that shall hold for an object instead of how they are achieved.

To make this vision clearer consider the following example. Suppose we know that the variable x contains an object that represents a book and that we want to know the price of the book in EUR. Given we know that the function to determine the price of a book is `getPriceOfBook`, that the price is determined in USD, and we know the function to convert USD into EUR is `USD2EUR`, we can write

```
p := getPriceOfBook(x);
y := USD2EUR(p);
```

However, it would be much more convenient to simply write something like

```
y :: PriceOf(y,x) & EUR(y)
```

Instead of writing *how* the desired information is computed, we only write what we want to be sure of. Here `::` is a new assignment operator with the semantics to set the left hand side variable(s) in a way that the formula on the right hand side is satisfied.

The second notation has several advantages:

- The statement is closer to the developer's goal: It states the semantics of the content of the variable instead of describing *how* it is computed.
- Decoupling of description and implementation: Neither do we need to know the existence of `getPriceOfBook`, nor is there any problem if its name or location is changed.
- Less knowledge necessary: We do not even need to know the currency that is returned by the function that determines the price. It already may be EUR, but if not, we can (try to) automatically convert it (as in the example).
- The code that will replace the statement will be correct by construction.
- Shorter code; in particular, the helper variable `p` is not needed at all.
- The background knowledge is a valuable documentation of the domain.

Clearly, the fact that programs of the above type contain non-executable statements implies that we cannot build executables merely based on classical translation techniques. Leaving internal compiler optimization techniques out of the question, in classical programming, the developer implicitly defines the machine-code. The developer (team) has already decided which routines are used in the program, and the compiler only translates their definitions into machine code. In the above program, however, we have parts in the code that are not associated with concrete routines. There is no routine with the name `PriceOf(y,x)`, which we could use to resolve the corresponding statement. Consequently, a common compiler would not be able to build an executable here.

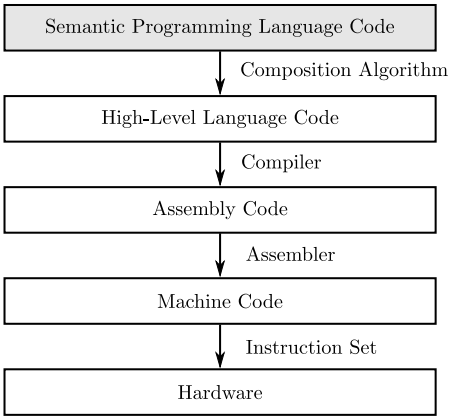


Figure 1.2: Semantic code imposes a new layer in the development stack.

In other words, we need to modify the development stack by adding a new layer that involves the *search* for adequate implementations. Figure 1.2 illustrates this extension. Before we can compile the program, a composition algorithm must *replace* the declarative code fragments by executable code of which it is sure to guarantee the desired property. If this succeeds, the compiler can translate the obtained program in the usual way. If the composition algorithm fails to find valid implementations for the declarative statements (in the given frame of time), the developer must write them manually.

Of course, this kind of programming only works if existing functions have descriptions that allow to reason over their suitability to accomplish a particular task. The search algorithm cannot simply "guess" which routines satisfy `PriceOf(y,x)`. The usual way to annotate existing functions is to use (first-order) logical postconditions and maybe with preconditions. We also may use background knowledge in order to express relations between the predicates.

In spite of the syntactic similarity between programs of the above style and logic and declarative programming, these pursue very different goals. In logic and, as I understand it, also in declarative programming, statements define database queries [46]. The query statement is a logic formula with variables, and the tool, e.g. Prolog, computes all known groundings of this formula that are true in the given database; i.e. we are interested in *data*. However, in the above case, there is no notion of any database. We are not interested in an *evaluation* of the predicate `PriceOf(y,x)` in terms of groundings, but we want to *deduce code* that allows to compute such a grounding at runtime. Several such solutions may exist and have different qualities while there is usually only one such derivation both existent and required in Prolog.

My motivation to solve this problem is rooted in the broader context of the collaborative research center (CRC 901 – On-The-Fly Computing). The goal of this CRC "is to develop techniques and processes for automatic on-the-fly configuration and provision of individual IT services out of base services that are available on world-wide markets" [1]. The role of this work within the research center is obviously to provide the functionality for the configuration, i.e. composition, of services based on the requirement specification given by a client. In Section 1.3, I introduce a running example that describes who the client is and how such a specification and solution looks like. Most of the aspects considered in the CRC are relevant for embedding the composition technique but not its algorithmic solution. Hence, I focus on the algorithmic solutions and not so much on their conceptual relation to other modules within the CRC.

1.2 State of the Art & Contribution

Many approaches solve a problem they call the automated service composition problem, but it is obvious that they do very different things. In [88], I have carried out an exhaustive literature review, which also contains a detailed classification of the different approaches. In this thesis, I only discuss the relation to other approaches as far as necessary to understand the major conceptual differences and refer the reader to the survey for more details. While the exhaustive discussion of related work is covered in Chapter 8, I now only sketch the research frontier for the questions I want to tackle.

Automated service composition is frequently seen (and sometimes even solved) as a planning problem, but there are several important differences to classical planning. The most important differences are the facts that operations *create* new constants, that plans are associated with non-scalar and non-additive costs (Quality of Service - QoS), that compositions are not sequential in general but have alternative branches and loops, that the closed world assumption does not apply, and that the invoker is not only interested in one but several or even all plans. It is still reasonable to interpret the composition problem as a kind of planning problem. However, a mere encoding into the standard language PDDL is grossly inadequate as the above aspects are not natively covered by standard planners [84, 87]. Hence, it is more reasonable to develop (planning) algorithms tailored for this class of planning problems.

As a consequence, in **Chapter 2**, I provide a detailed formal description of the composition problem. The formal model is expressed in a bottom-up fashion departing from the domain of software composition, identifying the relevant components and their relation, and finally defining the semantics in terms of *conditions* that must hold on compositions in order to satisfy a given query. This is somewhat opposed to the common top-down style in AI where we start from the standard components of a planning problem (state transition system, belief evolve relation, etc.) into which the actually considered model is then fit [59, 97].

The analysis of the field of automated service composition reveals that many approaches address (some of) the specialties encountered in automated composition but make heavily simplifying assumptions. For example, the well-known line of research of the Roman model [10] considers alternative branches but completely ignores data flow. In addition, it is assumed that the solution structure is already known in advance. Similarly, the research line around Pistore and Bertoli also developed an approach for composition with alternative branches [12, 97, 117] whose formal model (plans are state-action tables) even allows for compositions with loops. However, like in the Roman model, the data flow between the operations is ignored (assumed to be predefined), which in their case even implies that the operations occurring in the solution have been selected in advance. Also, while loops *can* be *encoded* using policies, actually *finding* compositions with loops is not supported [12]. There are several dozens of other approaches less powerful but making similar restrictions; I provide a detailed discussion of all these approaches in **Chapter 8** and in [88].

The works that constitute the state of the art solutions to the problem I tackle in this thesis are the approaches by McDermott [84] and the line of research carried out by Hoffmann, Sirbu, and, most notably, Weber [59, 60, 107, 122]. McDermott solves the composition problem by modifying the planning language PDDL to the needs of service composition and writing a planner that supports those modifications [84]; his approach allows to find compositions with limited alternative branches but no loops. The approaches in [59, 60, 107, 122] apply *forward* search to solve the composition problem. Alternative branches are considered to a certain extent (cf. Section 8.1.4). Loops are not considered, and the effective creation of objects is treated above all theoretically. None of these approaches considers service quality aspects.

My contribution to augment this state of the art is to answer the following research questions:

RQ 1. *How can we solve the sequential composition problem under quality constraints?*
In sequential service composition, we only consider compositions that are sequences of operation invocations, i.e. without branches or loops. Leaving quality constraints out of the question, this question was answered for forward search in [107] and [122]. I complement these works by answering the question for backward search in **Chapter 4** and for partial order planning in **Chapter 5**. To the best of my knowledge, neither backward search nor partial order planning has been used for *any* non-monadic planning domain in which operations were allowed to create new objects. Also, service quality was not considered before.

In order to not present two algorithms with largely overlapping descriptions, I present a generalized search framework tailored for automated service composition (or any other search problem with the respective properties) in **Chapter 3**, which also gives a brief introduction to search. The concrete composition techniques are then *instances* of this framework.

RQ 2. *How can we find compositions with branches and, in particular, with loops?*
Sequential compositions have their application but are significantly limited, because they cannot insert alternative branches or loops to the control flow. **Chapter 6** provides an answer to this question. First, based on McDermott's ideas [84], I explain how we can extend the sequential composition techniques in order to find also compositions with branches. Second, I incorporate the results of my work on the instantiation of abstract loop templates with prethought invariants [91, 92] in order to add loop constructs to compositions.

The developed solutions provide correct and, to a certain extent, complete algorithms for the composition problem. The obvious question is whether these algorithms can be used in reality or whether complexity renders their application impossible. Two research questions arise:

RQ 3. *Is automated service composition computationally feasible?*
This question can be (and is) tackled from the theoretical and practical viewpoint. The theoretical viewpoint asks for the complexity of the composition problem, which I analyze in **Chapter 2**. The practical viewpoint asks for performance measures associated with applicability such as runtime and space consumption, which I analyze in **Chapter 7**. While the undecidability result is not much of a novelty since it was shown for a very similar setting before [59], the main contribution of this thesis with respect to this research question is an answer to the practical feasibility, which is generally positive. In particular, the experimental analysis we conducted suggests that automated service composition works sufficiently fast in real world setups.

RQ 4. *Is one of the composition algorithms superior to the others?*
Given the previous works and the two approaches discussed here, it is natural to ask for a "best" strategy. However, a fair comparison to the previously developed approaches in the sense of solving a common set of benchmark problems was not possible for technical reasons. Hence, the answer is limited to a comparison of the (variants of the) two strategies presented in this thesis. The results in **Chapter 7** show that partial order composition performs significantly better than backward search under any considered evaluation condition.

1.3 Running Example: The Bookstore Finder

In the discourse of this thesis, I use a running example that covers all aspects about service composition that are relevant for the approaches I discuss. Our role in this scenario is that we are software developers who want to create a specific program. In order to carry this task out, we make use of programs previously developed at our site as well as external services.

The program we want to create should compute a list of nearby book shops that have a particular book on stock. More precisely, we want a service with one operation. The operation has three inputs (a position, an author name, and a book title) and one output (the book shops). The invocation of the operation should not last more than 200ms and cause at most a cost of 1 EUR. Figure 1.3 shows a graphical description of the query. Intuitively, the precondition and postcondition of the operation are described by simple predicate logic formulas, and the non-functional quality requirements are defined in a vector; the exact formal definition can be found in Section 2.1.

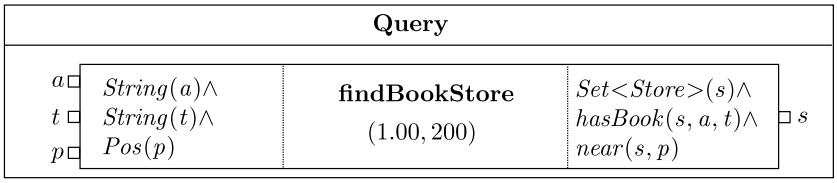


Figure 1.3: The service we want to obtain

There are four services available to compose the desired operation. Figure 1.4 shows a graphical description of them in the same notation as above.

1. *GeoService* provides an operation that computes the city that belongs to a position. The operation is gratis and has average response time of 20ms.

2. *YellowPages* provides an operation that computes all the companies of an industry sector in a city. Using the operation costs 0.5 EUR, and the response time is 10ms.

3. *LiteratureService* provides an operation to compute the ISBN from an author's name and the title of one of her books. Using the operation costs 0.1 EUR, and the response time is 100ms.

4. *BookShopService* provides an operation to filter a given set of book shops by the ones that have books with the given ISBN on stock. The notation $Set[t]$ indicates a set of objects of type t like generics in modern programming languages.

It is fairly reasonable to assume that the first three services are available from external sources. The fourth service is more specific and may have been developed it previously at our site.

One intuitive solution for this problem is as follows (see Figure 1.5). First, determine the ISBN i that belongs to the book identified by the author a and title t. Then determine the city belonging to the input position and use *YellowPages* to compute all the book shops in that city. Third, filter those shops by the ones that have the desired book at stock, and get the one of them that is nearest to the input position. Finally, use *getPrice* to determine the price of the book in the respective store.

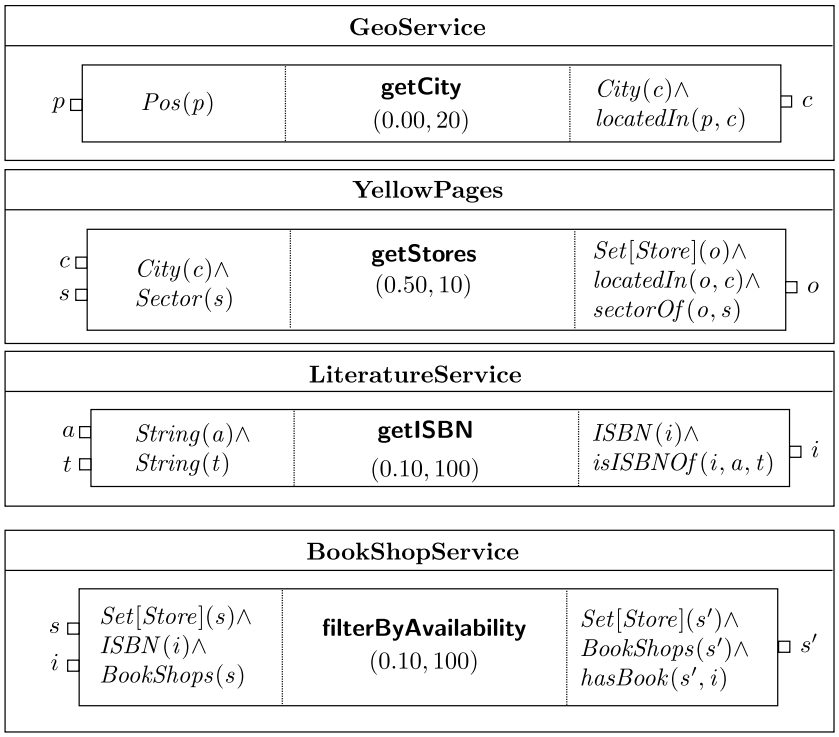


Figure 1.4: An example repository of services

However, the semantic descriptions do not allow us to build this composition automatically. For example, the only operation that provides the predicate *hasBook* is filterByAvailability, but filterByAvailability specifies this predicate in terms of an ISBN whereas the query requires it based on the author and the title. Also, filterByAvailability requires objects that satisfy the *BookShops* property as an input, but no operation provides objects with this property.

Therefore, we need to specify formal knowledge that can be used to derive the solution. Intuitively, one can think of the knowledge as a set of logic implications; in the formal model, these will be considered in form of clauses. For the above example, we consider the following knowledge base (parentheses denote the usage of constants):

- $Sector(\text{`}bookshops\text{`})$
- $Set[Store](s) \wedge sectorOf(s, \text{`}bookshops\text{`}) \rightarrow BookShops(s)$
- $hasBook(s, i) \wedge isISBNOf(i, a, t) \rightarrow hasBook(s, a, t)$
- $locatedIn(x, z) \wedge locatedIn(y, z) \rightarrow near(x, y)$
- $locatedIn(S, z) \wedge S' \subseteq S \rightarrow locatedIn(S', z)$

Given this knowledge, we can prove the correctness of the solution for the above query.

This example shows that queries for automated service composition do not necessarily imply a canonical solution. Automated service composition often faces the reproach to solve queries which already encode their own solution to some extent. While this is natural in

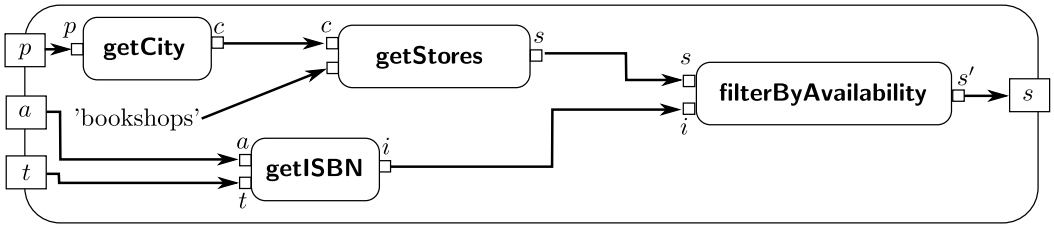


Figure 1.5: A possible solution for the above query

approaches like the Roman model [10], which explicitly assume the solution structure to be given, this is not as obvious for approaches not relying on such a template. The implicit argument behind this is that using automated composition does not alleviate the developer from writing formal descriptions, and, if he needs to write formal specifications, why should he not directly write the code himself? In addition to the arguments I gave in favor of composition in Section 1.1, we can see in this example that the query does by no means encode how the solution should look like. For example, the query does not say anything about how the set of stores from which we compute the relevant book stores is determined; using a yellow pages service is *one* possible way. This degree of freedom may, of course, also yield to solutions that are not desirable even though formally correct; I discuss this aspect of composition in Section 2.2.3. However, we can always use composition to make *suggestions* for solutions, which the developer may not even be aware of.

I will use this example problem throughout the rest of this thesis. More precisely, in Chapter 4 and Chapter 5, I will explain how we can solve the above query using backward composition and partial order composition respectively. In Chapter 6, I will explain how we can *automatically derive* the fourth of the above services (of computing the subset of stores having a book on stock) if we only have a service given a simpler service that determines for a single store whether or not it has a given book on stock. So the example serves not only to demonstrate how the sequential composition algorithms work but also showcases a query in which a composition with loops is required and how such a composition can be achieved.

Even though in this thesis I assume that the existing services are given in form of an explicit set, a more frequent assumption in service composition is that these need to be detected within a separate step called *discovery* [122]. In fact, in previous work [90] and in the implementation, the algorithm does not even receive a set of service to operate on but a discovery interface that may be queried for relevant services and that—asynchronously—returns answers based on a search in a market. This puts the developed composition techniques into the context of service markets as considered in the CRC 901. However, for the conceptual part of composition covered in this thesis, this additional aspect is of marginal importance and, hence, completely ignored. It can be easily put on top without changing the contributions made in this work.

2. Problem Definition and Analysis

This chapter defines the automated service composition problem based on a first-order logic setting. It describes the syntax and semantics of service operations, the background knowledge, queries, and compositions as solution candidates for queries.

Section 2.1 contains the formal problem description, which is the basis of all the algorithms presented in this thesis. Section 2.2 describes implicit assumptions imposed by the model in Section 2.1. Finally, Section 2.3 discusses the theoretic complexity of the composition problem.

2.1 Formal Problem Statement

The problem statement for a service composition task must define four concepts. That is, it must define what *services*, *queries*, and *compositions* are, and it must describe the conditions under which a composition is a *solution* to a query.

To keep the formal problem simple, I define the service composition problem rather as an *operation* composition problem. Services are nothing else than sets of operations, which are the core of the composition activity. There is no striking argument to maintain the additional concept of services in the formal model. So instead of sets of services with operations, we only consider sets of operations.

The big picture of the problem statement as shown in Figure 2.1 is as follows. The core of the composition problem are the operations, each of which is described through

- inputs and outputs, which are variable names that are associated with some domain-related type, e.g. *Book*, *Flight*, etc.;
- preconditions and postconditions, which state necessary conditions before and guaranteed conditions after its invocation; and
- non-functional (quality[1]) properties such as price, execution time, throughput, etc. that "occur" each time the operation is used.

The second important building block is background knowledge from the domain that may be relevant for the composition task. For example, if we know that the distance between two coordinates c_1 and c_2 is d and that c_1 and c_2 are the coordinates of some objects p_1 and p_2 respectively, then we also know that the distance between p_1 and p_2 is d.

I use *predicate* logic to describe background knowledge and operation behavior. Many approaches apply propositional logic [10, 12], but propositional logic is inherently inadequate in service composition, because we cannot express information about the data processed by operations. In particular, the composer cannot decide how data should flow between them.

[1] Even though the term "quality" is often used in the respective literature [19, 127], it is not always appropriate e.g. for the price "quality". In the following, I rather use the term "non-functional properties", but the intended meaning is actually equivalent.

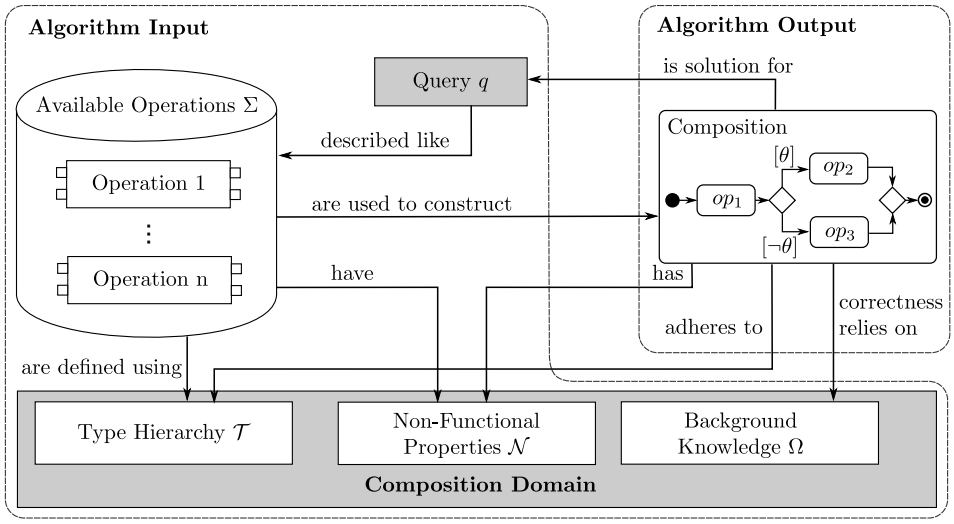


Figure 2.1: The big picture of the automated service composition problem.

Given the operations, the background knowledge, and a query, which is described like an operation itself, the task is to find an arrangement of operation *invocations* that satisfies the query. Intuitively, we can think of a composition as a simple program whose only program statements are operation invocations and control elements such as if-statements or loops. It has a finite set of (program) *states*, which are one initial state, possibly several final states, and one state between each pair of program statements. A composition is a solution to a query if we can label the states with logical assertions such that

- the initial state is implied by the preconditions given in the query,
- each of its final states implies the required postconditions,
- every program statement works given its preceding state, and
- if s' is the state after a program statement a whose predecessor state is s, then s' must be a consequence of s and the postcondition of a.

In addition, the used operations impose non-functional properties (price, execution time, etc.) of the whole composition, which must not exceed the bounds defined in the query.

This informal introduction devises the following road-map for the definition of the formal model. First, I define the *composition domain*, which consists of the background knowledge, the type system, and the system of non-functional properties, because without types and non-functional properties we cannot define operations. Second, I define what an operation is, which also includes the definition of queries, which are only operation descriptions. Third, I define the concept of states, operation invocations, and guards as conditions for if-statements and loops as elements of compositions. Finally, I define the conditions that a composition must satisfy in order to be a solution to a query.

2.1.1 Composition Domain and Operations

In the problem tackled in this thesis, services, requirements, and compositions rely on a *composition domain*. The composition domain consists of a type system, logical domain knowledge, and the relevant non-functional properties.

Definition 1. *A **composition domain** is a tuple $\langle \mathcal{T}, \Omega, \mathcal{N} \rangle$ where*

- *$\mathcal{T}$ is a finite set of clauses $\forall x : \neg t_{sub}(x) \vee t_{sup}(x)$ and ground unit clauses $t(a)$. The meaning of a clause is that every object of type t_{sub} is also of type t_{sup}. Unit clauses encode type information about constants.* — *Types*
- *Ω is a finite set of Horn clauses $\forall X_1, \ldots, X_k : l_1(X_1) \vee \ldots \vee l_{k-1}(X_{k-1}) \vee l_k(X_k)$. A literal $l_i(X_i)$ is a (possibly negated) predicate with variables X_i and constants; at most one of the literals of a clause may be positive. Predicates that occur in $\mathcal{T}$ may occur in Ω only negated.* — *Background Knowledge*
- *$\mathcal{N}$ is a finite list of tuples $\langle \mathbb{D}_p, \oplus_p, \ominus_p \rangle$ where $\mathbb{D}_p \subseteq \mathbb{R}$ are possible values of property p, $\oplus_p : \mathbb{D}_p \times \mathbb{D}_p \to \mathbb{D}_p$ is an associative and commutative aggregation function for p, and $\ominus_p : \mathbb{D}_p \times \mathbb{D}_p \to \mathbb{D}_p$ is a "consumption" function for p. Intuitively, $x \ominus_p y$ is what remains if we have a value of x for property p and "consume" y.* — *Non-Functional Properties*; *Prop. Aggregation*; *Prop. Consumption*

Remarks.

- Limiting the background knowledge Ω to Horn has practical reasons. It is much easier to set up a complete inference mechanisms specialized on Horn formulas than for arbitrary formulas. A relaxation of this type of background knowledge is important future work.
- Background knowledge here has a rather *constructive* nature as opposed to *constraining* knowledge as used in [59, 122] to model environmental conditions which are exploited in order to conduct belief revision. In both cases, the knowledge is used to model environmental conditions that always hold. However, I rather use this knowledge to *actively* derive new knowledge (which is implicitly entailed) while, in [59] and [122], knowledge is used to detect and resolved *inconsistencies* that arose in the application of an operation, which indicates that previous knowledge has become invalid.
- The type information contained in $\mathcal{T}$ could also be encoded into Ω, because the formulas allowed for $\mathcal{T}$ are a proper subset of the clauses allowed for Ω. However, we will see later that, in spite of the common usage of type predicates and other predicates, it is important and helpful to distinguish these predicates for technical reasons. First, in practice, we need to associate type predicates with syntax systems e.g. grammars in order to express the *structure* of objects described by them. Second, processing type knowledge is much more efficiently possible than less restricted knowledge, which also advocates for this separation.
- Property aggregation functions are usually monotone in software composition. For example, prices can be modeled with a domain $\mathbb{D}_{price} = \mathbb{N}_0$, aggregation $x \oplus_{price} y = x + y$, and consumption $x \ominus_{price} y = x - y$ where $x, y \in \mathbb{D}_{price}$. We also may have finite domains, e.g. for encrypted connections. Here, we may use $\mathbb{D}_{crypt} = \{1, 2\}$ where 1 and means encryption and 2 means no encryption, $x \oplus_{crypt} y = max\{x, y\}$, and $x \ominus_{crypt} y = x$.

The automated service composition problem is based on the concept of operations. An operation corresponds to a function in most programming languages with the difference that its description consists not only of the signature (inputs and outputs) but also of preconditions, postconditions, and non-functional properties. Inputs and outputs are parameter names that are not constant symbols or predicate symbols in Ω. Preconditions and postconditions are conjunctions of first order logic literals without functions. Variables in the preconditions must be inputs of the operation, and variables in the postconditions must be either inputs or outputs. The non-functional properties of an operation are expressed in an $|\mathcal{N}|$-vector. Since the non-functional properties $\mathcal{N}$ are ordered, each element of $\mathbb{D}_1 \times \ldots \times \mathbb{D}_{|\mathcal{N}|}$ defines a possible vector of non-functional properties.

Operation

Definition 2. *Let $\langle \mathcal{T}, \Omega, \mathcal{N} \rangle$ be a composition domain. An **operation** o is described by a tuple $\langle X_o, Y_o, Pre_o, Post_o, Z_o \rangle$ with*

- *disjoint sets of input variables X_o and output variables Y_o,*
- *preconditions Pre_o and postconditions $Post_o$ as conjunctions of literals without quantifiers and functions, with arbitrary constants, and with variables only from X_o and $X_o \cup Y_o$ respectively; and*
- *values of non-functional properties $Z_o \in \mathbb{D}_1 \times \ldots \times \mathbb{D}_{|\mathcal{N}|}$.*

Pre_o must contain exactly one positive type predicate (of $\mathcal{T}$) for each $x \in X_o$ with x as argument, and $Post_o$ must contain exactly one type predicate for each $y \in Y_o$ with y as argument; $Post_o$ must not specify type predicates for X_o.

The *semantic* of an operation is defined in terms of its usage. When using an operation, we must provide data for each for its inputs and may obtain some output values. The knowledge expressed in the preconditions must be true for the used inputs in order to be certain that the invocation will succeed. If this is the case, we may assume that the postconditions are true for the output values when the invocation finishes. The non-functional properties describe non-functional side-effects caused by the invocation of the operation, e.g. the costs, the time of execution, etc. The next section provides a formal definition of these aspects in terms of operation invocations.

2.1.2 Compositions

A composition consists of operation *invocations* and control flow elements in form of *guards* that organize these invocations. An invocation of an operation contains the information which operation is called, which values shall be used as inputs, and the variables where outputs are stored. So an invocation of an operation entails *variable assignments*. Structuring control flow elements are conditional statements as used in if-statements and loop headers in programming languages; I call these conditions *guards*.

The actually required output of the composition algorithm is often either an executable or a control flow graph. In software engineering, such a control flow graph is often documented as an UML activity chart, because it is easy to read for humans, which is also the reason why I adopt this notation throughout this thesis. However, for automated composition we find a more natural representation in *state charts*. In fact, as shown in Figure 2.2, this is almost an equivalent concept only that states are associated to nodes and operations are associated to edges, while activity charts have the opposite semantic. Hence, activity diagrams are useful for the reader, but the formal model will be defined using finite state machines.

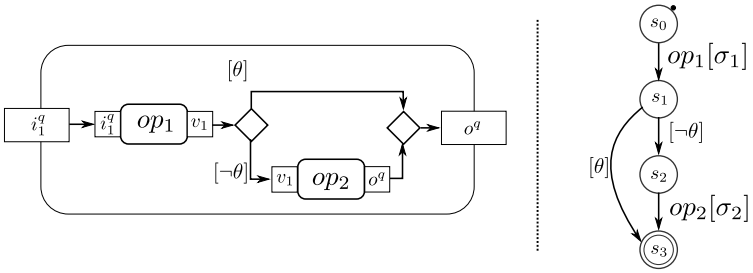


Figure 2.2: The same composition as activity chart (l) and state chart (r)

2.1.2.1 Composition Syntax

Data Containers and Domain Constants In a composition, operations are invoked with values from *data containers* or *domain constants*. Domain constants are constants like concrete numbers, names of cities, book titles, etc; these constants are sometimes referred to as *individuals*. Data containers are what we understand by variables in imperative programming. *Data Containers and Domain Constants*

In the composition problem, both data containers and domain constants are modeled as *logic constants*. The domain constants are simply the constants that occur in Ω. The constants that represent data containers stem from some previously defined set. Modeling programming variables as logic constants may seem unintuitive, but from the composition view point, these so called "variables" are simply objects that can be associated with content; so treating them as constants is quite natural and common practice in automated composition [59,60,107,122].

Definition 3. *Let Γ_{const} denote the (finite) set of constants contained in Ω and Γ_{data} be the (infinite) set of constants used as data container identifiers. We call the elements of Γ_{const} and Γ_{data}* ***domain constants*** *and* ***data containers*** *respectively.*

I chose the term data container in order to avoid confusion between the role of these objects in the model. Intuitively, data containers are programming variables. However, in order to avoid confusion with logic variables (as they occur in the definitions of operations), I call them data containers, which does not suggest the character of a logic variable.

Substitutions In order to ground operations and clauses from the knowledge base to these constants, we need *substitutions*. The term is defined as in first-order logic. *Substitutions*

Definition 4. *Let α be a formula and let σ be a (partial) mapping from variables in α to other variables or constants. We call σ a* ***substitution*** *and denote as $\alpha[\sigma]$ the* ***substituted formula****, which is α where each occurrence of variable u is replaced by $\sigma(u)$ if $\sigma(u)$ is defined.*

Note that substitutions can be used not only for grounding but are more powerful. That is, they may bind variables not only to constants but also to other variables. In particular, a substituted formula is not necessarily grounded in the sense that all its literals contain only constants as arguments. Hence, substitutions should be really understood in the spirit of Robinson as introduced for resolution in first-order logic [104].

Operation Invocations and Guards The core pieces of a composition are operation *invocations*. An operation invocation consists of the name of the operation, an input mapping,

and an output mapping. The input mapping defines the data containers and domain constants that are used for the invocation, and the output mapping defines the data containers where the results of the invocation will be stored.

Operation Invocations

Using a substitution, we can replace identifiers in preconditions or postconditions of operations and in clauses of the background knowledge. This allows us to ground the preconditions and postconditions of an operation to data containers or domain constants. We can then formally define an operation invocation as follows.

Definition 5. *Let Γ be the space of constants with $\Gamma_{data} \subseteq \Gamma$ being the data containers. An **operation invocation** is an operation $o \in O$ with a combined mapping $\sigma = \sigma_{in} \cup \sigma_{out}$ where $\sigma_{in} : X_o \to \Gamma$ is an input mapping and $\sigma_{out} : Y_o \rightsquigarrow \Gamma_{data}$ is a partial and injective output mapping; we write $o[\sigma]$.*

Guards

In addition to operation invocations, the composition may also contain *guards*. Guards are *conjunctive* logical formulas defined on data containers and domain constants. They can be used to define at design time the behavior of a composition based on the concrete values stored in data containers at runtime. This is exactly the same as the Boolean expressions used as conditions in if-statements or loops in all programming languages.

Definition 6. *A **guard** is a conjunction of FOL literals without quantifiers, functions, or variables, with constants only from Γ, and containing only* implemented *predicates (e.g. $=$, $\leq$, $\in$, predicates in $\mathcal{T}$, ...).*

The term "implemented predicate" means interpreted predicates for which we have a function that can compute its truth value at runtime. For example, the predicate $<$ is interpreted with the "less than" relation and can be computed. In contrast, a predicate *isAvailable* has a *suggested* interpretation based on its name but cannot be computed.

Compositions

Compositions As usual, I model a composition as a deterministic finite automaton (DFA) whose transitions are operation invocations and guards. In order to obtain a *syntactically* sound composition, the transition function is constrained in that

1. every transition is either an operation invocation or a guard;
2. if a state has an outgoing transition that is an operation invocation, then there is no other outgoing transition from that state (the represented program is deterministic and well-formed);
3. if a state has an outgoing transition that is a guard θ, it has exactly one additional outgoing transition with the negation of θ; and
4. every state must be reachable from the initial state, and a final state must be reachable from any state (no dead-locks).

This definition is along with all common approaches in literature [10, 12, 59, 69]. Some approaches restrict the syntax in that they only consider compositions without guards [59, 69].

This leads to the following formal definition of a composition:

Definition 7. *Let O be a finite set of operation descriptions. A **composition** c is a DFA $\langle S, \Sigma, \delta, s^0, F \rangle$, where*

- *S is a finite set of* opaque *states (i.e. nodes without any information),*

- Σ *is a finite set of state transition labels,*
- $s^0 \in S$ *the initial state,*
- $F \subseteq S$ *are the final states, and*
- $\delta : (S \setminus F) \times \Sigma \to S$ *is a state transition function, such that*
  1. *if* $\delta(s, a)$ *is defined, then* a *is an operation invocation or a guard;*
  2. *if* $\delta(s, a)$ *is defined and* a *is an operation invocation, then* $\delta(s, a')$ *is not defined for any* $a' \neq a$ *(there is no other transition from* s*);*
  3. *if* $\delta(s, a)$ *is defined, if* $a = \theta$ *is a guard, there is exactly one other outgoing edge from* s*, and that edge is labeled with* $\bar{\theta}$*, i.e. the negated version of* θ*; and*
  4. *every state is reachable through* δ *from* s^0*, and a final state must be reachable from every state.*

A composition is ***sequential*** *if none of its transitions is a guard; it can then be written as a chain* $\langle o_1[\sigma_1], \ldots, o_n[\sigma_n] \rangle$ *of operation invocations.*

We denote as $\mathcal{C}_O$ *the* ***set of compositions*** *that can be constructed with operations of* O*.*

Usually, we do not care about the concrete names of data containers in the compositions as long as the data flow remains consistent. In this sense, we consider two compositions *equivalent* if they are isomorphic modulo renaming of data containers.

Equivalence of Compositions

Definition 8. *Two compositions* c_1 *and* c_2 *are* ***equivalent*** *if* $c_1 = c_2$ *or if there is a composition* c' *with data containers not occurring in* c_1 *or* c_2 *such that there are injective substitutions* σ_1 *and* σ_2*, and replacing all data containers in* c_1 *and* c_2 *by* σ_1 *and* σ_2 *respectively yields* c'*.*

In this thesis, I will focus on compositions with particular properties. That is, we want to work with *valid* compositions in the sense that we know that the operation invocations are actually applicable in the states in which they constitute a state transition. Also, we want to ignore compositions that contain useless transitions. However, to specify these conditions, we must first define the semantics of compositions.

2.1.2.2 Composition Semantics

Composition (State) Labels The semantics of compositions is defined in terms of logic labels associated with states and constraints on these labels. In the above definition of compositions, states are opaque nodes, which is appropriate on the syntactic level. Now, we associate states with knowledge known to be true whenever the executing process arrives at them. Formally, such a state labeling is a conjunction of ground first-order literals, i.e. predicate whose arguments are constants. Then we can formalize a state labeling as follows.

(State) Labels

Definition 9. *A* ***(state) label*** *is a conjunction of FOL literals without variables or functions, with constants only from* Γ_{const} *and* Γ_{data}*, and with at least one type predicate from* $\mathcal{T}$ *for each data container in it. We denote as* $\mathcal{L}$ *the set of all such labels.*

Given a composition c *with states* S*, a function* $\lambda : S \to \mathcal{L}$ *is called a* ***(state) labeling*** *for* c*. The data containers that occur in a state* s *under labeling* λ *are denoted as* $\Gamma_{data}(\lambda(s))$*.*

States may contain general or value-specific assertions. At design time, we usually cannot know the particular value of a data container, so our states are mostly defined on a level of

information that is already available at design time, e.g. that c is the availability of a book b and that c is a Boolean. However, making use of guards, which will be explained next, would also allow to know at design time that, in a particular situation, the value of c is *true*. Value-specific information is not constrained to exact values but could also be a range information, e.g. the value of the data container d containing some distance is between 50 and 100.

Applicability of Operation Invocations and Guards

Semantics of Compositions On the basis of states, we can express the *applicability* of operation invocations and guards. An operation invocation is applicable in a state s, if each of the used inputs is at least as specific as the required inputs, if its *substituted* preconditions are contained in s, and if none of the outputs is written to a data container that existed already in s; this definition is equivalent to the definition of applicability in [59, 122][2]. A guard is applicable in s if all of its constants domain constants or contained in s.

Definition 10. *Let c be a composition, λ be a labeling for c, and s be a state of c. An **operation invocation** $o[\sigma]$ is **applicable** in s **under** λ if $\lambda(s) \wedge \mathcal{T} \wedge \Omega$ is consistent, $\lambda(s) \wedge \mathcal{T} \wedge \Omega \models Pre_o[\sigma_{in}]$, and $\sigma_{out}(Y_o) \cap \Gamma_{data}(\lambda(s)) = \emptyset$. A **guard** φ is **applicable** in s **under** λ if its constants are from $\Gamma_{data}(\lambda(s)) \cup \Gamma_{const}$.*

Remarks.

- The definition of applicability entails that a state satisfies the preconditions of an operation under the input mapping, and that no data containers contained in s are overwritten by the invocation.
- the set of operation invocations applicable in a state is not finite, because the number of output mappings is infinite.
- the set of guards applicable in a state is always finite, because the number of constants and predicates is finite.

Valid Labeling

The labeling for a composition is *valid* if the following two conditions hold. First, every operation invocation or guard belonging to an outgoing edge of a state must be applicable in it. Second, the label of each state can be inferred from the label of each predecessor state modulo background knowledge. Formally, this amounts to the following:

Definition 11. *Let c be a composition. We say that λ is a **valid labeling** for c if for every transition $\delta(s, a) = s'$ it holds that*

1. *a is applicable in s under λ,*
2. *$\lambda(s) \wedge \theta \wedge \mathcal{T} \wedge \Omega \models \lambda(s')$ if $a = \theta$ is a guard, and*
3. *$\lambda(s) \wedge Post_o[\sigma] \wedge \mathcal{T} \wedge \Omega \models \lambda(s')$ if a is an operation invocation $o[\sigma]$*

where θ and $Post_o[\sigma]$ must not contradict $\lambda(s) \wedge \mathcal{T} \wedge \Omega$.

Two remarks are due to relate this definition to common AI literature notation:

- AI literature typically does not use validity of state labelings but simply provides a concrete function for *the* correct labeling i.e. defines a reference labeling which is valid

[2] There, the type system is encoded within the (belief) state.

by construction [59, 107, 122]. The reason to deviate from that practice is that, in the composition process, we may obtain different labelings for the states depending on how we search. This difference arises from different ways how facts implied by the background knowledge are made explicit at different points of time. The above approaches refrain from making such knowledge explicit and define the semantics exclusively in terms of the operation postconditions that are added to the previous state. However, I find this unnecessarily restrictive and, hence, define conditions for labelings that must be true and leave the choice for the concrete labeling to the respective approach instead of enforcing a reference labeling.

- The fact that the postcondition of an action cannot contradict earlier knowledge implicitly defines the "Invocation and Reasonable Persistence" (IRP) assumption [68, 85]. This assumption says that knowledge we obtained once remains valid throughout the rest of the composition. This is in contrast to other approaches not making this assumption [59, 122] and adopting belief revision. I defer a detailed explanation and motivation of the IRP assumption to Section 2.2.2. Here, it is sufficient to know that it ensures a certain form of monotonic growth of knowledge along the execution path.

Definition 12. *A composition c* ***transforms*** *some initial state Pre into a goal state Post iff there is a valid labeling λ for c such that $Pre \models \lambda(s^0)$ for the initial state s^0 and $\lambda(s^f) \models Post$ for every final $s^f \in F$ of c.*

A composition c achieves this transformation ***minimally*** *if the composition induced by removing any transition $(s, a) = s'$ from c and re-rooting all transitions departing from s' to s does not transform Pre into Post anymore.* *Minimal Compositions*

We can make the following observation for minimal compositions:

Observation 2.1. *Let c be a minimal composition. Then, for every transition $o[\sigma]$ in it that is an operation invocation, there exists another transition that uses a data container as input or parameter that is written by $o[\sigma]$, i.e. bound in σ_{out}.*

In other words, a composition that achieves some transformation minimally, for every operation invocation at least one output is outputted or used by another operation invocation.

2.1.2.3 Non-Functional Properties of Compositions

Just as operations, compositions also have non-functional properties. Of course, the non-functional properties of a composition depend on the non-functional properties of the operations in it. Taking into account the structure of the composition, the non-functional properties of the operations contained in the composition are merged by an aggregation function. The general definition of an aggregation function is as follows:

Definition 13. *An* ***aggregation function*** *for compositions $\odot : \mathcal{C} \to (\mathbb{D}_1 \times \ldots \times \mathbb{D}_{|\mathcal{N}|})$ assigns a vector of non-functional properties to each composition in a composition space $\mathcal{C}$.*

Note the difference between property-specific aggregation functions and the aggregation functions for compositions. Property-specific aggregation is the basis for the aggregation function of compositions. In other words, the aggregation function for compositions relies on the property-specific aggregation functions. The difference will be made clear through the respective notation throughout the thesis.

There is no particular aggregation function that is used for every composition problem. For example, if a composition contains two branches created by guards with distinct final states at the end respectively, then at runtime only one of the branches will be executed. A pessimistic aggregation function could take the maximum values of the aggregated non-functional properties of the respective branches; it accounts for the worst case. Another aggregation function could estimate the likelihood of entering one of the branches and use a mean value instead. A detailed overview of aggregation functions can be found in [19]. Since several aggregation functions may be used, the concrete choice is part of the composition problem definition.

2.1.3 Queries

Intuitively, software composition means to answer composition *queries*. That is, some client specifies a desired service and poses conditions that should be satisfied by the algorithm.

A client's query consists of four elements.

1. *Desired Operation.* This part defines the desired functionality for which an implementation is searched and the non-functional properties that this implementation should exhibit. This simply corresponds to a description of an operation as in Def. 2.
2. *Answer Type.* The client may expect three types of answers:
   (a) only one solution,
   (b) all (minimal) solutions not dominated w.r.t. non-functional properties, or
   (c) all (minimal) solutions.
3. *Composition Structure.* The client may require that the algorithm searches for arbitrary compositions or limit the search to sequential compositions, which are easier to find.
4. *Aggregation Function.* As discussed, several aggregation functions are imaginable. However, the number of reasonable functions is rather small, so the client will not specify this function freely but select one from a rather small set of candidates.

This yields the following formal definition of a query.

Definition 14. *A **query** is a tuple $q = \langle o_q, at_q, as_q, \odot_q \rangle$ where o_q is an operation description, $at_q \in \{one, all\text{-}nondominated, all\}$ is an answer type, $as_q \in \{seq, complex\}$ is the answer structure, and $\odot_q$ is the aggregation function.*

In the remainder of this thesis, I will use q as a subscript of inputs, outputs, precondition, postcondition, and property bounds of the operation described in the query. That is, X_q, Y_q, Pre_q, $Post_q$, and Z_q will denote the inputs, outputs, precondition, postcondition, and property bounds of the query respectively.

2.1.4 The Composition Problem

I define the automated service composition problem as a specific search problem as formalized by Garey and Johnson [43]. Given a query, find (depending on the query) one or more compositions out of the space of admitted structures that minimally transform the query precondition into its postcondition, and observe the bounds on the non-functional properties defined the query (according to the aggregation function specified in it). Formally, this amounts to the following:

Definition 15. *A **composition problem** is a set of finite* instances $\langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$ *where* $\langle \mathcal{T}, \Omega, \mathcal{N}\rangle$ *is a composition domain, O is a set of available operations, and q is a query. A **solution** to an instance is a composition $c = \langle S, \Sigma, \delta, s^0, F\rangle$ that minimally transforms Pre_q into $Post_q$ and $\odot(c) \leq Z_q$. An algorithm solves the problem if it returns a solution (or, depending on q, every (non-dominated) solution) within finite time for every instance.*

Solution Composition

A composition problem is

- ***sequential** if $as_q = seq$ for all of its instances, and*
- ***positive** if the preconditions and postconditions of all operations and possible queries contain only positive literals and if Ω consists of definite Horn clauses.*

Positive and Sequential Composition Problems

The bounds Z_q specified in the query can be seen as upper bounds without loss of generality. Values for properties that should be maximized must be specified in a negated form. So if we want to have a throughput of at least 5, we must specify a value of -5 for the respective value in Z_q; the respective values of all the available operations are also assumed to be given as negative values.

The cost of sequential compositions is simply the sum of the individual cost values, because every operation invocation occurs exactly once. We can simply aggregate the values of non-functional properties along the sequence; no other aggregation function would make sense. Hence, in the discourse of this thesis, I assume the following aggregation function for sequential composition problems:

$$\odot(c)_p = (Z_{o_1})_p \oplus_p \ldots \oplus_p (Z_{o_n})_p$$

where c consists of a chain of operation invocations $\langle o_1[\sigma_1], \ldots, o_n[\sigma_n]\rangle$ and p is a non-functional property.

2.2 Implicit Assumptions

The above model makes assumptions that are not discussed in detail above but important for the correctness. This section explains these assumptions in more detail.

2.2.1 Open World Assumption

Most planners make the so called closed world assumption. The closed world assumption is defined for the context of finite logic languages (e.g. propositional logic or predicate logic with a finite set of constants, without existence quantifiers or functions) and amounts to extend the modeling relation. Formally, $\alpha \models_{cwa} \beta$ is true iff $\alpha \wedge \gamma \models \beta$ with $\gamma = \bigwedge_{\alpha \not\models L} \neg L$ where L is a positive ground literal. In other words, we may extend α by every negated version of a literal that does not follow from α. For example, in classical planning, every literal not contained in a state is not inferable and may be assumed to be false (by convention).

However, the closed world assumption cannot be applied to software composition [95]. We simply do not know what holds in the domain except what is given explicitly in Ω, so the fact that an information is not stored in a state does not say anything about the validity of that information. If some literal L is not contained in a state and cannot be inferred neither, L may or may not be true; we simply have no information about it.

Knowing that we work under open world assumption is important because this affects how we must model states and operations. The difference to classical planning is that we do

assume to have only *partial* interpretations of the world. The semantics of a state are affected in so far that literals not contained in it are not false but *unknown*. In particular, states may contain single negative literals in order to express that we know that a particular property does not hold. Moreover, negative literals in the preconditions and postconditions must be interpreted differently. Negative literals in the precondition mean that we must explicitly know that some information is not true; in classical planning it is sufficient to know that the positive version of the literal is not contained in the state. Negative literals in the postconditions mean that we may add the explicit negative knowledge to the state. In addition, operations may need a *delete list* of literals that are not explicitly true anymore (but also not false) after the invocation. In particular, the delete list may contain both positive *and* negative literals, which cease to be true.

In the planning literature, these cases are often modeled using *beliefs* [15]. Beliefs are nothing else than *sets* of states that are considered possible. Intuitively, a belief contains all the combinations of variables that are compatible with our current knowledge. In this thesis, I simply do not model facts that are not certain to be true or false. Hence, the states in this thesis are only a compact representation of beliefs in belief-space planning.

2.2.2 IRP Assumption: (No) Expiration of Information

Automated software composition faces the problem that acquired knowledge may become invalid over time. Information may become invalid either through some external event or through operations invoked by the composition itself. For example, the information that a seat ticket is available may change either due to the fact that somebody else reserves it or because some operation invocation in the composition itself reserves it.

Knowledge invalidation may impose strange inconsistencies. For example, consider that we have a state label $P(u, v) \wedge (v = \mathit{true})$ and Ω entails the rule $P(x, y) \wedge (y = \mathit{true}) \rightarrow Q(x)$. Obviously, the state entails $Q(u)$. Now suppose that we invoke an operation that has $\neg Q(u)$ as a postcondition. Then we may obtain a state label $P(u, v) \wedge (v = \mathit{true}) \wedge \neg Q(u)$, which is (implicitly) inconsistent. Of course, the state itself is consistent, but it yields a contradiction when combined with the knowledge base.

The underlying problem is that some information of the state is not valid *anymore*. In the example, the literal $P(u, v)$ expired when the reservation was made. In other words, we should not be able to apply the rule anymore that yields the contradiction. However, unless this expired information is removed from the knowledge base, we can draw false conclusions.

Some approaches resolve this problem using *belief revision*. For instance, this is done in [59] and [122] through the notion of the possible models approach (PMA) [125]. The idea is to remove as little previous knowledge as possible in order to make a belief consistent again.

However, I think that belief revision and other common techniques in the regard such as situation calculus are debatable solutions for this problem. While belief revision is reasonable in other planning domains, explicitly obtained knowledge never becomes actually invalid in a composition setup unless we overwrite data containers; hence, there seems to be no reason to revise knowledge. In fact, in the above example, the literals $P(u, v)$ and $v = \mathit{true}$ *are* still valid even when $\neg Q(u)$ becomes true. The problem is that a particular fact, here $P(u, v)$, has become *semantically* inconsistent. We could associate such a predicate with an *external* time stamp, e.g. $P(u, v, t)$. This does not make $P(u, v, t)$ a fluent in the sense of situation calculus, because t refers to an external and not an internal time stamp. For example, u could be a flight ticket and v the availability of the ticket. While v contains a valid information for u only

for a particular real world time stamp t, the fact that v *models* the availability of u at t stays correct throughout the whole composition, i.e. it is not only true in a particular situation in which the plan executor may reside; in particular, it does not need to be revoked. But of course, we need to take into account that the *actual* world state is not necessarily represented by our model, which restricts the conclusions we may draw. I am not aware of any planner that considers a belief model capable of expressing this kind of timed knowledge, and solving this issue seems to be highly non-trivial.

In this thesis, I avoid this trouble by applying the assumption of *invocation and reasonable persistence* (IRP) [68, 85]. This assumption says that every information that has been acquired remains valid for a reasonable time, which is usually defined as "until the composition terminates". In other words, we simply assume that information does not expire.

The benefit we gain from the IRP assumption is that we do not need belief revision or timed models. If we assume that knowledge acquired once remains valid forever (or at least through the composition), then we can only obtain an inconsistent state if the underlying composition model is inconsistent. That is, whenever we obtain a contradictory state, we know that the underlying model is inconsistent, because it allowed us to gather inconsistent information without changing anything in the world. Hence, we can simply add the postconditions of the operations to our knowledge with the only risk being that we gather the same information several times. In particular, we do not need delete lists or any concept of the like.

The IRP assumption implies a notable restriction, but there is still a significant set of relevant problems that can still be solved under this condition. First, the IRP assumption applies in every information gathering scenario. That is, whenever all the operations are only sensing operations that provide us with information about the world but do not change them, the IRP assumption holds. Second, even if we have operations that change the world, these do not always require information expiration treatment. For example, we may have an operation that adds a client to the database. Using this operation does change the world but does not necessarily invalidate previously obtained knowledge.

2.2.3 Semantic Incompleteness of the Query

The formal definition of the composition problem somewhat hides the fact that a solution of the formal model is not necessarily a solution desired by the client. That is, the client may have requirements about the software that have not (and possibly *cannot*) be adequately specified on the formal level. For example, the client wants a piece of software that applies image filters in a way that the color channels of sharply recognizable objects are turned to black and white [66]. Using logic descriptions, this goal can only be described very roughly, but whether or not a composition really satisfies the client's requirements can only be decided by the client itself after invocation and not on the basis of the formal model.

This means that the formal definition of a solution is only a *necessary* but not a sufficient condition. Unless the client explicitly says that all requirement definitions have been captured in the formalism, we cannot decide on the algorithmic level whether a solution is a "real" solution or not. The ultimate decision remains to the client. In the formal framework, the client has the possibility to declare the query as semantically complete by setting at_q to "one", i.e. to require only one solution.

Consequently, the composition algorithm cannot generally terminate when the first solution has been found. If a solution is found on the formal level, it can be announced to the client, and the client then decides whether the candidate is really a solution. The direct

consequence is that the composition algorithm does not generally terminate at all, because there may be an infinite number of solutions and, even if many are found, the algorithm only announces the solutions but does not terminate until it is signaled by the client that an acceptable solution was returned. Hence, the algorithm returns a solution *stream*.

2.3 Problem Complexity

Understanding the complexity of a problem is important to get an idea of the type of solution that can be expected and the conditions. Insights on complexity can either guide our strategy for possible relaxations or give us an argument that even though an algorithm is not efficient, it is (asymptotically) also not worse than any other algorithm that solves the problem.

Unfortunately, even the sequential composition is undecidable. In fact, this holds even for the sequential composition problem (see appendix for the proof, which is basically taken from [59]).

Theorem 2.2. *Let $p = \langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$ be the instance of a sequential composition problem. The decision problem whether a solution to p exists is undecidable.*

Obviously, this result generalizes to non-sequential composition problems. This is merely due to the fact that we do not need non-sequential compositions to simulate the Abacus and, hence, the above proof works equally for the case of compositions in general.

Corollary 2.3. *Let $p = \langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$ be the instance of a composition problem. The decision problem whether or not a solution to p exists is undecidable.*

Given that even NP-hardness sometimes provokes capitulation, the above result may suggest that automated service composition is a hopeless undertaking from the complexity viewpoint. Hence, we may come up with a conclusion like the following: "On the computational side, the problem seems to be semi-decidable, so it is not quite clear if the result has any practical importance."[3]. Then we would declare the property of undecidability a sufficient condition to render a problem irrelevant for the real world.

My viewpoint is very different from that one. I think that complexity results are interesting from the theoretical viewpoint but, except in shaping and steering expectations, without relevance in practice. Also in decidable setups, we cannot wait for the answer in the worst case. This is frequently the case for problem that are NP-hard or beyond but may even be an issue for problems in P. In practice, we will always operate with time bounds. If we return a negative answer because we hit the time bound, it does not matter whether we were simply not fast enough or whether we would have never found any answer. Of course, a possible way out is to search for relevant and more feasible subproblems as successfully done by Hoffmann et al. using strict forward effects [59]. However, if we do not want to diminish the potential abilities of the composition system, we must accept that we cannot expect the system to always deliver an answer. Hence, my approach here is rather to take the problem as given and try to make the best out of it; this thesis is about the results of this attempt.

This strategy is also common practice in other areas. As pointed out in [59], there is high activity on solving numeric planning problem in spite of their undecidability. Also, SMT solvers, which solve arbitrary first-order (sometimes even second-order) logic satisfiability problems, are important tools not only in science but also in practice, e.g. in verification [37].

[3] The quote is from a review we obtained on a submission at IJCAI 2015

3. Composition as a Search Problem

This chapter presents the search framework used by the composition approaches in Chapter 4 and Chapter 5 and gives a brief introduction to search for the uninitiated reader. It largely depends and builds on top of previous works on search, most notably Nilsson's textbook on artificial intelligence [93] and Judea Pearl's elaborations on heuristic search [94]. The reader familiar with search will find little new aspects. Some differences to the traditional material are that I adopt a dedicated pruning framework similar to the ones used in [48, 116], that costs are non-additively aggregated vectors instead of scalars, and that I apply a view that clearly separates the search graph from the strategy used to explore it.

The chapter consists of three sections. The formal description of a search problem in terms of a search *structure* as the basis of the search *algorithm* is described in Section 3.1. Section 3.2 describes the search algorithm itself. It implements a heuristic search guided by an *exploration strategy*. However, since the client may have several objectives, there is not a "best" such strategy for software composition. Therefore, Section 3.3 discusses different *intentions* with which a search structure related to software composition may be explored.

3.1 Search Based on a Search Structure

This section gives a brief introduction to search based on a search structure and how software composition matches into this setup. It is organized as follows. Section 3.1.1 briefly describes what a search problem is and gives an intuition of *search structures*, which are the basis for the search process. Second, Section 3.1.2 defines search structures on a formal level. Finally, Section 3.1.3 defines desirable properties of a search structure that are important in order to guarantee correctness and completeness of the algorithm used to explore it.

3.1.1 What a Search Problem is and how it can be Approached

Garey and Johnson define a *search problem* as follows [43, p. 110]:

> A search problem Π consists of a set D_Π of finite objects called *instances* and, for each instance $I \in D_\Pi$, a set $S_\Pi[I]$ of finite objects called *solutions* for I. An algorithm is said to *solve* a search problem Π if, given as input any instance $I \in D_\Pi$, it returns the answer "no" whenever $S_\Pi[I]$ is empty and otherwise returns some solution s belonging to $S_\Pi[I]$.

The first part of this definition is analogous to my definition of a composition problem. That is, the set D_Π corresponds to a set of composition problem instances of the form $\langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$, and there is a (possibly empty) set of solutions for these instances. So, intuitively, the composition problem is a search problem.

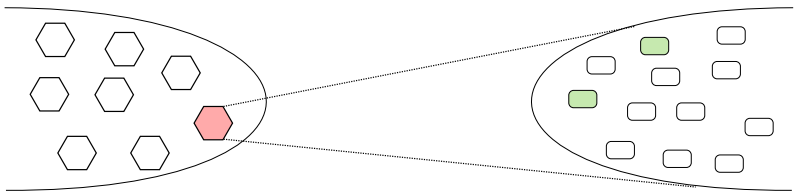


Figure 3.1: A problem instance with two solutions in the search space.

Problem Space
Search Space

In the following, I will use the terms *problem space* and *search space* for the sets of instances and solution *candidates* respectively. The problem space is simply the set of all possible composition problem instances, i.e. what corresponds to D_Π in the definition of Garey and Johnson. The set of solutions $S_\Pi[I]$ of an instance $I \in D_\Pi$ is obviously not known; it is a subset of an actually examined set not mentioned in the above definition and which I call the search space. In the case of composition, the search space is the set of all compositions that can be constructed. Figure 3.1 illustrates the relation between a particular composition problem instance and the corresponding search space.

In contrast to the first part of the definition, the second part, which refers to the "solves"-property of an algorithm, is not appropriate for the composition setting. The first problem is that Garey and Johnson apparently assume that the question whether or not the set of solutions is empty is decidable, which we proved to be false in our setup (cf. Section 2.3). The second problem is that the algorithm is only required to return *one* solution, while, in the composition setting, more answers may be requested (cf. Section 2.1.1), e.g. if the query is incomplete (cf. Section 2.2.3) or if the client wants to choose among several solutions that are Pareto optimal with respect to the non-functional properties.

In spite of this difference, the composition problem is still clearly a search problem. I would argue that Garey and Johnson describe one particular search problem class (decidable problems for which one solution is desired), while the composition problem is a search problem that is not decidable and where multiple solutions may be required. However, the basic idea of attempting the composition problem is the same as for classical search problems. That is, given a search (composition) problem instance, we somehow want to determine elements of the search space that satisfy the solution property.

Search Problem

Hence, I modify the above definition as follows: A search problem consists of a set $\mathscr{P}$ of finite objects called *instances* and, for each instance $\mathfrak{p} \in \mathscr{P}$, a set $\mathscr{S}$ of finite objects called *search space* for $\mathfrak{p}$ and a set $\mathscr{S}^* \subseteq \mathscr{S}$ called *solutions* for $\mathfrak{p}$. An algorithm solves the search problem if, given any instance $\mathfrak{p} \in \mathscr{P}$,

- if $\mathfrak{p}$ requires that *one* solution is found, it returns a solution s of $\mathscr{S}^*$ whenever $\mathscr{S}^*$ is not empty; and
- if $\mathfrak{p}$ requires that *every distinct* solution with a property ξ is found, it returns (in a stream) every s of $\mathscr{S}^*$ that satisfies ξ.

In other words, the algorithm does not need to decide on the existence of a solution but only must return one (or many) if the set of solutions is not empty. Note that, in the second case, the algorithm is allowed to return also solutions that do not satisfy ξ, but these do not *need* to be returned. Hence, the algorithm is *allowed but not obliged* to ignore solutions that do not satisfy ξ. Since the number of solutions may be infinite, the solutions are not collected in a set and finally returned but the algorithm outputs them in a possibly infinite stream.

Obviously, the composition problem is a search problem of this type. Whether a concrete composition problem belongs in the first or the second of the above cases depends on the answer type at_q specified in the query. Three values are possible:

1. $at_q = one$. This corresponds to the first case, e.g. one solution is required if one exists; otherwise, the algorithm is not required to halt.

2. $at_q = all$. This corresponds to the second case with ξ allowing to filter non-minimal compositions (cf. Def. 7 in Section 2.1).

3. $at_q = all\text{-}nondominated$. This corresponds to the second case with ξ allowing to filter non-minimal compositions and those that are dominated by others with respect to non-functional properties.

So the search algorithm must be able to work in different modes depending on the query.

Distinct Solutions

For the case that more than one solution must be returned, there may be equivalent search space elements of which we only need to consider one. This is what is meant by the requirement that every *distinct* solution is returned. We assume that there is an equivalence relation among the elements of the search space, and we only need to return one solution from each equivalence class induced by this relation. Naturally, we require that every equivalence class either contains no solution or that all of its elements are solutions.

In the case of software composition, we define the equivalence relation as equality modulo the naming of data containers. That is, two compositions c and c' are equivalent in this sense if they only differ in the names of data containers in the operation invocations and guards, i.e. renaming of data containers in one composition yields the other one (cf. Def. 7 in Section 2.1.2). The implication for search is that the data container names are irrelevant. We do not need to return two compositions that differ only in the data container naming; it is sufficient to return one of them.

3.1.2 Formal Definition of a Search Structure

Since the search space is usually not given explicitly, one cannot simply iterate over its elements until a solution is found. Instead, the elements of the search space must be somehow *created* until a solution is found.

The common technique is to convert the general search problem into a path search problem [93]. That is, the general strategy to identify solution elements is encoded into a graph definition, the *search graph*. The search graph is given either through an inductive definition or by a non-deterministic algorithm whose choice points induce successor nodes. Some nodes of the search graph are *goal* nodes, and the task is to find a path from a distinguished initial node to one of these goal nodes. The relation between the two problems is that paths in the graph correspond to elements of the search space. The actual search is then realized with a standard (shortest) path algorithm like A*. Of course, search graphs encode algorithmic

ideas, so we may use different search graphs for the *same* search problem. For example, this thesis discusses two possible search graph types for the sequential composition problem.

In order to connect the search graph with the search space, I apply a *transformation function*. The search algorithm identifies goal nodes in the search graph, but the paths to these nodes are not solutions to the search problem themselves. That is, a path to a goal node *contains* the information necessary to create a solution but it does not *encode* the solution itself. Hence, a transformation function is needed to derive a solution of the search space from the information encoded in the path to the identified goal node.

Search Structure

I call the search graph together with the transformation function and the goal node function the *search structure*. Let $\mathscr{P}$ be a search problem. A search structure $\mathcal{S}$ for $\mathscr{P}$ defines three elements for each instance $\mathfrak{p} \in \mathscr{P}$. That is, $\mathcal{S}(\mathfrak{p})$ is a 3-tuple with the following items:

1. the search graph G,
2. the transformation function TRANS to translate paths into solutions, and
3. the goal function $\star$, which decides whether paths to a node encode solutions

I now describe these elements formally.

3.1.2.1 Search Graph

The search graph is the graph that is actually traversed by the search algorithm. Since the graph is usually infinite, it is specified inductively through a root node and a function that generates successor nodes. Formally, we write

$$\text{GETROOT} : \rightarrow \widehat{N} \quad \text{and} \quad \text{GETSUCCESSORS} : \widehat{N} \rightarrow 2^{\widehat{N}}$$

where $\widehat{N}$ is the space of possible node encodings, GETROOT is the (parameter-less) function that generates the root node of the search graph, and GETSUCCESSORS is the function to compute the child nodes for any node n. In the following, we write n^0 for the root of the search graph for a concrete instance. The actual search graph is then $G = (N, E)$ where

$$N = \bigcup_{i=0}^{\infty} N_i \text{ with } N_0 = \{n^0\} \text{ and } N_{i+1} = \{n \mid \exists n' \in N_i \text{ such that } n \in \text{GETSUCCESSORS}(n')\}$$

are the nodes and

$$E = \{(n, n') \mid n' \in \text{GETSUCCESSORS}(n)\}$$

are the edges.

Paths of the Search Graph

The edges imply a set of *paths* in the search graph. Formally, we write

$$P = \{(n_0, \ldots, n_k) \mid n_i \in N \text{ and } (k = 0 \text{ or for every } n_i, n_{i+1}, (n_i, n_{i+1}) \in E)\}$$

as this set of paths. There is always a path of length 0 from a node to itself. We write P^0 to denote the set of paths where the first node is n^0. In the following, when talking about paths, I assume that these are from P^0; exceptions will be indicated explicitly.

Node Reachability

In the following, we will also be interested in the nodes *reachable* from a given node. Hence, we define

$$\mathit{desc}(n) = \{n' \mid \exists (n, \ldots, n') \in P\}$$

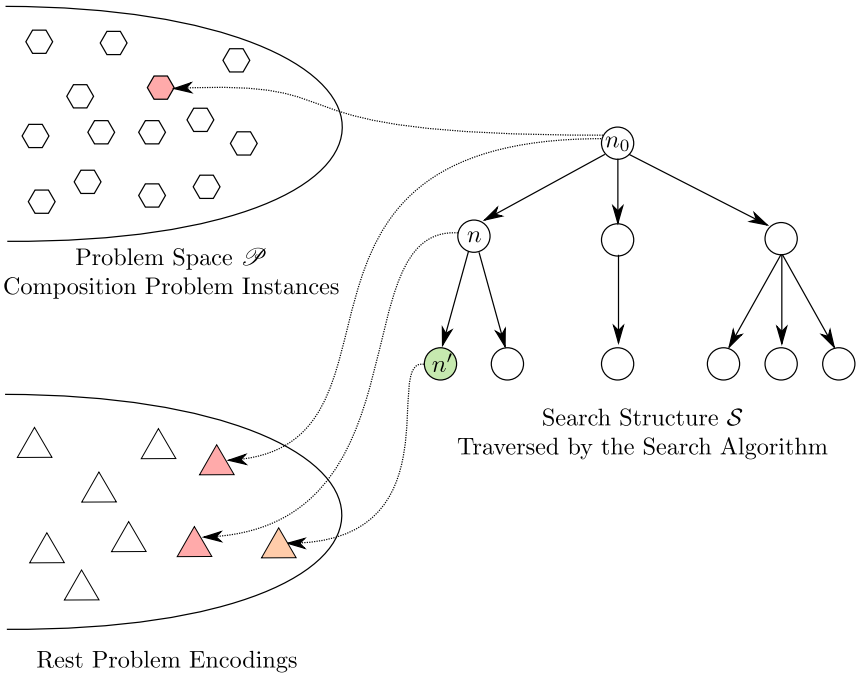


Figure 3.2: Every node in the search graph induces exactly one rest problem.

In other words, $desc(n)$ is the set of nodes to which there is a path from n. This definition implies that a node is always reachable from itself.

Nodes actually encode *rest problems*. What the rest problem means in terms of the domain depends on the semantics of nodes and edges. For example, the rest problem in the common encoding of the n-queens problem is described by the task to place k (with $0 \leq k \leq n$) queens on a $n \times n$ chess field that already contains $n-k$ queens such that no queen threatens another one. In particular, the rest problem belonging to the root node corresponds to the original search problem, potentially with a different encoding.

Figure 3.2 shows the relation between the search structure and the *rest problem space*. While the problem space reflects the clients' view on the problem, the rest problem space reflects the expert's internal view of the problem based on his perspective of how it is solved.

Besides the computation of GETSUCCESSORS, the node encoding is also important for other purposes. First, we can use it to estimate the distance to a solution. Based on its encoding, we can estimate how *promising* a particular node is in the sense that we are likely to find a solution close to the node. Second, we can use it for *pruning*. Pruning means to exclude a particular node from further considerations. Suppose that an algorithm that traverses the search graph discovered two nodes n and n'. Based on the graph structure, we may know that we can ignore n. For example, we may know that the existence of a goal node reachable from n implies the existence of a goal node reachable from n'.

Pruning

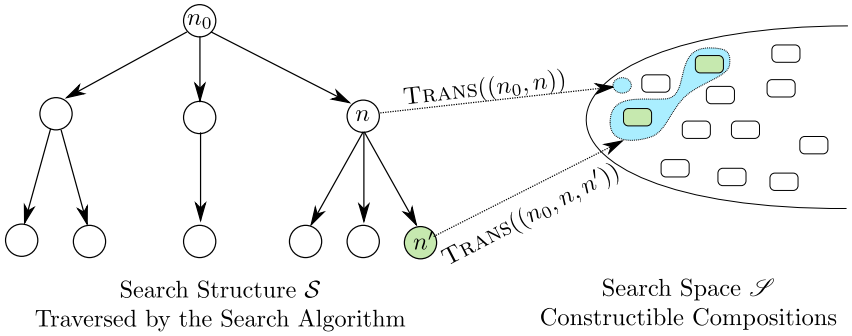


Figure 3.3: Nodes of the search structure may be associated with no, one, or many elements from the search space.

3.1.2.2 Transformation Function

The transformation function is the bridge between the search graph and the search space. Figure 3.3 provides a sketch of this function. It assigns a set of search space elements to every *path* in the search graph. Formally,

$$\textsc{Trans} : P^0 \to 2^{\mathscr{S}}$$

such that $\textsc{Trans}(p) = \{y \mid y \in \mathscr{S}, y \text{ is encoded in } p\}$. Note that the search structure defines the transformation function for a specific search problem instance $\mathfrak{p}$ to which the search space $\mathscr{S}$ belongs. In contrast to the goal property, the transformation function depends on the path from n^0 to a node n and not only n itself. Of course, it make sense to establish a condition between the goal property of a node and the elements obtainable through Trans to that node; I discuss this condition in Section 3.1.3.1.

It is possible that a path of the search graph can be transformed into more than only one element of the search space. For example, the path of the search graph may correspond to partially ordered compositions, i.e. compositions where the order of operation invocations is not (yet) fixed. $\textsc{Trans}(p)$ then should consider *every* totally ordered completion of the partially ordered composition encoded in p.

On the other hand, it is also possible that a path of the search graph can *not* be translated into *any* element of the search space. For example, in the 8-queens problem, every path corresponds to a partial positioning of (at most) 8 queens. However, every element in the search space corresponds to a board configuration with exactly 8 queens located. In this case, only leaf-nodes of the search structure have a non-empty value for Trans.

We make the following requirements for Trans:

1. The computation of $\textsc{Trans}(p)$ must be efficient in the sense that its runtime is polynomially bound in the length of p. This requirement avoids that a significant part of the search problem is outsourced into the transformation function. This also implies that $\textsc{Trans}(p)$ is finite.

2. If $\textsc{Trans}(p)$ offers several search space elements, these must be equally good for the client. In particular, if one of the elements is a solution, then every other element in the set must also be a solution. In other words, the client must be *indifferent* among

the objects in the set. This implies that the search structure is responsible to make all relevant decisions. It cannot happen that TRANS offers a set of candidates from which a (best) solution must be chosen.

The transformation function TRANS should not be confused with Pearl's subset view [94, p.17]. Pearl assumes that each path represents a set of solution candidates to which it can be complemented. However, TRANS has not the purpose to reflect the set of solution candidates to which a path can still be completed but the solution candidates encoded by *itself*.

3.1.2.3 Goal Function

The goal function maps every node of the search graph to a Boolean value. Formally, we have

$$\star : N \to \{false, true\}$$

We say that a node n is a *goal node* if and only if $\star(n) = 1$. Note that the goal property does not depend on the way how a node is reached.

3.1.3 Correctness and Completeness of a Search Structure

Being the basis of the search algorithm, the search structure itself may exhibit the classical formal properties of correctness and completeness with respect to a given problem. For the following, let $\mathscr{S}^* \subseteq \mathscr{S}$ denote the set of solutions for the search problem instance.

3.1.3.1 Correctness

A search structure is *correct* with respect to a search problem instance, iff every path to a goal node can be transformed into a non-empty set of search space elements each of which is a solution to the problem instance. Formally, we can write this as a constraint on $\star$:

$$\text{if } \star(n) = 1, \text{ then } \forall p = (n^0, .., n) \in P : \textsc{Trans}(p) \neq \emptyset \text{ and } \textsc{Trans}(p) \subseteq \mathscr{S}^*$$

The case that only some of the elements obtained from a node are solutions would conflict with the requirement of equally good elements. Obviously, solution elements are better than non-solution elements, so TRANS would not satisfy the required property of the client being indifferent among the candidates in the set if it returned such a mixture.

3.1.3.2 Completeness

Completeness of Search Structure

Completeness of a search structure means that it allows to reach all solutions of the search space. This requires to *cover* all solutions and to *recognize* the nodes leading to them as goal nodes. The requirements on completeness depend the required number of solutions. Formally, a search structure is *complete* iff for each problem instance it holds that

1. if the task is to find *one* solution and if $\mathscr{S}^* \neq \emptyset$, then there must be a path p such that $\textsc{Trans}(p) \cap \mathscr{S}^* \neq \emptyset$,
2. if the task is to find *all* solutions that satisfy the property ξ, then for every class C of equivalent solutions in $\mathscr{S}^*$, there must be a path p such that $\textsc{Trans}(p) \cap C \neq \emptyset$.

3. at least one of the nodes that are reached by a path $p = (n^0, .., n)$ where $\textsc{Trans}(p)$ contains solutions must be recognized by $\star(n) = true$.

Note that (2) implies (1). The difference is only relevant in the case that the search graph is built differently depending on the answer type. That is, the search graph can be built differently depending on whether one or all ξ-satisfying solutions are required. This can be useful to create smaller search graphs if only one solution is required. However, the search graphs in this thesis do not depend on the answer type, so only property (2) must be proven.

3.2 The Search Algorithm

This section describes the search algorithm used for composition in five parts. First, Section 3.2.1 shows the algorithm body and briefly explains its behavior. The algorithm contains two subroutines, namely *choose* and *prune*, which are described in detail in Section 3.2.2 and Section 3.2.3 respectively. In Section 3.2.4, I prove the correctness and completeness of the algorithm. Finally, since the algorithm significantly deviates from classical search algorithms, Section 3.2.5 compares the algorithm with the well-known A* algorithm [51].

3.2.1 Algorithm Overview

Given a search structure, the search algorithm tries to find a path between the root node and, depending on the query, one or all nodes that encode a solution that satisfies the condition ξ.

Algorithm 1: $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}}$

Input : Search Problem Instance $\mathfrak{p}$
Output: A solution (stream), given that a solution exists

```
1  OPEN ← {(getRoot_S(p)())}
2  CLOSED ← ∅
3  PRUNED ← ∅
4  while OPEN ≠ ∅ do
5      p ← choose path from OPEN and move n to CLOSED based on E
       // assume that p = (n^0, .., n)
6      foreach n' ∈ (getSuccessors_S(p)(n) \ PRUNED) do
7          p' ← (n^0, .., n, n')
8          pruning update of OPEN based on OPEN, CLOSED, p', P, and p
           // The previous step may have updated OPEN and PRUNED
9          if p' ∈ OPEN and if ⋆_S(p)(n') then
10             solution ← select any item of Trans_S(p)(p')
11             if p requires one solution then return solution
12
13             else if ξ(solution) then stream solution
14         end
15     end
16 end
```

The general structure of the algorithm is a form of best-first search. The algorithm maintains a list *OPEN* of unexplored paths, which is initialized with the path of length 0 containing

only the root node of the search structure. In each iteration of the main loop, one path p is selected for expansion, which means that the successor nodes of its last node n are computed, and the resulting paths are possibly inserted into the *OPEN* list themselves. For each potential successor n' of n, the algorithm checks whether the resulting new path p' should be inserted or if even the node n' can be pruned; that is, we distinguish between pruning paths and nodes, which I discuss below in more detail. If the new path p' is not pruned, its head node n' is first checked to be a solution, and, finally, p' is added into *OPEN*. *CLOSED* and *PRUNED* are sets that maintain *nodes* already expanded or pruned respectively.

SEARCH$_{\mathcal{S},\mathcal{E},\mathcal{P}}$ is a generic algorithm with three parameters:

1. *Search Structure* $\mathcal{S}$. The search structure defines the graph that is traversed, the goal function, the transformation function, and functions relevant for pruning nodes during the exploration procedure. Two concrete structures for sequential software composition are discussed in Chapter 4 and Chapter 5.
2. *Exploration Strategy* $\mathcal{E}$. The exploration strategy defines the criterion under which the search structure is explored, i.e. how the next elements are chosen in line 5 of the algorithm. I discuss the fundamentals of the exploration strategy in Section 3.2.2 and potential strategies for software composition in Section 3.3.
3. *Pruning Strategy* $\mathcal{P}$. The pruning strategy decides how nodes of the search graph are pruned in line 8 of the algorithm. I discuss the fundamentals of the pruning strategy in Section 3.2.3.

Note that the three parameters are not independent from each other. As Figure 3.4 shows, both the exploration strategy and the pruning strategy depend on the search structure, which is because they need to evaluate the rest problems encoded in the nodes. Also, the exploration strategy $\mathcal{E}$ may influence the way how pruning is applied by $\mathcal{P}$ in line 8; I explain this relation in detail in Section 3.2.3.3. The behavior of all three elements depends on the concrete search problem instance that is solved.

There are two modifications in comparison with Nilsson's GRAPHSEARCH procedure [93, p.64] . First, *OPEN* maintains *paths* instead of nodes. This is due to the assumption that the search space elements are associated to paths and not to nodes, and we want to be able to distinguish the search space elements that lead to the same node in the search graph. This makes the occurrence check in *OPEN* (and *CLOSED*) for new paths superfluous, because every path is considered at most once. Second, the above algorithm incorporates a pruning mechanic for nodes, which is not present in Nilsson's GRAPHSEARCH and no other search algorithm up to recent past [48, 116]. That is, we maintain a set of nodes that is explicitly excluded from further consideration, for which several reasons are possible (cf. Section 3.2.3).

Composition Algorithm

The above algorithm is a general search algorithm, but when using a particular search structure $\mathcal{S}$, we can speak of a *composition algorithm*. That is, in the above form, nodes have no semantics, but when instantiating the algorithm with a concrete search structure that reflects a software composition problem, the (instantiated) algorithm is a composition algorithm. Hence, in the following, when talking about *the* composition algorithm, I refer to SEARCH$_{\mathcal{S},\mathcal{E},\mathcal{P}}$ where $\mathcal{S}$ reflects the creation of compositions.

3.2.2 Path Selection Mechanism

Of course, the search process does not select the next path at random but based on some selection criterion. That is, the path selection function in line 5 of the above algorithm is

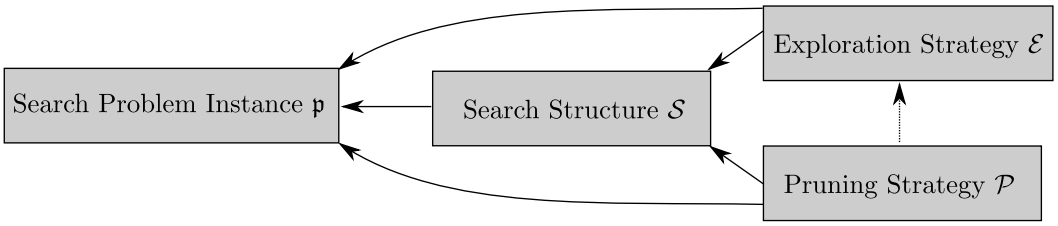


Figure 3.4: Relations among the three driving concepts of the search algorithm.

driven by some deterministic mechanic.

3.2.2.1 The Exploration Strategy

The basis of the decision process is a path evaluation function. For every path in the search structure, the path evaluation function computes an evaluation value considering the observations made so far during the search process and an estimate about the remaining work to be done. The evaluation value stems from an *evaluation space* $\mathbb{E} \subseteq \mathbb{R}^k$ for some $k \in \mathbb{N}$.

The necessity to consider higher-dimensional evaluation spaces arises from the fact that we may want to use non-functional properties as evaluation criteria. There is usually not just one but a couple of non-functional properties, which are encoded in form of vectors and cannot be reduced to a scalar without loss of information.

The evaluation of a path depends on information contained in itself as well as an estimate about the rest-path from its current head to a "best" solution. Here, the term "best" refers to the value of the path evaluation function, i.e. a reachable goal node minimizing this value.

History The first part, sometimes called *history*, is captured in a function $g : P_{\mathcal{S}} \to \mathbb{E}$ where $P_{\mathcal{S}}$ is the set of *paths* in the search graph. The estimate about the *rest solution* is captured in

Heuristic $h : P_{\mathcal{S}} \to \mathbb{E}$, which is sometimes called *heuristic*.

Aggregation Function In order to evaluate a path, the history and the heuristic are combined using an *aggregation function*. This aggregation function, which we denote as $\circ : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$, is often realized through addition, but may also adopt other functions such as maximum, etc. It can also be used to assign weights to the history and heuristic.

Tie-Breaker Since, two paths may be incomparable through $<$ with respect to their evaluation, a tie-breaker function is needed. Formally, this is a binary relation $<_{tb} : P_{\mathcal{S}} \times P_{\mathcal{S}} \to \{false, true\}$ that defines a *total order* on the paths. In general, there can be several (partially-ordering) tie-breakers chained together in the sense that, if two elements have the same value with respective to the first tie-breaker, they are compared with respect to the second and so on; only the final sub-tie-breaker must enforce a total ordering. At this point, we just assume that $<_{tb}$ encapsulates these sub-tie-breakers.

The history, the heuristic, the aggregation, and the tie-breaker are not built into the algorithm but constitute an *exploration strategy* with which the algorithm can be parametrized. Formally, an exploration strategy is

Exploration Strategy

$$\mathcal{E} = (g, h, \circ, <_{tb})$$

3.2.2.2 The Selection Mechanism in Dependence of the Exploration Strategy

In the following, we merge the functions of the exploration strategy into one single function. That is, given an exploration strategy $\mathcal{E} = (g, h, \circ, <_{tb})$, the search algorithm uses the function

$$f(p) = g(p) \circ h(p)$$

as a *primary* path evaluation function.

Among all the paths in *OPEN*, the algorithm selects one of the paths with minimal evaluation function value for expansion. That is, it selects one path from the following set:

$$PF = argmin_{p \in OPEN} f(p) = \{p \in OPEN \mid \neg \exists p' \in OPEN : f(p') < f(p)\}$$

Since this set contains all paths that are equally good with respect to f, it is often called *Pareto frontier*. *Pareto Frontier*

This definition amounts to the set of paths that are Pareto minimal with respect to the path evaluation function. Among a set of vectors V, a particular vector v is said to be Pareto minimal if and only if there is no other vector $v' \in V$ such that $v' < v$. Note that, in contrast to the scalar case, we cannot simply say that for every $v' \in V$ it holds that $v' \geq v$, because $v'_i < v_i$ may be true for some i. However, there is also at least one i' such that $v'_{i'} > v_{i'}$.

The set determined in this way may contain many elements. If the dimension of the evaluation space is 1, the problem vanishes somewhat in that one may consider all solutions with the same evaluation value equally good (and simply select one of them at random). But in the case of higher-dimensional evaluation spaces, it is much less clear which of the candidates to actually pick. This problem gets even worse by the fact that the size of the set of non-dominated candidates tends to heavily increase with the dimensions of the evaluation space.

Here, the tie-breaker comes into play. Out of the above set, we now choose the path with the best tie-breaker value. This refines line line 5 of the algorithm as follows:

$$\text{select the } n \text{ from PF that is minimal with respect to } <_{tb}$$

Since $<_{tb}$ defines a total ordering, there is exactly one such n.

3.2.2.3 Intention vs. Implementation of Exploration Strategies

I have presented exploration strategies as tuples of functions, but they should be actually seen as a more general concept. On a more conceptual level, we can understand an exploration strategy as an *intention* with several possible *concepts of realizations* each of which can be translated into an *implementation* in form of functions g, h, $\circ$, and $<_{tb}$. For example, one exploration strategy reflects the intention to find a solution as fast as possible while another has the intention to identify a solution with good non-functional properties. Figure 3.5 shows these three aspects of an exploration strategy.

Usually, an intention can be realized in different ways and each of these realization concepts needs a particular implementation depending on the search structure. For example, if the intention is to find *cheap* solutions, the concept of realization could be to consider the price itself or simply the *size* of solution in the expectation that the size and the price are correlated. Depending on the search structure, each of these concepts requires an individual implementation of the history, the heuristic, and the aggregation.

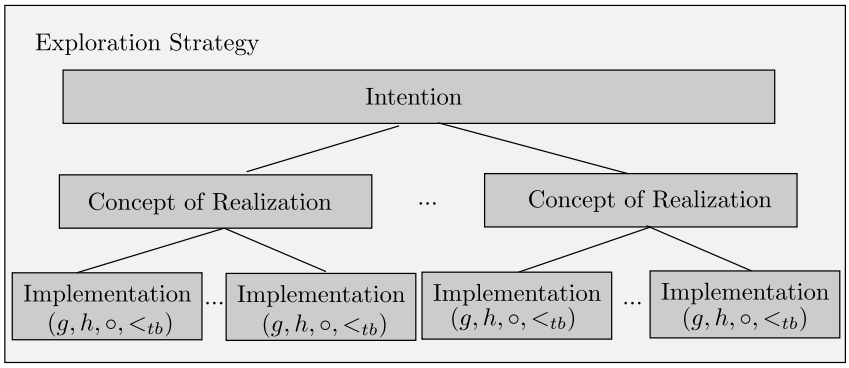


Figure 3.5: An exploration strategy has an intention that can be realized by different concepts. Their implementation depends on the search structure.

Since the intention and concept of realization of an exploration strategy are independent from the search structure, I discuss these once centrally, and only the concrete implementations for each search structure are discussed in the respective chapters. That is, in Section 3.3, I discuss the exploration strategy intentions and concepts of realizations, and Section 4.2.4 and Section 5.2.4 describe the respective implementations for the presented search structures.

3.2.3 Pruning Update Mechanism

The dedicated pruning mechanism is a major difference to classical best-first search algorithms. I first give an overview of what is necessary for pruning in general (Section 3.2.3.1), then discuss the criteria applied in this thesis (Section 3.2.3.2), and finally describe how the criteria are used in order to meet concrete pruning decisions (Section 3.2.3.3).

3.2.3.1 Overview

The high level view of the pruning step in line 8 is as follows. When considering a new path p' to the node n', four pruning decisions are imaginable.

1. We may prune the *node* n', which means to explicitly add it to the black list *PRUNED* and to not insert the new path p' into *OPEN*. For example, this may happen if a node is strictly "worse" than another one. Note that n' is not already in *PRUNED*, because we only consider the successors of n that are not in *PRUNED* yet (cf. line 6).

2. We may prune the *path* p' but not the node n', which means to simply not insert it into *OPEN*. For example, we may want to do this if we already generated the node n' before and are not interested in alternative paths to it. This holds in particular if we search only for one solution.

3. We may prune *nodes* that already *exist*, i.e. determine a set N_{prune} of nodes N_{gen} already generated that should be pruned due to the existence of n'. In that case, we would explicitly mark all nodes of N_{prune} as pruned by inserting them into *PRUNED*, remove all paths from *OPEN* that contain a node of N_{prune}, and insert p' on *OPEN*.

4. We do not prune anything and simply add p' to *OPEN*.

So the pruning step updates the two sets *PRUNED* and *OPEN*.

Pruning Criteria

The basis for pruning decisions are *pruning criteria*. For example, one pruning criterion is that a the partial solution has costs already higher than allowed in the query, or that every solution reachable from one node is "worse" than every solution reachable from another, etc. Pruning criteria can be based on the query, already existing nodes and paths, and general domain knowledge. In this thesis, I use three criteria, which I discuss below in Section 3.2.3.2.

Given the new node n' and new path p', for each of the pruning criteria, we get *prunability assertions*. Let N_{gen} be the set of nodes that have already been generated, i.e. contained in *CLOSED* or being the head of a path in *OPEN*. Then, for each criterion, we may ask

(i) whether n' is prunable (due to nodes in N_{gen} or other reasons),

(ii) whether p' is prunable (due to nodes in N_{gen} or other reasons), and

(iii) for a subset N_{prune} of nodes in N_{gen} prunable based on n'.

Whenever one of the questions is answered with yes or a non-empty set of nodes respectively, we have a justification for pruning but still need to make a concrete pruning decision.

In general, we cannot simply prune whenever a pruning inquiry is answered positively since this may cause inconsistencies. First, different criteria may come to contradictory answers. For example, one criterion says that n' is prunable due to a node n while another says that n can be pruned due to n'. Now if we prune both, then the pruning justification for both is gone. Second, even a single criterion may be inconsistent in the sense that it says that n' is prunable due to a node n and that an existing node n'' is prunable due to n', i.e. the questions (i)-(iii) could *all* be answered positively for the same criterion. Again, we cannot simply prune both n' and n'', because pruning n' would take away the justification for pruning n''. Hence, we need a mechanic that guarantees consistency in this respect.

To achieve a pruning decision, the pruning strategy consists of a set of pruning criteria and a pruning decision algorithm. Formally, we can write

$$\mathcal{P} = ((pc_1, .., pc_m), \textsc{Prune})$$

where Prune is the pruning decision function. The behavior of the function must be as follows. Invoked with $\textsc{Prune}(\mathit{OPEN}, \mathit{CLOSED}, p', \mathfrak{p}, (pc_1, .., pc_m))$, it *updates* the sets *OPEN* and *PRUNED* and returns *true* if p' was inserted into *OPEN* and *false* otherwise. Prune must make sure that completeness is preserved, i.e. that all relevant solutions can still be found. In particular, it must avoid the above conflicts that we prune two nodes whose pruning decision depends on each other.

Before describing the pruning function I adopted here, I describe the used pruning criteria.

3.2.3.2 The Pruning Criteria in Detail

In this thesis, I use three criteria:

1. the rest problem of a node is recognized to be unsolvable
2. we already found solutions that are "better" than anything achievable from a new node
3. a node n induces a more difficult rest problem than the one of another node n'

Of course, not every technique is always applicable; I now discuss each of them in more detail.

Pruning on Unsolvable Rest Problem Given that a concept such as costs is defined and that it increases monotonically, a partial solution that violates the cost cannot be completed to a solution anymore. In our setting, such a cost bound is given with the non-functional properties; if a candidate violates one of the properties, it cannot be completed to a solution anymore and, hence, be ignored. Of course, this requires that satisfying the property bounds actually *is* a necessary condition for a search space element to be a solution.

So, this pruning technique is applicable if and only if three conditions hold. First, there must be a cost bound $cost(q)$ specified in the search problem instance, and the cost of each partial solution must be encoded in the rest problem of a node; let $cost(n)$ be this encoding for node n. Second, $\star(n) = true$ must imply $cost(n) \leq cost(q)$ for every n. Third, costs must increase monotonically, i.e. $cost(n) \leq cost(n')$ if n' is a child of n.

However, there can also be other reasons for which there is no path from a node to a goal node. For example, we may create a composition with an operation that has a precondition that is neither contained in the preconditions of the query nor in any clause or in the effects of any other operation. It is then impossible to complete this candidate to a solution.

In order to detect this kind of dead-end, an additional routine is required in order to evaluate this criterion. The search algorithm cannot infer this property simply from the node encoding as in the case of property bounds. That is, we need a dead-end function $\perp : N_{\mathcal{S}} \rightarrow \{false, true\}$ that decides whether or not the node can be pruned due to insolvability. A typical way to identify this type of dead-ends is to solve relaxed versions of the rest problem. In heuristic search, this is often done for the computation of heuristic values of a node, and, for values of ∞, this is equivalent to pruning. I also apply this strategy (cf. Section 3.3).

Pruning on a Cost-Dominating Solution In a particular case, we can prune a node based on solutions already found. That is, we prune a partial solution because it is already more "expensive" than a solution that we have found earlier in terms of some cost measure. Naturally, any solution derived from this partial solution must be worse than the solution already found.

This pruning technique is applicable under three conditions. Again, we need the concept of cost where $\{cost(n_1^*), .., cost(n_k^*)\}$ is the set of (Pareto) optimal cost values of solutions found so far, and $cost(n)$ is the cost encoded in the current node. These are here naturally given through the non-functional properties. Second, costs must increase monotonically, i.e. $cost(n) \leq cost(n')$ if n' is a child of n. Third, the problem instance must ask for *all* solutions not dominated w.r.t. to the cost measure. That is, the ξ of the solution condition (cf. Section 3.1.1) says that solutions dominated with respect to this cost measure do not need to be returned. For example, in the composition setup, this is given for the non-functional properties if at_q = *all-nondominated*.

Pruning on Solution Reachability Dominance (SR-Dominance) Probably the most important pruning is the one based on the comparison of nodes with respect to the reachability of a goal node. That is, if we know that for every path from a node n to a goal node, there exists an at most as long path from n' to a goal node, then we can ignore n. For example, in planning, this has always been considered by pruning "subsumed" states, i.e. states in which one has strictly less knowledge than in another (in forward planning) or strictly more left to do than in another. However, in the absence of any semantics of the nodes, we need a general framework for this type of comparison.

SR-Dominance relation

The formal account for this solution reachability (SR) dominance relation is as follows:

$$\succeq : N \times N \rightarrow \{false, true\}$$

Intuitively, the relation $n \succeq n'$ means that n is preferable over n' in that n' is as least as hard to complete to a solution as n. This relation implies a *strict* comparison of the form:

$$\succ : N \times N \rightarrow \{false, true\} \quad \text{with} \quad n \succ n' \Leftrightarrow n \succeq n' \wedge \neg (n' \succeq n)$$

In other words, evaluating the basic node comparison in both directions, we know whether or not the comparison criterion holds strictly.

Completeness-Preservingness for $\succeq$

The SR-dominance relation must satisfy two conditions in order to be *completeness-preserving*. First, if $n \succeq n'$ holds and if a solution is reachable from n' over r edges, then a solution must be reachable from n over at most r edges. Second, the strict comparison $\succ$ must be well-founded[1] over the set of nodes N of the search graph. These conditions are necessary to ensure that we do not prune every solution before it can be found. Consider Figure 3.6 for two (fairly) special search graphs in which, if we do not have this condition, the algorithm chases infinite paths and never terminates with a solution even though one (in fact even infinitely many) exist. Note that a lower bound on the edge costs as used in A* does not help to resolve this problem, because, even though the paths become infinitely expensive, they are the only ones that are available.

The above pruning questions are then answered as follows. Let n' the node to be inserted and N_{open} be the head nodes of the paths in $OPEN$. The answer on question (i) and (ii) is "yes" iff $n \succ n'$ holds for at least one $n \in N_{open}$. On question (ii), the criterion returns the set of all nodes $n \in N_{open}$ for which $n' \succ n$ holds. Note that the relation must hold strictly in both cases.

This pruning technique is applicable whenever the search problem instances requires that one solution is returned. That is, in the composition problem instance, this is the case iff $at_q = one$. Otherwise we would possibly cut away relevant solutions that need to be returned.

Recently, very similar concepts to the SR-dominance relation have been proposed [48,116]. The basic idea behind these relations is the same, but there are two differences. The first is that, in this thesis, requirements on the comparison relation are related to path *length* instead of path *cost*. Path costs are considered there because they consider an optimization problem in which it is forbidden to prune solutions that are optimal with respect to the path costs. However, since I address a constraint satisfaction problem, I allow pruning of such solutions in this thesis. Second, the pruning mechanism itself works slightly different. In both of the above papers, pruning is done as soon as $n \succeq n'$ *and* $g(n') \geq g(n)$ where g is the history function while I *only* require that the relation holds *strictly*, i.e. $n \succ n'$ must hold. I need the strictness in order to preserve completeness (as shown below); they guarantee completeness by the restriction on g by which the cheapest solution in terms of f can never be pruned and, hence, is always found within a finite time horizon. Of course, this yields different pruning behaviors, and it would be interesting to compare the performances of the different strategies, but this not at the core of this thesis.

3.2.3.3 Obtaining the Pruning Decision

In this thesis, I use the following procedure to induce a pruning decision: First, ask criteria (1)-(3) questions (i) and (ii). If any of them answers "yes", do not insert the new path or

[1] A relation R is well-founded on a set S if every subset $S' \subseteq S$ contains at least one element m such that for no $s \in S'$ the relation sRm holds, i.e. m is *minimal*.

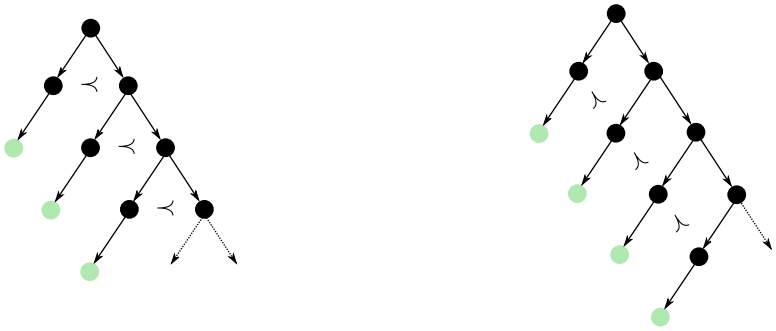


Figure 3.6: Pruning by node comparison may imply that the algorithm chases an infinite solution path. Chasing infinite paths can be avoided on the left by requiring that $n \succeq n'$ and the reachability of a solution from n' in r steps implies that a solution is reachable from n in at most r steps. On the right, it can be avoided by requiring that the node comparison relation is well-founded. Both assumptions together yield completeness of the search algorithm.

even prune the corresponding node respectively. Otherwise, ask criterion (3) question (iii); criteria (1) and (2) will only classify the new node or currently considered path as prunable, i.e. always answer $\emptyset$ on question (iii), so we do not need to ask this question for them. Then prune each of the returned nodes N_{prune} that is not on *CLOSED*, i.e. I do *not* prune nodes that have already been expanded even if they are strictly dominated in the spirit of $\succeq$. Here, we also remove all paths from *OPEN* whose head node is in N_{prune}. One can easily see that, given this strategy, criterion (3) will never answer "yes" to question (i) and a non-empty set to question (iii) at the same time as long as $\succeq_{\mathcal{S}}$ is transitive, which is the case for the relations I use in Chapter 4 and Chapter 5. Altogether, we obtain a consistent pruning.

Note that pruning criterion (3) reveals a potential conflict of interests. For example, for two nodes n and n', we may have $f(n) < f(n')$ and $n' \succ n$ at the same time. Then we know that we can prune n, but at the same time has a better value with respect to the evaluation strategy than n'. It is not clear whether it is better to prune or to keep n. The above procedure, which is also used in the implementation used for the evaluation, completely ignores the node evaluation function for pruning. However, in general this could be a parametrization of the search algorithm that can be set by the client.

This conflict is also present in the related papers in spite of considering the exploration strategy. As already said, pruning is never exclusively done based on $\succeq$ in [48] and [116] but also considers the exploration strategy. Also, they require that $n \succeq n'$ implies that $h^*(n) \leq h^*(n')$ where h^* is the cheapest path from a node to a goal node. While the latter assumption makes sure that no optimal solution is pruned, it can still happen that $h(n) \gg h(n')$, which would imply $f(n) > f(n')$ in spite of $n \succeq n'$ and, hence, result in the same conflict.

Pruning Strategy $\mathcal{P}_{\succeq}$

Since the above criteria and decision mechanism are the only one I consider in this thesis, I refer to it with $\mathcal{P}_{\succeq}$. Given a node encoding containing the non-functional properties, the pruning behavior for criteria (1) and (2) is obvious without further specification. Hence, the only aspect that needs to be specified in a concrete context is the SR-dominance relation $\succeq$.

3.2.4 Correctness and Completeness

I now show the correctness and the completeness of $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$. The proofs rely on the assumption that the underlying search structure $\mathcal{S}$ is correct and complete and that the exploration strategy $\mathcal{E}$ provides a strictly increasing history evaluation. Note that some proofs are only sketched for readability; the detailed versions of those proofs are found in the appendix.

3.2.4.1 Correctness

An algorithm is correct if a returned solution candidate actually *is* a solution to the posed problem instance. For the composition problem, we must show that the returned composition(s) are solutions to the query q.

Since the composition algorithm does not know the semantics of the nodes, it must rely on the correctness of the search structure. Hence, correctness can only be asserted assuming that the search structure is correct.

Theorem 3.1. *Let $\mathcal{S}$ be a correct search structure. Then $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ works correct for any exploration strategy $\mathcal{E}$.*

Proof. If a composition is returned, which happens in line 11, then the test in line 9 has been passed. If line 9 is passed for a node n', we know that $\star_{\mathcal{S}}(n') = \mathit{true}$. Since $\mathcal{S}$ is correct, we know that $\star_{\mathcal{S}}(n') = \mathit{true}$ implies that every element of $\textsc{Trans}_{\mathcal{S}}(n')$ is a solution. So the item returned in line 11 is a solution. □

3.2.4.2 Completeness

An algorithm is complete if it terminates with a solution given that one exists. In our case, there is a potentially infinite set of solutions that may be required (cf. Section 3.1.1), so we would require that each of them is outputted (in a stream) after a finite number of steps.

The completeness of $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ depends on the search structure $\mathcal{S}$, the exploration strategy, and the SR-Dominance relation $\succeq$. Intuitively, three conditions must be true:

1. for every solution, there must be a path in the graph that can be converted into it,
2. pruning must not make the set of solutions reachable within a finite horizon empty, and
3. if the algorithm does not halt before, every node that does not become pruned or unreachable through pruning is finally discovered.

The first two conditions are captured in the definition of a complete search structure and, using $\mathcal{P}_\succeq$ as a pruning strategy, in the requirement that the SR-Dominance relation is completeness-preserving.

Strictly Increasing Exploration Strategy

For the third condition, we need a *strictly increasing* exploration strategy like in A*. An exploration strategy is strictly increasing if there is some $\varepsilon \in \mathbb{E}$ with $\varepsilon > 0$ such that, for any path $p = (.., n)$, extending p by one edge increases the history by at least ε, and if the total evaluation of a node is at least its history. Formally, an exploration strategy is strictly increasing if the following two properties hold for every such paths p and p':

$$g(p') \geq g(p) + \varepsilon \quad \text{and} \quad f(p) \geq g(p)$$

These properties guarantee that exploring an infinite path will eventually yield a higher value for g (and for f) than the g-value of any other path in *OPEN*; the consequence is that, unless the algorithm has terminated successfully before (or the node has become pruned or unreachable), every node on is discovered (and expanded) after a finite number of steps. This is the commonly assumed lower bound on edge weights in A* [51] and algorithms of the like.

Note that a strictly increasing exploration strategy does not require that the f-values of a path are strictly increasing. That is, it is possible that $f((..,n)) \leq f((..,n,n'))$, but for the *history* it must hold that $g((..,n)) + \varepsilon \leq g((..,n,n'))$.

In the remainder of this section, I assume a strictly increasing exploration strategy. We can then make the following observation.

Observation 3.2. *Let $\mathcal{S}$ be a search structure and $\mathcal{E} = (g, h, \circ, <_{tb})$ a strictly increasing exploration strategy for $\mathcal{S}$ with evaluation space $\mathbb{E}$. Let $f(p) = g(p) \circ h(p)$. Then for every $M \in \mathbb{E}$, the set ${P_\mathcal{S}}^M = \{p \mid p \in P_\mathcal{S}, f(p) \leq M\}$ of M-bounded paths is finite.*

Proof. First, the strictly increasing exploration strategy implies that $f(p) \geq g(p) \geq \varepsilon \cdot |p|$ for every path. Then, a path p with $f(p) \leq M$ can have at most $\lfloor \frac{M}{\varepsilon} \rfloor$ edges, hence ${P_\mathcal{S}}^M \subseteq \{p \mid p \in P_\mathcal{S}, |p| \leq \lfloor \frac{M}{\varepsilon} \rfloor\} = \bar{P}_\mathcal{S}^M$. Since every node has only a finite number of successors, $\bar{P}_\mathcal{S}^M$ must be finite, which implies the finiteness of $P_\mathcal{S}^M$. □

I now show the completeness of $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ given a complete search structure $\mathcal{S}$ and a strictly increasing exploration strategy $\mathcal{E}$. Depending on the type of the query posed in the search problem instance $\mathfrak{p}$, I first consider the case that only one solution must be returned and then the case that every solution with a property ξ must be returned.

Completeness When One Solution is Required Let us assume that $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ receives an input that requires one solution (if one exists). Note that pruning based on $\succ$ is activated in this setting, which can repeatedly make currently considered solution paths invalid, such that one cannot focus on one particular path that eventually must be explored.

The road-map of the proof of completeness (for the case that one solution must be found) is as follows. The main argument of the proof is that, within a finite period of time, there must be subpaths of solution paths put on *OPEN* whose remaining length to a goal node strictly decreases. This fact is proven in Lemma 3.5. This requires that there is always a subpath of a path to a goal node on *OPEN* (Lemma 3.3) and that the algorithm does not terminate with "no solution" (Lemma 3.4). The proof of completeness for this case, i.e. that one solution is returned if one exists, is then given in Lemma 3.6.

Lemma 3.3. *Let $\mathcal{S}$ be a search structure with a goal node, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then, at each point of time before* $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *returns a solution, there is a path $(n^0_\mathcal{S}, .., n, .., n^*) \in P_\mathcal{S}$ such that $\star_\mathcal{S}(n^*) = true$ and $(n^0_\mathcal{S}, .., n) \in$ OPEN.*

Proof Sketch. The proof is by induction over the number of iterations. In the first iteration, the claim is obviously true, because every path to a solution start with n^0. In other iterations, a concrete subpath of a solution can only be removed from *OPEN* if it is expanded or pruned. In the first case, the successor path is on *OPEN*; in the other case, the path that caused the pruning must be on *OPEN*. □

Lemma 3.4. *Let $\mathcal{S}$ be a search structure with a goal node, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then* $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *will not terminate with "fail".*

Proof. Suppose that $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ terminates with "fail". By the previous Lemma, we know that *OPEN* has not been empty. But this is a contradiction to the condition for terminating the main loop, which is the only possibility for terminating with "fail". □

Lemma 3.5. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$, and let $p = (n^0_{\mathcal{S}}, .., n) \in$ OPEN such that there is a path of length r from n to a goal node. Then, after a finite number of steps, $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ does one of the following:*

1. *it returns a solution,*
2. *it puts a path $p' = (n^0_{\mathcal{S}}, .., n')$ on OPEN such that there is a path from n' to a goal node that is shorter than r, or*
3. *it puts a path $p' = (n^0_{\mathcal{S}}, .., n')$ on OPEN such that $n' \succ n$.*

Proof Sketch. One can show that condition (3) is enforced after a finite number of iterations if neither (1) nor (2) occurred. The only reason why condition (1) or (2) do not occur for a particular path p after a finite number of steps is that p is pruned. However, the only reason for p being pruned once it has been on *OPEN* is that its head node is dominated by another node through $\succeq$; the other pruning techniques would have implied that p would not have been put on *OPEN*. □

Lemma 3.6. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ returns a solution after a finite number of steps if one exists.*

Proof Sketch. This follows directly from the previous Lemma. Within a single transitive chain of paths, condition (3) can only occur for a finite number of iterations due to the well-foundedness of $\succeq$. Also, condition (2) can only become true a finite number of times, because the rest length eventually will be 0. In that case, a solution will have been found and is returned. □

Completeness When All Solutions are Required I now show that $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ is complete if the task is to find all solutions that satisfy a property ξ. In this case, pruning based on $\succ$ is deactivated, so the proof is similar to the proof of completeness of A*.

Lemma 3.7. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ outputs every solution for which ξ holds after a finite number of steps.*

Proof Sketch. The proof here is similar to the one of completeness of A*. By completeness of the search structure, for every solution s^*, there is an equivalent solution $\widehat{s}^*$ such that a path $p = (n^0, .., n)$ exists with $\widehat{s}^* \in \text{TRANS}(p)$ and $\star(n) = \mathit{true}$. Since the exploration strategy is strictly increasing, the number of paths with f-values at most $f(p)$ is finite. Since no node of p and none of its subpaths is ever pruned (except that a node that is generated twice), the parent node of the head of p is finally expanded and the solutions in $\text{TRANS}(p)$ are outputted. □

Completeness of the Composition Algorithm The above results now enable us to conclude that the algorithm is complete on complete search structures when run with a strictly increasing exploration strategy.

Theorem 3.8. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, and $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then the search algorithm* $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *is complete.*

Proof. Two answer types are possible for $\mathfrak{p}$. If one solution is required, this follows directly from Lemma 3.6. If all solutions satisfying ξ are required, it follows from Lemma 3.7. □

3.2.5 Differences to A*

A* is certainly the best-known search algorithm, so I use it as a point of reference also for other instances of best-first search [94]. Even though the overall structure of the above algorithm is similar to A*, there are significant differences with respect to both the basic algorithm layout and the path evaluation function.

3.2.5.1 Differences in the Algorithm

The differences in the main algorithm are that $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ considers all paths to a node instead of only one, that it may be configured to search for multiple solutions, that it uses f to *guide the finding of* but not to optimize solutions, and that I integrate an explicit pruning mechanic that exceeds the basic equality check of nodes.

1. *OPEN* consists of *paths* instead of *nodes*; in particular, I do not use back pointers. Classical best-first search considers only sub*trees* of the search graph called *traversal trees*, which maintain exactly one path to each node found so far. This path is remembered using a *back pointer* for each node, which points to the node from which it was reached [94, p.34]. As a consequence, we consider only one path for each node, and *OPEN* can be simplified to a set of nodes. However, since we consider all paths to (and over) a node, we cannot generally discard any of them. In particular, nodes do not necessarily have a single other node from which they are reachable, so we really maintain explicit paths instead of storing back pointers with the nodes.

2. The above algorithm has a built-in option to identify *all* relevant solutions. If the search problem instance requires that all solutions (satisfying condition ξ) are outputted, the algorithm does not halt after the first solution but continues. Moreover, it only prunes nodes that do not need to be returned. The default implementation of A* only allows to search for one solution, because it halts after the first solution, and even if solutions were outputted in a stream, A* discards solutions if several solutions share the same node in the search graph.

   Obviously, with a slight modification, one can achieve the same effect in A*. First, one needs to change the return statement into a stream-output statement. Second, the equals-relation used to identify the identity of two nodes must be fixed to *false* for every pair of nodes, because otherwise parent-discarding would prune potential solutions.

3. I do not apply a *delayed termination*. When A* discovers a goal node, it does not return it until every other more promising candidate has been examined. A* considers this node

comparison by checking the solution property of a node and returning it when it would be expanded and not, as done in the above algorithm, when it is discovered. Delayed termination is only necessary if the algorithm should return the optimal solution with respect to the path evaluation function, which is not the case in the composition problem considered in this thesis. Hence, solutions are outputted in the moment where they are created.

4. I allow an explicit pruning function, which is not supported in A*. In A*, pruning only takes place in form of "parent discarding" when two nodes are equal. However, this is only a very special case of the situations in which pruning is possible. The support for a dedicated pruning mechanic is beyond the classical A* algorithm. A corresponding extension of A* was presented in [48, 116].

3.2.5.2 Differences in the Exploration Strategies

The second aspect is that the path evaluation used here significantly differs from A*. The differences are that, in this setting, the history function g does not need to be recursive (or decomposable), that we do not optimize for the path evaluation criterion, that edge costs may be vectors instead of mere scalars, that the cost aggregation function is not additive, and that the heuristic function h is not required to be admissible. In detail:

1. The history does not need to be decomposable. A* assumes that the history of a path can be disaggregated and that its parts can be assigned to the edges; i.e. $g(p)$ is a function of the weights of the edge in p. However, doing this is not always easy. Suppose that a path reflects a composition with guards, i.e. with two alternative branches. If the non-functional properties are used for path evaluation, then adding an operation to one of the two branches changes the properties of that branch, but the properties of the whole composition may remain unchanged. For example, if the total aggregation takes the maximum values over the different branches, and the operation was added to the "cheaper" branch, this branch may *remain* cheaper even with the new operation and the total evaluation would not change. This example shows that a disaggregation of path evaluation to edges is at least not trivial and maybe not always possible.

2. The evaluation function is not subject to optimization. The role of the path evaluation function f depends on the objective of the algorithm. If the goal was to find an *optimal* solution with respect to some criteria, then f should be used to capture that criteria; i.e. the optimization criterion itself is used as the path evaluation. This is what is done in A*. However, if the goal is to find *any* solution, there is no criterion for which we optimize. In such a setting, f can be used either to identify any solution as fast as possible or to semi-optimize some criterion. The latter means that, during the search, we prefer the expansion of paths that are better with respect to that criterion, but if we identify the first solution, we return it (even though better solutions may exist).

3. We have a vector-based path costs instead of scalar values. That is, the path evaluation function f, the current path costs g, and the heuristic value h are vectors instead of numbers. This strongly increases the set of "minimal" paths and gives raise for a tie-breaker function. This aspect was previously considered in Multiobjective A* [112].

4. The aggregation function $\circ$, which defines f, is not necessarily additive. For example, considering the non-functional properties for evaluation, the aggregation function $\circ$ may apply the *max* operator for some of the properties instead of the addition.

5. The heuristic does not need to be admissible (i.e. $h(n)$ may be greater than the actual path cost to the (next) solution). Admissibility is required e.g. in A* in order to guarantee that returned solutions are optimal. Since we do not consider an optimization problem, admissibility is not required.

3.3 Exploration Strategies in Software Composition

In the following, I discuss three potential intentions that may be relevant when searching for software compositions.

1. *Find a solution that is optimal wrt. non-functional properties.* Here, the algorithm prioritizes "cheapest" compositions by always expanding a path that is Pareto optimal with respect to the non-functional properties.
2. *Find any solution as fast as possible.* The algorithm gives priority to the most *promising* paths in terms of estimated remaining "distance" to a solution. The edge weight is 1, and the heuristic estimates the remaining edges to a solution.
3. *Find a solution that is most likely what the user wants.* Given the fact that the user cannot specify every aspect of his request formally, it may be a good idea to prioritize by the estimated user satisfaction that can be expected by completing a partial solution.

Once again, this section provides a description of the exploration strategy *intentions*. That is, I do not describe how the path evaluation functions are *implemented*. These implementations depend on the choice of the search structure, i.e. are explained in Chapter 4 and Chapter 5 for the concrete structures respectively. This section focuses on the explanation of the different *semantic* choices of exploration strategies.

3.3.1 e_{nf}: Finding a Good Solution w.r.t. Non-Functional Properties

This strategy prioritizes by the non-functional cost vectors and is presumably appropriate if we are interested in high-quality solutions. The algorithm always tries to complete a composition to a solution that has currently the most promising quality values. Since there are usually several Pareto optimal compositions at a time (the set of these is called Pareto frontier), a tie breaker is needed to select the actually explored path. Note that even applying this exploration strategy, the algorithm does not necessarily return an optimal solution, because we do not apply delayed termination.

3.3.1.1 Evaluation Space and Evaluation Function

Evaluation Space The evaluation space here corresponds to the space of non-functional properties. That is, we have $\mathbb{E} = \mathbb{D}_1 \times \ldots \times \mathbb{D}_k$ where $\mathbb{D}_i$ is the domain of the i-th of k non-functional properties (cf. Def. 1).

History In this strategy, we equalize the weight of a path with the non-functional properties of the composition belonging to it. This requires that all compositions $\textsc{Trans}(p)$ derivable from the path p *do* have the same non-functional properties, i.e. $c_1, c_2 \in \textsc{Trans}(p)$ implies $\odot(c_1) = \odot(c_2)$.

Heuristic The heuristic tries to estimate the non-functional properties of the *remaining* composition that must still be added in order to reach a goal node. That is, to compute the heuristic of some path $p = (n^0, .., n)$, the rest problem belonging to n (, which is induced by all compositions belonging to paths from n^0 to n,) is computed, and the non-functional properties of a composition that solves that rest problem are determined.

In this thesis, I compute this heuristic based on a relaxed variant of the problem that is significantly easier to solve than the original rest problem. More precisely, the rest problem is transformed into a (less expressive) set-theoretic planning problem. We can use (a slightly adapted version of) the aggregation $\odot$ to compute the non-functional properties of the relaxed solution and use it for h. Details are given in Chapter 4 and Chapter 5 respectively.

Aggregation Function The aggregation function for this exploration strategy corresponds to the aggregation function of non-functional properties. That is, for each element of the vector corresponding to a particular property, the respective aggregation is used. Formally, we write $(v^1 \circ v^2)_i = v^1_i \oplus_i v^2_i$.

Tie-Breaker As a tie-breaker, I consider the number of steps necessary to complete a path to the solution found in the relaxed problem when the heuristic was computed. That is, for paths with equal values for f, we prefer the one that is allegedly closer to goal node. To this end, we can use the length of the solution path in the relaxed problem used to compute the heuristic. There might be solutions reachable with a shorter distance in the search graph, but, intuitively, it would not be worth the effort to solve a second relaxed problem only to compute a tie-breaker value. If these are still equal, the time of insertion into *OPEN* is used to enforce a total order on candidates.

3.3.1.2 Discussion

The obvious motivation to apply this strategy is to find a solution that not only satisfies the given bounds for non-functional properties but that is significantly better than average within the set of admissible solutions.

However, this exploration strategy has some major drawbacks.

1. The solutions identified are not optimal unless the goal check is deferred to the expansion step, which significantly delays the time until a solution is returned at all.

2. In the above form, this strategy can only be used if the compositions belonging to a path have exactly one final state. Otherwise, the sequential aggregations $\oplus_i$ cannot be used. For example, if a composition has two final states, then it can only completed by two compositions, but then it is not clear how the aggregation should be done.

3. Applying this search strategy generally renders the composition algorithm incomplete. The problem is that edge weights in this scenario may or may not increase monotonically but cannot be assumed to increase *strictly* monotonically. This is because the application of rules does not increase the g-value of the parent composition, and there are composition problems where an infinite sequence of clause applications exists. Once a path with this property is expanded, there is always at least one child path with the same values for both g and h, so the algorithm would get stuck in an infinite branch. As a consequence, applying this strategy is not complete in general.

4. The strategy is of highly limited utility in the case of incomplete query semantics. If a solution actually is not a "real" solution in the eyes of the user, the non-functional properties are completely irrelevant. In particular, there is no value in having found a very good one.

5. The strategy in this form only works for the construction of sequential compositions. The problem is that the aggregation function $\odot$ aggregates the non-functional properties of compositions that are concatenated, but this cannot be easily done for non-sequential compositions where one compositions may be *injected* into another one (say in a loop).

A particular consequence is that the strategy should be applied only in the case of semantically complete queries (cf. Section 2.2.3). The drawbacks in terms of runtime and complexity can only be justified if a found solution actually *is* a solution for the user. Put differently, the user will hardly accept long runtimes only to obtain non-functionally optimal solutions that do not satisfy his intention.

3.3.2 e_{fast}: Finding an Arbitrary Solution as Fast as Possible

This strategy prioritizes by the supposed closeness to a solution node in terms of edge numbers. In order to avoid running into infinite branches, the heuristic is somewhat regularized by the path cost to the root.

3.3.2.1 Evaluation Function

Evaluation Space The evaluation space are the natural numbers; i.e. $\mathbb{E} = \mathbb{N}$.

History The history is the length of p; i.e. $g(p) = |p| - 1$.

Heuristic 1 The first heuristic I propose for this strategy is to determine the rest problem "size". Size refers for example to the number of literals that must still be eliminated or produced (depending on how one is searching). Assuming that each step can reduce the rest problem by one, this heuristic then *is* the rest problem size. This heuristic is not optimistic, because operation invocations may reduce the state by more than one predicate.

Heuristic 2 The second heuristic I propose for this strategy is the tie breaker strategy used in the previous strategy. That is, we create a relaxed rest problem in form of a set-theoretic planning problem and use the solution *path* length in the (relaxed) search graph for $h(n)$.

In contrast to the previous strategy, it is reasonable that the relaxed rest problem reflects the original search structure. That is, the general semantics of the underlying search structure should be the same. Otherwise, it would not serve as an approximation for shortest distance to a goal node. For example, if the original search structure realizes a backward search and the relaxed problem realizes a partial-ordered search, the heuristic value does not generally reflect the path length to a solution in the original search graph.

Aggregation Function The aggregation function for this strategy is simply a weighted addition. That is, we have $\circ = w_g \cdot g(n) + h(n)$. There is no obvious choice for w_g. In this thesis, I simply assume that $w_g = 1$; experiments could recommend other values.

Tie-Breaker The tie-breaker for this strategy may simply be the heuristic that was not chosen as the primary heuristic for the path evaluation function. If this heuristic yields identical values, a final decision can be achieved again by using the time of insertion into *OPEN*.

3.3.2.2 Discussion

First note that if this exploration strategy is used by the above algorithm on a complete search structure, then the algorithm is complete. To this end, first make the following observation.

Observation 3.9. *The strategy e_{fast} is strictly increasing.*

We can assert this on the abstract level, because we already have fixed g with a fixed lower bound. Now we can conclude the completeness of $\textsc{Search}_{\mathcal{S},e_{fast},\mathcal{P}}$ given that $\mathcal{S}$ is complete.

Corollary 3.10. *Let $\mathcal{S}$ be a complete search structure and $\succeq$ be completeness-preserving. Then $\textsc{Search}_{\mathcal{S},e_{fast},\mathcal{P}_{\succeq}}$ is complete.*

Proof. Follows directly from Theorem 3.8 and Observation 3.9. □

So the two strengths of this strategy are that it is the only strategy that is complete and that it (presumably) returns a solution fastest. Since every edge has a weight of 1, there exists a constant lower bound, such that the term $w_g \cdot g(n)$ in the evaluation function increases with the path length at least by this lower bound. This implies that every path is eventually expanded (or pruned). Since the exploration is driven by a heuristic that estimates the closeness to a solution, it can be expected to find solutions fastest.

The downside of this strategy is obviously that we may return highly suboptimal solutions. We cannot make any assertions about optimality of the properties of solutions identified. It is quite possible that, even if a found solution satisfies the constraints, there would be other solutions that are much better.

Summarizing, constant edge weights prioritize by the remaining distance to a solution and, intuitively, it should be appropriate when we want to find a first solution as fast as possible. The strategy is complete due to strictly increasing paths costs. We cannot say anything about the quality of solutions, but this does not matter in our constraint satisfaction problem.

3.3.3 e_{rating}: Finding a Good Solution with Respect to User Rating

A third interesting possibility to guide the search in the case of semantic incompleteness of the query is to use a recommendation system that learns the user preferences over time [66, 90]. The underlying idea is that we can associate a solution with a *loss* value $l \in \mathbb{R}$ that indicates its deviation from an optimal solution. Given a concrete query and a partial composition, the recommendation system then could give a loss value that must be expected when completing the candidate to a real solution. The goal is to find a loss-minimizing solution.

3.3.3.1 Evaluation Function

Evaluation Space The evaluation space is the set of real numbers; i.e. $\mathbb{E} = \mathbb{R}$.

History The history of a partial solution is of secondary priority in this strategy. This is because the strategy looks on what is still possible; the current composition has not caused any loss itself. Hence, the history can be set to the path length to the root.

Heuristic The task of the heuristic is to *estimate* the loss of the "best" solution (with respect to the user rating) derivable from an extended version of the path. This value must be provided by some external database that compares the partial solution of a path with other (earlier) solutions for the same (or similar) original query and derives from the respective ratings a loss value that must be expected when completing the partial solution of the path to a real solution. That is, we assume the possibility to invoke a function GETRATING$(q, c(n))$ provided by the external module.

For good results of this strategy, it is necessary that GETRATING knows the search structure that is used by the composition algorithm. The reason for this is as follows. The external module associates the delivered partial solution $c(n)$ with other previously found solutions for similar queries. However, this makes only sense if $c(n)$ can be completed to any of these solutions (or similar ones) in the used search structure. Only if the experiences on which the external module relies on are based on the search structure used by the composition algorithm, it can drive the search towards solutions that can be expected to optimize the expected user rating.

Aggregation Function Similar to the previous strategy, we apply a weighted sum with a rather small weight for the current path. That is, we have $\circ = w_g \cdot g(n) + h(n)$. Again, there is no obviously good choice for w_g but this would have to be examined through experiments.

Tie-Breaker A tie-breaker for this strategy may be one of the heuristics of the previous strategy. That is, for two paths with equal quality estimation, we would expand the one that is supposed to be closer to a solution.

3.3.3.2 Discussion

The advantage of this strategy is obviously that it optimizes for the alleged desire of the client. That is, the other strategies are uninformed with respect to what the client probably wants with respect to what is not specified formally. The consideration of this aspect is a unique property, which may be highly relevant in several use cases.

There are several issues that could be hard to resolve.

1. The implementation of a learning recommendation system is a highly non-trivial job, and it must be semantically coupled with the search structure. That is, the answer of the recommendation system for a path obviously depends on what a path means, because the path information is used for a lookup. Consequently, the recommendation mechanic must be tailored for the respectively used search structure.

2. The recommendation system must be able to give (non-trivial) heuristic values for most paths of the search structure. Otherwise, the search would degenerate to a breadth-first search, which could significantly delay the time until a solution is returned.

3. A similar case may occur if there are many well-rated solutions, i.e. with equal rating, for similar problems. In this case, the algorithm would potentially explore all partial solutions completable to well-rated solutions simultaneously. As in the previous case, this could lead to a significant delay for the time until a first solution is detected.

4. Total Order Backward Composition

This chapter presents the search structure BW, which follows the idea to create sequential compositions backwards. The root node of the search graph corresponds to an empty composition whose precondition and postcondition correspond to the query postcondition. For an arbitrary node, the child nodes remove one or more literals from the precondition of the composition belonging to the parent by *prepending* a new operation invocation. Together with the precondition of the prepended operation invocation, this yields a new precondition for the new composition. A solution is found when a composition is created whose precondition is implied by the precondition defined in the query.

The content of this chapter is mostly novel even though I have already published some parts of it before [90]. The underlying idea of backward search is of course old and reaches back into the beginnings of STRIPS planning. The novelty is that operations now can create new objects, which was not considered previously. While the above cited work describes the basic ideas of BW, this chapter provides a significant extension of the pruning capacities, treatment of background knowledge, and proofs of correctness and completeness.

The chapter is organized in three sections. Section 4.1 gives an intuition of the search structure and shows its application to the running example. The formal definition is given in Section 4.2. Finally, Section 4.3 provides proofs of correctness and completeness.

4.1 Intuition

I give the intuition for BW in three steps. First, Section 4.1.1 explains the very basic idea of the search structure based on "backward programming". Section 4.1.2 illustrates a composition run with BW solving the running example. Next, Section 4.1.3 goes into some more detail in order to give an intuition for the pitfalls that must be treated in this search model.

4.1.1 Basic Idea

The idea of this search technique is to program backwards. Instead of developing a program by subsequently appending new commands, we *prepend* new commands to the existing program.

Intuitively, the search graph for BW is a *tree* whose nodes correspond to state labels and whose paths encode compositions. The state label associated with a node is the "precondition" of the composition corresponding to the path from it to the root. That is, the composition belonging to the path from the root to node n, denoted as $c(n)$, transforms the state associated with n into the query postcondition. The root node is labeled with the query postcondition $Post_q$, and the composition corresponding to its (empty) path is a composition with only one state and without transitions. Given any node n of the search structure with composition $c(n)$, there is a child node for every "prependable" operation invocation. Let $o[\sigma]$ be an operation invocation. Intuitively, there is a child node for the composition that concatenates $o[\sigma]$ and

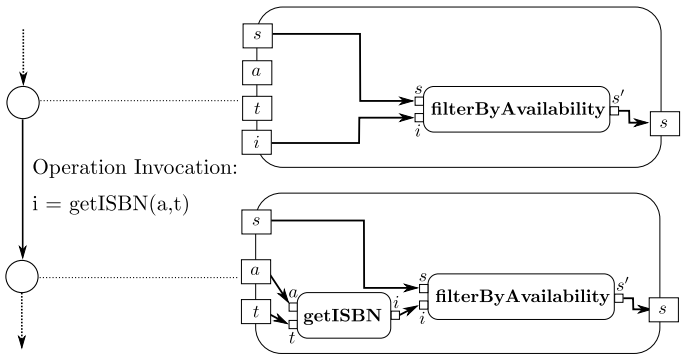


Figure 4.1: Prepending an operation yields a new composition.

$c(n)$ if the postcondition of $o[\sigma]$ contains a literal L that is also in the state label associated with n and, hence in the precondition of $c(n)$. Figure 4.1 shows this relation between two nodes. The label of the child is the label of n without L and with the precondition of $o[\sigma]$.

A node is a goal node if its state label is implied by the query preconditions. On one hand, every time we prepend an operation invocation in the above manner, we obtain a composition with a new initial state and label, which can be seen as a valid precondition of the composition. On the other hand, the postcondition of all derived compositions is unchanged and corresponds to the postconditions of the query. Hence, if we finally obtain a composition whose precondition is implied by the precondition of the query, it is also a solution. In particular, the node encoding it is a goal node.

Manual programming usually relies on domain knowledge that needs to be formalized in order to be exploitable by the composition algorithm. Suppose that a composition contains an operation invocation $o[\sigma]$. A human developer may know that its precondition $Pre_o[\sigma]$ is implied by the query precondition and postconditions of previous operations even if not every literal in $Pre_o[\sigma]$ is contained explicitly in them. The developer simply "sees" that the explicitly mentioned conditions must necessarily imply $Pre_o[\sigma]$. However, the algorithm cannot see this; the knowledge that allows this conclusion must have been made explicit in the domain knowledge Ω in order to be used by the algorithm.

In order to incorporate this knowledge in the composition process, the compositions are not only modified by prepending operation invocations but also by clause applications. That is, if we can derive a particular literal in the preconditions of a composition, we modify its initial state correspondingly. More precisely, we remove that literal from the initial state of the currently considered composition and add the negated remaining literals of the clause to it. The associated composition remains the same. Figure 4.2 shows an example of an edge that corresponds to a clause application.

As a consequence, the compositions are contained in the paths from goal nodes to the root. More precisely, the paths implicitly encode the compositions and what is needed to proof their correctness. Hence, we do not store the compositions explicitly in the nodes.

4.1.2 Example Run

Figure 4.3 illustrates an excerpt of the search structure BW for the case of the running example introduced in Section 1.3. The root node n^0 is labeled with the initial problem,

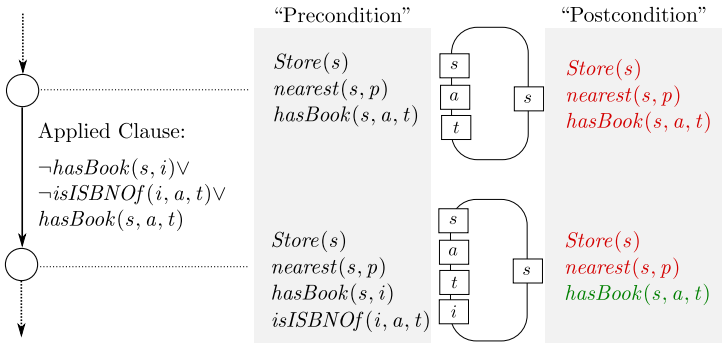


Figure 4.2: Prepending a clause yields a new precondition for a composition.

which encodes the query postcondition together with the type definitions of the inputs. The final node n^* is a solution node from where the solution composition can be reconstructed, walking along the path to the root. The only information we need to store with a node is the *state labeling* in which the associated composition can be executed, i.e. its precondition.

The first three edges correspond to clause applications. First, we exploit the knowledge that two objects are near if they are located in the same place. Then, we exploit the fact that, if every object stored in a set s' is located in city c and if s is a subset of s', then every object in s is also located in c. Finally, we use the fact that whenever a store has a book with ISBN i where i is the ISBN of a book with author a and title t, then the store also has the book with title t written by author a. In each of these cases, one of the literals of the clause is eliminated from the parent state (the positive ones), and the others are negated and added to the state of the next node.

Note that each of these clause applications adds new data containers to the state label of the child node. The first clause adds a data container c for the city. The second clause adds a data container s' for the superset of the outputted set of stores. And the third clause adds a data container i for the ISBN for the desired book. In the implementation, the algorithm uses sequentially numbered names for data constants from which I refrain here for readability.

Of course, the clauses do not produce these data containers in the sense of outputs but require them as parameters. The application of the clauses should be rather understood as a formulation of a condition that must be true in order to enforce that the one remaining literal holds. For example, considering the third clause, we know that a store has the book with author a and title t on storage if "there is an object i that is the ISBN of the item with author a and title t and the store has i on stock". If this condition holds, then one can use the clause to prove that the actually desired condition is implied.

Then, the four operation invocations that actually determine the composition are added in a row. First, the operation to compute, from a given set of stores, the subset of *available* stores is added; this will be the last operation of the composition. Then, getISBN, getStores, and getCity are added sequentially. In-between, we apply the clause that allows to infer the BookShops property of the set of stores.

Of course, the depicted path is only one of many possible paths. In fact, each of the nodes can be extended in different ways, and the systematic search must consider all of them unless they can be pruned. In particular, there are several solution paths even for the bookshop example. To illustrate the process, however, the path shown in Figure 4.3 is sufficient.

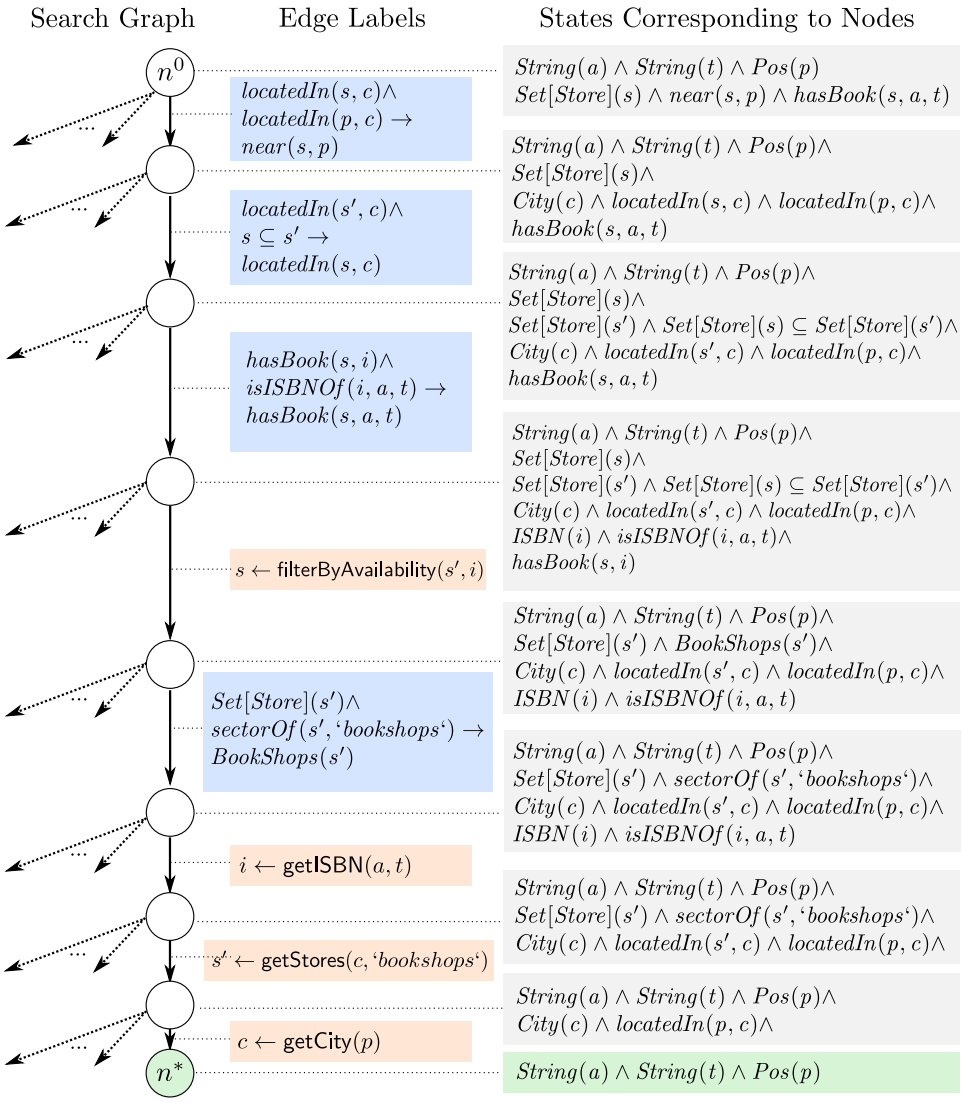


Figure 4.3: Creation of the running example composition using BW.

4.1.3 A Look at the Details

Before writing down the formal search structure definition, we must have in mind three important aspects. First, a strategy is needed to cope with operation inputs whose source is not yet known at time of invocation. Second, type information are a special kind of knowledge, which should be treated specially in the search structure for performance reasons. Third, prepending operation invocations may cause inconsistent states, which must be avoided. This section explains how these questions are resolved for BW.

4.1.3.1 Deciding the Sources of the Inputs

The main advantage of automated "backward programming" as shown here over automated forward programming is that it is more goal directed. In the above structure, we will never consider a composition that contains operation invocations whose postconditions are completely irrelevant for the rest of the composition or the query. This is something that easily happens in forward composition apart from the fact that, due to usually rather small operation preconditions, the branching factor in forward programming is usually significantly larger, which makes it only feasible with extraordinary heuristic support.

However, a significant conceptual challenge of backward programming is that one or more data containers used as inputs of an operation or clause are not available when the operation is added to the program. In common forward programming, the inputs of a new operation must be either inputs of the query (or function) or outputs of any previously inserted operation. This set is well-defined, so the input candidates are clear. However, in backward programming, we do not yet know which operations will be prepended later that may provide new data that may be used as an input for the operation we are inserting now.

As a consequence, the composition algorithm must, at some point of time, decide which are the concrete inputs of each operation or clause. This can either be at time of insertion of the operation or clause using a dummy data container for which it is not yet clear how it will be filled, or it can be made an own decision. In the latter case, the inputs remain unbound at time of insertion, and the algorithm can, at a later point of time, bind it to a data container whose defining output is known at that point of time; this is lifted backward search.

The composition technique discussed here assumes that the decision on the source of every input is made at time of insertion even if the sources do not exist yet. More precisely, when inserting a new operation, the mechanism assigns any input of the operation to one of the following three source types:

1. A query input or constant.
2. A data container that is not a source for any input of the previously inserted operations. In this case, a new data container is introduced.
3. A data container that has been introduced through (2) for a previously inserted operation. In this case, the mechanism decides that the input for this operation and the input of a previously inserted operation are the same even though the operation that produces that data container with one of its outputs is still unknown.

Figure 4.4 provides a snapshot of a summary of a couple of composition steps showing a particular way how these input decisions were made.

The names for new data containers are drawn from a pool of artificial identifiers, which are specific for the respective node. When deriving a node n' from an existing node n, the names of new data containers that may be used as new inputs are $v_{x_1}^{n'}, \ldots, v_{x_k}^{n'}$ where $x_1, .., x_k$ are inputs (variables) of the operation (clause) if n' is derived by prepending an operation (clause). Using the node names as namespaces avoids that an artificial identifier is used twice at different positions in the composition.

Since the actual name of the identifier of the new data container does not matter, it is predefined. That is, if some input x should be bound to a new data container in node n', then the name of the data container must be $v_x^{n'}$. Otherwise we would consider the names of the new variables being a choice.

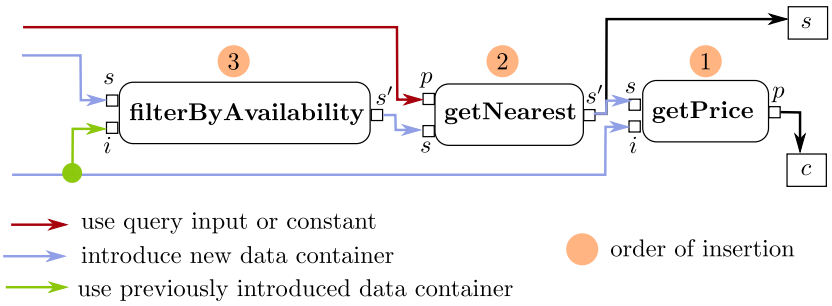


Figure 4.4: Inputs may come from the query inputs X_q or from still unknown operations, which are supposed to be prepended later.

In BW, the decision on the input source is final. That is, there is no mechanism that allows to revoke a variable binding decision at a later point of time.

In order to not miss a solution with a particular decision, each possible decision for the input sources is considered with an individual node at time of operation insertion. In this way, different decisions can be considered through different nodes and, hence, paths.

4.1.3.2 Treating Types

In theory, the above technique already can treat types. In the composition model considered in this thesis, variable types are part of the logical description of an operation. Types of input parameters are defined in the preconditions, and types of output parameters are defined in the postcondition of an operation. Clauses may contain *negated* type literals in order to restrict the types of variables contained in them. Considering type clauses as part of the knowledge base, we have full support for types.

However, treating type knowledge as ordinary knowledge is highly inefficient. The subtype relation between two types t_{sub} and t_{super} is expressed as $\neg t_{sub}(x) \vee t_{super}(x)$, so if we need that some data container is t_{super}, we could produce it using an operation that has an output of type t_{sub}. Without some kind of type matching, we would first have to apply this clause before prepending the desired operation. Even worse, if the operation produces a sub-subtype, we would need to apply two clauses before the operation can be applied. The problem here is that applying clauses is a decision, which causes a branch in the search structure.

Instead, I directly evaluate the type conformity when prepending an operation. That is, even though the required type predicate is not provided by the postcondition of the prepended operation, it is provided by the postcondition *together with* the type hierarchy system. Since the type heterarchy is a definite Horn formula without existence quantifiers or functions, checking the type conformity can be done in polynomial time.

As a consequence, clauses from the type heterarchy $\mathcal{T}$ can be excluded from the clause application part. The above technique makes an explicit type casting decision superfluous.

Note that this technique works only for a very special type of clauses, which is why the clause application technique remains necessary. The problem is that we need the possibility to introduce new data containers through the backward application of clauses. If we only checked whether a particular literal of a state label can be followed from the postcondition

of an operation and the knowledge base, this would not be possible anymore. The special property of type clauses is that they contain only one variable, which means that they cannot introduce new data containers to the precondition of the composition.

A second issue with types is that we may want to decide that two inputs of different operations have the same source but the types of the inputs are different. For example, one input has type t_1 and the other has type t_2 where none of them is a subtype of the other but there is a common subtype t_3 of both of them. In object oriented programming, this is for example the case if t_1 and t_2 are interfaces that are implemented by a class t_3. Intuitively, there is the necessity to determine whether two inputs may have the same source.

Fortunately, this analysis can be easily done by checking whether a common subtype exists. If no common subtype of t_1 and t_2 exists, it will not be possible to find an operation that provides the source for these inputs. Hence, when trying to prepend a new operation o that uses a known data container of type t_1 as input for a variable with type requirement t_2 in the preconditions of o, we perform this subtype check. If the test succeeds, we add the second type predicate, i.e. t_2 to the state of the child node. For example, if we first introduced the operation that uses t_1, then we inserted a data container, say v such that $t_1(v)$ is part of the new precondition. Adding the second operation and deciding that the inputs are equal would yield another predicate $t_2(v)$ in the precondition. Of course, this technique can be also used to add a third or fourth input with a still different type to the same source.

4.1.3.3 Elimination of Knowledge About Outputs

Another pitfall is the loss of information about a data container when it is written. When a data container is filled with content, everything we knew about it before is lost. This is exactly the same what happens in programming when we write a variable: Everything that was in the variable before gets lost and is replaced by the new content. Thereby, the knowledge about the object, which refers to a previous *version* (or state) of it, becomes invalid.

The consequence for programming backwards is that everything we need to know about a data container in the moment of its production must be provided by the operation that creates it. Gathering knowledge about that data container is possible when creating the data container and afterwards (in the sense of the composition execution order) but not before it is created. Hence, the postcondition of an operation invocation must cover every literal in the successor state that contains one of the data containers written by the invocation, i.e. every data container of that state two which an output of the operation is bound.

4.2 Search Structure

I now define the search structure elements based on a composition problem instance. That is, I define it point-wise for each composition problem instance $\langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$ as defined in Section 2.1.4. The following definitions rely on the elements of this instance.

The organization is as follows. I first describe the search structure in Section 4.2.1 (search graph G_{BW}), Section 4.2.3 (goal node function $\star_{\mathrm{BW}}$), and Section 4.2.2 (translation function $\textsc{Trans}_{\mathrm{BW}}$). Second, Section 4.2.4 explains the implementation for the exploration strategies e_{fast} and e_{nf} (cf. Section 3.3). Finally, I describe the the SR-dominance relation $\succeq_{\mathrm{BW}}$ (cf. Section 3.2.3.2) in Section 4.2.5.

4.2.1 The Search Graph G_{BW}

I first give the basic graph definition and then the additional node labelings that may be used by the heuristics or pruning function.

4.2.1.1 Core Graph Definition

State Label $\lambda(n)$ of nodes

Every node n in G_{BW} is associated with a *state label* $\lambda(n)$ (cf. Def. 9 in Section 2.1.2.2), which is the *precondition* of the composition corresponding to the path from that node to the root. It will turn out that G_{BW} is a tree, so this path is unique and the composition $c(n)$ can be associated with the node n. However, the nodes *encode* only the state labeling $\lambda(n)$ but not the respective composition $c(n)$, which is implicitly encoded in the path labels from a node to the root node.

Composition $c(n)$ of nodes

The following definition makes use of several symbols introduced in Chapter 2 and Chapter 3. For an easier overview, a short summary is as follows:

- Ω and $\mathcal{T}$ are the domain knowledge and type heterarchy respectively;
- for an operation o, the symbols X_o, Y_o, Pre_o, and $Post_o$ denote inputs, outputs, precondition, and postcondition respectively. The same symbols with subscript q refer to the respective element of the query;
- $\Gamma_{data}(n)$ are data containers that occur in a state labeling $\lambda(n)$ of node n—it holds that $\Gamma_{data}(n) \subset \Gamma_{data}$; and
- $\rightsquigarrow$ is used instead of $\rightarrow$ to denote *partial* function, i.e. a function that is not necessarily defined for every element in its domain.

For convenience, I will adopt set operators such as union and difference also to logic conjunctions such as Pre_o, $Post_o$, and $\lambda(n)$; the results are sets with the semantic of conjunctions.

Now the graph of the search structure is defined as follows:

Definition 16. *The **search graph G_{BW} for backward search of sequential compositions** is inductive:*

1. *G_{BW} contains a distinguished root node $\text{getRoot}_{BW}() = n^0_{BW}$ with $\lambda(n^0_{BW}) = Post_q$.*
2. *Let n in G_{BW} be a node. A node n' is a successor of n, i.e. $n' \in \text{getSuccessors}_{BW}(n)$, if there are an operation $o \in O$, an input mapping $\sigma_{in} : X_o \rightarrow (\Gamma_{const} \cup \Gamma_{data}(n) \cup \{v^{n'}_{x_1}, \ldots, v^{n'}_{x_{|X_o|}}\})$, and an injective output mapping $\sigma_{out} : Y_o \rightsquigarrow \Gamma_{data}(n)$ such that*
   (a) *$T = \{t(v) \mid \exists y \in Y_o : \sigma_{out}(y) = v, t(v)$ is a type literal in $\lambda(n)\}$,*
   (b) *$\lambda(n') = (\lambda(n) \setminus (Post_o[\sigma] \cup T)) \cup Pre_o[\sigma_{in}]$,*
   (c) *at least one literal of the previous precondition is created by $o[\sigma]$, i.e. $\exists L \in \lambda(n) : Post_o[\sigma] \wedge \mathcal{T} \models L$,*
   (d) *the types of produced data containers are consistent with the type hierarchy, i.e. $Post_o[\sigma_{out}] \wedge \mathcal{T} \models \bigwedge_{t \in T} t$,*
   (e) *produced data containers do not occur in the new state or in the query inputs, i.e. $\sigma_{out}(Y_o) \cap \Gamma_{data}(n') = \emptyset$ and $\sigma_{out}(Y_o) \cap X_q = \emptyset$, and*
   (f) *for every $x \in X_o$, if $\sigma_{in}(x) \in \{v^{n'}_{x_1}, \ldots, v^{n'}_{x_{|X_o|}}\}$, then $\sigma_{in}(x) = v^{n'}_x$.*

The edge (n, n') represents an operation invocation $o[\sigma]$ and is labeled correspondingly.

3. *Let n in G_{BW} be a node. A node n' is a successor of n, i.e. $n' \in \text{GETSUCCESSORS}_{BW}(n)$, if there are a clause $\alpha = \neg\alpha_1 \vee \ldots \vee \alpha_i \vee \ldots \vee \neg\alpha_n \in \Omega$ and a mapping $\sigma : vars(\alpha) \to (\Gamma_{const} \cup \Gamma_{data}(n) \cup \{v_{x_1}^{n'}, \ldots, v_{x_{|vars(\alpha)|}}^{n'}\})$ such that*

   (a) $\lambda(n') = (\lambda(n) \setminus \alpha_i[\sigma]) \cup \bigcup_{j \neq i} \alpha_j[\sigma]$,

   (b) $\alpha_i[\sigma] \in \lambda(n)$,

   (c) for any $x \in vars(\alpha)$, if $\sigma(x) \in \{v_{x_1}^{n'}, \ldots, v_{x_{|vars(\alpha)|}}^{n'}\}$, then $\sigma(x) = v_x^{n'}$.

   The edge (n, n') is a backward application of the implication $\left(\left(\bigwedge_{j \neq i} \alpha_j\right) \to \alpha_i\right)[\sigma]$ and is labeled correspondingly.

G_{BW} does not contain other nodes; in particular, $\text{GETSUCCESSORS}_{BW}(n)$ is completely described by (2) and (3).

Remarks.

- (2b) and (3a) are the definitions of the labels associated to the successor nodes in the graph. In case of (2), it corresponds to the precondition of the composition before the respective operation invocation. In case of (3), it corresponds to a different labeling for the initial state that allows to derive $\lambda(n)$ using the background knowledge Ω.

- (2c) and (3b) are the conditions that require a contribution of the action for the current precondition. (2c) says that the postcondition of the prepended operation must resolve at least one literal of the former precondition, and (3b) says that the conclusion of the induced rule must be in the former precondition. In the first case, the type heterarchy $\mathcal{T}$ may be used to infer the desired literal(s). In the second case, this is not necessary, because the head of a rule cannot be a type literal (since these must be negative), so type knowledge cannot be used to infer anything from the head literal.

- (2d) and (2e) are restrictions for prepending operations. (2d) says that every type literal of a data container that will be bound to an output of the prepended operation must be deducible from the respective type of the output of the operation, and (2e) restricts the output mapping such that it does not map outputs to input variables (data containers) defined in the query, and it requires that there is no knowledge about a data container before the operation that creates the respective container is invoked.

- the mappings σ_{in} in (2) and σ in (3) may introduce new data containers. In (2), there may be one new container $v_i^{x'}$ for each input $i \in X_o$ of the operation. In (3), there may be one new container for each variable occurring in the clause. These correspond to data containers that may be produced in the (yet unknown) previous part of the composition but that are not used after this operation invocation and, hence, not specified in the successor state (state of parent node).

- The last conditions (2f) and (3c) imply that we do not make the naming of data containers a choice point. We fix in advance the name of a new data container for the case that the input of the operation shall be bound to one. Consequently, it only remains a choice *whether* a new data container is used but not its name.

Besides the state node label λ, a node is labeled with the non-functional properties of the composition belonging to it, denoted as $\odot$. We can easily compute the non-functional properties using recursion. The empty composition has values 0 for every non-functional property, so $\odot(n^0_{\text{BW}}) = 0$. For any other node n with $n \in \text{GETSUCCESSORS}_{\text{BW}}(n')$, we can compute the property as follows:

$$\odot(n)_i = \begin{cases} \odot(n')_i \oplus_i (Z_o)_i & \text{if } (n', n) \text{ is an invocation of operation } o \\ \odot(n')_i & \text{else (edge is clause application)} \end{cases}$$

Recall from Section 2.1 that $\oplus_i$ is the sequential aggregation function of property i, and $(Z_o)_i$ is the value of property i of operation o. Since $\odot(c)$ is the sum (w.r.t $\oplus_i$ of the different properties) of the operation invocations in c, it holds that $\odot(n) = \odot(c(n))$.

4.2.2 The Transformation Function TRANS_{BW}

Reconstructing the composition belonging to node is straight forward walking along the path (cf. Alg. 2). It can be easily seen that the runtime is linear in the length of the path, because it simply walks the path from the head to the root. Edges that correspond to clauses are ignored, because they are relevant only for the proof of correctness of the solution but not for the composition itself. The algorithm returns a *set* that contains exactly one composition, which is a list of operation invocations. In fact, it returns a composition, but since the framework requires TRANS to return a set, we return a set of size 1.

Algorithm 2: TRANS_{BW}

Inputs : Path $p = (n_0, .., n_k)$ where $n_0 = n^0_{\text{BW}}$
Output: Composition $c(n)$

```
1  s^0 ← new State();
2  S ← {s^0};
3  Σ ← ∅;
4  δ ← ∅;
5  s^f ← s^0;
6  for i ← k to 1 do
7      if label(n_i, n_{i-1}) = o[σ] then
8          s ← new State();
9          S ← S ∪ {s};
10         Σ ← Σ ∪ {o[σ]};
11         δ ← δ ∪ {((s^f, o[σ]), s)};
12         s^f ← s;
13     end
14 end
15 return {new Composition(S, Σ, δ, s^0, {s^f})};
```

4.2.3 Goal Function $\star_{\text{BW}}$

A node is a *goal node* if its composition is a solution to the query. Recalling Def. 15 (cf. Section 2.1), a composition c is a solution to query q iff there is a valid state labeling λ for

c such that the query precondition implies the label of the initial state of the composition ($Pre_q \models \lambda(s^0)$), if the label of every final state implies the postcondition of the query ($\lambda(s^f) \models Post_q$ for every $s^f \in F$), and if the non-functional requirements are satisfied ($\odot(c) \leq Z_q$). So, for every node $n \in N_{\mathrm{BW}}$, we need that $\star_{\mathrm{BW}}(n) = \mathit{true}$ iff $c(n)$ satisfies these properties.

We can easily check this solution property without reconstructing the composition. The first condition $Pre_q \models \lambda(n)$ can be easily checked using the state label associated with the node modulo an implicit type cast for the inputs used from the query. The second condition $\lambda(s^f) \models Post_q$ holds for *every* composition obtainable through $\textsc{Trans}_{\mathrm{BW}}$. Finally, in the case of monotone aggregation of non-functional properties, every node n that does not satisfy the non-functional properties is pruned, so $\odot(c) \leq Z_q$ holds for every $c \in \textsc{Trans}_{\mathrm{BW}}(p)$ for every path p with non-pruned head. However, in order to keep the goal condition also valid for non-monotonic properties, the check is integrated into the goal condition. The check is then achieved by the following function:

$$\star_{\mathrm{BW}}(n) = \begin{cases} \mathit{true} & \text{if } \odot(n) \leq Z_q \text{ and } Pre_q \wedge \mathcal{T} \models \lambda(n) \\ \mathit{false} & \mathit{else} \end{cases}$$

In other words, if the state label of a node that does not violate the non-functional properties of the query is a subset of the precondition of the query, we have found a solution to the query. In particular, the solution is provably *correct* with respect to the query preconditions and postconditions.

4.2.4 Implementation of Exploration Strategies

I now describe the implementation of the exploration strategies e_{nf} and e_{fast} (cf. Section 3.3). An implementation of e_{rating} is not part of this thesis, but a sketch can be found in [66].

The history g for e_{nf} corresponds to the non-functional properties stored for the node at the end of the path. Hence, $g((n^0_{\mathrm{BW}}, .., n)) = \odot(n)$. The history for e_{fast} was already defined as the path length in Section 3.3.2.

The heuristics of e_{nf} and e_{fast} rely on a *relaxation* of the rest problem. The rest problem is relaxed to a simple set theoretic planning problem. The solution to the relaxed problem is a sequence of operations (instead of operation invocations) and propositional clauses (instead of ground first order clauses).

We compute the relaxed rest problem of a node n as follows: We use the function *prop*, which removes the the literal parameters from a given formula; that is, $\mathit{prop}(\alpha)$ is α where every predicate $P(X)$ is replaced by P (negations and junctions are not touched). In a preprocessing step, we compile a set of actions. This set contains one action for each relaxed operation, i.e. for each $o \in O$, there is an action with $\mathit{prop}(Pre_o)$ as precondition, $\mathit{prop}(Post_o)$ as postcondition, and Z_o as values of the non-functional properties. Also, there is one action for each clause of Ω with $\bar{L}_1 \wedge .. \wedge \bar{L}_m$ as precondition where $L_1(X_1), .., L_m(X_m)$ are the negative literals of the clause, and L as postcondition where $L(X)$ is the positive literal of the clause. The non-functional properties are 0. This relaxed planning problem is then $(\mathit{prop}(Pre_q), \mathit{prop}(\lambda(n)), Z')$ where $Z'_i = \ominus_i((Z_q)_i, \odot(n)_i)$ for every non-functional property i. These actions can be used to solve the rest problems of an arbitrary node.

The heuristic h differs between the two strategies. In e_{nf}, we set $h(n)$ to the *non-functional properties* of the solution to the relaxed problem. In e_{fast}, we set $h(n)$ to the *length* of the solution path (including edges for clauses). In the case of e_{fast}, the concrete value for $h(n)$ depends on the algorithm used to solve the relaxed problem.

4.2.5 SR-Dominance Relation $\succeq_{\text{BW}}$

The SR-Dominance relation for BW is based on the old idea of "having to do less is better". In set-theoretic planning, this would be to say that $n \succeq_{\text{BW}} n'$ holds iff the state in n is a subset of the state of n'. However, this relation should be refined in our setup, because we know that compositions are equivalent if their data containers can be appropriately renamed. Taking into account that also the non-functional properties must be semi-dominated, we define that $n \succeq_{\text{BW}} n'$ is true iff there is an *injective* mapping $\varphi : \Gamma_{data}(n) \to \Gamma_{data}(n')$ such that $\lambda(n)[\varphi] \subseteq \lambda(n')$ and if $\odot(n) \leq \odot(n')$. Injectivity is necessary, because the above way how data containers are created implies the unique name assumption; hence, we must respect the different names of the objects, which would not be the case for non-injective mappings.

Computing this relation for two nodes cannot be done efficiently in general. The number of candidates for σ is $|\Gamma_{data}(n)|! \cdot \binom{|\Gamma_{data}(n')|}{|\Gamma_{data}(n)|}$, which may become an infeasible number even for moderate states with, say, 10 data containers.

However, an intelligent filtering avoids that we need to look at all these mappings. More precisely, we can exploit the type information about the data containers in the state, because we only need mappings where data containers are mapped to others of the same type. That is, we can compose σ from $\sigma_1, \ldots, \sigma_n$, where σ_i only substitutes data containers that share the same type i. Using this technique, we can reduce the number of candidates by an order of magnitude and make the comparison feasible even for big states.

4.3 Theoretical Analysis

This section presents key results on the theoretic properties *correctness* and *completeness* of running the search algorithm using BW as search structure and $\succeq_{\text{BW}}$ as SR-dominance relation. Section 4.3.1 shows that $\textsc{Search}_{\text{BW},\mathcal{E},\mathcal{P}_{\succeq_{\text{BW}}}}$ is a correct algorithm for sequential composition, and Section 4.3.2 shows that it is complete for the exploration strategy e_{fast}. Again, the detailed versions of sketched proofs can be found in the appendix.

4.3.1 Correctness of $\textsc{Search}_{\text{BW},\mathcal{E},\mathcal{P}_{\succeq_{\text{BW}}}}$

$\textsc{Search}_{\text{BW},\mathcal{E},\mathcal{P}_{\succeq_{\text{BW}}}}$ is correct iff the search structure BW itself is correct. This was the result of Theorem 3.1 (cf. Section 3.2.4.1). Correctness of the search structure means that compositions belonging to paths to goal nodes must be retrievable in polynomial time and that these compositions satisfy the solution criteria, i.e. they transform the query precondition into the query postcondition minimally and adhere to the bounds on non-functional properties (cf. Section 3.1.3.1). Since there is exactly one composition $c(n)$ associated with each node n, given a query q, we only must show that $\textsc{Trans}_{\text{BW}}$ derives $c(n)$ in time polynomial in the length of the path from n^0_{BW} to n, and that $c(n)$ is a solution to q.

The proof goes in two steps. First, I show that, given *any* node n of G_{BW}, $\textsc{Trans}_{\text{BW}}$ efficiently computes a valid composition $c(n)$ that minimally transforms $\lambda(n)$ into $Post_q$. Second, I show that the computed $c(n)$ is a solution to q if $\star_{\text{BW}}(n) = true$.

Lemma 4.1. *Let q be a query and $p = (n^0_{BW}, .., n)$ a path in G_{BW}. Then $\textsc{Trans}_{BW}(p)$ computes in linear time w.r.t. $|p|$ a sequential composition that transforms $\lambda(n)$ into $Post_q$.*

Proof Sketch. It is obvious that $\textsc{Trans}_{\text{BW}}(p)$ computes a single sequential composition within linear time in the length of p since it walks along that path. The proof that the resulting

composition transforms $\lambda(n)$ into $Post_q$ is by induction over the length of paths. For the induction basis, there is only one path, which is the one containing only the root. Here $\lambda(n) = Post_q$, i.e. the condition is trivially true. For other nodes, the condition follows relatively straight forward from the node label definition. □

Together with the definition of the goal node function, this result implies the correctness of the search structure.

Theorem 4.2. *The search structure BW is correct.*

Proof Sketch. This follows directly from the previous Lemma and the definition of $\star_{\mathrm{BW}}$. Let $p = (n^0_{\mathrm{BW}}, .., n)$ be a path such that $\star_{\mathrm{BW}}(n) = true$. The goal condition gives us that $Pre_q \wedge \mathcal{T} \models \lambda(n)$ and that $\odot(n) \leq Z_q$, and the previous Lemma gives us that the composition c in $\textsc{Trans}_{\mathrm{BW}}$ transforms $\lambda(n)$ into $Post_q$. It is easy to see that the above implies that c also transforms Pre_q into $Post_q$, and, hence is a solution to the query. □

The consequence is that the search algorithm works correct using BW regardless the used exploration strategy.

Corollary 4.3. $\textsc{Search}_{BW,\mathcal{E},\mathcal{P}_{\succeq_{BW}}}$ *is correct for any strategy* $\mathcal{E}$*.*

Proof. By Theorem 3.1 (cf. Section 3.2.4.1), $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_{\succeq}}$ is correct whenever the search structure $\mathcal{S}$ is correct. Since BW is correct, $\textsc{Search}_{\mathrm{BW},\mathcal{E},\mathcal{P}_{\succeq_{\mathrm{BW}}}}$ is also correct. □

4.3.2 Completeness of $\textsc{Search}_{\mathsf{BW},\mathcal{E},\mathcal{P}_{\succeq_{\mathsf{BW}}}}$

Given a strictly increasing exploration strategy $\mathcal{E}$ (cf. Section 3.2.4), two more conditions must be satisfied in order to ensure completeness of the search algorithm $\textsc{Search}_{\mathrm{BW},\mathcal{E},\mathcal{P}_{\succeq_{\mathrm{BW}}}}$:

1. the search structure BW itself must be complete (cf. Section 3.1.3.2)
2. the SR-dominance relation $\succeq_{\mathrm{BW}}$ must be completeness-preserving (cf. Section 3.2.3.2).

I show these properties in the following two sections respectively.

4.3.2.1 Completeness of BW

Proving that every solution is contained in the search structure goes in three steps. First, in Lemma 4.4, I show that the clause edges simulate a complete backward chaining. Second, I use this Lemma to show that for any condition and any composition that transforms the condition into the query postcondition, there is a node that encodes that composition; this is the result of Lemma 4.5. Theorem 4.6 concludes the completeness from these results.

Lemma 4.4. *Let β be a finite conjunction of ground literals and n be a node in G_{BW} such that $\beta \wedge \Omega \wedge \mathcal{T} \models \lambda(n)$. Then there is a finite path from n to a node n' with $\beta \wedge \mathcal{T} \models \lambda(n')$ and $\lambda(n') \wedge \Omega \wedge \mathcal{T} \models \lambda(n)$.*

Proof Sketch. If $\beta \wedge \Omega \wedge \mathcal{T} \models \lambda(n)$ holds, then there is an SLD resolution refutation for $\beta \wedge \Omega \wedge \mathcal{T} \wedge \neg\lambda(n)$ [5] with $\neg\lambda(n)$ as top clause such that the refutation can be split into a non-unit part (side clauses are non-units from Ω and $\mathcal{T}$) and a unit part (side clauses are

only unit clauses from β). There is an edge from n for the first side clause from Ω in the refutation, and there is an edge from that node for the second side clause from Ω, and so on. Finally, we end up with a path of length k to a node n' for k side clauses from Ω. The label $\lambda(n')$ contains those units of β that are necessary to complete the refutation. In particular, no knowledge other than the type heterarchy is necessary to infer $\lambda(n')$ from β. □

Now I show that, for any condition β and any composition c that minimally transforms β into the query postcondition modulo some initial type definitions, G_{BW} contains a node whose path encodes c. This already is the core part of completeness, which only needs to be complemented by the *recognition* of such nodes as goals.

Lemma 4.5. *Let q be a query, β be a finite conjunction of ground literals, and $c \in \mathscr{S}$ be a sequential composition that transforms β into $Post_q$ minimally (cf. Def. 12 in Section 2.1.2.2). Then there is a node n in G_{BW} such that $c = c(n)$ and $\beta \wedge \mathcal{T} \models \lambda(n)$.*

Proof Sketch. The proof is by induction over the number k of transitions of c. If there is no transition, then the claim follows directly from the previous Lemma. For $k > 0$, we know by the induction hypothesis that such a node n' exists for the composition c' corresponding to c without the first transition, say $o[\sigma]$. Since c' *minimally* transforms β into $Post_q$, $o[\sigma]$ is necessary; in particular, there is a literal in the postcondition $Post_o[\sigma]$ that is needed in order to guarantee a sound execution of c' when departing from β. It is not hard to see that then there are nodes n'', n''', and n such that n''' is the node obtained by prepending $o[\sigma]$, and n'' itself and n are reached by clause edges in the spirit of the previous Lemma. In particular, since $c(n)$ is c' with $o[\sigma]$ prepended yields $c(n) = c$. □

Now that we know that every minimal composition that transforms any state into $Post_q$ modulo initial types is contained in the search graph, we can use this to show that every minimal *solution* to a query is contained and that it is recognized as such.

Theorem 4.6. *The search structure BW is complete.*

Proof Sketch. Let q be a query and c be a solution to q. The previous Lemma implies that there is a node n such that $c(n) = c$ and $Pre_q \wedge \mathcal{T} \models \lambda(n)$. Since c is a solution to q and $\odot(n) = \odot(c(n))$, $\odot(n) \leq Z_q$, i.e. $\star_{\text{BW}}(n) = true$, i.e. the solution is recognized. □

Completeness of the search structure is a necessary but not a sufficient condition for the completeness of the search algorithm. What remains to show is that $\succeq_{\text{BW}}$ preserves the completeness (cf. Section 3.2.3.2).

4.3.2.2 $\succeq_{\text{BW}}$ is Completeness-Preserving

The SR-dominance relation $\succeq_{\text{BW}}$ preserves completeness under two conditions:

1. if $n_1 \succeq_{\text{BW}} n_2$ is true and if there is a path of length r from n_2 to a goal node, then there is a path of length at most r from n_1 to a goal node, and
2. $\succeq_{\text{BW}}$ is well-founded on N_{BW}

Showing property (1) goes in four steps. First, I show that $n_1 \succeq_{\text{BW}} n_2$ implies that, for every outgoing edge of n_2 labeled with an operation invocation leading to n_2', it holds that $n_1 \succeq_{\text{BW}} n_2'$ or, for a direct successor n_1' of n_1, it holds that $n_1' \succeq_{\text{BW}} n_2'$ (Lemma 4.7). Then,

I show the same relation for the case that the edge from n_2 to n_2' is labeled with a clause application (Lemma 4.8). Third, I merge these results to show that if $n_1 \succeq_{\text{BW}} n_2$, then for every descendant n_2' of n_2, there is a descendant n_1' of n_1 such that $n_1' \succeq_{\text{BW}} n_2'$ holds (Lemma 4.9). This result directly implies the desired property (Lemma 4.9).

Lemma 4.7. *Let $n_1 \succeq_{BW} n_2$ be true and let (n_2, n_2') be an edge labeled $o[\sigma]$ in G_{BW}. Then $n_1 \succeq_{BW} n_2'$ or there is a direct successor n_1' of n_1 in G_{BW} such that $n_1' \succeq_{BW} n_2'$.*

Proof Sketch. Suppose that $n_1 \succeq_{\text{BW}} n_2'$ does not hold. Then we can use the same operation o (with a possibly different mapping σ') to reach a node n_1' such that $n_1' \succeq_{\text{BW}} n_2'$ holds. Intuitively, the mapping σ' is σ except that it introduces new data containers for the ones known in n_2 but not in n_1. Based on σ' and the mapping φ from data containers in $\lambda(n_1)$ to the ones in $\lambda(n_2)$, which exists by $n_1 \succeq_{\text{BW}} n_2$, it is not hard to construct a mapping φ' from $\lambda(n_1')$ to $\lambda(n_2')$ that proves that $\lambda(n_1')[\varphi'] \subseteq \lambda(n_2')$ and, hence, $n_1' \succeq_{\text{BW}} n_2'$ holds. □

So we now know that the comparison relation $\succeq_{\text{BW}}$ can be propagated using edges that correspond to operation invocations. We now show that this is also possible for the case that the edge from n_2 to n_2' is labeled with a clause application.

Lemma 4.8. *Let $n_1 \succeq_{BW} n_2$ be true and let (n_2, n_2') be an edge in G_{BW} with the label $\left(\left(\bigwedge_{j \neq i} \alpha_j\right) \rightarrow \alpha_i\right)[\sigma]$. Then $n_1 \succeq_{BW} n_2'$ or there is a direct successor n_1' of n_1 in G_{BW} such that $n_1' \succeq_{BW} n_2'$.*

Proof Sketch. The proof is largely analogous to the previous one. The only difference is that, instead of talking about an operation, we use the same *clause* departing from n_1. □

This gives us that the comparison relation $\succeq_{\text{BW}}$ can be propagated using edges that correspond to clause applications. We now generalize these results for the relation $\succeq_{\text{BW}}$ between nodes reachable over *paths* from nodes n_1 and n_2.

Lemma 4.9. *Let $n_1 \succeq_{BW} n_2$ be true. Then for any descendant $n_2' \in desc_{BW}(n_2)$, there is a descendant $n_1' \in desc_{BW}(n_1)$ of n_1 such that*

1. *$n_1' \succeq_{BW} n_2'$ holds and*
2. *the path from n_1 to n_1' is at most as long as the path from n_2 to n_2'.*

Proof. The proof is by induction over the length k of the path from n_2 to n_2'.

Induction Basis. Let $k = 0$. Then $n_2' = n_2$, so we choose $n_1' = n_1$, and $n_1' \succeq_{\text{BW}} n_2'$ holds.

Inductive Step. Let $k > 0$, and let n_2^p be the parent node of n_2'. By the induction hypothesis, we know that for n_2^p, there exists a $n_1^p \in desc_{\text{BW}}(n_1)$ such that $n_1^p \succeq_{\text{BW}} n_2^p$ holds, and the path length from n_1 to n_1^p is at most $k-1$. We know that $n_1^p \succeq_{\text{BW}} n_2'$ or n_1^p has a direct successor n_1' such that $n_1' \succeq_{\text{BW}} n_2'$ is true. If $n_1^p \succeq_{\text{BW}} n_2'$, we choose $n_1' = n_1^p$, so the length of the path between n_1 and n_1' is at most $k-1$. Otherwise, Lemma 4.7 and Lemma 4.8 imply that there is a successor n_1' of n_1^p such that $n_1' \succeq_{\text{BW}} n_2'$. The length from n_1 to n_1' is at most $(k-1)+1 = k$. □

In other words, $n_1 \succeq_{\text{BW}} n_2$ implies that for every node n_2' reachable from n_2 with r edges, there is a node n_1' reachable from n_1 with at most r edges such that $n_1' \succeq_{\text{BW}} n_2'$. The only thing that remains to show is that if a goal node is reachable from n_2 with $n_1 \succeq_{\text{BW}} n_2$ using r edges, then a goal node is reachable also from n_1 using at most r edges.

Theorem 4.10. *Let q be a query, $n_1 \succeq_{BW} n_2$ be true and assume that there is a path of length r from n_2 to a goal node n_2', i.e. $\star_{BW}(n_2) = true$. Then there is a path of length at most r from n_1 to a goal node n_1', i.e. $\star_{BW}(n_1') = true$.*

Proof. By $\star_{\mathrm{BW}}(n_2') = true$, we know that $Pre_q \wedge \mathcal{T} \models \lambda(n_2')$ and $\odot(n_2') \leq Z_q$. By Lemma 4.9, we know that there is a path of length at most r from n_1 to n_1', and $n_1' \succeq_{\mathrm{BW}} n_2'$; i.e. there is an injective function φ such that $\lambda(n_1')[\varphi] \subseteq \lambda(n_2')$ and $\odot(n_1') \leq \odot(n_2')$. It is not hard to see that φ can be the identity function, which implies $\lambda(n_1') \subseteq \lambda(n_2')$. Then we have $Pre_q \wedge \mathcal{T} \models \lambda(n'_1)$ and $\odot(n_1') \leq \odot(n_2') \leq Z_q$, which implies that $\star_{\mathrm{BW}}(n_1') = true$. □

So the first of the two above conditions is shown. For the second property, we need to show that $\succeq_{\mathrm{BW}}$ is well-founded on the set of nodes N_{BW}.

Theorem 4.11. *$\succ_{BW}$ is well-founded on the set of nodes N_{BW}.*

Proof. A relation R is well founded on a set M iff every non-empty finite subset of M has a minimal element with respect to R. Now let $\widehat{N}$ be a non-empty finite subset of N_{BW}. There is a node in $\widehat{N}$, say n, that has a minimal label *size* among the nodes in $\widehat{N}$, i.e. $|\lambda(n)| \leq |\lambda(n')|$ for every other node $n' \in \widehat{N}$. Now we can show well-foundedness by showing that $n' \succ_{\mathrm{BW}} n$ does not hold for any of the other nodes in $\widehat{N}$. Suppose the opposite, i.e. $n' \succ_{\mathrm{BW}} n$. Then there is an *injective* mapping φ and $\lambda(n')[\varphi] \subseteq \lambda(n)$ and there is no such mapping that the opposite holds. But then it must be the case that $\lambda(n')[\varphi] \subset \lambda(n)$; in particular $|\lambda(n')[\varphi]| < |\lambda(n)|$, which contradicts that $|\lambda(n')| \geq |\lambda(n)|$ for every $n' \in \widehat{N}$. Hence, n is the minimal element of $\widehat{N}$ with respect to $\succ_{\mathrm{BW}}$. □

The latter two theorems then directly gives us that $\succeq_{\mathrm{BW}}$ is completeness-preserving.

Corollary 4.12. *$\succeq_{BW}$ is completeness-preserving.*

This result now enables us to make assertions about the completeness of the search algorithm using BW as a search structure and $\succeq_{\mathrm{BW}}$ as a SR-dominance relation for pruning.

4.3.2.3 Completeness of $\textsc{Search}_{\mathsf{BW},\mathcal{E},\mathcal{P}_{\succeq_{\mathsf{BW}}}}$

I know show that $\textsc{Search}_{\mathrm{BW},\mathcal{E},\mathcal{P}_{\succeq_{\mathrm{BW}}}}$ is complete for every strictly increasing exploration strategy $\mathcal{E}$. By Theorem 3.8, we know that $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_{\succeq}}$ is complete if $\mathcal{S}$ is complete, $\mathcal{E}$ is strictly increasing, and if $\succeq$ is completeness-preserving. The above then holds, because BW is complete and $\succeq_{\mathrm{BW}}$ is completeness-preserving.

Since we know e_{fast} to be strictly increasing, we have a proof for the completeness of one parametrization of the composition algorithm.

Corollary 4.13. $\textsc{Search}_{BW,e_{fast},\mathcal{P}_{\succeq_{BW}}}$ *is complete.*

Proof. The completeness follows directly from Corollary 3.10 (cf. Section 3.2.4.2), which asserts that $\textsc{Search}_{\mathcal{S},e_{fast},\mathcal{P}_{\succeq}}$ is complete whenever $\mathcal{S}$ is complete, which is the result of Theorem 4.6, and $\succeq$ is completeness-preserving, which is the result of Corollary 4.13. □

Since e_{nf} is not strictly increasing in general, we cannot make the same assertion for $\textsc{Search}_{\mathrm{BW},e_{nf},\mathcal{P}_{\succeq_{\mathrm{BW}}}}$.

5. Partial Order Backward Composition

This chapter describes a search structure called PO (Partial Order) that finds compositions based on a least-commitment strategy. In contrast to the strategy presented in Chapter 4, this one allows to insert new operations at arbitrary positions of the existing partial compositions, and inputs are not bound all at once but only one at a time. It builds on top and is closely related to partial order planning.

The content of this chapter is completely novel. Of course, it is heavily inspired by partial-order planning [22, 83, 96], but, to the best of my knowledge, this is the first publication in which the composition problem is treated applying ideas from partial-order planning. In particular, it is the first version of a partial order planner that is able to cope with operations that create new objects.

This chapter is organized analogous to Chapter 4 in three sections. First, Section 5.1 gives an introduction to the search structure and explains how PO solves the problem defined in the running example. Second, the formal definition of the search structure is found in Section 5.2. Finally, Section 5.3 provides a theoretic analysis that covers proofs of correctness and completeness of PO.

5.1 Intuition

5.1.1 Basic Idea

The idea of composition I describe in this chapter is to split as many decisions of the composition creation process apart, and, hence, to pursue a least-commitment strategy. In fact, the technique used here is basically partial-order planning. Hence, readers familiar with (partial-order) planning will find many parallels. However, there are also significant differences e.g. that operations have not only inputs but also outputs, the typing of parameters, the absence of threats, and the consideration of (non-scalar) costs as non-functional properties.

5.1.1.1 Partial Compositions

Partial Composition

Bindings

The basis of PO are *partial compositions*. A partial composition is a partially ordered multiset of operations where (some of) the inputs of every *operation instance* are connected to inputs or outputs of the query, of other operation instances, or domain constants. These data connections are called *bindings*. So a partial composition is partial in the two aspects *ordering* and *grounding* of operation instances. Hence, partial compositions are not compositions in the sense of Def. 7 (cf. Section 2.1.2).

Operation Instance

The difference between an operation instance and an operation invocation is that an instance is only one of potentially many *copies* of the operation, but there is no grounding of the inputs or outputs. Such a grounding is contained in an operation invocation. Operation

invocations can be seen as operation instances together with a binding for every parameter.

5.1.1.2 Flaws and Resolvers in Partial Compositions

Flaws A partial composition may exhibit *flaws*. Flaws are literals occurring in the preconditions of operation instances of the partial composition or in the postcondition of the query for which it is not clear (yet) how they are satisfied. Intuitively, a partial composition is a solution if and only if it has no flaws.

Resolvers We can eliminate the flaws of a partial composition using *resolvers*. Given a flaw, a resolver for it is a (possibly new) operation instance or the query together with a binding of variables of the literal, which are inputs of the respective operation instance or outputs of the query, to inputs or outputs of the resolving operation instance or query inputs. More precisely, we may want to do the following things in order to resolve a flaw:

1. define the resolver as an existing or new operation instance with the flaw literal in its postcondition together with a binding that connects parameters of the flawed literal with the parameters of the literal in the postcondition of the resolving operation instance;
2. define the resolver as the query precondition together with a binding of the variables of the flawed literal to inputs of the query.

So, step by step, we insert new operation instances and connect them among each other and with the outputs required by the query. Thereby, we implicitly resolve unsatisfied conditions and, if we use a new operation instances as a resolver, possibly create new flaws; the added flaws are the literals in the precondition of an added operation instance. This process continues until there are no more unsatisfied conditions, i.e. no more flaws.

The above resolvers reflect a strategy of *least commitment*. Least commitment means that we make as few decisions as possible at one point of time. For example, the total-order search described in the last chapter decides in every step not only the operation that is applied but also where the inputs come from and completely fixes the order of operation invocations. In contrast, the strategy discussed in this chapter defers these decisions as much as possible.

In fact, some of the decisions are not even made at all during the search process. For example, the order of operations is not an explicit decision. That is, there is no explicit decision to order one operation instance before another. In contrast, the order is implicitly fixed by the above decisions.

5.1.1.3 Partial Compositions with Clause Instances

Again, the automation of the composition process requires that we use explicit domain knowledge within the process (cf. Chapter 4). That is, an operation may have a precondition that cannot be satisfied by any postcondition, but it "follows" from the postcondition of one or several other operations. While a human simply knows that this "follows"-relation holds and arranges the operations correspondingly, automation needs an explicit symbolic representation of this knowledge and its application. In other words, we need to consider the clauses defined Ω during the composition process.

Internally, we can treat operations and clauses as the same concept. In fact, we can interpret clauses as sets of implications, which can be seen as operations without outputs. That is, a clause $\alpha_1 \vee \ldots \vee \alpha_n$ can be seen as a set of n operations without outputs where the

i-th operation has precondition $\bigwedge_{j \neq i} \neg \alpha_j$ and postcondition α_i. Intuitively, we then interpret the clause as a rule of the form $\left(\bigwedge_{j \neq i} \neg \alpha_j\right) \rightarrow \alpha_i$.

Actions

We can then generalize the concept of a partial composition to a multiset of *actions*. Every operation instance is an action, and every transformation of a clause as described above is also an action. The partial composition is then a partially ordered multiset of actions where an action can be either an operation instance or a certain representation of a clause instance.

5.1.2 Example Run

Before explaining more details on the approach, let us see how this technique works in the case of the bookshop example (cf. Section 1.3). Figure 5.1 shows an exemplary excerpt of the search graph. We start with an empty partial composition, which is flawed because the outputs of the literals of the query postcondition are not satisfied.

The figure reads as follows. The left part illustrates the considered path in the search graph, which is only one of many possible paths. Each node is associated with a partial composition and, hence, with a set of flaws, which are shown on the right. Each edge corresponds to the application of a resolver to one of the flaws. The resolver, i.e. the action together with the binding, is shown in the middle, and the flaw treated in the each node is printed in green.

Note that every action brings its own globally unique variables, which need to be unified through the bindings in order to resolve flaws. The variables that constitute outputs of the query are prefixed with a^* while query inputs are prefixed with a_0. The variables of the i-th new action added to the partial composition are prefixed with a_i. Then of course, the variables are globally unique and, in order to resolve flaws, the variables need to be unified. This is done with the bindings, which are shown besides the applied action.

The two non-type flaws of the query postcondition are resolved with clauses from the background knowledge. In Figure 5.1, actions corresponding to clauses have a blue background. The flaw $\mathit{hasBook}(a^*.s, a_0.a, s_0.t)$ can only be resolved with a rule, because the only operation that produces a literal with the predicate *hasBook* makes assertions about ISBNs only but not authors and title; hence we first exploit the knowledge that we can infer the availability of a book described by author and title from the availability of a book described by an isbn together with the knowledge that the isbn belongs to the combination of an author and title. Analogously, the flaw $\mathit{near}(a^*.s, a_0.p)$ can only be resolved with a rule, because there is no operation at all that has a literal with the predicate *near* in its postcondition.

In the next two steps, we use operation instances to resolve two of the flaws that were introduced by the previous step. First, resolving $\mathit{near}(a^*.s, a_0.p)$ is done with a new clause resolver with a precondition literal $\mathit{locatedIn}(a_2.p, a_2.c)$, which is resolved by a new instance of the operation getCity (cf. Section 1.3). Then, resolving $\mathit{hasBook}(a^*.s, a_0.a, s_0.t)$ is done with a new clause resolver with a precondition literal $\mathit{isISBNOf}(a_1.i, a_1.a, a_1.t)$. The fourth step resolves this flaw using a new instance of operation getISBN. In Figure 5.1, operation instances are highlighted by boxes with orange background.

The following three steps follow the same pattern. The flaw resolved in the fifth step is $\mathit{locatedIn}(a_2.s, a_2.c)$ using as a resolver a new clause application instance corresponding to the inheritance of the respective literal from a subset to every subset of it. Sixth, this subset knowledge $a_5.s' \subseteq a_5.s$ is produced with a new instance of operation filterByAvailability. Finally, the literal $\mathit{locatedIn}(a_5.s, a_5.c)$ is resolved using a new instance of operation getStores, which implicitly resolves the type literal $\mathit{Set}[\mathit{Store}](a_6.s)$.

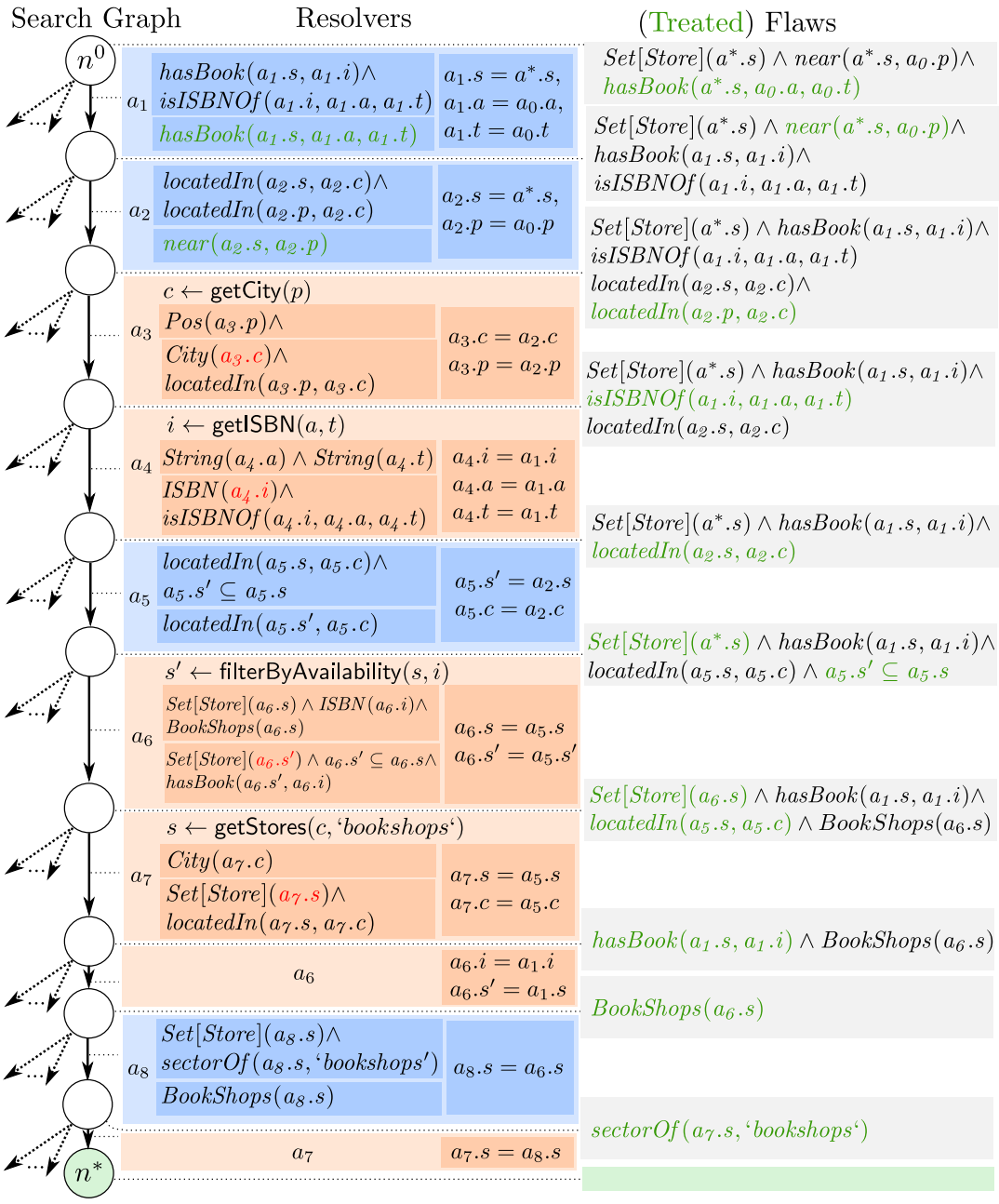


Figure 5.1: A solution path in G_{PO} for the bookshop example query.

In the last three steps, we use a new clause and two *existing* actions in order to resolve the remaining flaws. First, the flaw $hasBook(a_1.s, a_1.i)$ can be resolved using the above instance of filterByAvailability by simply adding additional bindings. Then, we resolve $BookShops(a_6.s)$ using a new clause action whose type flaw is automatically resolved. The remaining literal of the precondition of that clause can then be resolved by reusing the action a_7 of getStores.

Since the usage of an existing action does not add any flaw to the agenda, we obtain a partial composition that has no flaws.

The path shown in Figure 5.1 has nothing to do with the *ordering* of actions in the partial compositions. In particular, the order of operation invocations in the resulting composition does not correspond to their order of occurrence in the path. In general, we may order the occurring actions in an *arbitrary* way as long as every action a that is used as a resolver for a flaw of operation a' is ordered before a'. For example, the last clause action a_8 is actually ordered between the actions a_7 and a_6, because it infers a property required by a_6 from an output of a_7.

For this particular example, there are several possible compositions derivable from this plan. For example, getISBN may be the first operation invocation, but it may also come after getCity or even after getStores. So in contrast to BW, the path itself does not impose a total ordering of the operation invocations, but it induces a *set* of possible solutions.

5.1.3 A Look at the Details

Before going into the formal search structure definition, some conceptual aspects of the technique should be made clear. First, Section 5.1.3.1 explains how the query is encoded into partial compositions in form of dummy actions. Section 5.1.3.2 then gives a formal definition of partial compositions, which we need to define the search structure. Third, Section 5.1.3.3 explains the notion of causal links and the agenda, which compactly defines the set of flaws. Section 5.1.3.4 discusses how the flaw addressed in a node is selected. The last three subsections deal with special issues about how the flaws are resolved. Section 5.1.3.5 shows the relation between using existing or new actions as resolvers; Section 5.1.3.6 discusses the fact that type literals may (but need not) be resolved implicitly; and Section 5.1.3.7 explains why the IRP assumption allows us to leave threat treatment as used in classical partial order planning out of the model.

5.1.3.1 Encoding the Query and the Domain Facts in Dummy Actions

For simplicity, we encode the query and the domain facts as two dummy actions that simulate the beginning and the end of the composition. This technique is common practice in partial order planning, because it simplifies the check whether or not a partial composition is a solution to the query [96].

Init and Finish Actions

The init action provides the inputs and precondition of the query as well as the domain facts, and the finish action requires its outputs and postcondition. That is, a_0 is an action *without* inputs or precondition, and its outputs correspond to the query inputs X_q together with the domain constants Γ_{const}, and its postcondition corresponds to the query precondition Pre_q and the domain facts, e.g. clauses of Ω with size 1. Analogously, a^* is an action without outputs or postcondition, and its inputs correspond *parameters* in the query postcondition $Post_q$, and its precondition correspond to $Post_q$ itself. Naturally, every (other) action must be ordered after a_0 and before a^*.

This encoding has two advantages. First, the goal check is very simple, because we know that a partial composition is a solution to the query as soon as it has no more flaws. Second, we can now define resolvers entirely in terms of actions. A resolver is then an action (operation instance, clause instance, or init action a_0) together with a binding that maps the (still unbound) variables of the flaw literal to the inputs and outputs of the resolving action.

The a^* action seems to require a special treatment in that we interpret the query inputs that occur in the query postcondition as already bound. Usually, the precondition of an action refers only to the inputs of the corresponding operation or clause. However, the inputs and precondition of a^* refer to the outputs and postcondition of the query respectively. But the postconditions of the query usually also contain the inputs of the query, so a^* is a distinguished action in the sense that its preconditions contains variables that are not inputs of the action.

We can avoid this special role of a^* by using an *initial binding* that ties the inputs occurring in the postcondition to the params in a_0. We will allow only one output parameter in each equivalence class induced by the bindings, so these parameters cannot be bound to other outputs later.

5.1.3.2 Formal Definition of Partial Compositions

Given the above intuitive definition of a partial composition, we now define it formally. The symbol $\rightsquigarrow$ denotes a partial function, i.e. a function that does not necessarily define a value for each element in the domain.

Definition 17. *Let $\langle \mathcal{T}, \Omega, \mathcal{N} \rangle$ be a composition domain and O be a set of operations, and let q be a query. A **partial composition** is a triplet $\langle AS, OR, BI \rangle$ where*

- *$AS = \{a_0, a^*\} \cup f$ where $f : \mathbb{N} \rightsquigarrow O \cup \Omega$ is a finite set of numbered operations or clauses,*
- *the inputs, outputs, precondition, and postcondition of each action $a \in AS$ are as follows:*
  1. *for the dummy action a_0, we have $X_{a_0} = \emptyset$, $Y_{a_0} = \{a_0.y \mid y \in X_q\}$, $Pre_{a_0} = \emptyset$, and $Post_{a_0} = Pre_q[X_q/a_0.X_q]$;*
  2. *for the dummy action a^*, we have $X_{a^*} = \{a^*.x \mid x \in vars(Post_q)\}$, $Y_{a^*} = \emptyset$, $Pre_{a^*} = Post_q[vars(Post_q)/a^*.vars(Post_q)]$, and $Post_{a^*} = \emptyset$*
  3. *if $a = (\cdot, o)$ with $o \in O$, i.e. a corresponds to an operation, then we set $X_a = \{a.x \mid x \in X_o\}$, $Y_a = \{a.y \mid y \in Y_o\}$, $Pre_a = Pre_o[X_o/a.X_o]$, and $Post_a = Post_o[X_o/a.X_o, Y_o/a.Y_o]$*
  4. *if $a = (\cdot, \bigwedge_{j \neq i} \neg\alpha_j \rightarrow \alpha_i)$ with $\alpha \in \Omega$, i.e. a corresponds to an arranged clause, then $X_a = \{a.x \mid x \in vars(\alpha)\}$, $Y_a = \emptyset$, $Pre_a = \bigwedge_{j \neq i} \neg\alpha_j[vars(\alpha)/a.vars(\alpha)]$, and $Post_a = \alpha_i[vars(\alpha)/a.vars(\alpha)]$*
- *$invars(AS) = \bigcup_{a \in AS} X_a$ and $outvars(AS) = \bigcup_{a \in AS} Y_a$,*
- *$OR \subseteq \{a < a' \mid a, a' \in AS\}$ are ordering constraints for AS, and*
- *$BI \subset (invars(AS) \cup outvars(AS) \cup \Gamma_{const})^2$ is a (partially defined) equivalence relation.*

*A partial composition is **consistent** if*

1. *OR can be completed to a transitive and asymmetric relation,*
2. *$a_0 < a$ is consistent with OR for every $a \in AS$ other than a_0, and $a < a^*$ is consistent with OR for every $a \in AS$ other than a^*,*
3. *for actions a and a' with input x and output y respectively, if $a.x$ and $a'.y$ are in the same class of BI, then $a' < a$ must be consistent with OR,*
4. *each class in BI contains at most one item of $outvars(AS) \cup \Gamma_{const}$,*

5. *for each equivalence class defined by BI, there exists a common subtype for all the type definitions of items in the class according to $\mathcal{T}$, and*

6. *optionally, we may require that each equivalence class defined by BI contains at most one input per action.*

Remarks.

- The first and the second constraint are common in partial-order planning expressing the consistency of the ordering; the others are special for our setting.
- The third constraint deals with the fact that operations make objects come into existence. Obviously, an operation or clause may only use objects as inputs that have been produced before, so the producing operation must be an predecessor.
- The fourth constraint ensures that we know *how* an object is created. That is, the (input) parameters of an operation or clause must be clear, which means that there is at most one (and in a solution exactly one) output bound to such a parameter in the bindings BI. The condition also makes sure that we do not rewrite domain constants.
- The fifth condition makes sure that the data flow is consistent with respect to type compatibility. Note that, whenever an equivalence class induced by BI contains an output variable, the common subtype *is* exactly the type of that output. However, if such an output does not exist in a class, it may be that none of the elements in the class actually has a type that is a subtype of all the others.
- Finally, the, sixth and optional condition enables a kind of unique name assumption. This may be useful if we have a setting in which we say that we do not use the same object for two different inputs of the same operation (or clause).
- Classical partial-order planning requires that the bindings BI are consistent, which is rooted in the fact that bindings also define *forbidden* equalizations of parameters. This is necessary if states may contain negations, which we ignore in this thesis; hence, the consistency of bindings is limited to satisfying the previous constraint.

5.1.3.3 Causal Links and the Agenda

During the search process, it is helpful to keep track of flaws that have already been resolved and *how* they were resolved. So we need the list of remaining flaws for each partial composition in order to determine what is still to be done.

This bookkeeping can be easily achieved by equipping every partial composition with *causal links* [83]. Let $\langle AS, OR, BI\rangle$ be a partial composition. A causal link is a triplet of the form $\langle a', L, a\rangle$ where $a, a' \in AS$ are actions of the partial composition and $L = p(v_1, .., v_m)$ is a literal of the precondition of a and $L' = p(v'_1, .., v'_m)$ is in the postcondition of a'. The literals L and L' can be unified by $v_1 = v'_1, .., v_m = v'_m$. A causal link $\langle a', L, a\rangle$ means that the literal L necessary for action a is established by action a' under the binding $v_1 = v'_1, .., v_m = v'_m$. Hence, the set of causal links can be understood as an explanation of how preconditions of actions are satisfied within the partial composition. *Causal Links*

A causal link induces a canonical binding. For a causal link $\langle a', L, a\rangle$, this binding σ is the function $\sigma(v_1) = v'_1, .., \sigma(v_m) = v'_m$. The pairs $(v_1, v'_1), .., (v_m, v'_m)$ are stored explicitly in BI; other pairs contained in BI due to symmetry and transitivity can be checked on demand but are not computed explicitly.

Agenda

The set of flaws for which no causal links are defined is called the *agenda*. Let $\langle AS, OR, BI\rangle$ be a partial composition and CL be a set of causal links defined for this composition. Then, the agenda for this composition under these links is defined as

$$AG = \{\langle L, a\rangle \mid a \in AS \wedge L \in Pre_a \wedge \neg\exists a' : \langle a', L, a\rangle \in CL\}$$

Note that, in this model, the causal links and, hence, the agenda are not a part of the partial composition itself. In each node, we have both a partial composition and a set of causal links, which then induce an agenda.

5.1.3.4 Selecting Flaws and Choosing Resolvers

In AI literature, the terms *select* and *choose* are sometimes used to distinguish deterministic and non-deterministic decisions on an option pool respectively. Selecting an option means that all the options must be considered eventually, such that there is no need to revoke such a select-decision. On the contrary, choosing an option means to make a decision for an option that potentially needs to be revoked in order to find a solution.

In partial order search, flaws correspond to select-options, and the resolvers are choice-options. Eventually, every flaw needs to be resolved, so, from the viewpoint of completeness, it does not matter which of the flaws we treat first; of course, the efficiency consequences can be tremendous. On the other hand, choosing a particular resolver may or may not yield a solution. It is possible that we need to revoke a resolver decision at a later point of time if we find that it does not yield a solution; hence, resolvers are choice-options.

The consequence for the search graph is that the successors of a node should be the different choice-options for *one* select-option. That is, instead of having one successor for each resolver of each flaw, we select a distinguished flaw treated in a node, and the successors are only the resolvers of that flaw.

The strategy I pursue in this thesis is to treat non-type literals first, since these are often implicitly resolved together with other resolvents. That is, if the set of non-type flaws is not empty, one of these is selected by some function SelectFlaw. If the flaw list contains only type flaws, one of them is selected by SelectFlaw. In the implementation used for the experimental evaluation, SelectFlaw randomly takes one element of the pool where all have the same probability. However, in the theoretic part, we simply think of SelectFlaw as a black box function of which we do not know how it selects the flaws.

Note that the black box view on the selector function implies that the search graph is not necessarily deterministic in the query input. For example, a randomized implementation of SelectFlaw means that, for the same set of flaws, a different one may be selected in different runs. In particular, running the algorithm twice for the same input yields different search graphs. As a consequence, I will not speak of *the* graph G_{PO} for a query, but one *instance* of G_{PO} , which is rather seen as a production function of these graphs.

Search Graph Instances

In BW we did not make this difference between selecting or choosing open literals. The reason is that, in BW, the literal to be resolved next actually *is* a choice point; choosing the wrong literal here may lead into a dead end due to the total ordering, which is not the case in PO. As a consequence, the least commitment strategy pursued here implies a significantly

smaller output degree of the nodes in the search graph.

5.1.3.5 New Action vs. Existing Action

When resolving a flaw, we may be in the position to do this by adding a new action or by using an already existing action. For example, suppose that a partial composition contains an action a with precondition $hasBook(s, i)$ and an action a' with postcondition $hasBook(s, i)$. Then we can decide to use a' as a resolver for this literal, and add the respective causal link $\langle a', hasBook(s, i), a\rangle$. However, we could also insert a *new copy* of the operation or clause of which a' is an instance. Suppose that a' is an instance of operation o. Then we could just create a new instance of o, say a'', and establish the causal link between this new action and a, i.e. $\langle a'', hasBook(s, i), a\rangle$.

More precisely, adding new actions is *always* a possible option. That is, if a flaw can be resolved at all, it is always possible to do this through a new action. In particular, if the flaw can be resolved by an existing action a', it can also be resolved by a new copy of the same operation or clause belonging to a'.

Of course, using resolvers with new actions usually increases the agenda. This is because, at the time of insertion, every literal in the preconditions of the new action becomes a new flaw; none of them is resolved. In contrast, if a flaw is resolved with an existing action, then this action may have already some or even all of its precondition literals been resolved. But even if none of the literals were resolved before, then still the agenda does not increase.

In the following, I will use a function NewActions to refer to a routine that creates the set of new actions of a node. Intuitively, for any node n be of the search graph, NewActions(n) is simply the set of *all* operations and clauses prefixed with the depth d of n, i.e. its distance to n^0_{PO}. In particular, NewActions(n) also contains actions that cannot be used as a resolver for the flaw addressed in n; it is simply the pool of candidate actions that *may* be used as resolvers. Formally,

$$\textsc{NewActions}(n) = \{(d(n), o) \mid o \in O\} \cup \{(d(n), \bigwedge_{j \neq i} \neg\alpha_j \rightarrow \alpha_i) \mid \alpha \in \Omega\}$$

defines the set of new actions that can be considered as resolvers for flaws in the partial composition of the node in addition to the actions that are already there. By prefixing the operations and clauses with the depth of n, we make sure that every action has an id that is unique in the path to n, and, hence, unique in the partial composition belonging to n.

The above definition of NewActions implies that the search graph becomes a tree. In every node n, we address a particular flaw $\langle L, a\rangle$ and resolve it with a particular action. Suppose that n'_1 and n'_2 are successors of n. Then they treat the same flaw using different resolvers; the resolvers differ by the action, the bindings, or both. But then there is no node n'_3 reachable from both n'_1 and n'_2. If, w.l.o.g., n'_3 is reachable from n'_1. Since the way how a flaw is addressed cannot be changed over a path, the partial composition of n'_3 resolves $\langle L, a\rangle$ in the same way as the partial composition of n'_1 and, at the same time, in a different way that n'_2. In particular, n'_3 is not reachable from n'_2, so the search graph of PO is a tree.

5.1.3.6 Implicitly Resolving Type Literals

There is generally only one flaw, i.e. one literal, explicitly resolved per step, but several type literals can (and must) be resolved implicitly in it. Suppose that L is a literal in the

precondition of an action a for which we have found a resolver with action a' and the canonical binding σ (see above). The binding σ may induce a relationship between inputs of a and inputs or outputs of a'. But when we bind an input x of a to an output y' of a', we resolve the type flaw of x even though we did not address it explicitly. The precondition of a contains a type literal $t(x)$, and the postcondition of a' contains a type literal $t'(y')$. Binding x to y' when resolving some (non-type)-flaw means that we resolve $t(x)$ using $t'(y')$ together with the type heterarchy. This aspect is not considered in classical partial order planning [22, 83, 96].

What is more, whenever we use a resolver where the parameters of the flaw are bound to outputs of the resolving action, *all* type literals associated with these parameters become also resolved. This is true not only for the type literal in the precondition of the flawed action but also for every other action with inputs that are in the same equivalence class in BI. For example, we may use action a' with output y' to resolve a flaw of action a with parameter x, and x was previously bound to some other parameter x'' of another action a''. Then there may be a type definition $t(x)$ in the precondition of a and a definition $t''(x'')$ in the precondition of a''. We now resolve both $t(x)$ *and* $t''(x'')$ at a time.

In order to obtain a consistent partial composition, we must make sure that the type of the output is a subtype not only of the input type of the action whose flaw we address but of all parameters that are bound to it. Suppose that the canonical binding of the resolving step defines $\sigma(x) = y$ where x is an input of the action whose flaw is addressed and y is an output of the resolving action with $t(y)$ in the postcondition. Moreover, suppose that $t_1(a_1.x_1), .., t_m(a_m.x_m)$ are the types of variables that are in the the equivalence class of x according to BI. Then t must be a subtype of all the types $t_1, .., t_m$ in $\mathcal{T}$. If this is the case, then, by construction, all variables in the equivalence class $\{y, a_1.x_1, .., a_m.x_m\}$ have t as a common subtype, and the partial composition is consistent with respect to this criterion.

5.1.3.7 IRP Avoids Threats and the Necessity of Demotion or Promotion

In traditional partial order planning, inserting a causal link, e.g. by adding new actions, may cause threats. This means that a resolver undoes the achievement of other resolvers in the sense that the literal is the negation of the literal covered by another link. Threats are resolved themselves by promotion or demotion, which forces an ordering of the threatening action before or after the other actions involved in this threat relation respectively.

However, threats cannot occur under the IRP assumption, so we do not need to treat them. The reason is that we assume that nothing that we knew once to be true can become false within one composition. But then it is not possible that the same literal occurs once positively and once negatively in the postconditions of an operation *invocation* (or clause applications). So there cannot be two actions with opposite literals in their postconditions if the variables are bound to the same data containers or constants; if such a construction was possible, the IRP assumption would be violated. Consequently, we do not need to check whether a causal link is threatened by an action.

5.2 Search Structure

I now define the search structure elements based on a composition problem instance. That is, I define it point-wise for each composition problem instance $\langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$ as defined in Section 2.1.4. The following definitions rely on the elements of this instance.

The organization is as follows. I first describe the search structure in Section 5.2.1 (search

graph G_{PO}), Section 5.2.3 (goal node function $\star_{\text{PO}}$), and Section 5.2.2 (translation function TRANS_{PO}). Second, Section 5.2.4 explains the implementation for the exploration strategies e_{fast} and e_{nf} (cf. Section 3.3). I have not defined an SR-dominance relation for PO, so $n \succeq_{\text{PO}} n'$ is assumed to be $false$ for any pair of nodes.

5.2.1 The Search Graph G_{PO}

I define the search graph of PO based on the above explanations. Every node n is associated with a partial composition $c(n)$ and a set of causal links $CL(n)$ inducing an agenda $AG(n)$.

Definition 18. *The **search graph** G_{PO} is defined inductively in terms of* GETROOT_{PO} *and* $\text{GETSUCCESSORS}_{PO}$.

$n^0_{PO} = \text{GETROOT}_{PO}$ *is defined as follows:*

- $c(n_0) = \langle \{a_0, a^*\}, \{a_0 < a^*\}, \{(a_0.x, a^*.x) \mid x \in X_q \cap vars(Post_q)\}\rangle$ *and*
- $CL(n_0) = \emptyset$

a_0 *and* a^* *are the* names *of the initial and final action discussed above respectively.*

Now we define $\text{GETSUCCESSORS}_{PO}$. *Let* $n \in N_{PO}$ *be a node in this graph, let* $c(n) = \langle AS(n), OR(n), BI(n)\rangle$ *be the partial composition,* $CL(n)$ *be the set of causal links, and* $AG(n)$ *be the agenda associated with* n *respectively. Moreover, let* $\langle p(v^a_1, .., v^a_m), a\rangle = \text{SELECTFLAW}(n)$ *be the selected flaw from* $AG(n)$.

Then $n' \in \text{GETSUCCESSORS}_{PO}(n)$ *iff there is an action* $a' \in AS(n) \cup \text{NEWACTIONS}(n)$ *such that*

1. *the partial composition* $c(n') = \langle AS(n'), OR(n'), BI(n')\rangle$ *is consistent where*
   - $AS(n') = \begin{cases} AS(n) & \text{if } a' \in AS(n) \\ AS(n) \cup \{a'\} & \text{else; i.e. if } a' \in \text{NEWACTIONS}(n) \end{cases}$
   - $OR(n') = OR(n) \cup \{a_0 < a', a' < a^*, a' < a\}$, *and*
   - $BI(n') = BI(n) \cup \{(v^a_i, v^{a'}_i) \mid 1 \leq i \leq m\}$; *and*
2. *there is a literal* $p'(v^{a'}_1, .., v^{a'}_m) \in Post_{a'}$ *such that* $p'(v^{a'}_1, .., v^{a'}_m) \wedge \mathcal{T} \wedge BI(n) \models p(v^a_1, .., v^a_m)$.

The causal links for n' *are defined as follows:*

- *every link of* $CL(n)$ *is in* $CL(n')$,
- $\langle a', p, a\rangle$ *is in* $CL(n')$,
- *for every action* $a'' \in AS(n)$ *with an input* x *that is bound to an output of* a' *in* $BI(n')$, *the link* $\langle a', t(x), a''\rangle$ *is in* $CL(n')$ *where with* $t(x)$ *is the type of* x *in* $Pre_{a''}$.

The edges in this graph are not labeled, because the nodes already encode everything necessary to recover the composition. However, in the implementation, I do not store the whole partial composition of every node n but only the *augment*, i.e. the causal links belonging to the edge. It contains the new action (if there is any) and implies both the ordering constraint and the binding BI. This corresponds to a kind of edge labeling.

In contrast to BW, we can encode the non-functional properties based on what is stored in the node itself. Let $o_1, .., o_m$ be the operations for which there exists an action in $c(n)$,

and let $u_1, .., u_m$ the *number* of such actions, i.e. the number of actions being an instance of operation o_j. Then we define

$$\odot(n)_i = \underbrace{(Z_{o_1})_i \oplus_i \ldots \oplus_i (Z_{o_1})_i}_{u_1 \text{ times}} \oplus_i \ldots \oplus_i \underbrace{(Z_{o_m})_i \oplus_i \ldots \oplus_i (Z_{o_m})_i}_{i_m \text{ times}}$$

where we compute each property i separately due to the possibly diverging aggregation functions. Recall from Section 2.1 that $\oplus_i$ is the sequential aggregation function of property i, and $(Z_o)_i$ is the value of property i of operation o. Since $\odot(c)$ is the sum (w.r.t $\oplus_i$ of the different properties) of the operation invocations in c and since the aggregation functions are commutative, $\odot(n) = \odot(c(n))$ holds even if we have not fixed the total order of the composition.

5.2.2 The Transformation Function TRANS$_{\text{PO}}$

In contrast to the total-ordered backward search, we cannot derive a complete composition from every node. The problem is that some inputs are unknown, i.e. not every operation instance can be ground to an operation invocation. In other words, the partial composition associated with a node can be converted into a "real" composition iff every equivalence class of the bindings contain exactly one domain constant or operation output.

Alg. 3 shows the top level view of the transformation algorithm.

Algorithm 3: TRANS$_{\text{PO}}$ (p)

Inputs : Node n with partial composition $\langle AS(n), BI(n), OR(n)\rangle$
Output: Composition $c(n)$

```
1 map ← getParameterMap(⟨AS(n), BI(n), OR(n)⟩);
2 comps ← ∅;
3 foreach comp ← getTopOrderings(AS(n), OR(n)) do
4 |   comps ← comps ∪ {createStateMachine(comp, map)};
5 end
6 return comps;
```

It consists of the following three steps:

1. Create a data flow based on data containers. This is what is done in the subroutine Alg. 4. For every output of an action contained in the partial composition, a new data container is introduced with a corresponding output mapping for the action. This only affects the initial action a_0 and the actions that correspond to operation invocations; others have no outputs and are ignored. Then every input variable is ground to the data container corresponding to the output with which it is in the same equivalence class according to BI. There is exactly one output (or domain constant), and this was bound to a data container in the first substep; so this grounding is well-defined.

Serialized Partial Composition

2. Compute all possible serializations of the partial composition. In the following, I will denote such *serialized partial compositions* with causal links CL as $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$.

3. Finally, a state machine is derived for every serialization. Intuitively, every operation action of the serialized partial composition induces one *state* in the composi-

Algorithm 4: getParameterMap

Inputs : Partial composition $\langle AS, BI, OR \rangle$
Output: Parameter Map

```
1  map ← new Table();
2  i ← 0;
3  foreach a ∈ AS do
4  |   foreach y ∈ Y_a do
5  |   |   map[y] ← v_i;
6  |   |   i++;
7  |   end
8  end
9  foreach a ∈ AS do
10 |   foreach x ∈ X_a do
11 |   |   map[x] ← getSupplier(x, BI, outmap);
12 |   end
13 end
14 return map;
```

tion. So, suppose the serialization is $\langle a_0, a_1, .., a_l \rangle|_{BI,CL}$. Then there will be a subsequence $\langle a'_1, .., a'_n \rangle$ where $\{a'_1, .., a'_n\} \subseteq \{a_1, .., a_l\}$, and each a'_i corresponds to an operation invocation $\kappa_{BI}(a'_i)$. Then the (compact notation of the) sequential composition is $\langle \kappa_{BI}(a'_1), .., \kappa_{BI}(a'_m) \rangle$. Note that κ is an injective function, so $\kappa^{-1}_{BI}(o_i[\sigma_i])$ is the action belonging to operation invocation $o_i[\sigma_i]$ of a composition produced this way.

5.2.3 Goal Function $\star_{\mathsf{PO}}$

Intuitively, the goal check for PO comes down to check the emptiness of the set of flaws. Recalling Def. 15 (cf. Section 2.1.4), a composition c is a solution for query q iff c transforms Pre_q into $Post_q$ and if the non-functional requirements are satisfied ($\odot(c) \leq Z_q$). It is not too hard to see that a partial composition transforms Pre_q into $Post_q$ iff the agenda is empty. The additional check on the bound of non-functional properties assures that we do not mark compositions as solutions that violate those bounds; however, in monotone setups, such nodes can be pruned such that this check is obsolete.

Formally, this amounts to the following definition:

$$\star_{\mathrm{PO}}(n) = \begin{cases} true & \text{if } \odot(n) \leq Z_q \text{ and if } AG(n) = \emptyset \\ false & else \end{cases}$$

Section 5.3.1 shows the proof that this relation is in fact sound.

5.2.4 Implementation of Exploration Strategies

I now describe the implementation of the exploration strategies e_{nf} and e_{fast} (cf. Section 3.3). Once again, an implementation of e_{rating} is not considered, but a sketch can be found in [66].

Like in BW, the history g for e_{nf} corresponds to the non-functional properties stored for the node at the end of the path. Hence, $g((n^0_{\mathrm{PO}}, .., n)) = \odot(n)$ for e_{nf}. For e_{fast}, we already defined $g((n^0_{\mathrm{PO}}, .., n)) = |(n^0_{\mathrm{PO}}, .., n)|$ in Section 3.3.2.

Also similarly to BW, the heuristics of e_{nf} and e_{fast} rely on a *relaxation* of the rest problem. The rest problem is relaxed to a simple set theoretic planning problem. The solution to the relaxed problem is a sequence of operations (instead of operation invocations) and propositional clauses (instead of ground first order clauses).

We compute the relaxed rest problem of a node n as follows: We use the function *prop*, which removes the the literal parameters from a given formula; that is, $prop(\alpha)$ is α where every predicate $P(X)$ is replaced by P (negations and junctions are not touched). Using this function, we perform the following steps:

1. In a preprocessing step, we compile a set of actions. This set contains one action for each relaxed operation, i.e. for each $o \in O$, there is an action with $prop(Pre_o)$ as precondition, $prop(Post_o)$ as postcondition, and Z_o as values of the non-functional properties. Also, there is one action for each clause of Ω with $\bar{L}_1 \wedge .. \wedge \bar{L}_m$ as precondition where $L_1(X_1), .., L_m(X_m)$ are the negative literals of the clause, and L as postcondition where $L(X)$ is the positive literal of the clause. The non-functional properties are 0.

2. Now, compute propositional groundings $c(n)^p$ and $CL(n)^p$ by applying *prop* to every precondition and postcondition of any action in $c(n)$ and any literal in a link of $CL(n)$;

3. Compute the conjunction F as the propositional grounding of the part of the agenda induced by $c(n)$ and $CL(n)$ that needs to insert new actions. So, compute the proposition set $F = \bigcup_{\exists \langle L,a \rangle \in AG(n)} prop(L) \setminus \bigcup_{\exists a \in AS(n): L \in Post_a} prop(L)$. This encoding is highly optimistic in the sense that it assumes that every literal of the agenda that is in one of the postconditions of the existing actions can be resolved with existing actions; given this assumption, these flaws do not even need to be considered in the relaxed problem. While this assumption usually does not hold due to the consistency constraints, this type of optimism is important to avoid that we consider a rest problem unsolvable even though it is not. The problem is that the relaxed problem does not contain information about possible reuse of existing actions in the plan, so it may unnecessarily introduce new actions, which may violate the cost bound even though this could be avoided by reusing an existing plan action. In order to avoid this trouble, we consider the other extreme and assume that actions are reused for all literals of the agenda where applicable.

4. Then, define a (cost-based) set theoretic planning problem with initial state $prop(Pre_q)$, goal state F, actions A, and cost bound Z where $Z_i = (Z_q)_i \ominus_i \odot(n)_i$ for every non-functional property i.

The solution to the relaxed problem is a sequence of operations (instead of operation invocations) and propositional clauses (instead of ground first order clauses).

The heuristic h differs between the two strategies. In e_{nf}, we set $h(n)$ to the *properties* of the solution to the relaxed problem; hence, we applied a simplified version of $\odot$. In e_{fast}, we set $h(n)$ to the *length* of the solution path (including edges for clauses). In the case of e_{fast}, the concrete value for $h(n)$ depends on the algorithm used to solve the relaxed problem.

5.3 Theoretical Analysis

This section presents key results on the theoretic properties *correctness* and *completeness* of running the search algorithm using PO as search structure. Section 5.3.1 shows that $\textsc{Search}_{\text{PO},\mathcal{E},\mathcal{P}_{\succeq \text{PO}}}$ is a correct algorithm for sequential composition, and Section 5.3.2 shows

that it is complete for the exploration strategy e_{fast}. Again, the detailed versions of sketched proofs can be found in the appendix.

5.3.1 Correctness of $\text{SEARCH}_{\text{PO},\mathcal{E},\mathcal{P}_{\trianglerighteq\text{PO}}}$

I show the correctness of $\text{SEARCH}_{\text{PO},\mathcal{E},\mathcal{P}_{\trianglerighteq\text{PO}}}$ in two steps. First, for a given serialization of a supposed solution, I describe the construction of a labeling of actions in the plan that describes the knowledge after each step. Second, I describe how we can derive the correctness of PO from this, which directly implies the correctness of the algorithm when run with PO.

5.3.1.1 Labeled Serialized Partial Compositions

Given a serialized partial composition (cf. Section 5.2.2), we can define the *labeling* of it as follows: Let $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$ be a serialized partial composition and let ψ be the mapping obtained by $\text{GETPARAMETERMAP}(\langle AS, OR, BI\rangle)$. Then

$$\tilde{\lambda}(a_i) = \begin{cases} Post_{a_0}[\psi] & \text{if } i = 0 \\ \tilde{\lambda}(a_{i-1}) \cup Post_{a_i}[\psi] & \text{if } 1 \leq i < l \\ \textit{undefined} & \text{else} \end{cases}$$

The labeling $\tilde{\lambda}$ imposes a straight forward semantic that says what is true after the application of an action (i.e. operation or clause instance). Note that $a_l = a^*$ has no label, but all other actions have exactly one.

Now, we convince ourselves that this labeling makes sure that the precondition of action a_i is true after having executed $a_0, .., a_{i-1}$.

Lemma 5.1. *Let G be an instance of G_{PO} for a query, n be a node in G with an empty agenda, and let $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$ be a serialized partial composition of $c(n)$. Then*

1. *if $\langle a_i, L, a_j\rangle \in CL(n)$, then $L[\psi] \in \tilde{\lambda}(a_i)$, and*
2. *$\tilde{\lambda}(a_{i-1}) \models Pre_{a_i}[\psi]$ holds for every i with $1 \leq i \leq l$.*

Proof. I prove the two claims separately:

1. Let $\langle a_i, L, a_j\rangle \in CL(n)$ and $L = p(v_1, .., v_m)$. Then there are a literal $L' = p(v'_1, .., v'_m) \in Post_{a_i}$ and $(v_i, v'_i) \in BI(n)$ for $1 \leq i \leq m$. Since ψ maps every two parameters v_i and v'_i to the same data container, we have that $L'[\psi] = L[\psi]$. By $L' \in Post_{a_i}$, we have that $L'[\psi] \in \tilde{\lambda}(a_i)$, which, by $L'[\psi] = L[\psi]$, means that $L[\psi] \in \tilde{\lambda}(a_i)$.
2. Let a_i be an action in the plan and $L \in Pre_{a_i}[\psi]$ be a literal in the mapped precondition of a_i such that $L = L'[\psi]$. The agenda is empty, so there is a causal link $\langle a_j, L', a_i\rangle$ such that $j < i$ and, by (1), $L'[\psi] \in \tilde{\lambda}(a_j)$. In particular, $j \leq i - 1$, which implies $\tilde{\lambda}(a_j) \subseteq \tilde{\lambda}(a_{i-1})$. Since $L = L'[\psi]$, and since $L'[\psi]$ is in $\tilde{\lambda}(a_j)$, L is in $\tilde{\lambda}(a_{i-1})$.

□

The next step is to show that PO makes correct inferences. More precisely, we can show that a sequence $a_i, .., a_j$ within a serialized composition $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$ establishes a modus ponens based proof for the assertion $\tilde{\lambda}(a_{i-1}) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_j)$.

Lemma 5.2. *Let G be an instance of G_{PO} for a query, n be a node in G with an empty agenda, and let $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$ be a serialized partial composition of $c(n)$ with $a_i, .., a_j$ being a sequence of clause actions in it. Then $\tilde{\lambda}(a_{i-1}) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_j)$.*

Proof Sketch. The proof is straight forward by induction over the length of the chain. □

These two Lemmas can now be used to prove the correctness of the search structure PO.

5.3.1.2 Correctness of PO and $\text{SEARCH}_{\text{PO},\mathcal{E},\mathcal{P}_{\succeq_{\text{PO}}}}$

The main Lemma for the correctness of PO consists of the proof that, for a given query q, a path that points to a node with an empty agenda is only associated with compositions that transforms Pre_q into $Post_q$. Based on this observation, the correctness then follows straight forward.

Lemma 5.3. *Let G be an instance of G_{PO} for query q, and let $p = (n^0_{PO}, .., n)$ be a path in G such that n has an empty agenda. Then every composition $c \in \text{TRANS}_{PO}(p)$ transforms Pre_q into $Post_q$.*

Proof Sketch. The idea is to define a state labeling λ for c based on $\tilde{\lambda}$ as a witness for this property and, hence, to show that λ is valid, that $Pre_q \models \lambda(s_0)$, and that $\lambda(s_m) \models Post_q$ where s_0 and s_m are the init state and final state of c respectively. λ can be defined straight forward by simply setting the label of state s_i to what is known after the i-th operation invocation action in c for $1 \leq i < l$, i.e. $\tilde{\lambda}(\kappa^{-1}_{BI}(o_i[\sigma_i]))$ where $\kappa^{-1}_{BI}(o_i[\sigma_i])$ is the action of the serialized partial composition corresponding to operation invocation $o_i[\sigma_i]$ as per TRANS_{PO}, and to set it to $\tilde{\lambda}(a_0)$ for $i = 0$ and $\tilde{\lambda}(a_l)$ for $i = m$. It is not hard to see that the previous two Lemmas imply the validity of λ and $\lambda(s_m) \models Post_q$. Since $Pre_q \models \lambda(s_0)$ holds by definition, there is nothing more to show. □

This Lemma then gives us directly the correctness of the search structure PO.

Theorem 5.4. *The search structure PO is correct.*

Proof. A search structure is correct if the following holds (cf. Section 3.1.3.1): Let q be a query, and $p = (n^0_{\text{PO}}, .., n)$ be a path in G_{PO} with $\star_{\text{PO}}(n) = true$. Then $\text{TRANS}_{\text{PO}}(n)$ is non-empty, and each of its elements is a solution to q. Clearly $\text{TRANS}_{\text{PO}}(p)$ is not empty if $\star_{\text{PO}}(n) = true$, so let c be a composition in $\text{TRANS}_{\text{PO}}(p)$. By Def. 15 (cf. Section 2.1.4), c is a solution iff it transforms Pre_q into $Post_q$, which is assured by the Lemma 5.3, and if $\odot(c) \leq Z_q$, which holds by definition of $\star_{\text{PO}}$. □

Again, the correctness of the search structure directly implies the correctness of the parametrized search algorithm $\text{SEARCH}_{\text{PO},\mathcal{E},\mathcal{P}_{\succeq_{\text{PO}}}}$.

Corollary 5.5. $\text{SEARCH}_{PO,\mathcal{E},\mathcal{P}_{\succeq_{PO}}}$ *is correct for any exploration strategy $\mathcal{E}$.*

Proof. The claim is a direct consequence from Theorem 3.1 (cf. Section 3.2.4.1) and the above Theorem 5.4. □

5.3.2 Completeness of $\text{SEARCH}_{\text{PO},\mathcal{E},\mathcal{P}_{\succeq_{\text{PO}}}}$

Given a strictly increasing exploration strategy $\mathcal{E}$ (cf. Section 3.2.4), two more conditions must be satisfied in order to ensure completeness of the search algorithm $\text{SEARCH}_{\text{PO},\mathcal{E},\mathcal{P}_{\succeq_{\text{PO}}}}$:

1. the search structure PO itself must be complete (cf. Section 3.1.3.2)
2. the SR-dominance relation $\succeq_{\text{PO}}$ must be completeness-preserving (cf. Section 3.2.3.2).

Due to our definition of $\succeq_{\text{PO}}$, the second condition is trivially true.

Showing the completeness of PO goes in three steps. First, I show that PO is complete with respect to queries for which solutions as pure resolution steps exist. That is, we can use PO as a theorem prover for definite Horn formulas. Second, I use this result to show that every minimal composition that solves a given query can be created using PO. Third, I combine this result with the definition of $\succeq_{\text{PO}}$ to show that PO is complete.

Lemma 5.6. *Let q be a query such that $Pre_q \wedge \Omega \wedge \mathcal{T} \models Post_q$ and let G be an arbitrary but fixed search graph instance of G_{PO} for a run on q. Then there is a path p from n^0 to a node n in G such that $AG(n) = \emptyset$ and $AS(n)$ does not contain any operation actions.*

Proof Sketch. The proof is similar to the one of Lemma 4.4 for BW (cf. Section 4.3.2), i.e. by induction over the number of side clauses from Ω used in an SLD resolution refutation. The main difference to the proof of Lemma 4.4 is that we do not know which flaw is chosen in which node and, hence, must be flexible with respect to the order in which the clauses are applied. The trick is to take a virtual solution (partial composition) obtained by a subquery equal to the original one except the postcondition, which corresponds to the resolvent of the first SLD resolution step; such a solution exists by the induction hypothesis. It is not hard to see that we can use that partial composition to show that a path to a node n with empty agenda must exist in G. This is because for the flaw of each node we may encounter on a path to n, the partial composition contains a causal link that explains how that flaw can be resolved also in the currently considered graph G. □

The direct consequence of this is that each query that can be satisfied with the empty composition has a corresponding node in the search graph.

Observation 5.7. *Let q be a query, and suppose that the empty composition $\langle\rangle$ is a solution to q. Then there is a path p from n^0 to a node n in G such that $\langle\rangle \in \text{TRANS}_{PO}(n)$.*

We can now use this result to show that, for each minimal solution c of a query q, there is some node that encodes c. This proof is inspired by the completeness proof for UCPOP [96].

Lemma 5.8. *Let q be a query and c be a composition that solves q minimally. Moreover, let G be an instance of a search graph of G_{PO} for q. Then G contains a path $p = (n^0_{PO}, .., n)$ such that $c \in \text{TRANS}_{PO}(p)$ and $\star_{PO}(n) = true$.*

Proof Sketch. The proof is by induction over all solutions of length k. For $k = 0$, the claim follows directly from Lemma 5.6. The core idea of the proof for $k > 0$ is similar to the one use in Lemma 5.6. Given a solution c to query q with a first operation invocation $o[\sigma]$, we define one subquery for the problem to get from Pre_q to $Pre_o[\sigma]$, and a second subquery for the problem to get from $Pre_q \cup Post_o[\sigma]$ to $Post_q$. The solutions for both subqueries are smaller than k and, hence, are known to exist and to be returned due to the induction hypothesis.

Now, starting from n^0, we can walk along a path defined by particular resolvers chosen for the flaw addressed in a node. The chosen resolver is the one defined in the causal link for the respective flaw in one of the two subsolutions; there can be no flaws for which none of the subsolutions has a recipe. In particular, we use exactly the operation instances used for the second subquery and $o[\sigma]$. Since no flaw is resolved twice, we obtain a node n with empty agenda after a finite number of steps. □

Together with the definition of $\succeq_{\mathrm{PO}}$, this Lemma directly yields the completeness of PO.

Theorem 5.9. *The search structure PO is complete.*

Proof. This is directly implied by Lemma 5.8, because it entails completeness for the case that only one solution is required (cf. Section 3.1.3.2). □

We can then assert the completeness for $\textsc{Search}_{\mathrm{PO},\mathcal{E},\mathcal{P}_{\succeq_{\mathrm{PO}}}}$ for any strictly increasing exploration strategy $\mathcal{E}$. In particular, completeness holds for e_{fast}.

Corollary 5.10. $\textsc{Search}_{PO,e_{fast},\mathcal{P}_{\succeq_{PO}}}$ *is complete.*

Proof. By Theorem 3.8 (cf. Section 3.2.4.2), we know that $\textsc{Search}_{\mathrm{PO},e_{fast},\mathcal{P}_{\succeq_{\mathrm{PO}}}}$ is complete if PO is complete, if e_{fast} is strictly increasing and if $\succeq_{\mathrm{PO}}$ is completeness-preserving. Completeness of PO is the result of the previous Theorem 5.9. That e_{fast} is strictly increasing was concluded in Corollary 3.10 (cf. Section 3.3.2.2). Finally, $n \succeq_{\mathrm{PO}} n' = false$, i.e. it is trivially completeness preserving as defined in Section 3.2.3.2. □

Also here, since e_{nf} is not strictly increasing, we can not make assertions about the completeness of $\textsc{Search}_{\mathrm{PO},e_{nf},\mathcal{P}_{\succeq_{\mathrm{PO}}}}$.

6. Searching Non-Sequential Compositions

The previously described mechanisms are able to find sequential compositions only. Of course, virtually no real program consists only of sequences of operation invocations but also contains conditional statements or loops. This chapter proposes techniques to overcome this limitation and to create also non-sequential compositions. However, taking into account non-sequential control structures imposes another drastic increase in complexity, and covering this topic in an exhaustive fashion is beyond the scope of this thesis. Hence, the approaches presented in this chapter, even though they constitute significant progress, are by no means final but should be considered initial work.

The chapter is organized in two sections. First, Section 6.1 presents a technique to create compositions with alternative branches. The presented technique is actually not a novelty itself but rather an integration of McDermott's idea [84] into my composition framework. Second, Section 6.2 shows how we can create compositions with simple but useful loops. The presented technique summarizes my previous work on template-based composition [89,92,121] and shows how it can be integrated into the composition framework into order to obtain compositions with loops.

6.1 Finding Compositions with Alternative Branches

Creating compositions with alternative branches means to create compositions that are able to test properties of data containers at runtime and to react on the results of these tests. The good news is that we can reuse the techniques for sequential compositions and only need moderate adjustments in order to find (a particular type of) compositions with branches. The bad news is, however, that composition with branches only makes sense for non-positive composition problems for which the sequential algorithms are not complete.

In Section 6.1.1, I describe semi-formally what I understand by compositions with branches. Second, Section 6.1.2 describes an algorithm to create that type of compositions (cf. Section 2.1.4). Finally, Section 6.1.3 explains the relation of composition with branches to non-positive domains and how to cope with such a case.

6.1.1 An Intuition for Compositions with Branches

Under alternative branches in a composition, I understand the following:

Consider a a state s of a composition with guards, say θ and $\neg\theta$; if such a state does not exist, the composition is sequential. Now the *branch* of θ is the subcomposition induced by all states that are not reachable from any initial state *without* passing $\delta(s, \theta)$. In other words, the branch of θ is the subcomposition for which θ is a necessary condition to be reached.

What we require here is that, if there is a state in the branch from which we can leave it, then that state must not be reachable *without* passing any of the branches departing from s.

Intuitively, this means that branches cannot induce cycles in the composition.

This definition is closely related to the viewpoint of other approaches on composition with branches such as [12, 63]. In fact, those approaches do not make explicit use of guards but of possible outcomes of a non-deterministic operations. If we see the exit signal of an operation as its (deterministic) output and then pose one guard for each of its possible values, we simulate this behavior. Note that the fact that we only allow two guards for each node is not a limitation since we can simply nest the condition. That is, if we want to check a variable $x \in \{1, 2, 3\}$, we can use $\theta_1 : x = 1$, and for the case of $\neg\theta_1$ use another guard $\theta_2 : x = 2$ departing from the state reached by $\neg\theta_2$, and so on.

In the following, we will focus on compositions with diverging branches. I say that a branch of a condition θ in a state s diverges if no state reachable from $\delta(s, \theta)$ can be reached from $\delta(s, \neg\theta)$. The term "diverging branch" is probably not a standard, but I am not aware of another term used in literature to describe this case.

For the moment, also suppose that we work with compositions that have no loops, i.e. without cycles in the transition graph. We will see later that loops as considered here cannot contain diverging branches anyway, but for the moment we should simply ignore loops at all.

An important property of such a diverging branch is that the conditions under which a state is reached is unique. In a composition where all branches diverge, the necessary and sufficient condition to reach a state s is the conjunction of all guards $\theta_1, .., \theta_m$ on the path from the initial state to s.

Clearly, focusing on diverging branches is not a limitation in the sense of completeness. Every composition where different branches have common reachable states can be "unfold" into a functionally equivalent composition with diverging branches only. So this divergence property only affects the compactness of representation but not the completeness or even the correctness.

6.1.2 An Algorithm for General Composition

The following algorithm implements McDermott's idea of automated service composition with alternative branches [84]. His idea was to insert verify-steps during the sequential composition process that may or may not be true at runtime and need to be reconfigured in case of failure.

6.1.2.1 Algorithm

The idea is to construct compositions with branches by creating sequential ones that may contain tests to be executed at runtime for which a default outcome is assumed and whose other outcomes must be considered in a reconfiguration step. That is, we construct sequential compositions that may contain not only operation invocations but also special actions, i.e. if-actions. Each if-action is associated with a condition θ, which will represent a guard, and we complement this if-statement by an else-statement whose body is filled with the solution for $Pre_{\hat{q}} = Pre_q \wedge \lambda(s) \wedge \neg\theta$ and $Post_{\hat{q}} = Post_q$. Here s is the state of the composition where the guard θ is applied and $\lambda(s)$ is the label associated to it by the respective search structure. In other words, we allow everything as a precondition that was known when the guard was invoked.

We can easily integrate this into the existing approaches using special operations. More precisely, for each evaluable predicate (i.e. predicate that can be used in a guard), we create an artificial operation with empty precondition (except some most general types for the

Algorithm 5: ComposeWithBranches

```
1 comp ← getRelaxedSequentialComposition(D, O, q);
2 ∀i : (Z_q̂)_i = ⊖_i((Z_q)_i, ⊙(c)_i);
3 foreach (s, θ) ∈ comp do
4 |   Pre_q̂ = Pre_q ∧ λ(s) ∧ ¬θ;
5 |   Post_q̂ = Post_q;
6 |   subComp ← Compose(D, O, q̂);
7 |   comp ← comp ⋄ subComp;
8 end
9 return comp;
```

arguments) and without output and with only this predicate in its postcondition. This allows the search structure to simply make this type of predicates true using these special operations. In a post-processing step, we can simply replace invocations of these operations by guards.

Given such a sequential composition, we can derive a composition with alternative branches. Let c be a composition obtained in the above fashion with guards $\theta_1, .., \theta_n$. c is not well-defined because it does not contain $\neg\theta_i$ for any of the guards. Hence, we need to find the implementations for exactly these missing alternatives. We can do this by submitting a recursive subquery with precondition $Pre_q \wedge \lambda(s) \wedge \neg\theta_i$ and postcondition $Post_q$ for each such guard θ_i. As described above, we can assume for this subquery not only Pre_q but actually everything we know to be true at the point where the guards split the control flow. That state is denoted as s and the knowledge we have in it as produced by the algorithm is its labeling $\lambda(s)$.

This process continues recursively until every subcall returns a sequential composition without guards. We obtain a tree-shaped composition, and each node with more than one successor corresponds to an if-else block in which the body of the else-block was obtained through a subcall.

Of course, this method is technically constrained in the choice of literals that may be "assumed". That is, not every predicate can really be tested at runtime. For example, a predicate like $isAvailable(x, y)$ cannot be tested, because predicates are just (abstract) names but not names of implemented routines. As a consequence, we restrict this method to guards imposed by literals that are *implemented* like $\leq, \in$, *isEmpty*, etc.

Given such an extension of the search structures discussed previously, we can apply the algorithm Alg. 5 in order to find compositions with branches. It solves the relaxed sequential problem and then builds a subquery for each guard contained in the relaxed version. Considering also an update of the remaining non-functional properties for the else-branch, it invokes itself with the subquery and this updated non-functional properties. The resulting composition is then merged into the relaxed result obtained in the beginning.

6.1.2.2 A New Benefit: Automated Query Relaxation

When applying the above algorithm in practice, one must propagate some kind of timeout to the subcalls. The client will not wait forever, so we can assume that some maximum time t^{max} describing the time until a solution must be delivered is specified. Suppose that it takes t_1 to find a relaxed solution c, then $t^{max} - t_1$ remains to find solutions for the counterparts of the guards of c.

An interesting case occurs when no solution is found within the given timeout but a

relaxed one was identified. In other words, if we would have applied the purely sequential algorithm, we would not have obtained a solution at all, but when employing the relaxed composition algorithm, we can find a composition that achieves the desired postcondition at least in *some* situations (based on the variable values evaluated by the inserted guards). Instead of returning nothing, we can now return this relaxed solution, i.e. a non-sequential composition where some of the else-branches are empty (namely those where no (sub-)solution was found in time).

The composition returned then is not a solution for the original query, but a conditional one. Suppose that the query postcondition was $Post_q$ and we find a composition including a guard θ but did not find any solution for the else case based on $\neg\theta$. Then we can still say that we solved the problem with postcondition $\theta \rightarrow Post_q$. This is *much* better than returning nothing. In other words, if the client asks for too much, we do not tell him that it's impossible to solve the query but tell him how the query must be adjusted in order to provide a solution, and directly give him the solution for that adjusted query.

In planning literature, this problem is addressed through the notion of strong and weak plans respectively [25]. A strong plan always yields the goal state, and this would be very desirable. However, in the discourse of developing the SAM framework for software composition at SAP, Hoffmann et al. showed that in many cases of software composition, strong plans do not exist [63]. The solution is to escape to weak plans, which is the same as query relaxation.

In practice, this kind of automated query relaxation may often be the only way to return something useful at all. It may be that the user requests something that cannot be achieved unconditionally. Using this technique, we can propose him a solution bound to some condition and he may acknowledge or decline this one.

6.1.3 Coping with Negative Domains

I briefly describe the negation problem and then possible solutions and challenges that arise.

6.1.3.1 The Negation Problem

The problem here is now that $Pre_{\hat{q}}$ contains a negative literal. So it does not match the composition problem defined in Chapter 2.

The root of this problem is that creating compositions with alternative branches *necessarily* means to relax the assumption of a positive domain. Suppose that we have a composition c with a guard $\neg\theta$ in a state s. A valid labeling λ for c must satisfy the condition $\lambda(s) \wedge \Omega \wedge \mathcal{T} \wedge \neg\theta \models \lambda(s')$ for the successor state $s' = \delta(s, \theta)$. It can be easily seen that $\lambda(s')$ cannot contain any positive literal that we could not also get without $\neg\theta$. But then, for every positive literal $L \in \lambda(s')$, we have that $\lambda(s) \wedge \Omega \wedge \mathcal{T} \models L$, and the guard $\neg\theta$ is needless. In other words, even if we allow the negation in the query precondition but have only positive operations, we cannot obtain any solution we not also would get without $\neg\theta$. As a consequence, software composition with branches does only make sense in non-positive environments.

Unfortunately, the completeness of the algorithm is based on the assumption of a positive domain. That is, some of the *proofs* involved in the completeness theorems assume Horn formulas, which are only given if state literals are positive. This can only be guaranteed if the postcondition of the query and the preconditions of the operations do not contain negations.

In fact, we can construct cases in which the algorithm, applying BW or PO, does not find a solution even though one exists. Consider the case that the query q has the (sequential) so-

lution $\langle o_1[\sigma_1], .., o_n[\sigma_n]\rangle$, and the precondition of o_1 contains a literal $\neg a$. For simplicity of the argument, I ignore the literal arguments here. Now suppose that the background knowledge contains a clause $\neg a \vee \neg b$ and a clause $\neg a \vee b$, i.e. we implicitly know that $\neg a$ must be true. Then even if we already identified the (partial) composition belonging to the above solution, neither BW nor PO is able to proof that $\neg a$ holds and, hence, will not return the solution.

To make this clearer, consider the resolution steps that would belong to the steps by the two search structures in Figure 6.1. Both algorithms would make use of the clauses of the background knowledge and end up with a state/agenda containing a instead of $\neg a$; recall that the resolvents in the resolution proof are the negation of the state labels belonging to the search graph nodes. But instead of recognizing that a cannot be true (and that we could have simply removed $\neg a$ from the state), it stops exploration in the particular node (of course assuming that there are no more literals and that a is not provided by any clause or operation). In particular, the solution would not be returned.

6.1.3.2 Possible Solutions to the Negation Problem

The most trivial way to treat this problem is to simply ignore it. This option is generally viable, because both search structures can, in general, work with states (or flaws) that contain negative literals. At no point in the definition of the structures we exploit the fact that all literals are positive. To see this, consider a simplified version of the above example. That is, suppose that a solution exists where *one* of the operations has $\neg a$ as a precondition and that we only have the clause $\neg a \vee \neg b$. Now we will, at some point, obtain a state/agenda with literal b, which is in the postcondition of some other operation invocation of the solution. So, we eliminated the negative literal by applying a non-definite Horn clause backwards. In other words, we can apply the above algorithm but must accept that existing solutions are not always found.

Even though ignoring the problem *can* be a solution, ideally, we would be modify the search structures such that they also work for non-positive problems. If we achieve this, we also have a complete algorithm for composing with alternative branches.

However, redesigning the search structures in this way is not a trivial thing to do. More precisely, there are at least two situations that need special treatment:

1. *Inconsistent Rest Problems.* Even if the underlying knowledge base is consistent, backward chaining may encounter contradictory situations. For example, if we know a, b, $\neg b \vee c$, $\neg c \vee \neg d \vee e$, $c \vee d$, and $\neg a \vee d$ and want to derive e. Then we may try to get e by the clause $\neg c \vee \neg d \vee e$, which would mean that we need to show $c \wedge d$. By $c \vee d$, we could then try to get d by $\neg c$, but then we would have to show $c \wedge \neg c$. This rest problem is inconsistent but the above knowledge base *is* consistent. So we somehow need to backtrack here.

2. *Derivation of Entailed Literals.* Things become even stranger if we get some literal a for a state that does not contain $\neg a$ but some previously reached state in the same path contained $\neg a$ as in Figure 6.1. Intuitively, this also looks like a contradiction, but it is somewhat different. From the above refutation, we can see that such a case may occur and that it would mean that a already follows (implicitly) from the knowledge base such that we never really had to solve a. The used way of backward chaining would, however, not eliminate a but simply introduce $\neg a$, even without realizing that a was a goal before.

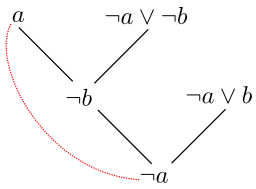


Figure 6.1: Simulating SLD resolution makes BW and PO incomplete in non-positive domains.

In both cases, we need a non-obvious backtracking strategy. In the first case, we do not exactly know at which point in the path the contradiction became unavoidable. Hence, we should not simply step back to the immediate parent and continue even though this would be possible. In the second case, we would need an occurrence check that would make sure that we at least *realize* this kind of situation and the respective predecessor node n. We could then erase all the work, go back to n and remove the implied literal a from the agenda. But perhaps some of the steps were useful and not associated with supplying a. To avoid this, we could also only stop exploration for this particular node, i.e. do not expand it.

Clearly, treating these issues is only partially possible with the presented algorithm (and much less with common best-first algorithms like A*). Of course, we can design the traversed search graph in a way that, if such kind of contradiction is detected in a node n, then n has no successor nodes. However, it would be probably better to either detect the existence of these nodes in advance, i.e. even before they are generated, or to create the possibility of transforming the search graph during search. But these techniques are beyond the formal capacities of a search structure (and the algorithm working on it) defined in this thesis, which is already more powerful than classical best-first algorithms.

Having this said, this alternative constitutes highly relevant future work. For the time being, however, it is beyond the scope of this thesis.

6.2 Finding Compositions with Loops

This section describes my approach to identify compositions with loops. In Section 6.2.1, I define what I understand by (structured) loops within a composition. Second, Section 6.2.2 gives a brief overview of the technique applied to find such compositions, which is followed by a formal model of templates in Section 6.2.3. Section 6.2.4 provide a short discussion of the treatment of non-functional properties. Then, Section 6.2.5 explains how template instantiation is used to create compositions with loops using the above search structures BW and PO. The correctness and completeness of the approach is discussed in Section 6.2.6.

6.2.1 Definition of Compositions with (Structured) Loops

Under loops in a composition, I understand a cycle in the graph belonging to it. So a composition has a loop if and only if its graph has a cycle.

From the definition of a composition, a cycle must contain a guard. Otherwise there is no *other* path (because every node has only one outgoing edge), and no goal node can be reached. Since a goal node must be reachable from every node, the cycle contains a guard.

Typically, we want to consider *structured* loops. A loop is structured if it has a (re)entrance

state $s_{entrance}$, which is the state from which the first transition belonging to the loop departs, a head state s_{head} in which the loop condition is checked, and an exit state s_{exit}, which is the first state whose departing transitions are not part of the loop. We require that

1. s_{head} must have guards θ and $\neg\theta$ corresponding to the condition to stay in or to leave the loop respectively; $\delta(s_{head}, \neg\theta) = s_{exit}$.
2. every path from the initial state to s_{head} goes over $s_{entrance}$; that is, we cannot jump into the loop from the outside
3. every state reachable from $s_{entrance}$ without going over s_{exit} must be reachable only by going over $s_{entrance}$; that is, we cannot jump out of the loop to a previous point.
4. every path from $s_{entrance}$ to some state reachable from s_{exit} actually does go over s_{exit}; that is, we cannot jump out of the loop to a later point without leaving the loop adequately.

This is exactly what is realized by while-loops ($s_{head} = s_{entrance}$) and do-while-loops ($s_{head} \neq s_{entrance}$) in programming languages.

This definition deviates from those used by others related planning approaches. Other approaches work on *universal plans* [105] and *policies* [25], which are simply state-action tables. This is almost the same as the above but without the constraints I made. Without these constraints, it is difficult to convert the composition into real programming language constructs. The naive implementation of a state machine goes using the goto-command, which has been removed from many languages. Hence, in order to obtain compositions that can actually be translated into real programming languages, we need these constraints, which are often called "controlled" loops.

6.2.2 Creating Compositions with Loops Using Templates - The Idea

On one hand, one may think that it is a tough if not impossible task to find compositions with loops. We can arbitrarily nest loops, define arbitrary loop conditions, and put arbitrarily many operation invocations (or even alternative branches) within the body of a loop. It is not clear how this kind of composition space can be reasonably explored.

On the other hand, one can argue that, unless relevant knowledge is provided, the planner would not construct loops anyway. We would somehow need the motivation to believe that a loop can help to derive some desired fact.

Instead of going for a general algorithm that tries to solve composition problems based on arbitrary loops, I propose the usage of *loop templates*. A loop template reflects common tasks performed using loops like identifying objects of a given set that maximize or minimize some property, or simply filtering them. For example, in the bookshop scenario, we use an operation that computes the subset of stores that have a particular book on stock. The implementation of this operation, which we do not know, could be a loop that iterates over the given shops and checks the availability of the book in that store; that is, the implementation would be an *instance* of a general filter pattern.

So, templates, as I understand them in this context, are generic and domain independent programs that perform some kind of activity on a loop that can be *instantiated* with concrete operations. They can be thought of as compositions with placeholders for operation invocations and guards. The basic composition structure is given, but we can still concretize the

labeling of (some) edges. Replacing these placeholders by concrete operation invocations and guards yields concrete compositions; I call this process the *instantiation* of a template.

Based on this technique, we can augment the set of existing operations by the ones that can be derived from templates. That is, every instantiation of a template yields an operation that can be seen as an atomic building block.

Even though the set of possible (and reasonable) template instantiations is usually finite, we would rather instantiate them on demand. That is, we would not add these derived operations to the set of operations but provide a module that can compute *particular* instantiations relevant in a particular situation on demand.

As a consequence, we can create non-sequential compositions by passing non-sequential building blocks to BW or PO. The search structure does treat these blocks as if they were existing operations, but, in fact, they are derived on the fly and are not yet stored anywhere in form of code. In particular, they will induce a loop in the resulting compositions.

6.2.3 Formal Model

I explain the formal model of templates in two steps. First, Section 6.2.3.1 defines the elements templates and the elements belonging to it. Then, Section 6.2.3.2 defines what the *instantiation* of a template is.

6.2.3.1 The Template Model

A template is a *generic program* together with a *generic black box description* and *consistency rules* for instantiation. The generic program describes a control and data flow between used operations and serves as a blueprint for the implementation of the derived (new) operations. The black box description of the template is a blueprint for the description of the derived operations and is expressed like an operation itself (cf. to Def. 2 in Section 2.1.1). Consistency rules are conditions that every template instantiation must satisfy in order to be considered valid; they are encoded as a first-order logic Horn formulas.

As an example, consider the FILTER template in Figure 6.2. This template describes programs that compute, from a given set A, a subset A' that contains all elements of A that satisfy a particular property. For every $a \in A$, the (still undetermined) operation s is invoked and determines the value of some (still undetermined) property of a. The obtained value y is tested against some (still undetermined) condition F. The item a is added to A' if this test has a positive result. Figure 6.2 also defines some constraints, which are not relevant for the first intuition and which we explain below in detail.

Templates have *placeholders* for operation invocations, Boolean expressions, and auxiliary predicates. The syntax of placeholders for operation invocations is explained in Def. 19; we refer to these placeholders as *generic operation invocations.* Placeholders for Boolean expressions and auxiliary predicates are predicates themselves. While placeholders for Boolean expressions occur in the workflow, the description, or the consistency rules of a template, the auxiliary predicates only occur in its description and the consistency rules.

However, not all predicates that occur in a template are placeholders. First, a template may contain *domain specific predicates*, that is, predicates that occur in the knowledge base Ω. Of course, only *abstract predicates*, that is, predicates that do not occur in Ω or in the descriptions of existing operations, can be used as placeholders. Second, every generic operation invocation s reserves two distinguished predicates Pre_s and $Post_s$ to represent the

Name : FILTER
Inputs : A, v
Outputs : A'
Precondition : $\{Pre_s(A, v)\}$
Postcondition: $\{Post_s(A', v) \wedge R(A', v)\}$
Properties : $e \cdot Z_s$
Constraints$^+$: $\{Pre_s(x, y) \wedge Post_s(x, y, z) \wedge F(z) \rightarrow R(x, y)\}$
Constraints$^-$: $\{Pre_s(x, y) \wedge Post_s(x, y, z) \rightarrow R(x, y)\}$

```
1 A' := ∅
2 foreach a ∈ A do
3 |  (u) := s(a, v)
4 |  if F(u) then A' := A' ∪ {a} end
5 end
```

Figure 6.2: Generic list filter template. Placeholders are blue; dependent properties are purple.

precondition and postcondition of the operation that will replace s. While these predicates are abstract, they directly depend on the generic operation invocation and, hence, are not placeholders themselves.

For example, the FILTER template has three placeholders. First, there is an operation placeholder s, which is a generic operation invocation. Second, the abstract predicate F for the Boolean expression is used for the test on the result of the generic operation invocation. Finally, the abstract auxiliary predicate R is used for the postcondition of the template. The predicates Pre_s and $Post_s$ are no placeholders, because they belong to the generic operation invocation s. The precondition and postcondition of the template itself are generic in the sense that the precondition of an instantiation will correspond to the precondition of the operation that replaces s, and the postcondition will preserve this knowledge and add the knowledge that replaces R.

Since specifying a template in form of an automaton is an unreasonably tedious task, we write the generic program of a template in a simple imperative language (Def. 19). We allow *variables* to be of either some scalar type (like Boolean, integer, custom data types), or a (finite) set type; the types correspond to concepts of the ontology encoded in Ω. We allow the basic set operations union, intersection, and difference.

Definition 19. *Assuming the usual semantics of these programs, a* ***generic program*** *can be written as a product of these rules:*

$$W ::= \text{skip} \mid u := t \mid W; W \mid (o_1, \ldots, o_n) := s(i_1, \ldots, i_m) \tag{6.1}$$

$$\mid \textbf{if } B \textbf{ then } W \textbf{ else } W \textbf{ end} \tag{6.2}$$

$$\mid \textbf{while } B \textbf{ do } W \textbf{ end} \mid \textbf{foreach } a \in A \textbf{ do } W \textbf{ end} \tag{6.3}$$

where $u, o_1, \ldots, o_n, i_1, \ldots, i_m$ with $m, n \geq 0$ are variables, t is a basic program term (variable or arithmetic/set expression), $(o_1, \ldots, o_n) := s(i, \ldots, i_m)$ is a generic operation invocation, B is an abstract predicate, and A is a set.

The template also contains constraints, which are partitioned into two sets of *positive* and *negative* constraints respectively. Every constraint encodes some logical relation in form of a Horn implication. The positive constraints indicate conditions that every valid instantiation must satisfy. They are important to obtain *correct* instantiations. The negative constraints

express conditions that every instantiation must *not* satisfy. These conditions are important to obtain *useful* instantiations.

For example, the FILTER template has two constraints. The positive consistency rule requires that the predicate that replaces R must logically follow from the postcondition of the operation used for s and the positive test of the predicate that replaces F. The negative consistency rule requires that the predicate that replaces R must not already be derivable from the postcondition of the operation used for s alone. That is, a *useful* instantiation actually must decide the membership of a in A' based on the result of the test F. The postconditions of the template define A' as the set $\{x \mid x \in A \wedge R(x)\}$. If the result of the test F is not necessary to decide if a belongs to A', then the test is not required in the code to compute A', and the template is not an appropriate choice to compute a set with property R.

We can then formally define a template as follows:

Definition 20. *A **template** t is a tuple (D_t, W_t, C_t) where D_t is a description as in Def. 2[1], W_t is a program as in Def. 19, and C_t is a tuple (C_t^+, C_t^-) where C_t^+ is a set of positive and C_t^- is a set of negative constraints. This specification induces a set G_t of generic operation invocations, a set B_t of generic Boolean expressions, and a set H_t of auxiliary predicates.*

Note that, although we call the predicates in the template description "abstract", the approach is completely based on first-order logic. The distinction between abstract (domain-independent) and domain specific predicate is only relevant for the design task.

6.2.3.2 Template Instantiation

A template instantiation substitutes the placeholders of a template with concrete operation invocations, Boolean expressions and domain specific predicates. Generic operation invocations are substituted by existing operations and a binding between the inputs and outputs of the operations and the variables in the operation invocations. Boolean expressions are substituted by formulas of *evaluable* domain specific predicates; that is, formulas of predicates for which a programmatic implementation is known, such as the predicate $\leq$ over the domain of integers. Auxiliary predicates in the precondition, postcondition, or consistency rules are substituted by formulas containing arbitrary domain specific predicates from Ω. A set of placeholder substitutions is a *template instantiation* or simply instantiation.

We first introduce the notion of predicate bindings and operation invocation bindings, which we then merge into instantiations. Intuitively, one can think of predicate bindings roughly as logically defined string replacements.

Definition 21. *Let $\gamma(x)$ be an abstract predicate and Ω be a knowledge base. A **predicate binding** for $\gamma(x)$ is a formula $\forall x : \gamma(x) \leftrightarrow \delta^{\gamma(x)}$ where $\delta^{\gamma(x)}$ is a formula over concrete predicates from Ω without quantifiers or functions, containing exactly the variables x, and with $\Omega \rightarrow \forall x : \delta^{\gamma(x)}$ satisfiable. If $\gamma(x)$ is a Boolean expression, then $\delta^{\gamma(x)}$ must contain only evaluable predicates.*

The above definition allows to resolve abstract predicates to complex logical expressions, which are limited only in the number of variables. For example, the condition $F(y)$ in the above template can be resolved to $y \leq 100$ but not to $y \neq x$ (unless x is a constant). Note that the number of (logically equivalent classes of) formulas that can be bound to an abstract predicate is finite.

[1] With the slight difference that the properties are a mathematical expression.

Name : FilterBooks
Inputs : A, v
Outputs : A'
Precondition : $\{Set[Store](A) \wedge ISBN(v)\}$
Postcondition: $\{Set[Store](A') \wedge isAvailable(A', v)\}$
Properties : $e \cdot (0.01, 0.1)$
Constraints$^+$: $\{Store(x) \wedge ISBN(y) \wedge AvailabilityOf(z, y, x) \wedge z{=}\text{'true'} \rightarrow isAvailable(x, y)\}$
Constraints$^-$: $\{Store(x) \wedge ISBN(y) \wedge AvailabilityOf(z, y, x) \rightarrow isAvailable(x, y)\}$

```
1 A' := ∅ ;
2 foreach a ∈ A do
3 |   (y) := getAvailabilty(a, v);
4 |   if y = 'true' then A' := A' ∪ {a} end;
5 end
```

Figure 6.3: Instantiation of the filter template that filters available books

Definition 22. *Let o be a generic operation invocation and O be operations. An **operation invocation binding** for o is a concrete operation $\hat{o} \in O$, a surjective input mapping $\sigma_{in} : X_{\hat{o}} \rightarrow X_o$, and a surjective output mapping $\sigma_{out} : Y_o \rightarrow Y_{\hat{o}}$. This binding induces two predicate bindings $Pre_o(X_o) \leftrightarrow Pre_{\hat{o}}[\sigma_{in}]$ and $Post_o(X_o, \sigma_{out}(Y_o)) \leftrightarrow Post_{\hat{o}}[\sigma_{in}]$.*

Remarks.

- The precondition and postcondition of the generic operation invocation are only predicates while the precondition and postcondition of the concrete operation are (conjunctive) *formulas*.
- The purpose of requiring surjectivity is to maintain the instantiation model simple because we avoid the case of "unused" inputs and outputs. If σ_{in} is surjective, then every input of the generic operation invocation is used; hence, $X_o = \sigma_{in}(X_{\hat{o}})$. If σ_{out} is surjective, then every output of the concrete operation invocation is used; hence $\sigma_{out}(Y_o) = Y_{\hat{o}}$. These conditions are necessary to have the induced predicate bindings being well defined. For the same reason, the induced predicate binding for the postcondition $Post_o$ uses the output variables of the concrete operation invocation instead of the ones of the abstract operation invocation.

Putting the pieces together, we can define a template instantiation as follows:

Definition 23. *Let (D_t, W_t, C_t) be a template with G_t, B_t, and H_t as in Def. 20. An **instantiation** of t is a set of operation invocation bindings for generic operation invocations in G_t and predicate bindings for predicates in B_t and H_t.*

A *total* instantiation yields a new operation with semantic descriptions. An instantiation of template t is total if it defines a binding for each of the elements in G_t, B_t, and H_t. Figure 6.3 is an example for a total instantiation. The generic operation invocation is replaced by the operation *getAvailability*, the filter predicate $F(y)$ implements the test $y =$ 'true', and the postcondition predicate $R(A', v)$ is replaced by $isAvailable(A', v)$. The result is a

new operation with its description and its implementation, and the implementation can be considered a composition.

Note that we have applied a small type conversion technique here. In fact, the literal Pre_s was bound to $Set[Store]$ in the template precondition and to $Store$ in the constraints. This conversion is important for the correctness and can be obtained by specifying that there must be a type z and the first parameter of Pre_s must be $Set[z]$ in the (ground) template precondition and z in the constraints. So the abstract predicate is actually ground *twice*, depending on the occurrence of the predicate. However, one of the two groundings is a functional and deterministic dependency of the other, so the predicate is only ground once; the second occurrence is a transformed version of the first one. This is a rather trivial technical detail making the formal model unnecessarily complicated at this point, so I omit a discussion on this in the following.

One strength of our approach is that we can show that a template instantiation is correct *by construction* if the positive constraints of the template are implied by the domain knowledge for the particular instantiation. Since we already elaborated the technical aspect of the mechanism in our previous works [92, 121], I refer to those works for details. The important result is that we only need to check that the positive template constraints are true for a particular instantiation in order to be sure that even the respective instantiation is correct. The negative constraints are not relevant for the correctness.

In order to separate the term correctness of templates from the constraint check for instantiations, we introduce the notion of *valid instantiations*. A template instantiation is valid if the domain knowledge Ω entails the template constraints for that particular instantiation.

Definition 24. *Let Ω be the domain knowledge, $\hat{t}$ be a template instantiation of template t, C_t^+ be the positive constraints and C_t^- be the negative constraints of t, and ψ be the predicate bindings induced by $\hat{t}$. $\hat{t}$ is a **valid instantiation** if the formula $\Omega \wedge \psi \rightarrow c$ is always true for every positive constraint $c \in C_t^+$ and not always true for every negative constraint $c \in C_t^-$.*

Given the notion of templates and their valid instantiations, we can now explain how template instantiation helps find compositions with loops. Before describing our instantiation approach, I briefly discuss the role of non-functional properties in this setting.

6.2.4 Non-Functional Properties of Compositions with Loops

Non-functional properties become hard to handle when loops are involved. In fact, treating these is so challenging that the research community specialized in optimizing non-functional properties of *given* workflow templates (different templates than the ones here) has ignored loops for quite a while. The problem is that we do not know how often a loop will be executed, so it is not clear how the properties of the operations in the body can be aggregated to an overall-property.

The state of the art is to assume that the (expected) number of loop iterations is known. That is, among the approaches that consider loops [6, 18], it is assumed that the properties of the composition contained in the loop body can simply be multiplied by some constant factor, which corresponds to the assumed number of iterations.

Even though I think that this assumption is grossly inadequate, I also make use of it within this thesis. Of course, one needs a more complex expression to express that number and not just a constant factor. However, this also imposes changes in a model, because the non-functional properties then are not only numbers but become functions. This is an interesting

extension for future work but not the focus of this thesis. As a consequence, I assume a constant factor e to be predefined together with the template and that is used for instantiation.

Note that my previous publication [89] on which this section relies does not consider non-functional properties. Hence, the algorithm needs to be adapted, or we need a post-processing step that checks whether the operation itself exceeds the remaining budget.

6.2.5 Integrating Template Instantiation Into the Composition Process

I now explain how templates can be used to enrich the sequential composition techniques in order to find compositions with loops. Section 6.2.5.1 gives a brief intuition and an overview of the instantiation routine, which is explained in more detail in Section 6.2.5.2. In, Section 6.2.5.3 I give a brief discussion on the approach.

6.2.5.1 Intuition for the Integration

The idea is that, given templates of the above form, we can solve *subproblems* during the composition process by template instantiation. More precisely, we can enrich the search structures BW and PO by the possibility to resolve literals not only by existing operations but by template instantiations. Given a literal L that still needs to be produced, we can check whether we can obtain L by instantiating one of the available templates. In other words, we need a function SearchForTemplateInstantiations that receives a conjunction of literals that need to be produced and that returns a set of template instantiations that achieves (some of) these literals. We extend BW and PO in that the set of considered operations is not only O but $O \cup \text{SearchForTemplateInstantiations}(\mathcal{L})$ where $\mathcal{L}$ are the open literals in a node.

The instantiation of such domain independent templates involves quite some work. We have proposed an instantiation routine for templates for this type of query in [89]. On a high level, the check for the suitability of a template consists of three steps of instantiation:

1. Instantiate the abstract predicates in the template description such that desired literals would be derivable in case of a successful instantiation.

2. Instantiate the remaining abstract predicates in the constraints of the template such that the positive constraints hold in Ω and the negative ones do not hold in Ω.

3. Identify operations for the generic operation invocations.

Figure 6.4 shows these steps for the extended running example as explained above. In that example, we assume that the operation filterByAvailability does *not* exist for a set of stores but only for a single one, and we try to automatically derive filterByAvailability from that one. However, we assume the existence of an operation getAvailability that computes whether or not a book is available in a given store; the literal in the postcondition would be $AvailabilityOf$. Moreover, we assume that the background knowledge contains a clause $\neg Store(x) \vee \neg ISBN(y) \vee Boolean(z) \vee \neg AvailabilityOf(z, y, x) \vee \neg(z = \text{`}true\text{`}) \vee hasBook(x, y)$. Intuitively, the rule says that, if z is the availability of book y in store x, and if z is true, then y is available in x.

In the following, I explain the instantiation routine and some of the pitfalls of the instantiation problem in more detail. The focus here is on the intuitive level. A detailed (more) *formal* description of the approach is found in [89].

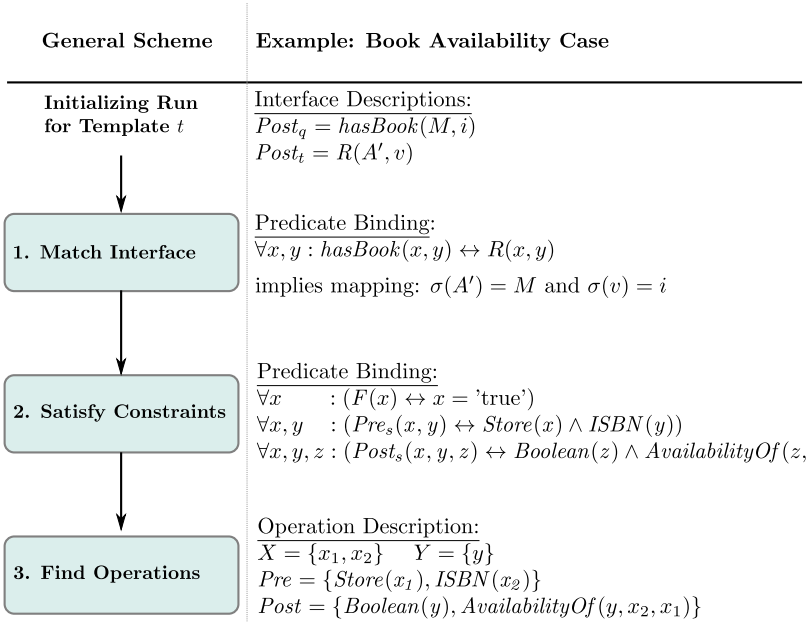


Figure 6.4: Sketch of the instantiation routine

6.2.5.2 The Instantiation Mechanism

In the following, I denote the desired functionality as the *goal state*. Just as state labels in the formal model, the goal state is characterized by a conjunction of positive first order logic (FOL) literals without quantifiers and functions. The goal state corresponds to the node label in BW and the node agenda in PO. We are interested in a template instantiation that guarantees that the goal state holds after its execution; we want it to *entail* (parts of) the goal state.

I describe the instantiation algorithm in a non-deterministic fashion. Each of the steps constitutes a choice point, which we may need to a backtrack if the set of possible choices in the next step is empty. For example, if we cannot find a solution in Step 3, we go back to Step 2, compute the next solution of Step 2 and then continue with the new solution in Step 3. If no solution was found for a template, we start again with Step 1 of the next template until all templates were tried.

Step 1: Choose an Interface Matching Intuitively, the first thing to do is to search for interpretations of the abstract predicates in the template postcondition in terms of domain specific predicates from Ω that allow to resolve at least one literal in the goal state. That is, we choose a binding for the predicates in the template postcondition to the predicates that occur in the goal state. This is a necessary and sufficient condition for the template instantiation to be relevant for the goal state.

Consider the FILTER template as an example. The postcondition contains the abstract predicate $R(A', v)$, and let $hasBook(u_1, u_2)$ be the goal state where u_1 and u_2 are data containers. Then, the only reasonable binding is to set R to *hasBook* and to map the parameters based on their position. The binding would be $\forall x, y : R(x, y) \leftrightarrow hasBook(x, y)$ (cf. Figure 6.4).

Two aspects should be discussed in some more detail. The first regards the *complexity* issue and the second the role of *outputs* of the template instantiation.

1. In general, the number of options among which we must choose here is not exponentially bound. Of course, in the above example there is no complexity issue, because one literal is mapped to another. But, in general, we split the n (more precisely, up to n) literals of the goal states into m partitions corresponding to m literals of the template postcondition. This yields $\sum_{i=1}^{n} \binom{n}{i} \cdot m^i$ possible matches; for $n = m = 15$, this is about $1.15 \cdot 10^{18}$. As a consequence, we cannot generally compute the number of candidates explicitly and choose one.

   However, the number of literals is typically rather small. The template postconditions will rarely have more than two or three literals. Also, we can bound the number of literals of the goal state that shall be achieved through the template. In fact, in many scenarios it will only be possible to infer one concrete literal for every abstract one, which changes the above term to $\sum_{i=1}^{m} \binom{m}{i} \cdot \binom{n}{i} \cdot i!$. For example, for the FILTER template, we would have to check only $\sum_{i=1}^{2} \binom{2}{i} \cdot \binom{n}{i} \cdot i!$ where n is the number of literals in the goal state. This term resolves to $n^2 + n$, so we only need to consider a polynomial number of combinations; i.e. 110 for a goal state with 10 literals and 10100 for a goal state with 100 literals.

   In addition, we can use syntactical information to reduce the combinations even more. For example, we can use the generic type prefix *Set* in order to denote a set type as already suggested by the notation Set[Store] to denote a set of stores. Then we can require that the output of the template must be of a set type, which strongly reduces the number of candidates.

2. Defining these predicate bindings implicitly defines a matching of the data containers in the goal state and the input and output variables of the template. For example, if the goal state is $hasBook(S, b)$ and we have defined the binding $\forall x, y : R(x, y) \leftrightarrow hasBook(x, y)$, then we can replace all occurrences of R respectively; in particular, we obtain the literal $hasBook(S, b)$ in the replaced template postcondition. In particular, it defines which of the data containers are produced by the operation induced by the template instantiation and which are inputs of the template itself. That is, the goal state itself makes no assertion about who is supposed to create the data containers that are mentioned in it; the decision is open, and each predicate binding induces such a data container production decision.

   Note that not all the literals of the goal state that contain the mapped outputs must be contained in the instantiated template postcondition. Recall that the elimination of all these literals was a crucial criterion in the search structure BW. However, the routine I describe here is rather a form of operation *discovery* that is independent of how the composition into which it will be encoded is created exactly. Hence, BW could indeed reject an instantiation that does not eliminate all of the literals. But, for example, in the case of PO, we may resolve some of the literals of the goal state with the template, and others (with the data containers corresponding to outputs of the template) at a later point of time.

After this step, we have *some* of the abstract predicates bound to the domain, but there are still unbound abstract predicates. More precisely, the template constraints contain predicates that do not occur in the template postcondition. Binding those predicates is the next step.

Step 2: Choose A Solution for the Template Constraints Step 1 made sure that the template instantiation (given that it succeeds) will be useful for the particular situation, i.e. our goal state. Now we must make sure that the template constraints are actually satisfied, because this is a critical condition for the *correctness* of the operation obtained by instantiation.

To this end, we compute the possible predicate bindings of unbound predicates in the constraints of the template such that the constraints are satisfied in the domain (cf. Def. 24). So we proceed in a similar way as in the first step, but we now choose bindings for the predicates that have not been bound already considering the template constraints.

Consider again the constraints of the FILTER template. The four abstract predicates are $Pre_s, Post_s, F$, and R. Suppose that R was bound in the first step, so we need to bind the remaining three. In the above example, the bindings were

- $\forall x \quad : (F(x) \leftrightarrow x = \text{'true'})$,
- $\forall x, y \quad : (Pre_s(x, y) \leftrightarrow Store(x) \wedge ISBN(y))$, and
- $\forall x, y, z : (Post_s(x, y, z) \leftrightarrow Boolean(z) \wedge AvailabilityOf(z, y, x))$.

There are two main differences between the binding technique in this and in the first step, which impose a significant increase in complexity.

1. The abstract predicates are not bound to the predicates in the goal state but to *arbitrary formulas* that can be built using domain predicates that occur in Ω or in the descriptions of operations in O. In the above example, we bind $F(x)$ to a single predicate $x = \text{'true'}$, but $Pre_s(x, y)$ is bound to a formula $Store(x) \wedge ISBN(y)$.

   Obviously, the number of formulas here is *very* far beyond what can be exhaustively analyzed. Even on the propositional logic level, there are roughly $2^{2^n} \gg n!$ many CNFs for n literals[2]. In the predicate logic case, this augments by the possibilities of parameter mappings.

   It is absolutely hopeless to consider even a fixed relative subspace, say 0.01%, of these bindings. Hence, we need to make restricting assumptions on which bindings should be considered.

2. The evaluation of a concrete predicate binding requires not only a check of coverage (i.e. that a literal of the goal state is entailed) but that the ground *constraints* are entailed by the background knowledge. That is, we must perform checks of the form $\Omega \wedge \mathcal{T} \models \bigwedge_{\alpha \in \Im(C_t)} \alpha$ where $\Im(C_t)$ are the *replaced* template constraints. Of course, this check must be performed on *each* considered predicate binding.

   For example, in the above case, we need to check that the formula $\alpha = Store(x) \wedge ISBN(y) \wedge AvailabilityOf(z, y, x) \wedge z = \text{'true'} \rightarrow isAvailable(x, y)$ is entailed by the domain. In the extended example, we assumed that Ω contains exactly this rule itself, so $\Omega \wedge \mathcal{T} \models \alpha$ holds.

   The problem here is that, even though the template constraints are Horn clauses, the replaced constraints are *not* Horn clauses anymore. This is because the literals in the constraints are replaced not by single literals but by formulas, which means that the replaced constraint is not even guaranteed to be a clause at all. In particular, the formula

[2] There are $2^n - 1$ non-empty clauses, and each non-empty subset of these is a CNF.

$\Omega \wedge \mathcal{T} \wedge \neg\alpha$ is not a Horn formula if $\alpha \in \Im(C_t)$ is a replaced template constraint. As a consequence, the formulas that need to be checked do not have the properties that guarantee an efficient treatment in general.

Unless one can show that only a logarithmic part of the possible predicate bindings needs to be considered, an instantiation that is both complete and (somewhat) efficient is impossible. In other words, we must heavily restrict the set of bindings that are actually should be considered and then exploit as much knowledge about inference as possible in order to minimize the work that has to be done by the algorithm. Obviously, this is a highly non-trivial task.

In [89], I describe a technique that addresses the problems discussed above. The procedure mainly consists of two substeps. First, the binding is solved on a simplified model on the propositional level, i.e. parameters are ignored; a solution on the simplified model is a necessary condition for a solution on the predicate level. In the second step, the bindings are completed by parameter mappings. A more detailed description of the two substeps is as follows:

1. *Finding a Propositional Solution.* In this substep, the abstract predicate *names* are bound to propositional logic formulas where the atoms correspond to predicate names occurring in the domain. Intuitively, if the propositionalized knowledge base and type system do not entail the propositionalized (mapped) constraints, the actual knowledge base cannot entail the actual constraints. Hence, we can comparatively efficiently check whether this necessary condition is satisfied.

   To this end, the knowledge base, the type system, and the template constraints are simplified to a propositional form. This is simply achieved by omitting the parameters.

   Already in this simplified model, we only consider particular predicate bindings:

   (a) predicates belonging to (abstract) operation preconditions and postconditions are bound to conjunctions only,

   (b) bindings for Boolean expression predicates are bound to formulas with only evaluable predicates,

   (c) formulas do not contain negations.

   The formula size of the chosen candidates is iteratively increased; i.e. we first try small bindings, and only if these fail, we try bigger ones.

   In practice, one would choose an upper bound for the size of the bound formulas. In the simplest case, we restrict the length to 1 modulo type information, i.e. bind every abstract predicate to exactly one concrete predicate plus at most one type predicate per parameter. Even in this simplified setting, which takes away most of the complexity, we can produce instantiations for reasonable queries [92]. For more general cases, we may fix the maximum length to some small integer.

2. *Completion of Parameter Mappings.* Once we have a valid binding on the propositional level, i.e. a binding that satisfies the (propositionalized) template constraints, we identify mappings of the parameters of the corresponding predicates. For every such choice, again, we check whether the template constraints hold.

   In order to make this more efficient, we do not solve one satisfiability problem but one for each constraint. As soon as one fails, we reject the current binding and try another one.

If no completion of predicate bindings can be found, we reject the propositional binding and consider another one. This case may occur, because a solution on the propositional level is not a sufficient criterion for a solution on the predicate level.

At the end of this process, if we have chosen a predicate binding that makes the mapped template constraints being satisfied in the domain, we usually have all abstract predicates of the template bound to domain specific formulas. Indeed, there may be predicates in the template that do neither occur in the constraints nor in the postconditions, e.g. in the template precondition, Boolean expressions of the workflow, or preconditions or postconditions of generic operations. However, this would mean that these predicates are irrelevant for the correctness of the template, which should not be the case for reasonably defined templates. In fact, if there were unbound predicates remaining, we could bind them to an arbitrary formula, and the template instantiation would still be correct. Hence, we assume that every abstract predicate of a template occurs in the template postcondition or at least one constraint; then, every predicate was bound in the first or the second step.

The template instantiation is then almost complete. The only thing that remains to be done is to find concrete operations that can be used for the generic operation invocations. This is done in the last step.

Step 3: Choose Appropriate Operations The predicate bindings computed in the previous steps have defined the preconditions and postconditions of operations that may replace the generic operation invocations. That is, if s is a generic operation invocation in the template, then the predicates Pre_s and $Post_s$ have become defined in the previous steps.

We now use these bindings to identify operations that can be used for the respective placeholders. We define the query for placeholder s as follows: The precondition Pre_q is the formula to which Pre_s has been bound. The inputs X_q are the variables in Pre_q. The postcondition $Post_q$ is the formula to which $Post_s$ has been bound. The outputs Y_q are the variables in $Post_q$ that are not in X_q. In the above example, there is only one generic operation invocation, and we would have the query $X_q = \{x_1, x_2\}$, $Y_q = \{y\}$, $Pre_q = \{Store(x_1), ISBN(x_2)\}$, $Post_q = Boolean(y), AvailabilityOf(y, x_2, x_1)\}$ (see Figure 6.4).

The set of choices for these operations is also restricted by the non-functional properties. That is, we do not only have a goal state but also a bound Z for the non-functional properties imposed by the situation in which we apply this technique. A concrete choice of operations allows us to evaluate the expression for the properties of the template. For example, in the FILTER template, we would obtain $e \cdot Z_{\mathsf{getAvailability}}$, which must be at most Z.

If these conditions are satisfied, we have generated the implementation of an operation that can be used by the sequential composition algorithm. It can be seen as a building block that was created on the fly. Either we directly deploy the implementation as a new operation, or we keep it in memory and, at the end of the composition process, replace all invocations of operations in the solution that were create in this manner by the respective implementations.

6.2.5.3 Discussion

The above instantiation routine is a first proposal and its performance strongly varies depending on the structure of the background knowledge. Some of the steps involve computationally complex activities such as solving SAT problems. In most cases, these problems are only tiny and can be solved very fast, but the problem is that we may have to solve a lot of them. In parts, it is also a simple matter of combinatorics to try out many bindings, which would be

even infeasible if the suitability of a single candidate could be checked in constant time. The experimental studies we carried out in [89] show that much of the complexity seems to depend on the structure of the background knowledge, e.g. if it is possible to derive the same literal in more than one way. These results are explained in detail in Section 7.2 in Chapter 7.

As a consequence, it will be necessary to optimize the runtime of the instantiation algorithm in order to avoid an unacceptable bottleneck. Since the instantiation routine is invoked in every node of the search structure, its runtime must be very small. Ideally, for most cases the runtime could be in the range of, say, 10ms by recognizing the unsuitability of templates based on syntactical properties like the data type. For example, we can detect that a particular type indicates a *set* of items, and literals are only tried to be resolved with a template if at least one of the parameters is a set variable. Then, if using a template for a particular problem is really promising, it is also ok if the runtime for that particular call is between 50 and 100ms.

Also, the template instantiation routine exhibits some parameters that allow to reduce the complexity significantly. For example, the length of formulas that can be bound to an abstract predicate can (in fact, it *must*) be fixed. In an extreme case, it can be set to 1, which leads to a runtime of a few milliseconds only. Another screw is the formula structure; may the formulas contain disjunctions or only conjunctions?

However, non-sequential composition *is* a complex task, and, apart from all efforts, we cannot generally expect solutions to arrive within a few milliseconds for this problem class. Of course, we can observe the algorithm behavior at runtime, analyze bottlenecks, and try to fix them by applying more sophisticated techniques. But this cannot hide the fact that finding compositions with loops exposes a tremendous complexity, which is not the fault of the algorithm that tries to solve it.

6.2.6 Correctness and Completeness of Composing with Loops

To show the correctness of this approach, one needs to show that the compositions obtained by instantiations really transform the instantiated precondition into the postcondition. Proving this is sufficient, because we already proved the correctness of both BW and PO, which implicitly assumed that the implementation for the used operations is correct. Now if we show that the implementation of the obtained loop blocks and, hence, for the operations created on-the-fly is correct, then the whole composition approach remains correct.

The nice property of the approach used here is that the template instantiations are correct by construction if the template itself is correct. That is, the actual verification task is to check that the template code transforms the template precondition into the template postcondition given the (positive) template constraints as background knowledge; i.e. the template is actually verified on the abstract level. Since the proof relies on the positive constraints, we must make sure that these constraints in fact hold in the domain for a specific instantiation, i.e. for a particular binding of the abstract predicates to domain predicates. In other words, verification goes in two steps:

1. Verify the template manually on the abstract level; the positive consistency rules may be used for this task.

2. For a concrete instantiation, check that the background knowledge Ω entails the positive consistency rules under the predicate binding imposed by the instantiation.

A proof for the correctness under these conditions can be found in [121].

Since we consider the first step been carried out by an expert and the second one assured by the instantiation routine (validity is a perquisite for an instantiation to be returned), we can be sure that the template instantiation is correct. It can be easily seen that the instantiation mechanism presented in [89] does only return valid instantiations.

While the presented composition technique is correct, it is obviously not complete. Of course, the presented method is complete in the sense that it detects all compositions with loops that can be built with the present templates. This is simply because the instantiation mechanism *enumerates* all relevant instantiations. However, the great majority (in fact, an infinite set) of compositions with loops is obviously not detected by the presented method because we have no template for the contained loops. Consequently, the approach is not complete in the sense that it finds every solution but only those with a particular structure.

7. Experimental Evaluation

This chapter presents experimental evaluations of three of the four techniques presented in this thesis. In Section 7.1, I provide an exhaustive comparative evaluation on the techniques for sequential composition techniques described in Chapter 4 and Chapter 5. Section 7.2 contains an evaluation of the template instantiation approach described in Chapter 6. I do not present experimental results for composition with branches since this is not a core contribution of this thesis; we carried out several example runs for compositions with alternative branches in order to verify the implementation.

Even though the idea is to connect sequential composition with loop instantiation, the evaluation considers both techniques in separation. The main reason is that a naive integration of loop instantiation into sequential composition as sketched in Section 6.2 is theoretically simple and easy to implement but would be still highly inefficient since the instantiation technique is still rather slow. Hence, in order to separate concerns and to enable reasonable conclusions, I consider sequential composition and template instantiation separately. I discuss the concrete practical obstacles of the integration and possible solutions in Section 7.2.3.

7.1 Experimental Analysis of Sequential Composition

This section presents the results of the practical experiments we[1] carried out with the sequential composition techniques presented in this thesis. In particular, it compares the search structures BW and PO with each other as well as the heuristics used with them, i.e. e_{fast} and e_{nf}. The main research question behind this study was whether or not the approaches can be expected to be usable in practice in the sense that they deliver results in acceptable time. The results presented here suggest that the answer to this question is affirmative, but they must be interpreted with caution as they are based on entirely synthetic benchmarks.

The section is organized in four subsections. Since creating a reasonable benchmark environment for this evaluation was a highly non-trivial problem, I first give a high level overview of how the benchmark was done in Section 7.1.1. As far as I know, there are no benchmarks available in the community, so we needed to create a rather complex benchmark environment that is able to generate reasonable synthetic composition problems. I then give a detailed description of how this problem generation works in Section 7.1.2. The presentation and discussion of the actual results is covered in Section 7.1.3. Finally, I briefly summarize the important insights of the evaluation in Section 7.1.4.

[1] In this section, when using the pronoun "we", I refer to myself together with my student coworkers David Niehues and Marcel Wever who invested many dozens of hours of high quality work to realize this evaluation.

7.1.1 Overview of the Experimental Analysis

In contrast to most tasks related to classical planning, there are no common benchmarks for the problem of software composition. In fact, I am not aware of any benchmark problem in planning where object creating operations are considered. Exhaustive evaluations of composition approaches have been carried out [12, 59], but the setups used in those papers are of limited utility for the composition problem discussed here due to the lack of object creation.

As a consequence, we have developed a parametrizable benchmark environment that generates composition problems synthetically. Before describing the details of this benchmark environment, I give a high level overview of the involved components.

7.1.1.1 The Three Dimensions of Analysis

Besides the simple number of operations and clauses in the background knowledge, there are many parameters that could be subject to analysis. First, on the syntactic level, these could be the distribution of the *number* of inputs and outputs of an operation or clause; the number of literals in the precondition, postcondition, or clause body. Second, with respect to the *semantics*, parameters could be the number and the connectivity of types, their frequency, and the distribution of predicates over the operations and clauses. Finally, on the level of possible *solutions*, these could be the number of different solutions, the average length of solutions, sets of minimal lengths of different types of solutions, etc.

In order to conduct a reasonable evaluation, I decided to focus on three parameters. That is, in the following, I fix all parameters except the *market density* of the domain, its *query potential*, and the *minimum solution length* for the actually sent queries. Market density refers to the (rough) number of operations that implement the same functionality and only differ in their non-functional properties. Query potential refers to the number of reasonable different queries we can send to the system, i.e. non-trivial queries for which we will get a positive result. The minimum solution length of a query is a lower bound on the length of *any* solution to it. Intuitively, these three parameters are positively correlated with the "difficulty" of the resulting composition problem. It is hard to present results in a reasonable fashion if the system complexity evolves in more than two dimensions, but focusing on only these three aspects is already highly compressed, and fixing even one of these dimensions would overly constrain our conclusions.

Market Density

Query Potential

Minimum Solution Length

7.1.1.2 Experiment Procedure Overview

Our evaluation involves two major steps. In the first step, we *generate* a set of composition problems for a specific difficulty setup. In the second step, we *solve* a reasonable subset of these problems for different solver configurations (i.e. once using PO or BW, using or not using pruning, etc.). Figure 7.1 sketches this process.

The problem generator is expected to generate composition problems based on the maximal minimum solution length, the query potential, the market density, and a set of non-functional properties. The requirement is that the set of returned queries contains m queries for each minimum solution length between 2 and n where m is the query potential and n the maximal minimum solution length; i.e. it should contain $m \cdot (n-1)$ many queries. The types, operations, and the background knowledge should be created in a "reasonable way", i.e. containing the elements necessary to answer the queries plus additional operations and clauses that may cause additional workload for the solving algorithm. The additional work-

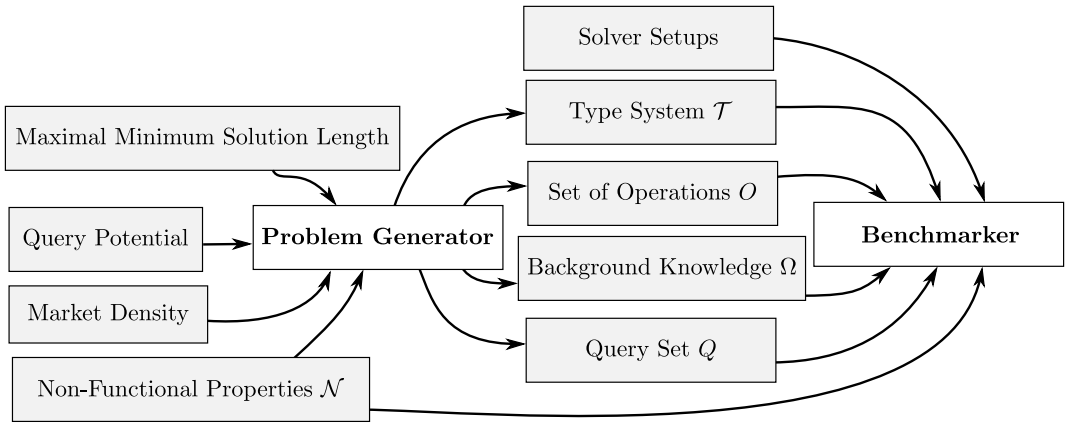


Figure 7.1: Overview of the experimental analysis process.

load is controlled by the market density parameter. Creating composition problems in this way is a non-trivial task, which I describe in more detail in the following section.

Once obtained a set of composition problems, the benchmarker tries to solve a reasonable subset of the queries using different algorithm setups. Theoretically, we could solve all of the queries in a problem, but based on the way how they are generated, they could be dependent on each other. Hence, it is better to run the problem generator several times and only evaluate a subset of the queries of each instance. In our experiments, we selected one random query for each minimum solution length and solved that query using the different algorithm setups. An algorithm setup consists of (i) the used search structure, i.e. BW or PO, (ii) whether one or all solutions shall be found (as many as possible within a given timeout), (iii) whether or not pruning is used, and (iv) the exploration strategy, which may be blind (breadth first search), e_{nf}, or e_{fast} as discussed in Section 3.3.

For our experiments, we ran this process 61500 times using different parametrizations of the problem generator. We fixed the maximal minimum solution length to 10, used query potentials from 1 to 30, and market density[2] values from 0 to 10 with step size 0.25. For each of these 1230 input parameter combinations, we started the problem generator 50 times with different seeds in order to obtain a reasonable sample set, which then yielded a total of 61500 problem setups.

I now first describe the exact evaluation setup and then the results obtained on these. Having restricted ourselves to the three parameters of market density, query potential, and minimum solution length of queries, we need to say something about all the other parameters in our setup. I give a summary of all the variables that were relevant in the evaluation and which we decided (had to) to fix in order to present a reasonably summarized evaluation.

7.1.2 Experiment Setup

The setup description consists of three parts. First, I describe the relevant parameters that were fixed for the evaluation. I then describe how we generated the addressed composition problems and, finally, which experiments were run under which conditions.

[2]I have not explained the semantics of this criterion. I will describe this in more detail below in Step 4 of the generation routine.

7.1.2.1 Fixed Parameters

The first question is how the generated operations should look like. The operation layout for all the experiments is as follows. Every operation has a number of inputs in $|X| \sim Round(Pareto(0.5, \sqrt{3}))$, which yields values in $\{1, .., 5\}$, and outputs $|Y| = 1$ with probability 0.8 and $|Y| = 2$ with probability 0.2. These distributions are arbitrary, but the input distribution results input signatures as found in the Java core library. There are no preconditions but only one type requirement per input. The postcondition of each operation consists of a type predicate for each output and one or two *basic task predicates*, which are defined in the generation process. We will refer to basic task predicates simply as predicates that may occur positively only in operation postconditions. Since each operation carries out some task, these predicates are called task predicates. The considered non-functional properties are *execution price*, *execution time*, *availability* (uptime), and *scalability*, which are drawn randomly by a network-based mechanism I will describe below.

Basic Task Predicate

The clause layout is also fixed among all the experiments. Every clause must have between two and four literals. Background knowledge will typically not contain two long rules; hence, it would be unnatural to have oversized clauses in the knowledge base. Of course, the concrete threshold of four may be rather small. However, the generator will actually produce clauses that exceed the threshold of four clauses. Such a clause is then split up into two (or more) clauses with a new literal that "connects" the clauses. So the threshold is not a semantic but only a syntactic one.

For the type system, we need to fix three parameters. The algorithm we will use to create a type system requires the number of types we want to generate and distributions on the number of both supertypes and subtypes of a type. We fixed the number of types to 1000. For both distributions, we chose an exponential distribution with event rates of 0.8728 for the number of supertypes and 0.9995 for the number of subtypes. The idea behind this was to produce a type system that exhibits the same count of supertypes and subtypes per type as in the type heterarchy of the Java core library, which is achieved by these parameters. Figure 7.3 shows an example of such a type system. Of course, this does not generate a type system that is homomorph to the Java core library type system but that has at least somewhat related properties.

The vocabulary in the domain consists of "basic" predicates, inferable predicates, and type predicates. First, the "basic" vocabulary in the domain is fixed to a value of 250 basic task predicates. That means, there are 250 predicates that occur *only* in the postconditions of operations but do not occur positively in clauses, i.e. cannot be obtained by background knowledge. Of course, this number is again arbitrary, but in fact the value does not matter so much. It should be not too small in order to simulate the difference between sparse and dense domains along increasing query potentials. Second, every clause in the system entails one inferable predicate, namely the one that occurs positively in it. Third, there is obviously exactly one predicate for each type in the system. The exact number of predicates is, hence, 1250 plus the number of clauses generated, which is a random variable that depends on the generation process described below.

We make no assertion about the exact number of operations and clauses in the problem domain. Their number stochastically depends on the vocabulary size, the query potential, the market density, and the generation process described below. In fact, we even *could* generate certain numbers of these items using fill-up algorithms that would create non-dummy operations and clauses, e.g. by copying operations and assigning different non-functional properties. However, there is no particular gain in considering a fixed such number. In order to get a

feeling of the numbers of operations and clauses, I will provide these in addition to the actual performance measures in the results section.

7.1.2.2 The Generation Process

Based on the parameters fixed above and the ones provided in the input of the generator, its task is to create a set O of operations, the type heterarchy $\mathcal{T}$, background knowledge Ω, and a set Q of predefined queries that can be sent to the system for benchmarking.

The basic idea of our implementation of the problem generator is to first create solutions and then derive queries from them. On one hand, this gives us the guarantee that there are reasonably difficult queries for which a solution exist. Obviously, we also output these queries in order to enable the benchmark to use them. On the other hand, based on the assumptions we already made in the part of fixing parameters, we can guarantee a particular hardness for these queries. More precisely, we can guarantee that every solution to it has a predefined minimum length.

The problem generation process consists of the following four substeps:

1. *Create Type System.* Here we derive the type system $\mathcal{T}$, which only depends on the number of types and the distributions on the number of supertypes and subtypes for each type, which we already fixed above.

2. *Create Type-Less Blueprints for Operations and Clauses.* This step creates blueprints for the core operations and clauses, i.e. the operations and clauses that form the essential functional body of the problem domain, are generated as elements of solutions for imaginary queries. We perform this step m times for each minimum solution length where m is the query potential and, hence, is an input parameter of the generation process. For each such minimum solution length *minlength*, the idea is to create a sequence of blueprints for operations and a query that is solved by that sequence and that cannot be solved by any sequence of operations with length less than *minlength*. So this step creates preliminary (untyped) versions of the sets O, Ω, and Q.

3. *Type Operation and Clause Blueprints.* In order to not overly constrain the possible solutions, types are ignored in the second step. Using types already in the second step would dramatically reduce the probability that operations can be used in solutions for different problems. Hence, the second steps creates blueprints for operations and clauses without types and only builds a *dependency graph* among inputs and outputs of the operations. Using this graph, we can now define types for the blueprints in order to guarantee that type-compatible solutions exist. The output of this step is then the actual query set Q.

4. *Derive Concrete Operations and Clauses.* Up to now, we have not created any concrete operations and clauses but only templates for them. In this final step, we create the concrete sets O and Ω.

Figure 7.2 sketches the connections between these four steps and the inputs and outputs of the process. I now discuss these steps in more detail.

Step 1: Create Type System Generating the type system is straight forward. We first create the required number of types without any connection; in our case, we fixed the number

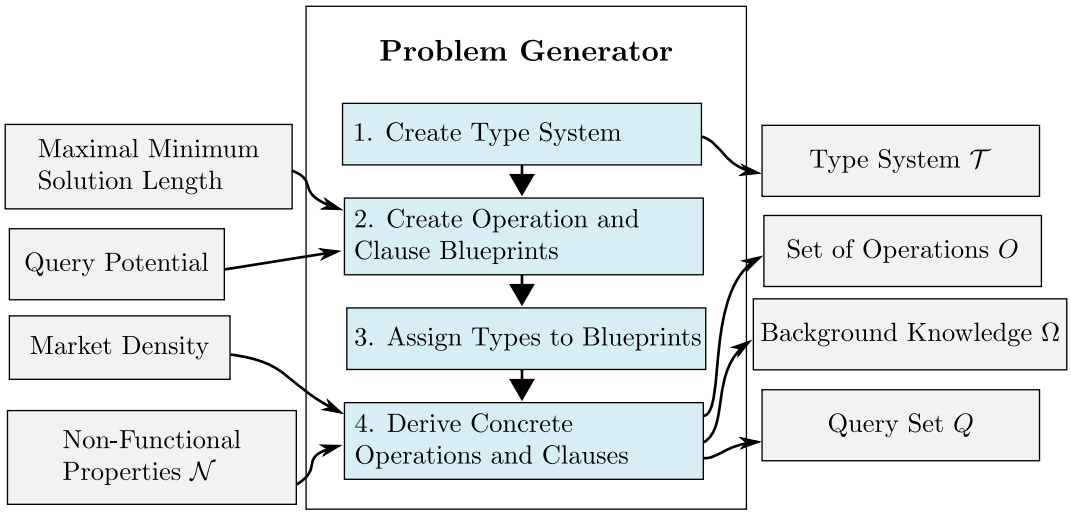


Figure 7.2: The four steps of the problem generator.

of types to 1000. Based on the given distributions on the number of supertypes and subtypes, we draw an "ideal" number of subtypes and supertypes for each type. This number is basically the input and output degree in a type heterarchy in which nodes are types. Then, we order the types and sequentially draw supertypes for the i-th type among all types with index greater than j whose ideal number of subtypes has not been reached until the number of ideal supertypes for i is reached. This way, we obtain a type heterarchy that looks like the DAG shown in Figure 7.3. In the figure, nodes are types and arcs indicate subtype relations. Nodes that have no supertypes are blue, and nodes that have no subtypes are green.

One may argue whether or not such a type system is realistic or not, but I do not think that one should go into too much detail here. As argued before, we found that its layout is somewhat similar to the type heterarchy in the Java core language. In a business domain model, one may have a very different structure, which may be for example more tree-like shaped. Also, I think that the role of the type system in the evaluation is only to simulate *some* typing but not exaggeratedly complex type heterarchies. Taking these arguments together, this type heterarchy works perfectly fine for our purposes.

Step 2: Create Untyped Blueprints for Operations and Clauses The idea for the creation of operations and clauses is to first create sequences of "operation blueprints" that are solutions to imaginary queries and then to deduce real operations and clauses from such sequences. More precisely, given a minimum solution length of *minlength*, i.e. that we want to create a solution for an imaginary query $\widehat{q}$ for which there can be no solution with size smaller than *minlength*, we create a structure similar to a sequential composition that solves $\widehat{q}$. The backbone of this procedure is our assumption that every operation will have at most two non-type predicates in its postcondition. This gives us the guarantee that if solving the query $\widehat{q}$ requires "producing" $2 \cdot \mathit{minlength} - 1$ many literals, there can be no solution to it smaller than *minlength*. What we do in the following is to make sure that there *does* exist a solution to $\widehat{q}$ (of size at most $2 \cdot \mathit{minlength} - 1$).

I describe the activity of this step for one particular run and for a given minimum solution

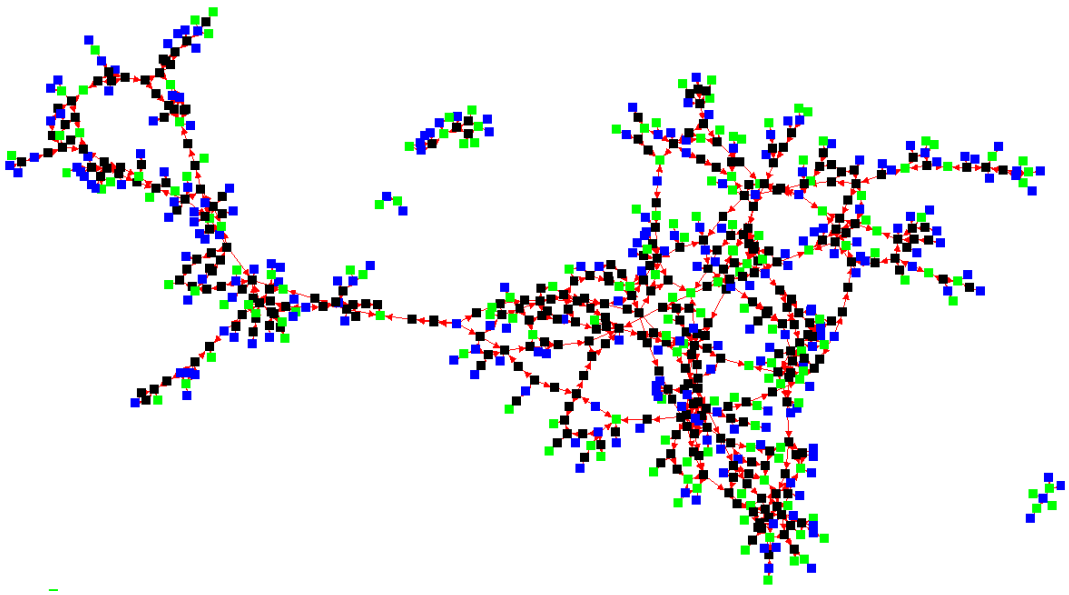


Figure 7.3: A randomized type system for 1000 types generated in step 1.

length *minlength*. In fact, this routine is run m times for each $minlength \in \{2, 3, .., 10\}$ where m is the query potential given in the generator input. The domain $\{2, .., 10\}$ for the minimum solution length is simply the result of the fact that we fixed the maximal minimum solution length to 10 (as explained above) and that a minimum solution length of 1 does not make sense (this would not be a composition but a discovery problem).

The top view of this step is as follows. First, we draw a sequence of non-type predicates of *score* $2 \cdot minlength - 1$ where each predicate contributes an positive integer to the score, and then we combine the predicates arbitrarily into groups of size 1 or 2. Second, going forwards through this group sequence, we define the data flow between the i-th group and the preceding ones. Third, we capture this "solution" in a new clause, which also introduces a new predicate. This predicate can be thought of as a "name" for this solution will be the postcondition predicate of the query $\widehat{q}$. These newly obtained predicates can then be used in the following iterations; there, we will call them *subquery (SQ) predicates*. Figure 7.4 shows an example of a whole run of this step. I now describe the substeps in more detail.

Subquery Predicates (SQ)

1. *Create Propositional Solutions.* Let *minlength* be the minimum solution length considered in this run. *minlength* will be a lower bound for *any* solution to $\widehat{q}$.

   Creating the propositional solution consists again of two substeps. The available propositions are the names of the 250 basic task predicates and the names of the subquery predicates that were newly generated in the third substep in earlier iterations. First, we draw a sequence of these propositions as follows: Set the *score counter* to 0, and then iteratively draw propositions (without putting back) uniformly from the pool. Increase the score counter by 1 if a basic task predicate was drawn and by k if a subquery predicate was drawn whose solution length is known to be k. Terminate when the score counter reaches a value of $2 \cdot minlength - 1$. In the second substep, we tie the propositions belonging to *basic task predicates* together into groups of size one or two. In the following, I will refer to these groups as basic operation bags (BOBs), which are

   Basic Operation Bag (BOB)

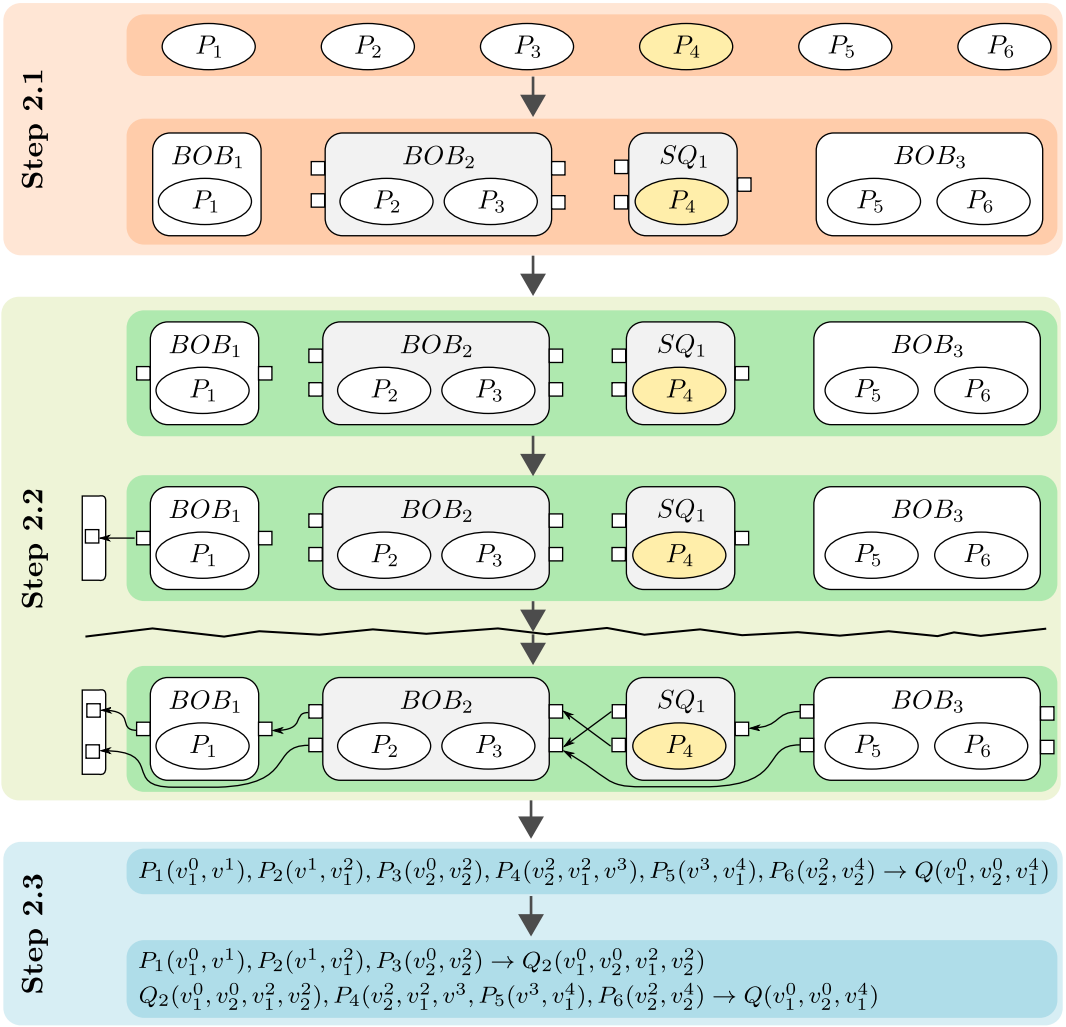


Figure 7.4: Creation Process of Operation and Clause Blueprints

basically blueprints for operations.

As an example, consider the case of *minlength* = 4 shown in Figure 7.4. As an output of the first substep, we may have obtained a sequence of propositions $\langle p_1, .., p_5, p_6 \rangle$, where p_1,p_2,p_3,p_5, and p_6 are basic task predicates and p_4 is a subquery predicate introduced in an earlier iteration with minimum solution length 2 for which we know that a solution of length 2 exists. Since p_4 increases the the score counter by 2 and all other predicates increase it by 1, we reached the value $2 \cdot 4 - 1 = 7$ after having drawn p_6, which completed the sequence. Then, merging the basic task predicates randomly in the second substep resulted in a sequence $\langle BOB_1, BOB_2, SQ_1, BOB_3 \rangle$ where BOB_1 encapsulates p_1, BOB_2 encapsulates p_2 and p_3, SQ_1 encapsulates p_4, and BOB_3 encapsulates p_5 and p_6.

Intuitively, the BOBs are our blueprints for operations. The predicates merged within a BOB will be the literals in the postconditions of the operations we will derive from it. Subquery predicates do not induce blueprints, because we know, so to say by induction,

that there are BOBs that, if we derive operations from them, can produce the predicate. Put differently, subquery predicates are like placeholders for subcompositions that were assembled in earlier iterations.

Connections among predicates in the synthesized domain are achieved in two ways. First, as already explained, we reuse subquery predicates introduced in the third substep of previous iterations. Using the predicate p_4 means that the solution to the earlier created query for which p_4 was created will be a subsolution to the query we are constructing now. This resembles that we create a solution to the imaginary query that will share operations with solutions of other queries. Second, we reuse BOBs once they have been created in later iterations. Whenever a BOB sequence contains a BOB $\{p_i, p_j\}$ and whenever a BOB for these predicates was already created in an earlier iteration, it is reused. For example, in Figure 7.4, we reuse BOB_2. This way, we avoid that queries are isolated and that there is no connection between predicates in the domain.

2. *Create Data Flow for Solutions.* Creating the data flow goes in a two-step loop. Going forwards through the created sequence of BOBs and subquery predicates, perform the following substeps for each such sequence item:

   (a) If the item is a new BOB, i.e. it has not been created in an earlier iteration, sample the number of inputs and outputs the BOB shall have; i.e. we draw this number according to the above distributions (and round the result in case of inputs). Otherwise, if the BOB has been created earlier, skip this substep.

   If we drew inputs and outputs for the BOB, we also need to link these with the arguments of the respectively grouped task predicates. Up to now, predicates were only considered by their names, but actually they reflect relations that make assertions about objects. We partition the arguments of a predicate into input and output arguments even though this is obviously not visible on the formal level.

   Mapping the BOB inputs and outputs to the predicates works as follows. Suppose that we drew a number of m inputs and n outputs for a BOB. Then all of the m inputs become arguments of the predicates grouped in it. The n outputs (actually 1 or 2) are partitioned uniformly over the predicates. Of course, predicates that were parametrized before are ignored here.

   (b) Decide the source for each of the inputs of the currently considered item; i.e. we fix where each of the inputs of the current BOB or subquery predicate comes from.

   There are several sets of possible sources for an input. Possible sources are (i) the outputs of a preceding BOB, (ii) the "output arguments" of a preceding subquery predicate, and (iii) the inputs of the still imaginary query $\widehat{q}$. Suppose that we need to parametrize the sequence item at position k. For every preceding BOB or subquery predicate, outputs are known by induction, i.e. we have a set O_l for each $1 \leq l < k$ that contains the outputs of the respective BOB or subquery predicate at position l. An additional special set O_0 contains the possible inputs of the query, which has one element for every input of every BOB or subquery predicate in the sequence up to position k.

   To illustrate how this substep works at an example, consider the green layer in Figure 7.4. It shows how we draw inputs and outputs for the first BOB and connect the inputs to the query inputs. The third sequence in the green layer shows how the sequence looks like after the last iteration of this substep. Note that the figure shows only the query inputs in O_0 that are actually used by the sequence. Also,

the green layer does not show the distribution of arguments on the parameters for readability, but they can be seen in the literals shown in the third layer.

The actual source of an input is choose randomly among the possible outputs in $O_0, .., O_{k-1}$. The probability for inputs being obtained from the possible outputs is as follows. A typical property of programs is that the parameter of a function is the output of a function invoked one or few steps before, which is why we give priority to these BOBs, which can be formalized as the condition $Pr(O_{k-1}) > Pr(O_{k-2}) > .. > Pr(O_1) > Pr(O_0)$; e.g. we can see the outputs of the BOBs as stochastic events with increasing probability in the position of the sequence. The probability among outputs within one output pool is uniform, such that the total probability of an output to be chosen is $\frac{Pr(O_l)}{|O_l|}$. One can easily show that for any probability distribution Pr, this overall distribution is also a probability distribution. For our setting, we used the geometric distribution with $p = 0.5$, i.e. $Pr(O_{k-i}) = 0.5^{i-1}$ for $1 \leq i \leq k$, which obviously satisfies the above property.

At the end of this substep, we have complemented the BOB postconditions with inputs and outputs and a connection among them within the solution. We have, however, not yet set the *types* of the inputs and outputs.

Note that, after having completed this step, not all of the basic task predicates have been used necessarily. The more iterations we make, the more likely it is that we use all of the 250 basic task predicates at least once. This way, we can control the "density" of the domain by increasing the query potential. The next section shows results on how many predicates were actually used in the different setups.

3. *Derive Clauses and Update Predicate Pool.* In our evaluation, we consider background knowledge that defines *shortcuts*. A shortcut is a single predicate that summarizes several other ones and possibly compiles away parameters. For example, if we have two BOBs with $P(x, y)$ being the postcondition of the first one and $Q(y, z)$ being the postcondition of the second one, we create a new predicate $R(x, z)$ and a clause $\neg P(x, y) \vee \neg Q(y, z) \vee R(x, z)$ that allows to infer that new predicate. In other words, we create clauses that "summarize" a solution with one single predicate; in the following, I call this the *goal predicate*.

   The goal predicate arguments are chosen minimally. It contains one argument for each query input in O_0 used by some BOB or subquery predicate and *one or no* argument for each output *set* in $O_1, .., O_l$ where l is the length of the sequence. With respect to the latter one, it will contain exactly one argument for each BOB or subquery predicate whose outputs O_i are not used by any other BOB or subquery predicate in the sequence; for each of them, one item of O_i will be chosen uniformly at random. For example, if a BOB has a postcondition $P(x, y_1, y_2)$, and y_1 and y_2 are not used as sources for inputs of successors, then either y_1 or y_2 will occur in the target predicate.

   If the resulting clause is too large, it is split up into several subclauses of acceptable length. Solutions for imaginary queries with a minimum length will produce relatively large clauses. Suppose that this length is k. If the final clause has size greater than k, it is not inserted, but instead the predicates are grouped into partitions of length at most $k - 1$. From each of these partitions, we derive a new clause containing the up to $k - 1$ literals and a new auxiliary predicate. Then, we insert the actual target clause as a clause containing the negated auxiliary predicates and the positive actual target predicate; here, it is important to retain the original data flow. If the resulting clause

would again exceed a size of k, the procedure is repeated. Since each iteration decreases the number of literals in the target clause strictly, this procedure always terminates.

As an example, consider the case that the sequence contains 10 predicates and $k = 5$. The target clause would have size 11, namely 10 plus the goal predicate and, hence, exceed the bound of $k = 5$. Let $p_1, .., p_{10}$ be the predicates from the sequence and q_1 be the target predicate. We now obtain a partition, e.g. $\{p_1, .., p_4\}$, $\{p_5, .., p_8\}$, and $\{p_9, p_{10}\}$. The resulting clauses are $\neg p_1 \vee .. \vee \neg p_4 \vee q_2$, $\neg p_5 \vee .. \vee \neg p_8 \vee q_3$, and $\neg p_9 \vee .. \vee \neg p_{10} \vee q_3$, where q_1, q_2, and q_3 are the introduced auxiliary predicates. The, the target clause is $\neg q_1 \vee \neg q_2 \vee \neg q_3 \vee q_1$; i.e. we insert 4 clauses in total. For simplicity, I omitted the data flow but this must be obviously considered here.

The criterion to partition the predicates is the number of data links within them. This number should be maximized in order to reasonably minimize the arity of the auxiliary predicates and the final goal predicate.

As indicated earlier, at the end of this substep, the predicates that were introduced in this clause creation process are added to the predicate pool. If we added auxiliary predicates, these are also added to the pool. In this way, we obtain a reuse of BOBs and subsolutions. This will increase the connectivity in the problem domain and allow for more potential solutions to the same query. Note that this does not undermine the requirement of minimum lengths for solutions, because we will not create BOBs based on non-task predicate and, hence, not create operations that produce these predicates.

Since subquery predicates are reused in later iterations, the clauses will contain both basic task predicates and subquery predicates. Without this reuse, we would only obtain clauses that allow to derive subquery predicates, and for each such predicate there would be a unique way to deduce it. However, by reusing these predicates, clauses will contain both basic task predicates and subquery predicates, which increases the connectivity in the synthesized domain and is certainly a more realistic scenario.

At the end of this step, we have a set of blueprints for untyped operations and clauses. Moreover, we have a data flow among them that is applied for solutions of (still imaginary) queries, which impose constraints on the types that may be assign to the variable parameters of the blueprints.

Step 3: Assign Types to Parameters in Operation and Clause Blueprints Given the untyped blueprints of operations and clauses, we now need to assign types to their inputs and outputs. Types must be assigned in a way such that for all data flows between BOBs established so far, the subsumption relation holds between outputs and inputs. That is, for every output o and every input i, if o is ever used as a source for i and if $t(o)$ and $t(i)$ are the types of o and i respectively, then $t(o)$ is as least as specific as $t(i)$ in $\mathcal{T}$, i.e. $t(o) = t(i)$ or $t(o)$ is a proper subtype of $t(i)$.

The challenge here is to define an assignment that is somewhat reasonable in that it covers as many types of the type system as possible. Since the assignment is not a matching in the sense that it must be injective but a type may be assigned arbitrarily often, there is a trivial solution to the above problem by simply picking one type and assigning it to every input and output. However, this would basically mean to disable types, so what we want is a more scattered typing where possibly many types are considered.

We achieved this using a (strong) modification of Andersen's pointer analysis algorithm [4]. We interpret the data flow created in the previous step as a graph of pointers (nodes are inputs

and outputs, and directed edges are the assigned flow). Now, if a node u points to several nodes v_i with $i \in \{1, .., n\}$, then all the nodes v_i will be *merged* into one new node h. This is repeated until no more merging is possible. The result of this procedure is a DAG, and we could now easily go backwards from the sinks and draw types that are subtypes of the successors. All (sub)nodes of merged nodes would receive the same type.

In order to obtain a more differentiated typing, we do not assign the same type to all (sub)nodes within merged nodes but consider the merged nodes in more detail. The (sub)nodes contained in a merged node induce a subgraph of the original pointer graph. We now apply the above algorithm recursively on this subset. The recursion cancels if no more nodes are merged. This procedure gives us a cyclic free nested subgraph of the original data flow graph containing all its nodes but only a subset of its edges.

Using this nested graph, we can now easily assign types in the way described above. That is, in a depth first fashion, we recursively draw types from the sinks to the sources. For example, if we have a path r, h, s in this graph where r is a source, s is a sink, and h is a merged node that has a path u, v. Then we would first draw a type $t(s)$ for s, then draw a type $t(v)$ for v as a subtype of $t(s)$, then draw $t(u)$ as a subtype of $t(v)$, and finally draw $t(t)$ as a subtype of $t(u)$.

Note that, like in the case of predicates, we do not necessarily make use of all types. The higher the query potential and, hence, the higher the number of solutions we produce, the more likely we are to cover all types, but this is by no means guaranteed.

Step 4: Derive Concrete Operations and Clauses Up to now we have not generated any real operation or clause. Everything that we did above was to create blueprints for these in order to easily derive "reasonable" concrete operations and clauses.

The first thing to do now is to create the final clauses. Since it does not make sense to create multiple instances of clauses, we insert exactly one for each clause blueprint produced in the second and typed in the third step. So the remaining task is to create the operations and the query pool.

Based on the market density, we now first determine how many operations will be derived for each of the previously generated BOBs. The number of operations per BOB is the maximum of 1 and a sample from a Gaussian random variable with mean 0 and standard deviation corresponding to the market density parameter; of course, the sample is taken individually for each BOB. Using a Gaussian here is arbitrary. I argue that one can consider the "normal" case as the one in which there is no operation that achieves some basic task predicate but that, depending on the market density, there may be some or even many. However, one could also choose a completely different parameter here. Obviously, we then have at least one operation per BOB and the probability of having more decreases.

While all operations derived from a BOB are functionally equivalent (same preconditions and postconditions), we generate different non-functional properties for each of them. We generate non-functional properties using a network of non-functional properties similar to a Bayesian network. The idea is that every node is associated with a non-functional property and a function to compute it from the properties it depends on. Naturally, these dependencies are modeled by the edges between the nodes. The function associated with a node may exhibit constants or non-functional properties it depends on.

In order to achieve a reasonable distribution of non-functional properties among the operations, we also need properties that are not contained in the final set of non-functional properties but that determine their values. More precisely, we can identify three layers of

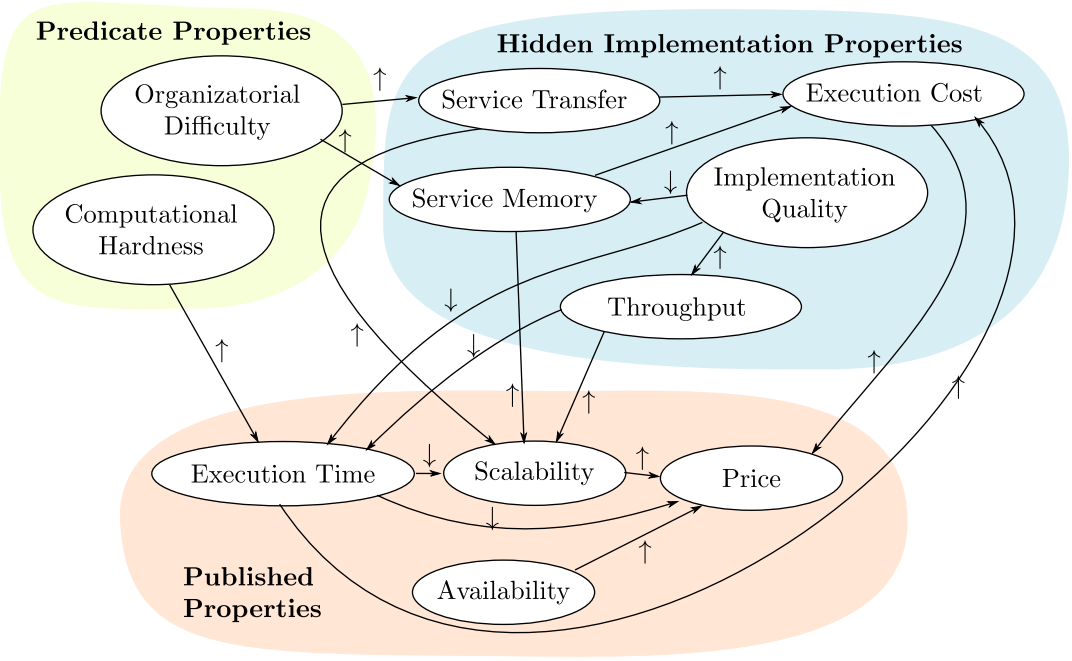


Figure 7.5: Exemplary Dependency Network for Sampling Non-Functional Properties.

nodes. The first layer contains only properties that are not actually non-functional properties but rather "aspects" on which the non-functional properties are based. They refer to properties of the problem *solved* by the respective operations (and that are hence identical for all operations that solve them). The second layer contains solution-specific but hidden properties as for example the "computation resources" of the operation (if we assume that it is executed externally). The third layer contains the actual non-functional properties.

Figure 7.5 shows an abstraction of the network we used for the generation process. The green layer corresponds to the first one describing the properties of the predicates themselves. Computational hardness says something about how hard it is to compute output values for the predicate from the complexity viewpoint. Organizational difficulty tries to capture the effort that is necessary to provide the functionality at all, e.g. how much data must be collected or known to achieve it? For example, solving mathematical equations is easy where getting weather information or making suggestions for new contacts in a social network is more difficult. The hidden implementation properties and published properties shown in the blue layer should be self-explanatory. The small arrows used to label the edges show how an increase of one property influence the other. Since the exact formulas we used for the computation are arbitrary, showing them here would unnecessarily hamper the readability such that I omitted them.

We then fix the aspect-properties for each basic task predicate and sample the non-aspect properties using the network with these values fixed. That is, for each basic task predicate, a vector of these aspect values is sampled from the network. Fixing these values, one can now evaluate the rest of the network to get different samples of non-functional properties based on the same problem aspects. Now for each operation we derive from a BOB, we draw the non-functional properties with these aspects fixed. The different operations then differ

in the way how they address the same problem and, hence, may for instance trade time for price. When creating the operations for a BOB, we check that no operation dominates or is dominated by another for the same BOB with respect to the non-functional properties. That is, each operation is a Pareto-optimal implementation for this basic task predicate.

For operations that are derived from BOBs with more than one predicate in the postcondition, we aggregate the non-functional properties. That is, we use the aggregation functions for the respective properties and postulate the results as the non-functional properties of the operation. Intuitively, this corresponds to the interpretation that the operations consist of two invisible substeps that produce the output literals in a sequence.

Once created the model, we can also output a set of queries. To this end, we simply take the goal predicate of the final main clauses as a postcondition and the type predicates of the query input pool O_0 of the respective loop run as a precondition of the query. The inputs and outputs are the trivial ones, i.e. the ones occurring in the goal predicate. The bounds on non-functional properties are obtained by randomly taking one derived operation of each BOB and aggregating their non-functional properties, which also yields a sample solution.

7.1.2.3 Evaluation Process

The evaluation then goes in two steps. First, we apply the previously described problem generator routine for a fixed number of 50 sample problems for each data point in the analysis space. Then, we solve these problems in parallel by picking specific queries of each problem and running the implementation of our algorithm of $\textsc{Search}_{\mathcal{S},\mathcal{E},w}$ ith the respective input and setups for $\mathcal{S}$ and $\mathcal{E}$. I briefly describe the exact experimental setup, the observed variables, and the hardware used for conducting the experiments.

The Examined Analysis Space (Setup of Independent Variables) Recalling the assumptions made above, the range of input we used for the problem generator was as follows. for the market density, we defined a range between 0 and 10 with a step size of 0.25; the case of 0 corresponds to a certainty of 1 that each BOB is instantiated exactly once. For the number of queries per minimum solution length, we defined a range between 1 and 30. Recall that the number of queries reflects the number of queries for *each* minimum solution length between 2 and 10. For example, if the parameter is set to 5, we will generate $9 \cdot 5 = 45$ solutions with their respective queries.

Each such problem set was addressed using different algorithm parametrizations and with different minimum solution lengths. More precisely, we applied each combination of the search structures BW and PO, the number of solutions (1 or all), activation of pruning, and the used heuristic (blind (breadth first), e_{fast}, or e_{nf}). For each problem set, we ran an experiment with one query for the minimum solution lengths of 2, 3, 5, 7, and 10 respectively.

In total, we conducted about 5.5 million experiments. The problem setup is the set $\{0.0, 0.25, 0.5, .., 9.75, 10\} \times \{1, 2, 3, .., 29, 30\}$, which induces 1230 points of evaluation. Out of the $2 \cdot 2 \cdot 2 \cdot 3 = 24$ solver parametrizations, we left out the combinations of PO with pruning, since we did not implement a specific pruning technique here, which yielded a total of 18 parametrizations. Multiplying these numbers with the 5 different minimum solution lengths and 50 samples per setup resulted in $1230 \cdot 18 \cdot 5 \cdot 50 = 5.535 \cdot 10^6$ experiments.

The Observation Space (Setup of Dependent Variables) In each experiment, we measured several properties. These can be categorized by the major concern, which was

either time, space, or quality.

- *Time to solutions.* Of course, we are mostly interested in the time we needed to find a first solution. However, in the case of BW, we also measured the time between the first and the second solution (if a second existed and was found).
- *Nodes generated, expanded, and pruned.* In addition to runtime, we were also interested in the size of the explored search graph and the state of the nodes. Therefore, we measured the number of nodes that had been generated, expanded, and pruned. We took this measure once when the first solution was identified and again on termination (due to timeout or because the graph had been completely explored).
- *Number and quality of solutions.* In order to learn something about the quality of solutions returned by the different configuration algorithms, we considered two performance measures. First, we counted the number of Pareto optimal solutions returned within the timeframe, i.e. how many different solutions are offered. Second, we compared the quality of solutions among different configurations for the same run in order to determine whether one algorithm parametrization delivers solutions that dominate all solutions delivered by the algorithm with a different parametrization.

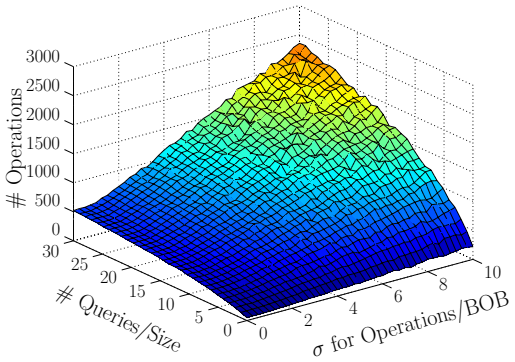
Experiment Execution Calculations leading to the results presented here were performed on resources provided by the Paderborn Center for Parallel Computing[3]. The experiments were conducted using 71 nodes of the High Throughput Cluster (HTC). Each experiment was run on an exclusively available node with an Intel(R) Xeon(R) CPU at 2.53GHz with 8 cores and 8MB cache size. The experiment process was allowed to allocate 8GB of main memory. We checked that running 4 experiments in parallel on each machine could be done without doing harm to the performance of each process, so we parallelized the experiments this way in order to accelerate the evaluation process. Since some experiments do explicitly ask for all solutions and the algorithm cannot be guaranteed to terminate, all experiments were run with an timeout of 60 seconds; of course, experiments asking for only one solution only ran until a solution was found. The choice of 60 seconds is arbitrary, but we found it to be a reasonable choice for an acceptable upper bound in practice.

7.1.3 Results

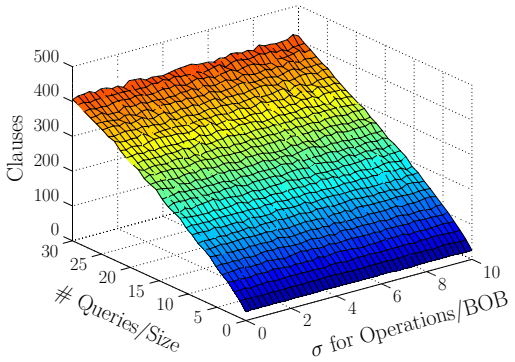
In order to put the results into a context, I first provide some setup plots for the settings. Figure 7.6 shows the evolvement of the most important dependent parameters in the respective settings. The top row shows the sizes of the sets O and Ω respectively. Considering the plots in (7.6a) and (7.6b), we can see that both query potential and market density impact the number of operations with a highly synergistic pattern, and the number of clauses is roughly linear in the query potential. A look at the plots in (7.6c) and (7.6d) in the middle row shows that also the number of BOBs is roughly linear in the query potential[4] and that the number of used basic task predicates increases rapidly such that all the predicates were used by at least one BOB for query potentials of 15 and more. The effect of this is that we have higher density of BOBs and operations per predicate for the higher query potentials, which is reflected in

[3] https://pc2.uni-paderborn.de/ – Accessed 2016-08-15

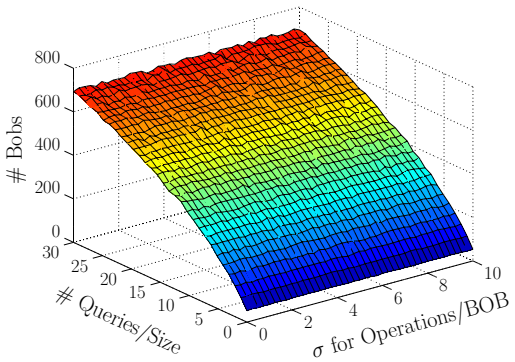
[4] Of course, the number is only roughly linear in the observed area. There is a maximum number of BOBs of $\frac{n^2}{2}$ where n is the number of basic task predicates, so for a sufficiently large query potential this maximum value is reached and stays constant.



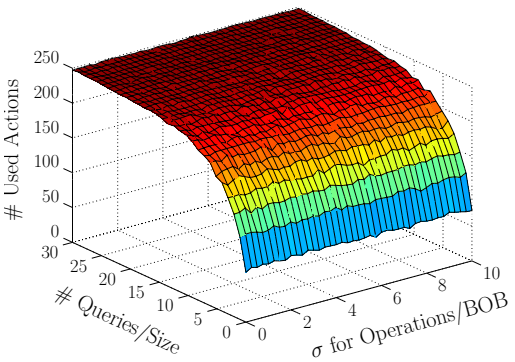
(a) Number of total operations



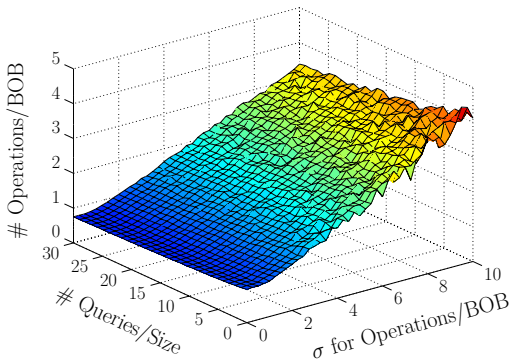
(b) Number of total clauses



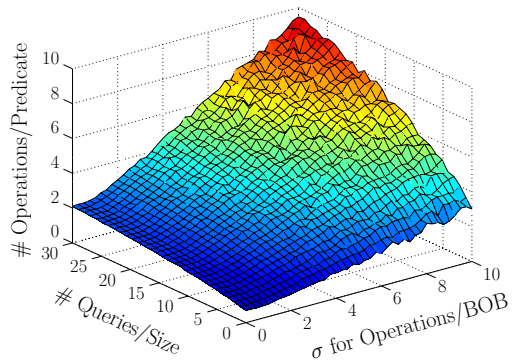
(c) Number BOBs used for deriving operations



(d) Number of used predicates



(e) Number of operations per BOB



(f) Number of operations per predicate

Figure 7.6: Average statistics for the generated problems.

the bottom right plot in (7.6f). Intuitively, the number of operations per predicate reflects the difficulty of a domain, and under this interpretation we can clearly see that query potential and market density show a synnergetic effect in this difficulty. The average number of copies derived from each BOB is shown in the plot in (7.6e).

In the following, I discuss the results in the order of the measurements defined above. That is, I discuss results on the runtime of the approaches in Section 7.1.3.1, the results on node generation count in Section 7.1.3.2, and finally results on the quality of solutions in Section 7.1.3.3. The rough structure of these subsections is similar in that they consist of four discussions: (i) a comparison of BW and PO for the easy and intermediate minimum solution length, (ii) an analysis of PO for stronger minimum solution length, (iii) a comparison of the measures with respect to the used exploration strategy (i.e. blind vs e_{nf} vs e_{fast}), and (iv) an analysis of the effect of dominance pruning based on $\succeq_{\text{BW}}$ in BW.

All measures are presented as inter-quartile means of the respective samples. Here, I take the range between the 0.1 quartile and the 0.9 quartile (instead of the 0.25 and 0.75 quartile for which the IQM is usually defined); the idea is to exclude outliers if they are rare but to consider as many of the data points as possible at the same time. That is, of the 50 sample data points per measure, we drop out the 5 lowest and the 5 highest values. The plotted value is then the mean of the remaining 40 sample points.

Several results are summarized using color maps. While I used surface plots wherever appropriate, there were several cases where color maps were a better choice. Unfortunately, for space reasons, the color maps come without a scale. However, the respective scale is always mentioned in the description text of the plots.

Note the changed meaning of the symbol σ. In the conceptual part of this thesis, I used the symbol σ to denote a mapping of data containers. Now, it represents the parameter for the market density, which is the standard deviation of the normally distributed random variable used to compute the number of derived operations per BOB. Since I do not talk about parameter mappings at all in this chapter, there is no danger of confusion.

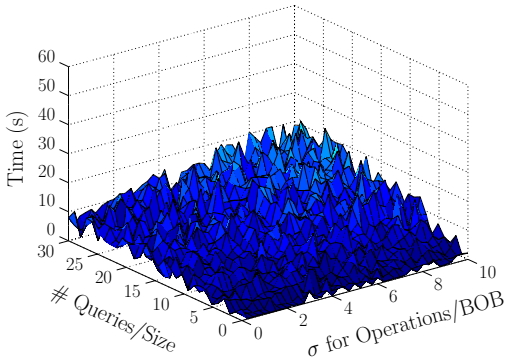
7.1.3.1 Runtime Results

The overall result on runtime is that PO is much more efficient than BW. Using PO is better than using BW in almost any occasion. What is more, the runtime of PO increases slower with the difficulty such that it can also solve cases where using BW is a hopeless undertaking.

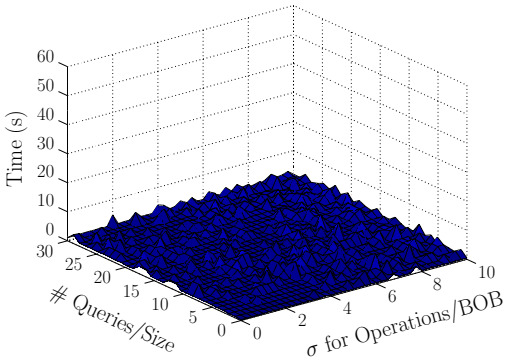
Due to this observation, I organize the result presentation as follows. First, I compare BW and PO for minimum solution lengths of 2, 3, and 5. Then, I discuss the behavior of PO in the more difficult settings of minimum solution length 7 and 10. Since BW was only very rarely able to find solutions for queries with minimum solution length 7 or 10 within 60 seconds, there is nothing to say about it in these cases. Third, I discuss the difference in runtime between e_{fast} and e_{nf}. Finally, I discuss the effect of pruning on the runtime of BW.

The runtime of all experiments lies between 0 and 60 seconds. A value of 60 means that the timeout was reached and no solution was found. The color maps come without a key, but, since they show a relative information, it is sufficient to know that the color scale is the same as for the surface plots, i.e. blue means 0 seconds and red means 60 seconds.

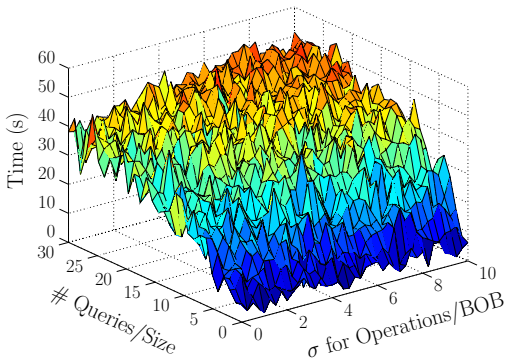
BW vs. PO It turns out that PO generally outperforms BW in terms of runtime. Consider Figure 7.7. The left and the right column show the averaged runtimes applying BW (using pruning) and PO respectively. The shown results were obtained running the algorithm with



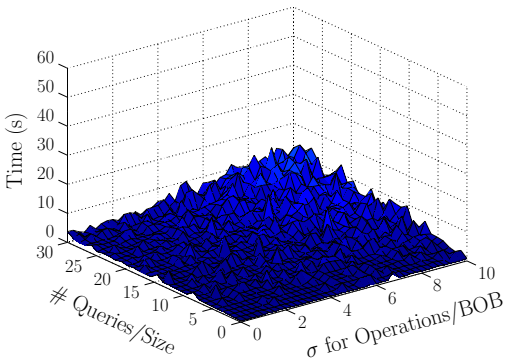
(a) BW, e_{fast}, pruning, minlength = 2



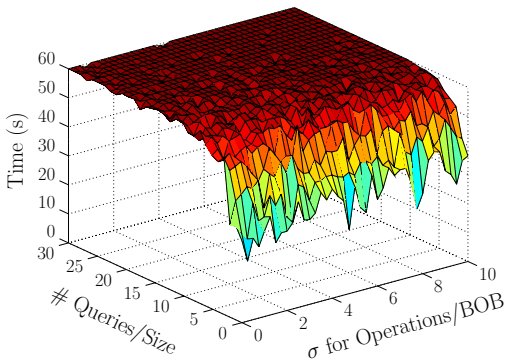
(b) PO, e_{fast}, minlength = 2



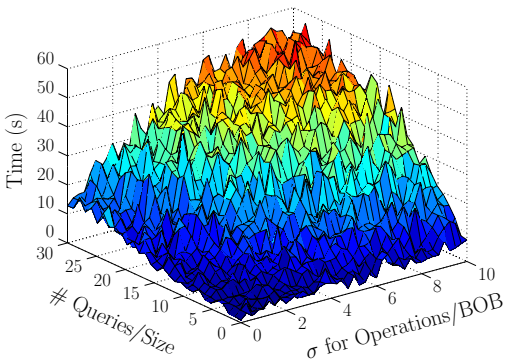
(c) BW, e_{fast}, pruning, minlength = 3



(d) PO, e_{fast}, minlength = 3

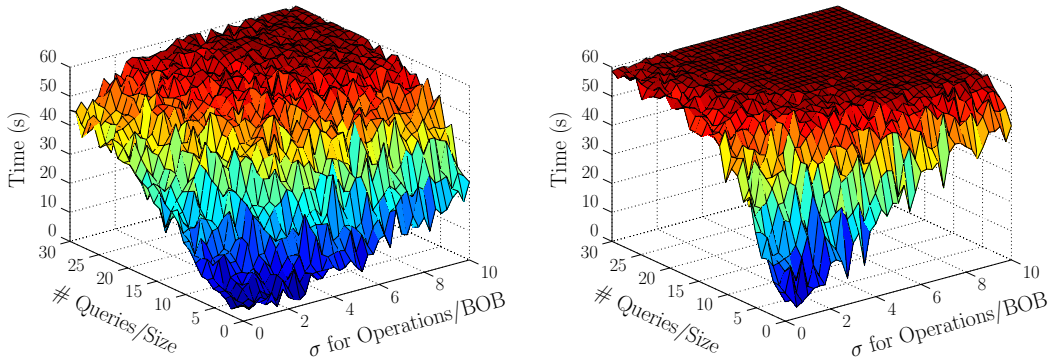


(e) BW, e_{fast}, pruning, minlength = 5



(f) PO, e_{fast}, minlength = 5

Figure 7.7: Times until first solution is found using BW and PO respectively.



(a) PO, e_{fast} on queries with minlength 7

(b) PO, e_{fast} on queries with minlength 10

Figure 7.8: PO is also applicable for settings with a minimum solution length of 10.

e_{fast} as exploration strategy. The rows correspond to the minimum solution lengths 2, 3, and 5 respectively. In the most trivial cases, the runtime is equally low, but increasing the difficulty in the domain or minimum solution length, BW performs significantly worse than PO.

It can be clearly seen that the advantage of PO over BW is far from being constant but rather linear. The runtime of PO seems to increase roughly proportional with the runtime of BW but with a coefficient less than 1. That is, in the case of a minimum solution length of 2, the plot of BW looks similar to the one of PO for a minimum solution length of 3, but for a minimum solution length of 3, the plot of BW looks already similar to the one of PO for a minimum solution length of 5.

In spite of the comparatively poor performance of BW, an interesting observation is that the market density does not affect BW as much as the query count or the minimum solution length. Increasing only the number of operations per BOB but nothing else only slightly tangles the capacity of BW to find a solution. This can be seen particularly well in the case of a little query count where even in the most difficult case of $\sigma = 10$ solutions are found within the timeout even for a minimum solution length of 5. In fact, the "vulnerability" of BW in this aspect is comparable to the one of PO.

However, PO shows this kind of resistance also with respect to the domain complexity. Consider, for instance, the BW plot (7.7c) and the PO plot (7.7f). One can clearly see that if we focus on the regions where $\sigma = 0$, the increase is much less in (7.7f) than in (7.7c). This shows that PO handles this degree of complexity better than BW.

PO in more difficult settings We have seen that, in this synthetic setting, there is no hope for BW to do anything interesting for queries with a minimum solution length higher than 5, but how does PO behave under such conditions? The results summarized in Figure 7.8 give a clear answer to this question. What we see is that PO also has problems with increasing minimum solution length, but the increase is much less dramatic. Even for queries with a minimum solution length of 10, we can still find solutions within less than 10 seconds for some settings.

From a different viewpoint, one case say that PO can handle also difficult situations efficiently as long as not *all* of the complexity factors rise but only at most two of them. Figures (7.7b) and (7.7d) clearly show that as long as the minimum solution length is low,

solutions are returned fast even if the other conditions become rather complex. On the other hand, Figures (7.8a) and (7.8b) show a relatively moderate increase of runtime along the axises for PO. The interpretation of this is that even for minimum solution lengths of 7 and 10, we can still find solutions in "easy" environments rather fast where easy means that not *both* complexity factors query potential and market density increase at a time.

Comparison of BFS, e_{nf}, and e_{fast} Naturally, we are also interested in the impact of heuristics on the search process. Since the search structure can be parametrized with different such strategies depending on whether the main focus is on runtime or quality, we would like to learn something about the relation between using no heuristic and using e_{fast} or e_{nf}. In particular, we want to know whether there is a significant advantage of using e_{fast} over e_{nf}.

Little surprisingly, it turns out that using a heuristic does significantly improve the runtime. Consider Figure 7.9. The three rows show runtimes for breadth first search (BFS), e_{nf}, and e_{fast} respectively. The left column shows runtimes for a minimum solution length of 3 where the right one shows result for a minimum solution length of 7. Comparing (7.9b) with (7.9d) and (7.9f), one clearly sees the advantage of the heuristics over breadth first search. Indeed, the difference is less striking than one may expect, but still one can see that, using a heuristic, still many problems can be solved within the time bound while most problems remain unsolved using BFS.

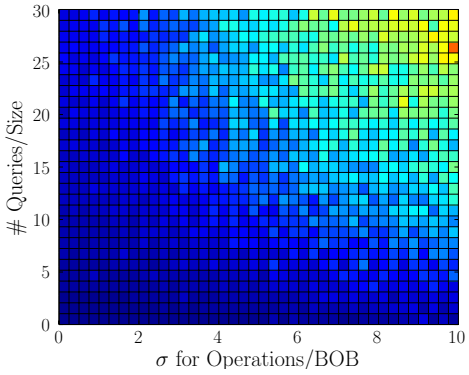
In general, the advantage of the heuristics becomes particularly visible with a higher problem complexity. Comparing the plot in Figure (7.9b) with the ones in (7.9d) and (7.9f), there is a tremendous advantage of e_{nf} and e_{fast} over BFS in the intermediate zones where BFS did not find any solutions within the time bound. In other words, the runtime increase in harder settings is much less for e_{nf} and e_{fast} than for BFS.

Also, there is a small but notable advantage of e_{fast} over e_{nf}. Indeed, Figures (7.9c) and (7.9e) show that there is only a small difference for simple queries. Considering, however, Figure (7.9d) and (7.9f), we can see that there is a quite visible advantage in more complex settings. Especially in the border cases, i.e. where market complexity or query potential are small, there are better chances to stay in the time bound using e_{fast}.

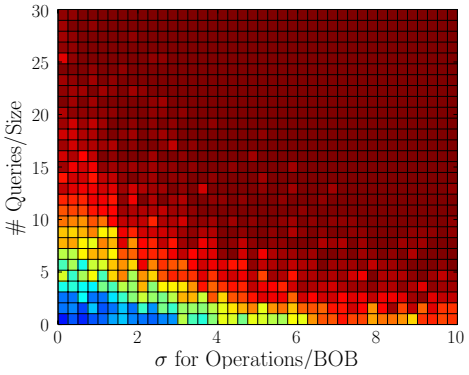
The Benefit of Pruning in BW A final point to discuss here is the impact of pruning on the runtime. Note that pruning here refers only to the pruning based on the node comparison $\succeq$ relation and not on pruning based on the non-functional costs etc. since this is cheap and the advantage is obvious. However, computing the pruning possibilities based on $\succeq$ tends to be costly, and its advantage is unclear.

The somewhat surprising result is that the impact of pruning is, even though observable, rather marginal. Consider the plots in Figure 7.10 to see this in more detail. The three rows show the runtime of BW for minimum solution lengths of 2, 3, and 5 respectively. Pruning is disabled on the left and enabled on the right. Amazingly, the plots are almost identical[5]. Since all other parameters are identical, the only explanation for this is that the time required to perform the actual pruning for node, in particular to check the conditions for pruning, roughly corresponds to the time required to expand it (and its descendants) even though this increases the nodes that need to be stored exponentially.

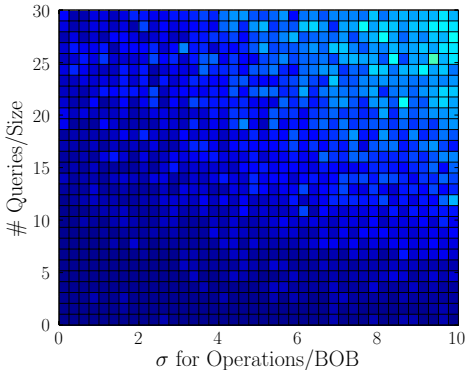
[5] One may suspect that this may indicate that pruning perhaps simply does not work even if it is supposed to be activated. However, we used a tool that visualizes the search graph and the pruning, and the plots on generated nodes below show that pruning actually *has* an impact.



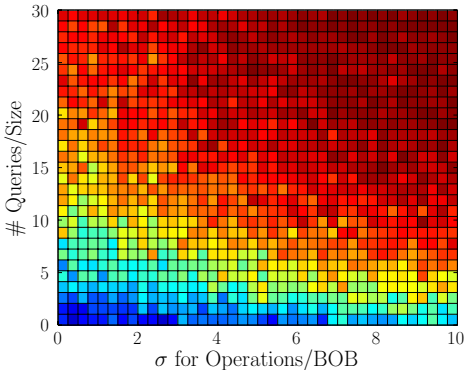
(a) BFS, minlength = 3



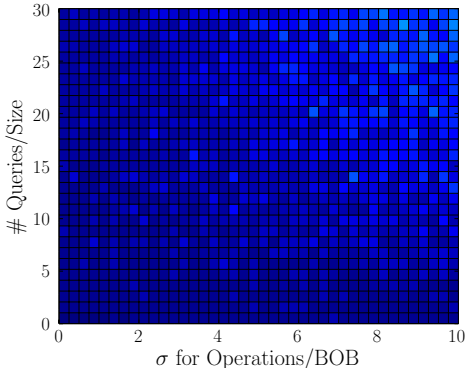
(b) BFS, minlength = 7



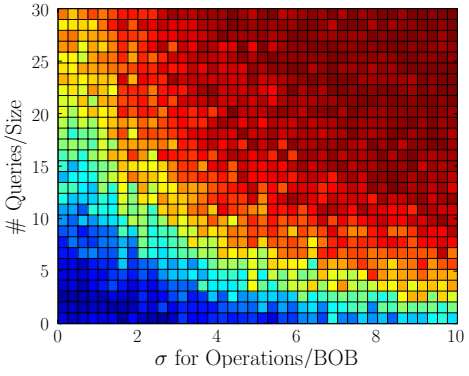
(c) e_{nf}, minlength = 3



(d) e_{nf}, minlength = 7

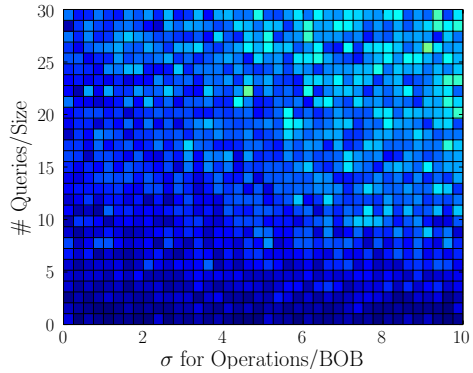


(e) e_{fast}, minlength = 3

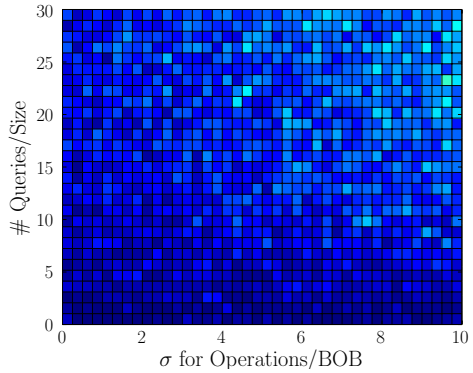


(f) e_{fast}, minlength = 7

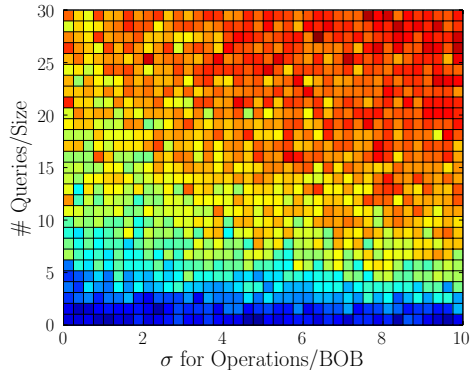
Figure 7.9: Comparison of BFS (top), e_{nf} (middle), and e_{fast} (bottom) using PO



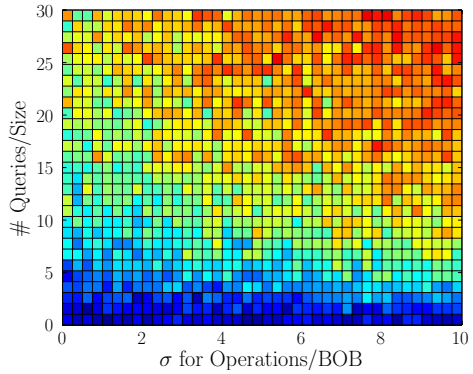
(a) BW, e_{fast}, no pruning, minlength = 2



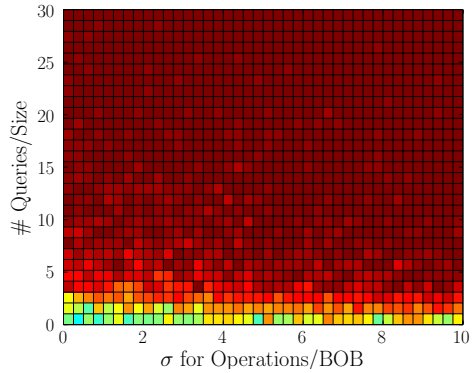
(b) BW, e_{fast}, pruning, minlength = 2



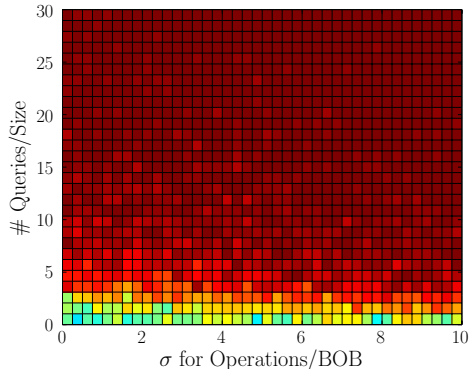
(c) BW, e_{fast}, no pruning, minlength = 3



(d) BW, e_{fast}, pruning, minlength = 3



(e) BW, e_{fast}, no pruning, minlength = 5



(f) BW, e_{fast}, pruning, minlength = 5

Figure 7.10: Pruning using $\succeq_{\text{BW}}$ does only marginally improve runtime performance.

7.1.3.2 Space Results

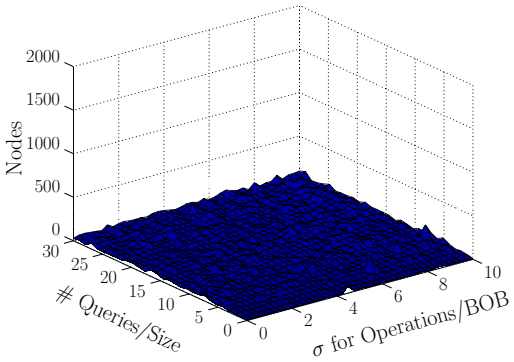
BW vs. PO The first observation on the node generation behavior considered in isolation for BW and partial-order planning respectively is that the number of generated nodes increases not as fast as the runtime under the same conditions. Consider Figure 7.11, which covers the same setups as Figure 7.7 and only shows generated nodes instead of runtime. Comparing the increase of generated nodes in BW (left column of Figure 7.11) with the increase in runtime (left column of Figure 7.7), we can clearly see that the increase of generated nodes is more moderate than the one of runtime. The same observation holds for PO (right columns of Figure 7.11 and Figure 7.7 respectively). In other words, the more nodes we generate, the longer it takes to find a solution. Clearly, this is, if at all, a correlation but not a causal relation. That is, we cannot deduce from this observation that an increase in the the number of generated nodes *implies* an even higher increase in runtime. At this time, we cannot be completely sure about the source of the higher runtime.

We can see that the node generation behavior of the two approaches is similar to their relation in runtime. Consider the results plotted in Figure 7.11. Figures (7.11a) and (7.11b) show that there is almost no difference in simple settings with minimum solution length 2. This is mainly because the number of nodes generated in such a simple setting is typically very small. However, comparing (7.11c) and (7.11d) already shows a significant difference between BW and PO in that BW generates more than twice as much nodes than PO even though pruning is active. The last row shows a similar result but is less informative since, in the difficult regions where (7.11e) and (7.11f) seen to exhibit similar values, the timeout prevented BW from generating much more nodes. In this sense, the figure is somewhat biased.

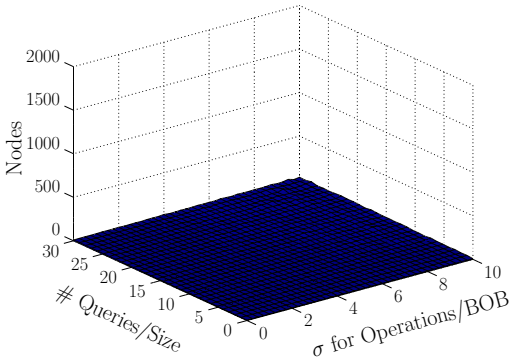
PO in more Difficult Settings When looking at the plots of the generated nodes for more complex problems, we can make three interesting observations. The first is that the general increase of generated nodes is quite moderate and much less dramatic than the increase in runtime. The second is that the number of generated nodes mainly depends on the minimum solution length and not so much on the query potential or the market density. This can be easily seen comparing Figures (7.11b), (7.11d), (7.11f), (7.12a), and (7.12b). In each of the plots, the level is relatively uniform over the whole grid, and an increase of the minimum solution length induces a lift of the level of this distribution. However, and this is the third observation, in the case of minimum solution lengths of 7 and 10, we can even see a *decrease* in the number of generated nodes with increasing query potential and market density.

The third of these observations is quite puzzling. It is intuitive to expect an increasing number of nodes moving from the left to the right in the plots as can be observed in the bottom lines of (7.12d). However, intuitively, we would not expect that this number decreases again going from bottom to top. Quite the contrary, considering the plots in Figure 7.11, we see that the number of generated nodes increases as expected.

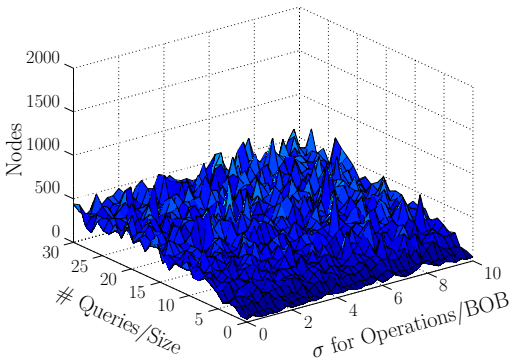
The most intuitive explanation for these results seems to be that the expansion time *per node* increases in difficult setups. That is, the more difficult a setting becomes in terms of the number of operations and clauses that are available, the longer it takes to perform a single node exploration step. This is intuitive since more operations and clauses must be considered on their suitability for a specific rest problem. As a consequence, in harder setups, there are less nodes generated within a given timeframe. This answer also explains why this effect does not occur for low minimum solution lengths. The reason here is simply that in all the cases where we have an unusual node generation behavior, the timeout was reached before the first solution was found; in the easier setups, the whole generation process was completed such



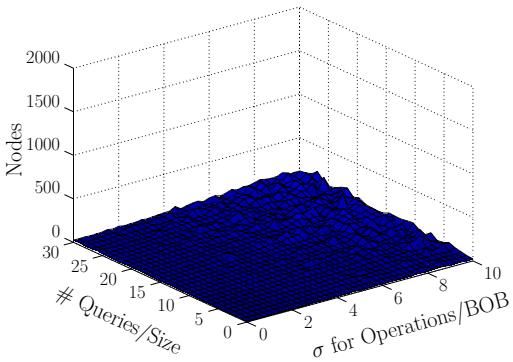
(a) BW, e_{fast}, pruning, minlength = 2



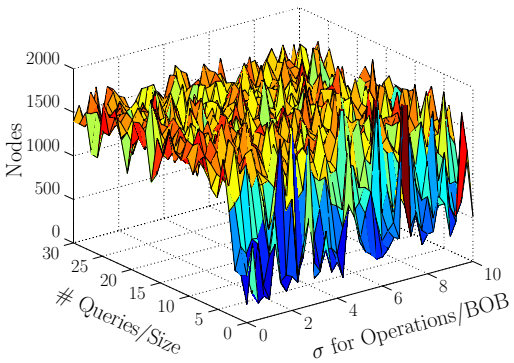
(b) PO, e_{fast}, minlength = 2



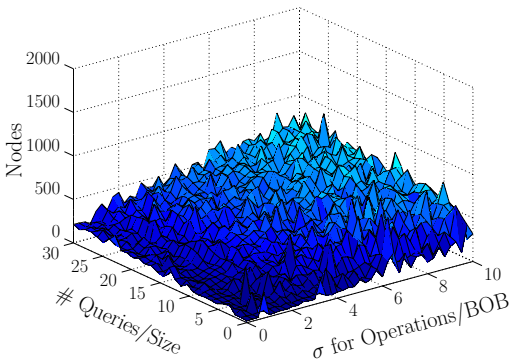
(c) BW, e_{fast}, pruning, minlength = 3



(d) PO, e_{fast}, minlength = 3

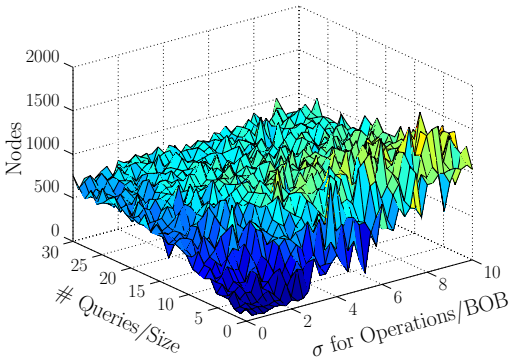


(e) BW, e_{fast}, pruning, minlength = 5

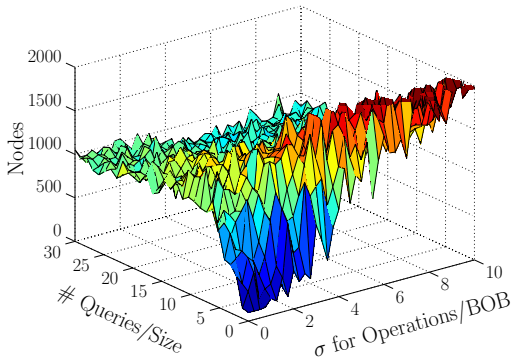


(f) PO, e_{fast}, minlength = 5

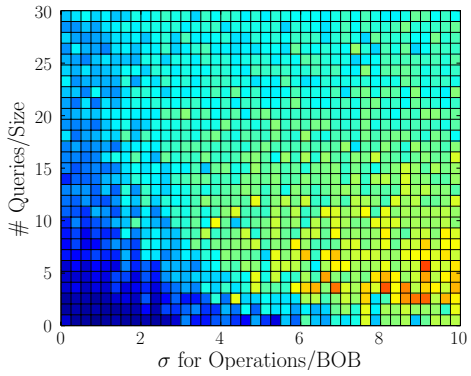
Figure 7.11: The number of generated nodes on termination by BW and PO respectively.



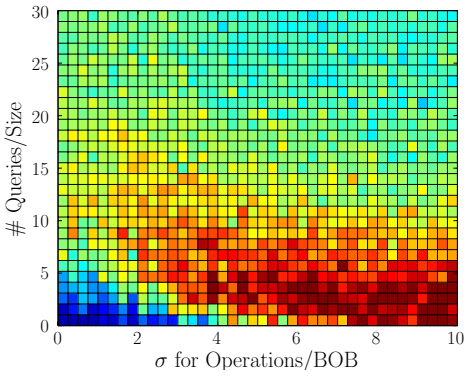
(a) PO, e_{fast} on queries with minlength 7



(b) PO, e_{fast} on queries with minlength 10



(c) PO, e_{fast} on queries with minlength 7



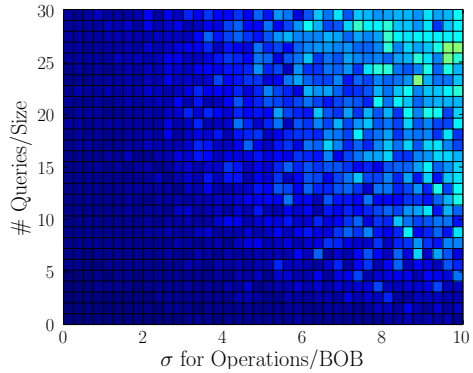
(d) PO, e_{fast} on queries with minlength 10

Figure 7.12: Strangely, PO generates *less* nodes in more difficult settings.

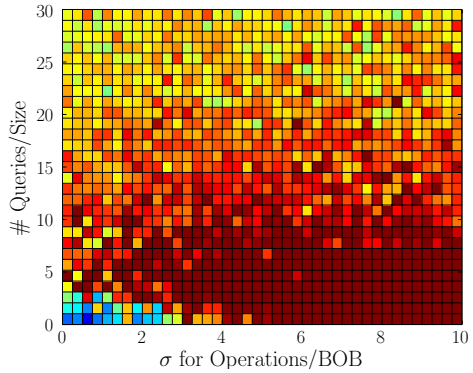
that the results were not biased.

However, another explanation could be that we expand a lot of dead-end nodes, which consume expansion time but do not produce new nodes. In fact, when running such settings with our visualization tool, we noticed that the algorithm spends a lot of time with expanding nodes that have no children. These checks take a lot of time, which could also explain the above phenomenon. Obviously, this is behavior gives rise to identify possibilities to recognize and prune this node type in the sense of dead-end pruning. An implementation of such a pruning is, however, beyond the scope of this thesis.

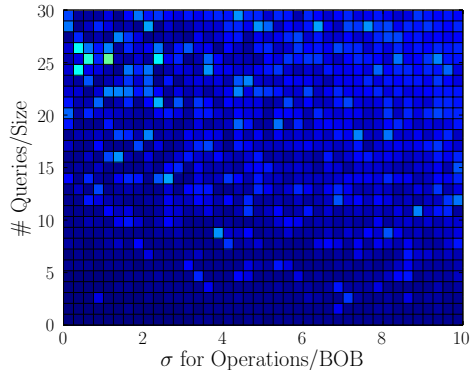
Comparing BFS, e_{nf}, and e_{fast} How do blind search, e_{nf}, and e_{fast} relate to each other with respect to the number of generated nodes? Figure 7.13 shows an answer to this question for minimum solution lengths 3 and 7. Again, the settings covered in this figure are analogous to those of the color maps for runtime shown in Figure 7.9; in particular, all results are for PO only. In these plots, dark blue, yellow, and red mean that 0, 1000, and 2000 nodes were generated respectively. Comparing the top row plots (7.13a) and (7.13b) for blind search with the plots for e_{nf} in (7.13c) and (7.13d) and the plots for e_{fast} in (7.13e) and (7.13f), we can



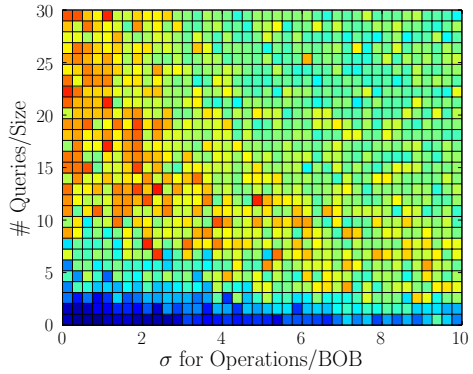
(a) PO, BFS, minlength = 3



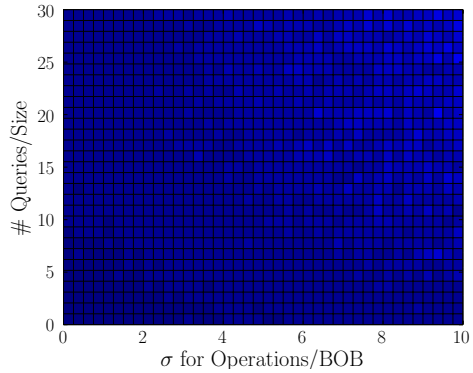
(b) PO, BFS, minlength = 7



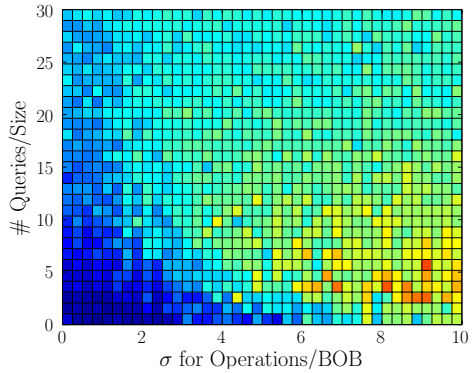
(c) PO, e_{nf}, minlength = 3



(d) PO, e_{nf}, minlength = 7



(e) PO, e_{fast}, minlength = 3



(f) PO, e_{fast}, minlength = 7

Figure 7.13: Nodes generated by PO using BFS, e_{nf}, and e_{fast} respectively.

clearly see that, as expected, BFS generates significantly more nodes than e_{nf} and e_{fast} in all setups. Moreover, one would expect e_{nf} to explore more nodes than e_{fast} before a first solution is found, and, in fact, comparing (7.13c) with (7.13e) seems to confirm this intuition for a minimum solution length of 3. However, comparing the results between e_{nf} and e_{fast} for a minimum solution length of 7 in (7.13d) and (7.13f) shows a different image, which requires a closer look.

Like the previously discussed results, the node generation results for a minimum solution length of 7 look quite disturbing at first sight. We already discussed above that the number of generated nodes in the difficult region is much less than in the easier regions (of high market density). In fact, while the plots on the left of Figure 7.13 document the expected increase of generated nodes for a minimum solution length of 3, the right column shows the same weird results for each of the heuristics for a minimum solution length of 7. As above, the explanation for this is probably that the generation time per node is higher in difficult setups.

However, even though that effect seems to be independent from the used heuristic, the three heuristics still exhibit significantly different behaviors. Using blind search, we generate significantly more nodes than using a heuristic. The conclusion of this must be that the node generation time does not only depend on the general composition setup but also on the region in which we search. Otherwise BFS could not generate so much more nodes in the same time.

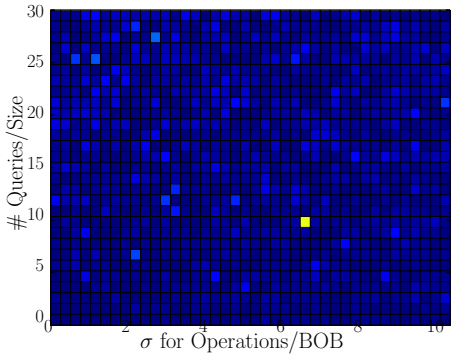
We must be very careful with interpretations of these results. In particular, we cannot say anything about "good" or "bad" node generation behavior. Usually, one would expect that generating less nodes is better than generating more. However, this seems to be a dangerous interpretation here since we do not know anything about the *reasons* for which more or less nodes were expanded.

Summarizing, the most important message to take away here is that the reason for high runtimes is not an explosion of the search graph. Instead, we saw the fact that exploring *single* nodes takes longer in those difficult setups and that we apparently lose a lot of time in expanding dead-end nodes. This motivates to look for possibilities to improve the node expansion and pruning process.

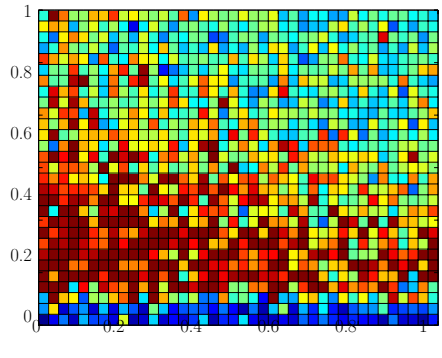
The Impact of Pruning We have already seen that pruning has only marginal impact on the runtime of searching with BW, but what about the number of generated nodes? Taking a look at Figure 7.14 shows that pruning has a tremendous impact on the number of generated nodes in BW. Since pruning does not play a huge role when posing queries with a minimum solution length of 2, I only discuss the cases of lengths 3 and 5. The plots in the top row of Figure 7.14 show the absolute numbers of pruned nodes (dark red = 200). The plots in the second row show the number of nodes generated when pruning was enabled divided by the number of nodes generated when pruning was disabled; so the values range between 0 and 1.

The first observation is that pruning effectively happens and that its application increases for queries with higher minimum solution length. To see this, consider (7.14a) and (7.14b). In (7.14a), we can see that pruning only occurs sporadically even though all over the setup grid. Comparing this plot with (7.14b) shows that an increase in the minimum solution length also implies that more nodes are pruned. Again, we have the anomaly in the top right region where we would expect more nodes to be pruned than on the left or on the bottom line. Leaving this effect aside, we can see a dependency between the setup complexity and the number of pruned nodes. It is very likely that we would see more pruning in the top right region if we did not use a timeout.

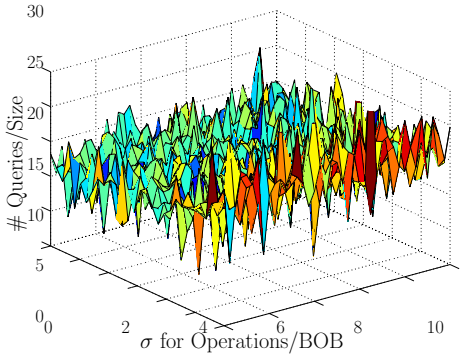
The impact of pruning on the number of effectively generated nodes is enormous. Figure



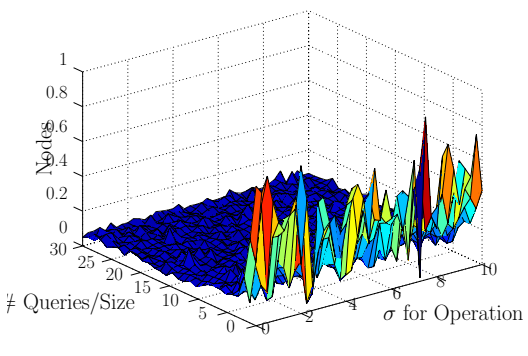
(a) pruned nodes (absolute), minlength = 3



(b) pruned nodes (absolute), minlength = 5



(c) ratio of generated nodes, minlength = 3



(d) ratio of generated nodes, minlength = 5

Figure 7.14: Pruning behavior of BW using e_{fast}.

(7.14c) shows that, except for low query potentials, the number of generated nodes is only half of the number of nodes when pruning is enabled. Given that so few nodes are pruned, this is quite impressive and suggests that pruning some few nodes saved the algorithm from creating several generations of nodes (not only children but also grandchildren). For a minimum solution length of 5, (7.14d) shows an even more drastic improvement. Here, the number of generated nodes is only about 10% in most setups when pruning is enabled.

The pitfall here is that we actually see two different measures within single plots. In fact, we are interested in the concrete ratio between nodes generated with pruning enabled and disabled at a fixed point of time or event. However, the measures here were taken at the point of time when the first solution was found *or* on the timeout if no solution was found. So there is a tendency that one region, the "easy" one, measures the number of nodes generated until a first solution was found and that another second region, the "difficult" one, measures the number of nodes generated within the timeout. Considering the enormous difference in the ratio between pruning and not pruning, the two measures should be clearly distinguished. In particular, we cannot infer that pruning decreases the node count by 90% from the fact that this holds at a particular point of time before having found a solution.

These insights on both (no) runtime improvement and node generation reduction imposed by pruning are important even though BW is highly inferior to PO, because we may use it to improve PO. Unless we believe that all the nodes pruned in BW are covered by single nodes in PO, we have a natural interest in specifying a valid node comparison relation for PO.

However, even if we would be confident about the gains of pruning in PO, implementing it would still be debatable. On one hand, one can argue that there is probably no gain since the runtime of BW did also not improve by pruning. On the other hand, pruning significantly decrease the number of nodes without consuming more time, which becomes relevant in setups without a time limit where execution is only constrained by memory consumption.

7.1.3.3 Quality Results

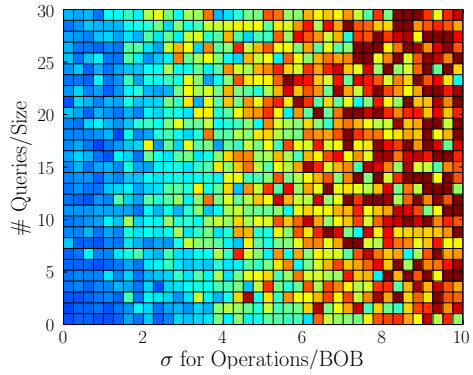
In this section, I present two types of quality results. First, I present results on the *number of different* solutions found within the given timeframe. Second, I present a comparative analysis between the heuristics e_{nf} and e_{fast} based on the Pareto dominance among the Pareto frontiers obtained by using them.

What one does not find in this section even though one might expect it are plots of Pareto frontiers of the non-functional properties. There are two reasons for this. First, we considered four non-functional properties, and we cannot plot Pareto frontiers for more than three dimensions; and even for three dimensions, plotting the frontier is cumbersome. Second, it would be completely unclear on which of the hundreds of setups we analyzed we should conduct such an analysis. As a consequence, I rather decided to show *dominance* results only, i.e. which algorithm parametrization found solutions that dominated all solutions found by another? Since we are neither interested in the concrete values of the non-functional properties nor actually in their relation, this type of comparison is absolutely sufficient for our purpose.

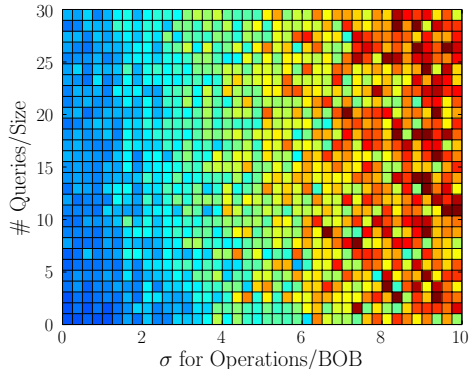
Pareto Front Size on Timeout - BW vs. PO We are now interested in the *diversity* of solutions in terms of non-functional properties, i.e. the size of the Pareto frontier of solutions obtained within the timeframe. Again, we want to first consider the difference between BW and PO in these regards, which are presented in Figure 7.15. The left column shows the Pareto frontier size after 60 seconds when BW was used while the right column shows its size when PO was used. Again, the three rows present the results for minimum solution lengths 2, 3, and 5 respectively. The color map for the plots ranges between 0 (dark blue), 2.5 (yellow), and 5 (dark red).

The first observation is that, in fact, a higher market density implies more solutions within the given timeframe in setups of low or moderate difficulty. Consider the first plot for BW in (7.15a) and the first two plots for PO in (7.15b) and (7.15d). It is evident that, going from the left to the right, we can observe a significant increase in the size of the Pareto front. While the value on the left ranges somewhere between 1 and 2, it ranges between 2 and 5 on the right. This is exactly the tendency one would expect.

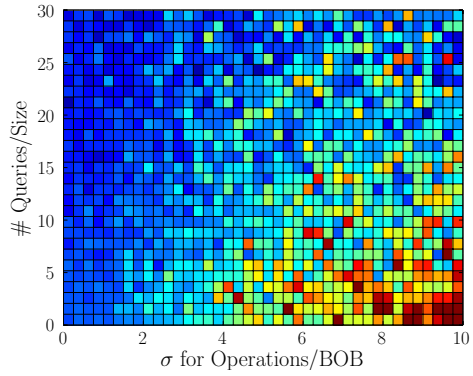
The fact that the values on the very left of the plots are not fixed to 1 but range between 1 and 2 is due to the fact that, in setups with high query potential, some predicates can be produced by several BOBs and hence several operations even with lowest market density. In fact, given that the market density in the leftmost column is 0 and we have exactly one operation for each BOB, one may wonder how there can be multiple solutions at all. This is because, if the solution we originally created in the problem generation process contains a predicate that can also be achieved from the query preconditions using an operation derived



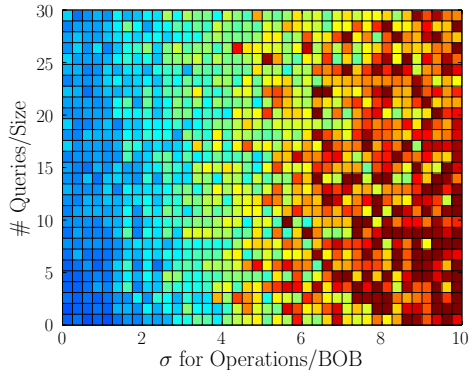
(a) BW, e_{fast}, minlength = 2



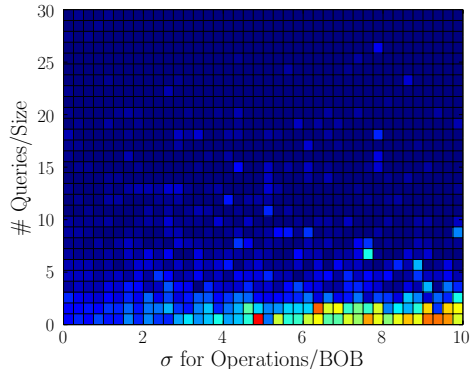
(b) PO, e_{fast}, minlength = 2



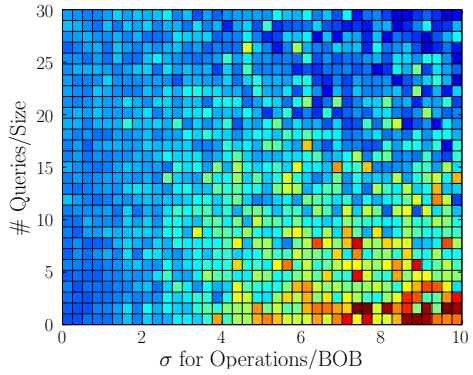
(c) BW, e_{fast}, minlength = 3



(d) PO, e_{fast}, minlength = 3



(e) BW, e_{fast}, minlength = 5



(f) PO, e_{fast}, minlength = 5

Figure 7.15: The size of the Pareto frontier after 60 seconds for the different search structures.

from another BOB, we may find more than only one solution even if for each BOB of the original solution blueprint only one operation was derived. For every predicate, there are 250 possible BOBs that may produce it, which are the BOB only containing the predicate and every BOB containing this predicate and another one. However, the subset of these BOBs that are actually created depends on the query potential, i.e. increasing market potential also increases the probability that a BOB is derived. Also, not all of them will have input types that can be achieved under the respective query. So this side effect is rather limited.

Little surprisingly, in the difficult setups, the number of solutions found at all reduces drastically. So the effect described above only works as long as a significant part of the solutions can be identified. If the search graph becomes too complex, which is also a consequence of a higher market density, the size of the Pareto front decreases to 0.

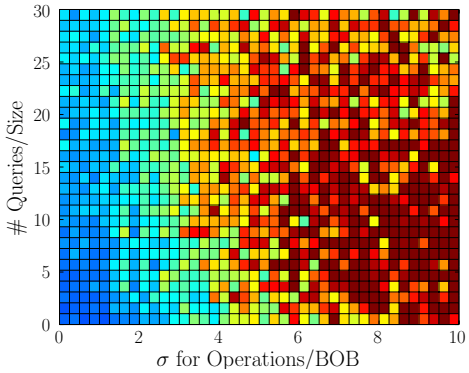
In other words, the effect of the market density on the size of the Pareto frontier is paradox in that it is both positive and negative at a time. While it generally increases the *theoretically* identifiable solutions, it also increases the search graph complexity, which yields a possible decrease or even a vanishing of solutions returned within the timeout. This can be seen particularly well in plot (7.15f). On one hand, in the 5 leftmost columns, we always find a solution but never more than two. Also, in the 5 bottommost rows, we always find a solution, and the size of the Pareto frontier increases going to the right, which we expect. However, if the conditions are adverse, i.e. in the middle and top rows of the plot, increasing market density implies that less or even no solutions are found anymore.

An additional observation we can make is that, if a solution is returned for a setup at all, then the algorithm often returns even several solutions. This holds for both BW and PO. Obviously, we have dark blue regions corresponding to the dark red regions in the runtime plots of Figure 7.7. However, in most other points, we have not only one but even more solutions on average.

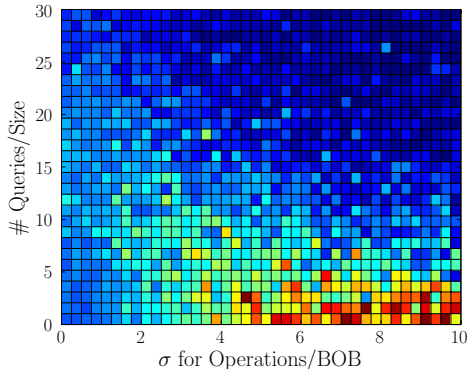
Pareto Front Size on Timeout - BFS vs. e_{nf} vs. e_{fast} After having obtained a first impression of the size of the Pareto frontier in general, we can now turn our attention to the difference imposed by the used heuristic. That is, we would like to know how much the used heuristic influences the *quantity* of found solutions. The corresponding results are shown in Figure 7.16; the results are for PO. The three rows show the results for BFS, e_{nf}, and e_{fast} respectively. The left column corresponds to a minimum solution length of 3 where the right one corresponds to a minimum solution length of 5. Note that the plots in (7.16e) and (7.16f) are the same as (7.15d) and (7.15f) in Figure 7.15 respectively; I show them again in order ease the comparison.

The first impression here is that using a heuristic has only little positive impact on the size of the Pareto frontier except that it enables finding solutions at all. That is, consider the regions in the plots of Figure 7.16 that are not dark blue and, hence, represent empty Pareto frontiers. The rough impression one has is that wherever BFS finds solution at all, it does not find (much) less solutions than e_{nf} or even e_{fast}. In particular, this is the case in the simple setup plotted on the left where BFS even finds significantly much *more* solutions.

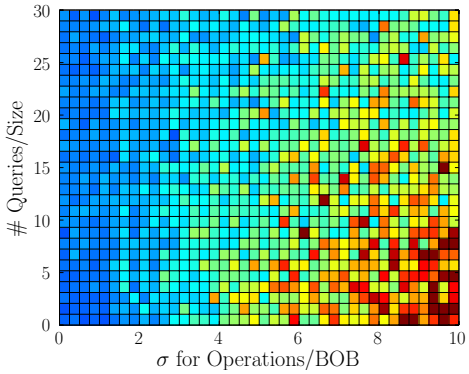
However, a closer look reveals that there actually are some differences with increasing minimum solution length. To see this, consider the plots in the right column of Figure 7.16. Now focus on the border line of the plot for BFS in (7.16b) where roughly 1 solution is still found and compare it to the same line in e_{nf} and e_{fast} in (7.16d) and (7.16f) respectively. Clearly, e_{nf} finds at least 1 solution more in average on that line, and the same holds for e_{fast}. In absolute values, this does not look like much of a difference, but if one can select



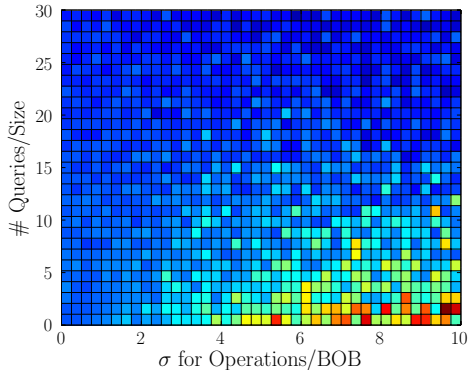
(a) PO, BFS, minlength = 3



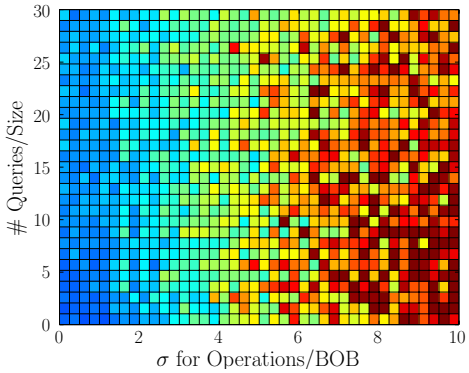
(b) PO, BFS, minlength = 5



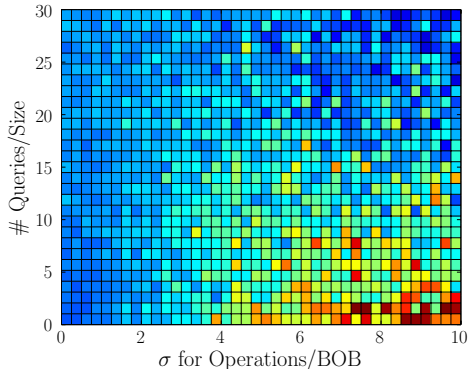
(c) PO, e_{nf}, minlength = 3



(d) PO, e_{nf}, minlength = 5



(e) PO, e_{fast}, minlength = 3



(f) PO, e_{fast}, minlength = 5

Figure 7.16: The size of the Pareto frontier after 60 seconds for the different heuristics.

either between one option or two options, there is much more of a choice in the second case. In other words, for difficult queries, using e_{nf} and e_{fast} may actually yield more alternatives that can be shown to the client.

Another observation is that e_{fast} offers notably more solutions than e_{nf}. Consider the plots in (7.16c) and (7.16e). There is actually no region where e_{nf} finds more solutions than e_{fast}, but there are many setups, in particular the difficult ones, where e_{fast} finds twice as much. The same picture even though less drastic can be seen on the right in (7.16d) and (7.16f).

The interpretation of this could be that the strong heuristic focus has the effect that less different solutions are found. Since BFS is not biased in its exploration, it returns more solutions. Likewise, since e_{fast} is only indirectly biased by the non-functional properties, it seems to be less vulnerable to this effect. Of course, the effect only plays out as long as BFS actually can find all solutions within the time bound and, hence, as discussed above, vanishes in the more difficult setups.

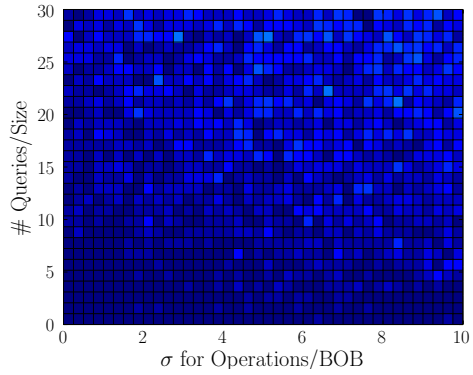
Quality of e_{nf} vs. e_{fast} The last results to discuss on sequential composition are about the qualitative advantage of one of the heuristics e_{nf} and e_{fast}. Intuitively, one would expect that using e_{nf} yields better results within the given timeframe. However, we have seen that e_{fast} offers a broader set of solutions, so there is the chance that one of them is even better than the ones found by e_{nf}. The following results are based on PO but do hold for BW as well.

In order to relate the two heuristics qualitatively, we compared the non-functional properties of solutions on the Pareto frontier obtained using e_{nf} and e_{fast} respectively for each sample point on the grid. Suppose that we obtained the Pareto frontier $\{v_1^{e_{nf}}, .., v_m^{e_{nf}}\}$ using e_{nf} and $\{v_1^{e_{fast}}, .., v_n^{e_{fast}}\}$ using e_{fast}, i.e. each of the entries is a vector of non-functional properties corresponding to an identified Pareto optimal solution. Then we checked whether there was a solution $v_i^{e_{fast}}$ that *strongly* dominated *all* solutions in $\{v_1^{e_{nf}}, .., v_m^{e_{nf}}\}$; that is, for each $v_j^{e_{nf}}$ it holds that $v_i^{e_{fast}} \leq v_j^{e_{nf}}$ and in one dimension the inequality is even strict. If this occurred, we said that e_{fast} won over e_{nf}. If the same condition held the other way around, e_{fast} lost against e_{nf}. Otherwise we scored a draw between the two. Using this scoring technique, we then counted the wins, draws, and losses for all samples over the whole grid.

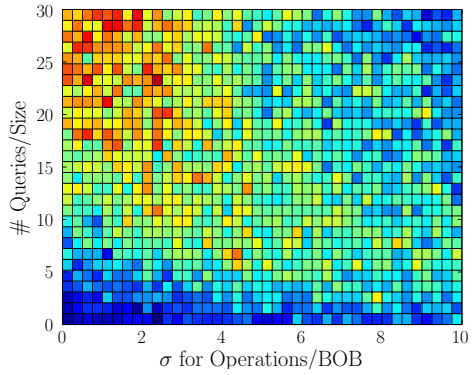
Figure 7.17 summarizes the results of these battles. The three rows show the wins, draws, and losses from the perspective of e_{fast} respectively. The two columns depict the results for different minimum solution lengths in order to show the effect of the minimum solution length on the battle results. For example, (7.17b) shows in which setups e_{fast} was better than e_{nf} and how frequently that was the case. The limit of the color scheme here is the number of samples, i.e. 50. Naturally, the sum of each point of the three plots in a column yields a constant value (the number of battles).

First, the most *obvious* observation is that the minimum solution length has a tremendous impact on this question. Consider the difference between (7.17c) and (7.17d). While the two heuristics deliver almost equally good results for a minimum solution length of 2, the results differ much more in quality for a higher minimum solution length of 5. In fact, for higher minimum solution length, we only have a draw in very simple and very hard settings. The first is probably the case because one returns the optimal solution regardless the used heuristic. The second is probably the case because we do only find one or even no solution within the given timeframe such that there are not much candidates that could dominate the opponent's solutions. Hence, for queries that have a short solution, any of the two heuristics finds a Pareto optimal one.

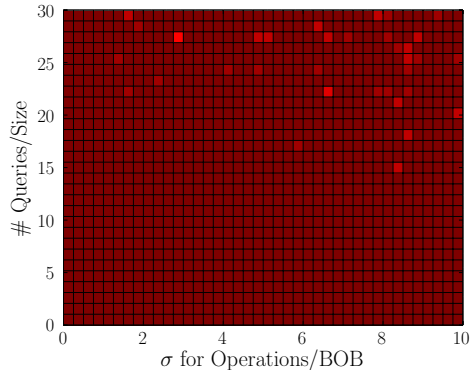
Second, the most *notable* observation here is that e_{fast} finds dominant solutions more often



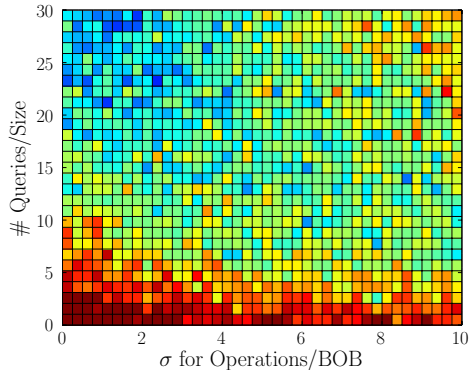
(a) PO, minlength = 2, e_{fast} beats e_{nf}



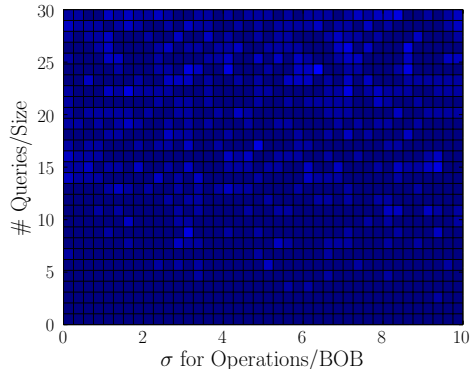
(b) PO, minlength = 5, e_{fast} beats e_{nf}



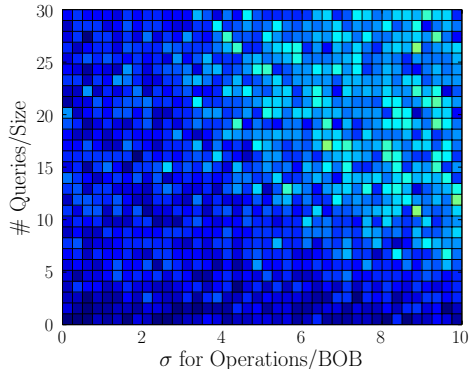
(c) PO, minlength = 2, draw



(d) PO, minlength = 5, draw



(e) PO, minlength = 2, e_{nf} beats e_{fast}



(f) PO, minlength = 5, e_{nf} beats e_{fast}

Figure 7.17: Best solutions per test set between e_{fast} vs. e_{nf}.

than e_{nf}. While e_{nf} has only a chance to win in the case of moderate or high query potential and moderate or high market density, e_{fast} has almost the same chances in those setups and clearly wins in most of the setups with high query potential and *low* market density. In other words, if there is no draw, e_{fast} tends to find better results than e_{nf} in those difficult regions. This is somewhat against what one would have expected. An explanation could be that optimizing for non-functional properties, which is multi-objective optimization problem, is highly non-trivial such that the strategy to find dominant solutions by simply finding as many as possible seem to play out quite well.

Summarizing, we can see that using e_{nf} does not provide a high probability or even a guarantee to find better solutions than e_{fast} within the given timeframe. Quite the contrary, in most runs that did not end with a draw (also considering the minimum solution lengths not shown here), we found better solutions using e_{fast}. In fact, one is inclined to say that, if one dominates the other at all, then e_{fast} delivers better solutions on average.

7.1.4 Summarizing Discussion

The above results and observations can be summarized as follows. Of course, the interpretations are limited to our implementation, which may have introduced a bias.

First, PO significantly outperforms BW. This applies for any considered setup. As a consequence, we can also say that, for our implementation, the disadvantage of BW of producing more nodes is not compensated by the advantage of pruning.

Second, finding short compositions can be done more or less efficiently, but finding more complex compositions only works in limited settings. Solving queries with minimum solution lengths up to 5 can always be done in within one minute using PO for setups of the form considered in this evaluation; queries with minimum solution length of 3 or less can be solved in at most 20 seconds. However, for more complex queries, one must possibly calculate higher response times.

Third, trying to incorporate dominance pruning to PO may or may not help. Pruning is highly effective for BW in terms of space but not much in terms of time. If the effect is the same for PO, it is not clear whether it is worth to make the effort. In particular, memory overflows were not a problem in any of the setups, which again decreases the motivation to go into this direction. On the other hand, even though small, there *was* a slight effect of pruning in BW also for runtime, which may be a motivation to move here.

Fourth, e_{fast} is better than in e_{nf} in virtually every aspect. Not only is it faster than e_{nf} in finding the first solution and explores less nodes but even finds both *more* and *better* (Pareto optimal) solutions within a given timeframe. That is, e_{fast} beats e_{nf} even in the measures where we would expect it do be vice versa. Even though this advantage vanishes in difficult settings, it is quite visible for many setups and suggests that e_{fast} should be preferred to e_{nf} no matter what the overall objective is.

Summarizing, the predominant solver configuration for our setups is to use PO with e_{fast} independently from the domain difficulty and the query. Using this configuration, we can find solutions for queries with minimum solution length of about 10 in a range between some seconds and some minutes, depending on the density of the domain and the market. This clearly suggests a positive answer to the initial research question on the practical feasibility of automated service composition. But once again, this assertion is limited to our implementation and the synthetic setups we used. Even though nothing points into this direction, the solvers may exhibit different behaviors in real world applications.

7.2 Experimental Analysis of Template Instantiation

This section presents the results of the practical experiments we carried out for the template instantiation technique presented in Section 6.2. Again, the research question that drove this analysis was whether instantiating domain independent templates is efficiently possible.

In contrast to the analysis on sequential composition, this study is much more compact. First, the analysis space is much simpler, because there are only two sources of complexity, which are the size of the vocabulary, i.e. predicate names, available to fill these on one side and their density on the other hand. Second, in contrast to the experiments we conducted on sequential composition, the synthetic part of the experiments is very small. Third, we do not have different algorithm setups, because the template instantiator cannot be parametrized and does not use interchangeable heuristics. Also, we are not interested in several different measures but only in runtime. All these aspects allow us to answer the above research question with much less effort than in the case of sequential composition.

The following results and discussions for template instantiation are largely taken from my previous publication on the topic [89]. I only adapted it partially in order to fit to the limited presentation of the instantiation algorithm in this thesis.

The overall result is twofold. On one hand, the approach *is* feasible in reasonable time if the number of ways in which a goal can be reached is rather small. On the other hand, if this condition does not hold, the runtime of the basic instantiation technique increases very quickly and requires a refinement.

The section is organized in three subsections. I describe the experiment setup in Section 7.2.1, the results in Section 7.2.2, and discuss the results from the viewpoint of integration into sequential composition in Section 7.2.3.

7.2.1 Experiment Setup

We used two templates and instantiated them using different knowledge base sizes. I first describe the templates, the used background knowledge, the available services and the queries used in the evaluation. Then I describe *how* the experiments are conducted.

7.2.1.1 Templates, Background Knowledge, Operations, and Queries

Our evaluation is based on two templates. The first template is the Filter template introduced in Section 6.2.1 in Figure 6.2. The second template is shown in Figure 7.18 and is blueprint for workflows that determine, for a given set A, the element a^* that has the maximum (or minimum) value of a particular property among all the elements of the set. This property is determined by a generic operation call and compared in a generic Boolean expression, which is usually resolved to $\leq$. For example, it determines the object with the maximum (or minimum) price in a given set.

The PeakFinder template has five constraints. First, it requires that successfully comparing the results of the operation invocations for two different items a and a' through the test F must yield the effect predicate $R(a, a')$. Second, the test predicate F must be transitive. Third, it must not be possible to infer $R(a, a')$ without considering the determined property of a or a' in the test respectively. Finally, it must not be possible to infer $R(a, a')$ only based on the effect of the generic operation invocation.

Name : PeakFinder
Inputs : A
Outputs : a^*
Precondition: $\{P_C(A)\}$
Effect : $\{R(a^*, A)\}$
Constraints$^+$: $\{E_s(x,y) \wedge E_s(x',y') \wedge F(y,y') \rightarrow R(x,x'),$
$F(x,y) \wedge F(y,z) \rightarrow F(x,z)\}$
Constraints$^-$: $\{E_s(x,y) \wedge E_s(x',y') \wedge F(y,y'') \rightarrow R(x,x'),$
$E_s(x,y) \wedge E_s(x',y') \wedge F(y',y'') \rightarrow R(x,x'),$
$E_s(x,y) \rightarrow R(x,y)\}$

```
1  begin
2  |  a* := nil
3  |  tmp := nil
4  |  for a ∈ A do
5  |  |  (y) := s(a)
6  |  |  if F(y, tmp) then
7  |  |  |  a* := a
8  |  |  |  tmp := y
9  |  |  end
10 |  end
11 |  return a*
12 end
```

Figure 7.18: Template to select a distinguished object from set.

Besides the symmetry of $=$ and the transitivity of $=$ and $\leq$, the *basic* background knowledge Ω contains only the following two rules:

- $\mathit{AvailabilityOf}(y,x) \wedge y{=}\text{'yes'} \rightarrow \mathit{isAvailable}(x)$
- $\mathit{PriceOf}(y_1,x_1) \wedge \mathit{PriceOf}(y_2,x_2) \wedge y_1 \leq y_2 \rightarrow \mathit{CheapestOf}(x_1,x_2)$

where the semantics of $\mathit{CheapestOf}(u,v)$ is that u is the cheapest of the books of u and v where v may be a book or a set of books. Basic here means that this is the background knowledge before additional synthetic clauses are inserted. For the actual experiments, we *inflate* this basic knowledge. We describe how this is done below in the part of evaluation conditions.

We consider two operations in this setting. The first one is *getAvailability* with input b, output a, precondition $\mathit{Book}(b)$ and effect $\mathit{AvailabilityOf}(a,b)$. The second one is *getPrice* with input b, output p, precondition $\mathit{Book}(b)$, and postcondition $\mathit{PriceOf}(p,b)$. The number of operations has no effect on the runtime except for the last step of the instantiation algorithm, which is not the bottle neck here. Hence, there is no need to generate artificial operations, and we consider only the operations relevant to solve the queries.

We need two queries for the evaluation. The first query q_1 is the one specified in the running example, that is $\mathit{Post}_{q_1} = \mathit{Book}(M) \wedge \mathit{isAvailable}(M)$. The second one q_2 asks to identify the cheapest one of a set of books, formally $\mathit{Post}_{q_2} = \mathit{CheapestOf}(m,M)$. Obviously, q_1 can be (only) answered instantiating the Filter template with *getAvailability*, and q_2 can (only) be answered instantiating the PeakFinder template with *getPrice*.

7.2.1.2 Evaluation Process

Given the above environment, we conducted several experiments for different scenarios. We conducted experiments for every scenario resulting from a combination of the following parameters:

- *Considered Template (*Filter *or* PeakFinder*).*
- *Number of possible different solutions for the query (1, 2 or 3).* Here, possible solutions means that there exists a template that can be instantiated to this number of completely different solutions. The number does, however, not say whether the *considered* template can be instantiated to a solution at all. The role of this parameter is to simulate the density of the domain. The more ways exist to achieve a particular solution (on any template), the more partial instantiations are possible (even if these cannot finally be completed to solutions).
- *Goal state (*$Book(M) \wedge isAvailable(M)$ *or* $CheapestOf(m, M)$*).* The purpose of the this point is to simulate the instantiation for a template for both the case that a solution exists and the case that no solution exists.

For each of these scenarios, we performed repeated experiments for different sizes of *inflated* background knowledge bases. The inflated knowledge bases extend the basic knowledge base by synthetically generated rules. Rather arbitrarily, we decided to let every such rule define the symmetry property of a new binary predicate that does neither occur in a template nor in the basic knowledge base. It would have also been possible to use any other rule, such as transitivity, or no rule at all but only specifying that a predicate is true for some constant. Since the predicates introduced by these rules are new, they must be considered by Step 3 of the instantiation algorithm and produce additional workload. However, they cannot affect the solution itself, because they are not connected to the predicates within the basic knowledge base that are relevant for the solution.

Instead of performing an experiment for every possible size of knowledge bases, we group the different sizes into blocks of 10 and repeatedly increase the number of blocks. That is, the first experiment works on the basic setting without additional rules, the next includes 10 rules, the next 20 rules, etc. Increasing the number of rules in each turn, we ran 50 experiments for every profile. In order to avoid outlier results, we performed every experiment 100 times with different choice of the synthetically generated predicate names.

As described above, we address the satisfiability checks with resolution [104]. If the formula is detected to have Horn structure, we use unit resolution, which requires that one of the clauses used for resolution must have size one. The benefit is that this procedure is very efficient as long as it is applicable. If unit resolution is not possible, we apply "normal" resolution, which at least decides the query on this formula class.

We measured the time that was required to find a valid instantiation. Our algorithm is implemented in Java 8 and was performed on an Intel® Core™ i7-2600 with 3.4 GHz CPU and 8.0 GB memory in a MS Win 7 64 bit environment.

7.2.2 Results

Each of the figures contains the diagrams that describe the instantiations of one of the two templates. Figure 7.19a and Figure 7.19b show the results of instantiating the template Fil-

TER and PEAKFINDER respectively. The results for query q_1 are shown on the left and the ones for q_2 on the right of each figure.

Every diagram shows, for different knowledge base sizes, the time that was necessary to decide if the template can be instantiated to a solution. The abscissa is the size of the knowledge base as the number of different predicates occurring in it. The ordinate is the time in milliseconds that was necessary to either find a valid instantiation that satisfy the query or to prove that no such instantiation exists for the template.

The different colors correspond to the number of solution instantiations for *any* template. The green line represents the case where a template exists that can be instantiated in exactly one way to solve the problem. The blue and red line represent the cases where there exists a template that can be instantiated to 2 or 3 solutions respectively. For each of the cases, there is a thick line representing the *mean* of the 100 runs of the respective size of the background knowledge as well as a background color for the area between the *median* and the *maximum* value; i.e. 50% of the runs are contained in the colored areas.

The first observation is that, for the case that a solution exists, the time to find one increases roughly linearly. This case corresponds to the diagrams on the left of Figure 7.19a and the right of Figure 7.19b. Even though the experiments show a great variance, even the maximum can still be bound linearly. The variance can be explained through shuffle operations we perform on the database in a preprocessing step in order to avoid a bias based on the input structure. Note that the variance for the FILTER template is significantly higher than for the PEAKFINDER.

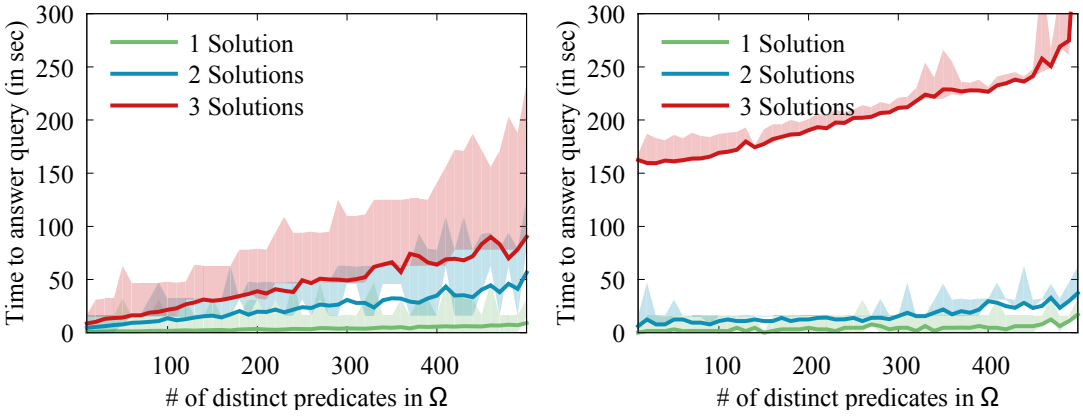
The fact that the increase search space does not significantly increase the runtime is not surprising. The solutions are always the same and the time to find them is only delayed by some preprocessing that causes linear overhead. Once the preprocessing is done, the solutions are almost found immediately.

Second, we can see that the time to prove that no solution exists for the template increases dramatically with the number of possible solutions (that exist for other templates). This case corresponds to the diagram on the right of Figure 7.19a and the diagram on the left of Figure 7.19b. While for one or two solutions, the time is similar to the case where a solution is found, in the case of three possible solutions, we already need 6 times more time. For a setup of 500 predicates in the knowledge base, this already implies an expected computation time of 5 minutes for the FILTER template. In fact, we conducted the experiment also for more solutions and observe a clear exponential growth. In contrast to the affirmative case, the variance here is quite small. The reason for this is that every candidate for predicate bindings must be considered, so the number of candidates considered in each experiment is equal (for the respective level). So, in a way, the mean can be seen as an approximation of the upper bound here.

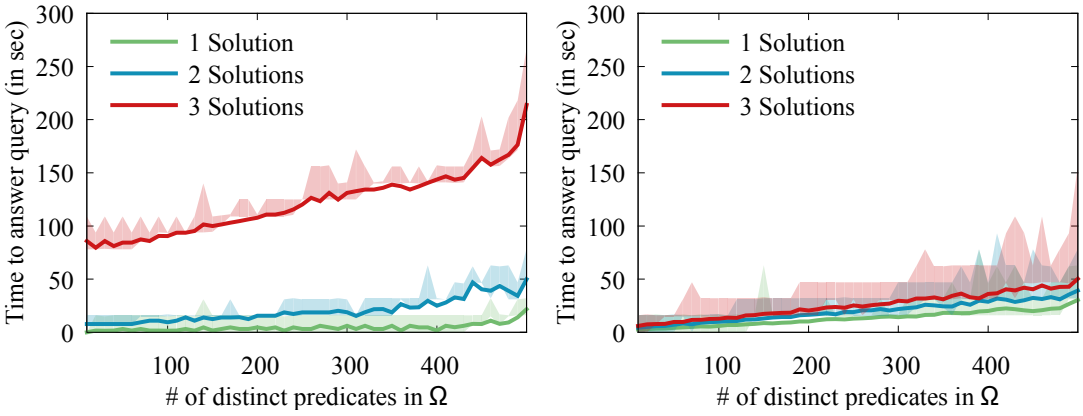
What we can conclude from this observation is that, not surprisingly, higher density in the domain increases the workload for the instantiation of a template even if the template cannot finally be instantiated to a solution. This is simply because the set of candidates for predicate bindings is higher even though none of them passes the validation (in this case because either the positive constraints are satisfied on the propositional but not on the FOL level).

In general, we can see that the instantiation process works faster for PEAKFINDER than for FILTER. In the case of success, we can find the solution faster for PEAKFINDER than for FILTER, and for the case of failure, we can find the proof faster. For large vocabulary sizes, instantiating PEAKFINDER goes almost twice as fast as instantiating FILTER.

Our analysis showed that the reason for the better performance is precisely the more com-



(a) Time necessary to decide if the FILTER template can be instantiated to a solution. The answer is positive on the left and negative on the right.



(b) Time necessary to decide if the PEAKFINDER template can be instantiated to a solution. The answer is positive on the left and negative on the right.

plex structure of the PEAKFINDER template. This sounds counter intuitive, but it turns out that the additional constraints of the template impose that less candidates pass the validation step in of the instantiation algorithm, so less bindings are constructed. Put differently, the results suggest that the more complex templates (i.e. templates with more negative constraints) allow for more effective pruning.

7.2.3 Summarizing Discussion

The overall conclusion that can be drawn from our observations is that the proposed template based composition method is computationally expensive but not in a hopeless manner. Having analyzed several cases, we can see that there is a number of settings where we have a good chance to find a valid solution in reasonable time if one exists. While we can improve the time to find a valid instantiation, the actual challenge is to reduce the time necessary to prove that no valid instantiation exists. But even without such optimizations, the algorithm can already be used as is for small and medium sized environments.

The problem is not so much with a single template but with a whole set of them. If we receive a query and can ground the first template we use to a solution, then everything is fine. But figuring out, which template would be appropriate is not trivial and maybe impossible in some cases. Then, we may first try a bunch of templates that do not work, and the working one is somewhere in the middle or even at the end of the chain. In the above case, if for q_2, if we first use the FILTER template, then we lose up to 300ms (in the last case) before switching to the "correct" template, which is then instantiated in less than 50ms.

Fortunately, we can completely parallelize the instantiation processes for several templates. That is, we may run the instantiation routines separately for each template. Since the number of this kind of highly general templates can be assumed to be rather small, this should be a reasonable option in almost every practical setting.

The observation that solutions can be found relatively fast if they exist suggest a simple timeout strategy. That is, set a timeout to "some few" seconds and wait for an answer until the timeout is reached. If no solution was returned, the chances are good that there is no solution. Of course, our viewpoint is the positive one: We are not interested in a proof that no instantiation exists, but only want an affirmative answer as fast as possible if one such can be given. In this sense, the results show exactly what we need.

Also, it is important to keep in mind that the figures reflect results of a still rather rudimentary search technique. Each of the steps of the instantiation algorithm has potential for optimization, in particular the one for satisfying the template constraints. In its current form, we did not apply any particular heuristic or highly sophisticated encoding or pruning techniques. So the figures show the results of a very straight forward implementation of the instantiation mechanism, and there is much space for improvement.

There are several concrete techniques that can help to practically reduce the runtime by some orders of magnitude. Potential improvements can be particularly made in the different steps of satisfying the template constraints. Currently, we only use a rather small part of information for pruning candidates of the propositional bindings. A more sophisticated analysis of which predicates may be relevant could significantly improve the set of considered candidates. Also the argument binding can be improved by pruning away combinations that do not work on single rules. For example, the argument binding could be created step by step per predicate and evaluated for the rules that contain the respective predicates. In this way, large numbers of argument bindings could be pruned. Going further into that direction and implementing this type of optimizations, it should be possible to achieved instantiation times of less than a second for all of the positive cases.

This type of optimization is indispensable for an integration into the sequential composition algorithms. We have seen in the previous section that the instantiation routine creates hundreds or even thousands of nodes. If one instantiation takes 10 seconds per template, checking the instantiation in one node would yield an unacceptably high overall runtime. So incorporating the above mentioned optimizing modifications is mandatory in order to make the approach work together with sequential composition.

But even if we achieve this optimization, there will probably be the need to use a smart instantiation system during search. Suppose that we are able to obtain instantiation times of 100ms per template. Even with such a highly optimized instantiation technique, the overall runtime would become very large if we execute the instantiation in each of the nodes. However, it should be possible to entirely avoid calling the instantiation algorithm under certain conditions. More precisely, we may be able to define equivalence classes of search nodes where two nodes are in a class if and only if a template instantiation for one of them has been

successful. If we can easily decide the membership of these classes, we can limit the number of nodes for which we invoke the instantiation routine at all.

Summarizing, the evaluation shows that the proposed template-based composition algorithm is ready to be tried in practice in isolation but must be enhanced if it is used in complex environments and, in particular, if it is integrated into sequential composition. Possible enhancements are the usage of heuristics, search space reduction, and timeouts. In spite of any optimization techniques used in the instantiation, the integration into sequential composition will only be practical with an intelligent management module that controls and limits the invocations of the instantiator.

8. Related Work

Automated composition as I understand and treat it in this thesis is an application area of the three large research fields *planning*, *search*, and *theorem proving*. In the very first place, automated composition is a form of automated planning. So every composition algorithm *is* a planning algorithm and can be used for other planning problems of the same kind, and we can solve automated composition through *existing* (standard) planning algorithms. At the same time, planning is closely tied to the field of *search* and *theorem proving*, so these areas are touched likewise.

On the other hand, software composition does have specific aspects that are not so common for other applications in the respective fields. Most notably, these are the creation of new constants, absence of closed world assumption, sensing abilities, non-scalar solution costs, etc. Hence, even though moving within the realms of quite elaborated research fields, software composition comes with its particularities, which also deserves a bottom up approach, i.e. developing the solution coming from the application domain, and not only a top down approach, i.e. making the problem somehow fit for existing techniques. This insight has led to a likewise large research field known as (automated) software/service composition and program synthesis.

In the following, I first discuss the position of my approach within the field of automated composition and then discuss relationships to achievements within the three fields of planning, search, and theorem proving. Making this distinction is far from being straight forward, because composition algorithms can (sometimes) serve as general planners or theorem provers and vice versa; hence, there are really no clear borders between the areas. The idea is as follows. In Section 8.1, I relate my work to approaches that focus on an application of their technique to the composition problem; i.e. that put a strong emphasize on the composition problem instead of solving planning or search problems in general. Opposed to this, in Section 8.2, I discuss general achievements of the major research fields and in how far these are relevant and connected to the problem of software composition as tackled in this thesis.

8.1 Related Work in the Field of Automated Composition

In the following, I discuss approaches that dedicate themselves to a significant degree to the problem of composition. That is, approaches that ground the developed techniques (even if applicable for other problems) in the scenery of automated composition.

The section is divided into four subsections according to the type of composition problem that is tackled. First, Section 8.1.1 discusses cases where the structure of the composition is already given in advance in form of a *template*. These approaches have nothing to do with what is done in this thesis, but they form a large subfield in the area of automated composition, which is why I discuss them. Second, Section 8.1.2 discusses approaches where the complete setup is (efficiently compilable into) a *propositional* model. This is a simplified case of the composition problem described in Section 2.1 in the sense that postconditions do not relate

inputs to outputs (either because inputs and outputs are empty, or by explicit restriction). A special case of this setting that uses modal (temporal) logics to describe goals and that does not fit into the composition problem in Section 2.1 is discussed in Section 8.1.3. Finally, Section 8.1.4 describes the approaches to software composition that address more or less the same problem as the one addressed in this thesis in the sense that they work in a *predicate logic* setting.

8.1.1 Composition With a Solution Template

The composition problem addressed in this thesis constitutes a (class of) planning problem(s). A planning problem is often formalized as a tuple $\langle \Sigma, s_0, S^* \rangle$ where Σ is a state transition system, s_0 is a state of Σ representing an initial situation, and S^* is a set of states of Σ representing goal states. The connection between composition and planning problems is quite obvious when we interpret Σ as the state transition system induced by the mechanic of applying operation invocations in states or using clauses of the background knowledge to reason about data containers and domain constants. Then, s_0 is simply the query precondition, and S^* is the set of states implied by the query postcondition.

However, other people have different understandings of what a composition problem is. In fact, the field of automated service composition can be split into two large subfields [88]. Approaches in the first field address some kind of planning problem of the above type and are some variant of the problem formalized in Chapter 2. In contrast, approaches in the second field assume that we already know the (rough) structure of the desired solution. The problem is then not given by an initial state and a goal state but by some kind of abstract workflow that has placeholders and a set of operations that can be used to replace these placeholders.

Approaches within the second subfield are motivated by different goals. Given the abstract workflow, they try to find replacements of the operation placeholders such that

- the instantiation is optimal with respect to non-functional properties [6, 11, 18, 30, 127];
- the selected operations satisfy behavior requirements [74], dependencies (or avoid conflicts) [2, 10, 39], or satisfy domain specific business constraints [21, 53, 115]; or
- the placeholder may be refined (possibly recursively) by complex sub-workflows [8, 85, 109, 126]

In fact, the goals are orthogonal and could be pursued together, but this is rarely done. One example is [39], which combines the requirement of realizing compositions with transactional properties and, at the same time, tries to optimize for quality of service.

In contrast, approaches that do not assume the solution structure given rather aim at finding a valid composition at all. That is, they assume that a precondition and postcondition are given in some logic form and then search for a composition that provably achieves the postcondition based on the precondition. In order to obtain this proof, the available operations are equipped with preconditions and postconditions themselves.

Having these approaches mentioned, there is no need to compare them to the work carried out in this thesis in more detail. The availability or absence of the goal structure imposes such a striking difference that there is no point in comparing approaches across the subfield borders. Indeed, some approaches in the first subfield even apply planning techniques, e.g. [126]. However, they use hierarchical task networks (HTN), which do not apply in planning

problems of the above form. In the following, I only consider settings where no solution template is given.

8.1.2 Synthesis Based on (Quasi-)Propositional Specifications

Many composition approaches, e.g. [7, 9, 12, 13, 32, 52, 60, 63, 70, 73, 75, 106, 114, 124, 128], define or can be transformed into set-theoretic planning problems in polynomial time. In the latter case, the problem description may contain unary predicates, which can be efficiently ground to a finite set of propositions for any finite set of constants. Clearly, given m unary predicates and n constants, we obtain mn many propositions from grounding the predicates. Note that these approaches address a special subproblem of the composition problem defined in Chapter 2 where preconditions and postconditions of operations contain propositions or unary predicates, and Ω is empty or contains only rules with propositions or unary predicates. Hoffmann et al. provide a polynomial reduction of composition under strict forward effects to conformant planning, which directly implies the possibility to compile the above approaches into set-theoretic planning problems [59].

Given one of the above composition problems, the corresponding set-theoretic planning problem looks as follows. The planning domain, given by a state transition system Σ, consists of three elements:

1. the state space contains the states, which are the possible sets of propositions or ground unary predicates;

2. the action set contains the (ground) operations and possibly the ground implication representations of the clauses (if these exist and if these are not encoded in the *problem* (see below)); and

3. the state transition function whose design depends on the question whether or not the composition domain satisfies the closed-world assumption (CWA). In general, the state transition function will add the positive literals of the operation effect to the state to which the action is applied. If negative effect literals exists, either their positive conjugation will be *removed* from the state if CWA applies, which is generally not the case as discussed in Section 2.2.1, or the negated literals themselves will be *added* to the state if CWA does not apply.

The planning problem is the above planning domain together with an initial state (or set of possible initial states in the case of conformant planning) and a goal state.

In the following, I distinguish these quasi-propositional approaches between simple and complex ones. Simple approaches assume that the descriptions

8.1.2.1 Simple Approaches

Many approaches address a positive and deterministic planning problem. Here, operations have deterministic postconditions and preconditions and postconditions contain only positive literals, i.e. knowledge that is achieved once remains valid forever. On one hand, this holds for type-production approaches [7, 9, 13, 52, 75, 114, 124, 128]. The semantics of a query in this setting is "given that we know $A_1, .., A_m$, find a sequence of actions (or type derivations) such that we know $B_1, .., B_n$". Actions require the existence of types and provide (objects of) new

types. Other approaches consider (positive) propositional preconditions and postconditions [70, 73, 106].

On the functional level, these problems can be easily encoded into PDDL and solved by every standard planner such as FF [62], LAMA [103], or Fast Downward [56]. Since all literals are positive, the question whether or not CWA holds is irrelevant; in particular we may apply tools that assume CWA to be true, which is the case for the above planners. The only additional aspect sometimes considered are non-functional properties and service-level agreements, which cannot be formulated in a straight-forward manner in PDDL.

The common assumption of these approaches and the model in Chapter 2 is that no delete-lists are used. In this thesis, this is due to the IRP assumption; I assume that data containers are not overwritten and that knowledge that has been obtained once never becomes invalid. This means that operations do not remove knowledge, so no delete-lists are necessary.

However, there are some crucial differences between those approaches and the one presented in this thesis. First, there is an enormous difference in the problem complexity. While the restriction to positive set-theoretic planning makes the problem solvable in polynomial time [40], the problem tackled here is undecidable (cf. Section 2.3). Second, from the viewpoint of service composition, the above approaches produce compositions that are little meaningful and sometimes not even executable. Some approaches ignore the inputs and outputs of operations [70], so no data flow is generated, and we do not obtain any executable composition. In general, these approaches work well and are sufficient if the semantics of a service can be captured in the involved data types or in propositional conditions.

8.1.2.2 Complex Approaches

Composition based on propositional descriptions is not always trivial. Things become instantly more difficult when negations or disjunctions (or both) are allowed in preconditions and postconditions of operations. In the following, I discuss four approaches that consider *uncertainty* in the form of disjunctive operation postconditions [12, 32, 59, 63].

The special property of these approaches is that they do not only find sequential compositions but also compositions with alternative branches. This is opposed to the simple approaches discussed above whose solutions are only sequences of operation invocations. Except [59], the following approaches generate tree-shaped compositions, where a node with more than one successor corresponds to a case distinction made at runtime. In [59], the output is still a sequential composition, but only a subset of the contained operation invocations will be really invoked at runtime; it is left to the execution environment to determine the applicability of the concrete operations at runtime.

Partial Matches Constantinescu et al. were the first to consider a more complex propositional composition problem through the notion of *partial matches* [31,32,33]. The assumption here is that types may *overlap*, i.e. their domain is the *union* of other types. For example, suppose that there is a type t and every element of type t is t_1 or t_2. If operation o_1 provides a t and operation o_2 requires a t_1, then the input condition may or may not be satisfied at runtime, because the actual output value of o_1 may result t_1 but also t_2. In this sense, the input of o_2 partially matches the output of o_1.

Their solution is based on a search in an AND-OR-tree where AND-nodes are induced by the concrete values a types may take. They encode the knowledge about which domain constants belong to which type in form of finite sets (in fact, they use intervals of reals, but

somehow these must be made finite). For example, a type definition $t = [t_l, t_u]$ is translated into some finite type domain $t = \{t_1, .., t_m\}$ with $t_1 = t_l$ and $t_m = t_u$. Now, instead of encoding the type information of an operation as a type literal, they use a literal for each possible *value* of a type and pose it as a disjunction; i.e. they encode the knowledge about the type domain into the operations. For example, an operation with output type t receives a postcondition of the form $t_1 \vee .. \vee t_m$ instead of simply t as I model it in this work. Applying such an operation yields an AND-node with one outgoing edge for each of these types. A solution tree rooted in such an AND-node induced by an output, say v, is referred to as a *switch*. A switch is seen as a complex service (the result of the composition) that is able to achieve a part of the original solution for every possible concrete value of v; so the switch *covers* all possible types values of v.

Later, Hoffmann et al. proposed a solution to a very similar setting through the notion of *strict forward effects* [59]. Instead of encoding the type knowledge into the operations, they represent the type definition in form of clauses belonging to background knowledge, which they call *integrity constraints*. Here, types are not defined in terms of their possible value but in terms of other types from which they form a union. The composition problem is encoded as a standard conformant planning problem, which is the problem of generating plans under uncertainty about the initial state and the operation postconditions. It has been shown that conformant planning can be solved by state-space search when states are lifted to *belief states*, which are simply sets of states [15]. The presence of standard tools for conformant planning such as Conformant Fast Forward (CFF) makes it reasonable to compile the composition problem into a conformant planning problem [61]. Hoffmann et al. prove this compilation to be sound and complete, which at the same time proves that all the above settings can be reduced to a propositional scenario, and apply CFF to solve it.

Non-Deterministic Operations A second line of research considers operations that actually *do* have disjunctive operation postconditions. This is opposed to the setting of partial matches where the operations themselves have deterministic effects and where uncertainty is *injected* by a formalization of type definitions.

Of course, there is a close connection between non-deterministic operations and partial matches. More precisely, if we have that $\neg P \vee \alpha_1 \vee .. \vee \alpha_n$ where P is a literal and $\alpha_1, .., \alpha_n$ are formulas, then a presumably deterministic operation with postcondition P also entails uncertainty if we exploit the above knowledge (as in the case of partial matches). In terms of what is infer*able*, it does not matter whether disjunctions occur directly in the postconditions or in the background knowledge in connection with literals that occur in the disjunction-free operation postcondition; this is just a matter of syntax and, of course, a matter of whether one wants to allow background knowledge or not. In the following, we consider operations that have postconditions with disjunctions themselves and no background knowledge is used. Compiling the disjunctive postconditions away into background knowledge clauses would yield the same setting as the one of partial matches (except the intended meaning of literals). Nevertheless, the algorithms for the two problems need to work differently, and the encoding may have a significant impact on the performance.

A lot of work in this line of research was carried out by Pistore, Bertoli, et al. [12, 97, 98, 100, 101]. In their papers, the disjunctive character of operation postcondition is captured in a non-deterministic state transition function of a state transition system (STS) that describes the behavior of a service whose operations correspond to the actions in the STS. Here, states are described by sets of propositions that typically describe the value of a variable, e.g. $x = 4$ [12]; note that this whole statement is seen as a proposition and not as a binary first-

order literal. So operations are not considered in isolation but as a part of a service whose state becomes modified through the invocation of their operations. For a set of services, the joint STS, which they call the *parallelized* STS, constitutes the environment the composition algorithm is supposed to interact with. The goal of composition is to find a *controller* for that parallelized STS that drives it into a goal state in as many cases as possible. Formally, the controller corresponds to a subgraph of the graph induced by the parallelized state transition system.

The approach is partially solved with tools that even allow to find cyclic plans, but it is not clear in how far compositions with loops can be found. For example, in [97], the composition problem is tackled with the MBP planner, which *is* able to find strong cyclic plans, i.e. plans that may contain a special type of loops [25]. Intuitively, this suggests that they are able to create compositions that may contain loops, and the authors say in the formal part of the paper that "We are interested in complex plans, that may encode sequential, conditional and iterative behaviors". However, in their own examples loops do not occur, and in [12], it is explicitly said that controllers do not have loops. In particular, they show that (at least some) service composition settings require a different definition of the StrongPreImage primitive, which is used to find strong (cyclic) plans in [25] (i.e. plans that *always* yield a goal state regardless the non-determinism of the domain). I am not aware that compositions with loops were ever constructed in this line of research.

Hoffmann et al. presented a second approach very similar to these works called SAM [63]. While Pistore and Bertoli returned to the setting of finding strong plans only [12] (an exception, CTL and EAGLE, is discussed in the next section), SAM allows for both strong and weak planning. Goals are specified in the usual way, i.e. as a set of propositions. They provide an algorithm to find strong plans and one to find weak plans. For weak plans, it is acceptable that goal literals are not achieved if it is proved that they *cannot* be achieved. Intuitively, this means that either a plan is provided or an explanation is given why this is not possible for some parts of it. Strong plans are searched via AO*, and the search for weak plans is carried out by a novel algorithm called SAM-AO* and related to AO*.

The above approaches work best in settings where relevant design decisions on the involved services and the data flow have been made in advance. In fact, the approaches require that the set of services that will be involved in the solution is already predefined. For example, these are the Producer and Shipper in [97, 100, 101] and the Customer Quote business object in [63]. Second, among these services, the data flow between their operations has already been fixed before (manually) [12, 97]; in [63], data flow is not considered.

8.1.3 Synthesis Based on Modal (Propositional) Logic Specifications

A very traditional field of program synthesis is found in the community of verification, more precisely model checking. Even though not restricted to such a setting, the term model checking is often referred to a scenario where a system is modeled through a Kripke structure M and a requirement specification φ is given in temporal logic. We want to know whether or not φ is satisfied by M; formally $M \models \varphi$. The verification problem is to check whether $M \models \varphi$ holds, i.e. both M and φ are given, whereas the synthesis problem is to *find* such an M for a given φ. Ironically, despite the fact that the synthesis problem seems like an extension of model checking, Church already defined the synthesis problem in the context of verification even before the term model checking was coined [23], and also Clarke et al. worked on the synthesis problem [28]. Hence, program synthesis has always been present within the verification community.

There have been many publications in this subfield within the last decades and also very recently. First solutions, e.g. by Büchi and Landweber [17], still relied on Church's arithmetic logic. Later approaches were then based on temporal logics like LTL [102], CTL [28], and others [97]. In the following, I will discuss approaches based on these temporal modal logics in more detail.

Before describing different approaches in (some) more detail, it is a good idea to discuss the meaning and suitability of these logics for program synthesis. The main motivation to introduce temporal logics was the desire to model *open* (or reactive) systems, which are in permanent interaction with their environment, instead of closed systems as considered in this thesis, which obey an invoke-response paradigm. In this regard, they are successful in the sense that they allow indeed for more complex specifications with respect to the *behavior*. That is, one can not only define what should be true at the end of an invocation but also what should hold within one or between several invocations of the system; a much more powerful description possibility than the one I use here. This power, however, comes at the cost that messages are only *binary signals*. The inputs of the compositions are signal sets, i.e. propositions that are known to be true, and the composition algorithm produces such signals as outputs; talking about objects and their relations is not part of the model. With this encoding, we cannot, for example, solve the very first example problem presented in Chapter 1 of computing the price of a book in EUR.

8.1.3.1 LTL-Based Program Synthesis

There is a community that recently tackles the problem of program synthesis based on temporal logics [14]. In a way, this is similar to the approach taken by Pistore and Traverso using CTL or EAGLE.

The strong advantage of temporal logics is that they allow to make assertions about what should hold *during* program execution and not only at the end. This was also the motivation in the above papers to use CTL. LTL is less expressive, and the non-deterministic aspect of operations does not come into play as above. However, this setup is still significantly more complex than the simple approaches discussed in the previous section.

It is not trivial to relate these approaches to the previous ones and in particular what their contribution to planning is. The application domains are typically very technical and hardware oriented. The applicability of the *current* approaches in this area for software development is, also due to the language limitations discussed above, rather questionable.

8.1.3.2 CTL-Based Composition and EAGLE

An important novelty in the field of both composition and planning that came along with non-deterministic operations was the use and definition of languages that allow for relaxed goal conditions. If the outcome of an operation is uncertain, it is often not possible anymore to guarantee a particular output. This is a highly relevant aspect in real applications. For example, one cannot guarantee that a particular ticket will be purchased, because this depends on whether a ticket is still available. It does not make sense to require that the ticket will be bought "no matter the conditions", because whether or not this is possible is not known at planning time (and may differ for different executions depending on the time of invocation).

To this end, Pistore and Traverso proposed to enable more expressive goal conditions by posing planning as model checking using computation tree logic (CTL) [99]. However, they recognized two shortcomings of CTL for planning, which are that CTL can neither express

that the plan should try to achieve the goal, and give up only if that is not possible nor does it allow to express preferences among goals [97]. To solve this issue, they developed a new language EAGLE, which enables goals of the form "TryReach G_1 Fail G_2" [34].

Up to now, this line of research has not been pursued any further. In their latest version [12], Bertoli and Pistore returned to strong plans without modal goal descriptions.

8.1.4 Composition Based on (Simple) First-Order Logic Descriptions

I now discuss composition approaches that consider operations with preconditions and postconditions that contain non-unary predicates. In this case, building a propositional model efficiently may be very costly in terms of the number of possible groundings of each predicate.

8.1.4.1 The $\mathcal{WSC}$ Approaches

$\mathcal{WSC}$ is an abbreviation used in a line of research [59, 60, 107, 122] to refer to the web service composition problem, which largely corresponds to the setup presented in Chapter 2. Approaches in this line of research come in two variants: Most approaches focus on composition with an only limited necessity to create new objects [59, 60], which is mainly due to the assumption of so called Forward Effects. However, there is also an approach in this line of research that does not make this assumption and is, hence, quite in a line with this thesis.

Composition Based on (non-strict) Forward Effects Sirbu and Hoffmann presented a technique for software composition allowing for descriptions with non-unary predicates based on *forward effects* [107]. Forward effects are a relaxation of the strict forward effects discussed above and constitute an environment where every postcondition literal of an operation must contain at least one output variable and where the variables of a literal in a clause are the same as for any other literal of the same clause. The main idea behind forward effects is that new knowledge obtained by operation invocations cannot be used to derive contradictions to previous knowledge.

The difference between forward effects and strict forward effects is that (non-strict) forward effects allow postcondition literals that relate operation inputs to operation outputs. The implication of this is that every operation may not be used only once but several times. Instead of defining one constant data container for every output parameter as in the case of strict forward effects, we must now express the postcondition of the ground operation depending on the inputs. Since the outputs produced in this way can be used again as inputs for others or even the same operation, we cannot easily say how many "instances" of each operation are necessary. In fact, they proved this version of $\mathcal{WSC}$ to be undecidable [59].

There are three differences between the setting applied by Sirbu and Hoffmann and the setting of this thesis. First, the layout of allowed background knowledge is different. On one hand, approaches based on forward effects heavily restrict the background knowledge in that all literals of a clause must talk about the same set of objects. For example, it is not possible to express transitivity of predicates, and most of the rules used in the example of this thesis (cf. Section 1.3) cannot be modeled either. On the other hand, they do not require definite Horn clauses but allow for any type of negation setup of clauses. Second, non-functional properties are not subject of interest. That is, finding compositions that are optimal with respect to, say, price is not considered in that line of research. Finally, the case of finding several solutions is not considered in their approach. But this is a minor difference and would be a straight

forward extension.

In a way, they also provide a technique for non-sequential composition in that case distinctions on types are considered. This is what was already done in the above cited approaches applying *partial matches*. It is then supposed that there is some technique that is able to check at runtime which of the operations can actually be executed, and only a sub-sequence of the composition is really executed. This implicitly corresponds to non-sequential compositions.

Due to the setting similarity, I have carried out an experimental comparison with their approach. Their setting contains one type-partitioning non-Horn clause of the form $\neg t \vee t_1 \vee .. \vee t_n$, which I encoded as $\neg t \vee t_1$, .., $\neg t \vee t_n$, i.e. one direction is preserved. It turns out that this is sufficient when used with the algorithm for finding compositions with branches. Applying partial order composition, even for the case of 6 + 30 operations (where each of the 30 new ones is a clone of the previous ones, i.e. producing the highest possible degree of additional workload), the first solution is found within some few seconds, where in [107] none is found within half an hour. A possible reason for this could be the need to frequently build and evaluating CNFs, which appears to be very costly. However, this observation should not be taken too seriously since the comparison was not made in form of a fair benchmark and there may be conditions we did not consider in this comparison.

Even though the formalism can do more in general, the evaluation setup in [107] actually constitutes an environment as tackled also by the above approaches using strictly forward effects [59,60,64]. In fact, there are some postcondition literals relating inputs to outputs, but these are not relevant for creating a solution in the given example. So at least the example query they presented can be solved in a setting of strictly forward effects. It is not clear how the runtime of the approach in [107] evolves in settings that are really beyond strictly forward effects, and where finding a solution *requires* using non-unary predicates.

Composition without Forward Effects In his PhD thesis, Weber presented an approach to automated composition without the limitation of forward effects [122]. As in the other works in the research line, he used a forward search algorithm to solve the problem.

The actually addressed composition problem is largely similar to the one I discuss here. The semantics of operations is the same, and background knowledge is considered.

However, there are still some quite significant differences. First, the structure of the background knowledge is different. On one hand, Weber allows for non-Horn clauses, which are not allowed in our setup here. On the other hand, he restricted the background knowledge to facts or clauses of size 2 for the implemented system. Second, his approach is better tailored to treat operations with negative literals. Instead of making the IRP assumption, he considers the case that contradictions may occur and applies belief revision in order to update previous knowledge in the spirit of the possible models approach (PMA) [125] even though this technique is debatable (cf. Section 2.2.2). Third, he also considers the case that operations have non-deterministic postconditions, which is reflected in an AO* algorithm to solve the problem. However, it seems that, in the evaluation, non-determinism in the operation postconditions is limited to disjunctions among type predicates as in the partial matches setup. Non-functional properties are not considered.

Despite the similarity of the composition model, a comparison between the approaches is difficult. The setups used to evaluate the approach are the VTA example, which is also used in [107], and a related one called TPSA, both of which do not require non-unary predicates (even though these occur in the descriptions). Hence, the unique aspect of the composition problem is not really reflected in the evaluation. In this sense, the aspect of relating objects

with each other is not covered in the same way as in the evaluation I conducted in this thesis. This makes the comparability of the benchmarks rather doubtful.

If we compare the runtimes nevertheless, we can observe notable differences. In [122], both problems were solvable with solutions of length 7, which corresponds to the experiments with a minimum solution length of 7 in this thesis. We do not know much about how the background knowledge in his experiments look like, but he also takes the market density as an evaluation criterion with a range of up to 1000 operations (apparently copies of the 7 basic operations). His results could then somewhat compared with our results plotted in Figure 7.8a in Section 7.1.3.1 along the axis for market density. If we do this, then we can see that the runtimes are quite comparable for the case of TPSA; in the setup for VTA, the results in [122] are much better. But, once again, the evaluation setups are too different or at least not sufficiently transparent in order to make more detailed assertions.

Weber proposes the computation of a heuristic based on a deletion-free relaxation as used in the FF planner [62], but it is not clear whether this heuristic can also be used for the approaches I present in this thesis. First, due to the IRP assumption, the approaches in this thesis already work in a setting that is relaxed in that sense. Second, the relaxation heuristic is tailored for forward search, and it is not trivially clear how this technique should be translated to backward search; I am not aware of any attempt in this direction.

8.1.4.2 PDDL-Based Approaches

Augmented PDDL The first composition approach that was based on PDDL was published by McDermott [84], i.e. one of the authors of PDDL himself. In that paper, he presents an algorithm called Optop ("Opt-based total-order planner") specifically for the purpose of software composition.

McDermott recognized that classical PDDL planners cannot be directly applied to the composition problem for two reasons. First, PDDL does not allow to express the creation of new objects, which is a key requirement in the composition domain. So, PDDL itself is no adequate input for planners that solve composition problems and needs to be extended. Second, he points out that planners must be able to construct plans that rely on the truth-value of some property, which cannot be guaranteed to be true at runtime. So the planner should create plans that are able to *react* on particular outcomes of operations.

With respect to the first shortcoming, he suggests an extension of PDDL introducing a new parameter `:value`. The assumption is that an operation has *at most* one output and that its value will be stored in a container accessible through the statement `(step-value` i`)` where i is the step of the plan. For example, `(step-value 4)` would be the output of the action at position 4 in the plan. The notation has become standard in PDDL and is also part of the new Opt language.

The second shortcoming is addressed by means of *verification statements*, which are added to plans. The idea is exactly the same as sketched in Section 6.1, namely to allow the planner to simply assume certain conditions to be true. It then must insert a check on that condition to the plan, which is done in terms of a statement `(verify (< (step-value` *step-id*`)` *value*`))`. For example, it could be written as `(verify (< (step-value 4) 2))` to check that the outcome of action at step 4 has a value less than 2.

There are four main differences between McDermott's approach and the one presented here. First, McDermott does not consider (non-scalar) costs. Costs can partially be encoded within PDDL using numeric expressions, but, apart from the fact that Optop does not support

numeric expressions, aggregating these in a fashion different from simple addition is not possible. Second, loops are not considered in Optop. Integrating loops in PDDL planning is, to the best of my knowledge, nothing that has been done so far even though they were considered in planning in general [25]. Third, the heuristic used to guide the search is based on a regression-match graph whereas I consider a relaxed planning graph (not in the sense of delete-free actions but of propositionalized operations). Fourth, operations can only have one but not more outputs in his model.

Simple PDDL In spite of McDermott's explanations on the shortcomings of pure PDDL for software composition, there were some attempts to do this a couple of years later [69, 95, 119]. Without discussing this explicitly, the underlying assumption is to simply treat output parameters as ordinary (input) parameters that are understood in some kind of call-by-reference fashion. In other words, instead of writing data to a new data container, an empty data container is passed as an input and the data is written into that object.

So the assumption is that we have an explicit set of all data containers that *may* be used by a composition in advance. From the theoretic viewpoint, this is no problem, because we can simply assume that the set of available (and explicitly given) data containers is huge. It only must be huge enough to contain sufficient elements for any composition we may find. Of course, it is completely unclear how much "sufficient" is; this could be 1 or could be 100. For example, in a version available for download of a project called OWLS-XPlan [69][1], there are 32 data objects (25 in initial state and 7 additional ones in the goal state) distributed over 16 types (some types have only one object, one has six objects). It is simply assumed that we know before that we will need that number of objects of each data type.

However, the practical feasibility of this approach is somewhat questionable. I created the PDDL encoding for the above problem and solved it using Fast Downward (FD) [56], which solved the problem within less than a second. This was possible, because we had very small numbers of each type at disposition; 8 of the types had only one object. Increasing the number of objects from 32 to 60 made the problem infeasible for FD. The problem, which I also discussed in [87], is that the data containers of each type are interchangeable, and the planner will consider the same operation invocation once for each possible combination of output mappings. For example, if we want to store an airport object obtained from some operation invocation $op(x)$ and we have 10 empty object of that type, say $o_1, .., o_{10}$, then all the groundings $o_1 = op(x)$, .., $o_{10} = op(x)$ will be considered even though all of them are equivalent modulo the naming of the data container where the output is stored.

To summarize, using PDDL-based standard planning for software composition may be an option, but there are also cases where the approach fails since the complexity of only grounding the model renders the problem infeasible. Precisely for these reasons, McDermott suggested an extension of PDDL to overcome these problems.

8.1.4.3 Deductive Synthesis

Another approach very close to the content of this thesis is a line of research called *deductive program synthesis*. Deductive program synthesis is rooted in Green's work on solving programming problems through theorem proving [47]. Some years later, this idea was elaborated in detail by Manna and Waldinger [81, 82]. While these attempts were still driven by very

[1] I do not pose a link here, since this may outdate over time. The project can currently be easily found on the web. The setting I refer to here is UseCase2 of the "health-scallops" example.

technical examples such as sorting a list, more than 20 years later, Waldinger showed that their original approach can be actually used for service composition, i.e. more business process like programs [120].

The main idea behind deductive synthesis is that we can write the whole composition problem as a query for a first-order logic theorem prover. The background knowledge and the operations are encoded into a huge formula, the "application domain theory", which contains one clause or rule for every item in the knowledge base and one rule for every operation. The premise and conclusion of rules introduced for operations correspond to the preconditions and postconditions of the operations except that the outputs already occur in an artificial precondition predicate used to indicate the invocability of the operation. The query is then a statement that is asked to be deduced from the domain theory. The technique requires a constructive theorem prover, which means that an *inference* path is provided that shows why the query is satisfied. From this proof, the composition needs to be extractable (taking only the rules corresponding to operation invocations).

The SNARK prover is such a system and was used for the purpose of composition in [120]. Snark is a highly parametrizable theorem prover for first-order logic with equality. It was developed by Mark Stickel at SRI in collaboration with Richard Waldinger, which suggests that it is a theorem prover specifically tailored for software synthesis. In particular, it is able to produce constructive proofs instead of simple yes/no answers.

In a way, leaving non-functional properties outside, it looks like deductive synthesis solves a more general version of the problem posed in Chapter 2. In fact, the setting is almost identical except that deductive synthesis ignores non-functional properties and makes no particular assumption on the structure of background knowledge; in fact, the knowledge base does not even need to be in CNF. Also, it allows for state and operation descriptions that do not only contain literals with variables but even literals with terms, i.e. arithmetic expressions. So apparently it entails the solution of this thesis.

While the observation that the setting of deductive synthesis is largely a generalization of the one in Section 2.1 is correct, the other side of the coin is that deductive synthesis heavily focuses on presenting a clean theoretic calculus rather than showing solution *strategies*. That is, the composition problem is solved by a theorem prover, which is here simply seen as a black box entity. We never learn anything about *how* the eventually compiled problem is solved by the prover. This is similar to the case of applying SAT solvers in order to solve planning problems. It is one thing to pose a formal reduction function that allows for a problem input conversion. Another question is whether that input has any specific properties of which the actual search algorithm could take advantage of, i.e. as the pruning I applied for backward composition. Deductive program synthesis does not go into detail with respect to how both (i) the search space and (ii) the search strategy look like. In particular, we do not know in how far a general theorem prover can employ heuristics to accelerate the composition process.

Unfortunately, we cannot say much about the performance of deductive synthesis. Indeed, [113] and [120] report some rudimentary runtime results, but these are far away from being representative or reproducible. An implementation of SNARK is still available for download, but it was not possible to run experiments with it since the code is not in a line with the documentation and there was no support by the authors. As a consequence, we do not really know how deductive synthesis compares to the approach presented in this thesis.

Also, the property of the theorem prover to not be tailored for composition is questionable. Intuitively, one would expect that, given that deductive composition works, any theorem prover that is able to produce constructive proofs can be used; this interchangeability of

the solving algorithm seems to be one of the main advantages of deductive synthesis. But then, [120] poses some additional constraints such as type hierarchy support, and, even more specific, an execution interface between the solver and the used operations. In particular the second aspect is very specific for the case of software composition rather than for general theorem proving. Either the theorem prover can be a general purpose solver that can be seen as a black box; then the constraints on it should be minimal and apply also in other settings. It is not at all clear whether this is the case. Or the theorem prover is tailored for the use case of software synthesis; then its behavior must be described in the scope of the description of the composition method. Currently, it seems that SNARK is the only prover that satisfies the requirements, but its behavior is not described in the above papers. In other words, deductive synthesis makes a clear cut between the formal model description and the theorem prover used to solve it, but, at the same time, there are requirements on the prover that are specific for software synthesis, which undermines the idea of separation.

One of the crucial merits of this line of research was the proof of concept that program synthesis can be applied in practice. In [113], we find the report of a practical experiment carried out at NASA Ames Research Center that applied the above encoding technique and SNARK to solve it for sequential composition problems.

However, there was no break through for deductive synthesis in practice. Neither did we see many publications on this line of research in the following years nor did deductive synthesis become a broader known development technique; not precisely a sign for striking success. It is not clear whether this has to do with the approach or tool itself or the general dilemma of a potentially disadvantageous ratio between formalization effort and gain from automation, by which every approach in this field is threatened.

8.2 Related Work in Planning, Search, and Theorem Proving

The automation part of composition takes techniques from several large research fields such as planning, search, and theorem proving. This section discusses approaches that describe general techniques of these areas heavily related to automated composition but not tailored for software composition. The question is in how far those techniques can be used to solve the composition problem out of the shelf. It turns out that there is currently no standard approach in any of the fields that is able to do this.

Since each of the fields is much too large to be treated exhaustively, I only focus on four questions that are potential obstacles for those techniques, i.e. conditions that are not directly compatible with the core techniques of the respective field. First, in Section 8.2.1 I discuss research on standard planners related to planning with non-sequential plans. In Section 8.2.2 and Section 8.2.3, I discuss advances in planning and search with pruning and non-scalar cost measures respectively. Finally, Section 8.2.5 discusses the ability of theorem provers to solve the problem mainly in terms of the obstacle to create constructive proofs from which a composition can be recovered.

8.2.1 Finding Non-Sequential Plans

The topic of finding non-sequential plans is largely covered by the areas of uncertainty and non-deterministic actions. In particular the second of these planning classes is important for software composition, because it reflects the semi-predictable behavior (or outcome) of operations and the computations they perform.

8.2.1.1 Branching and Loops in the Search Algorithm

A direct support for branching is found in the AO* algorithm [93, 94]. Here, the search graph is a rooted AND/OR-graph where a solution is not a path but a graph containing the original root, where every leaf is a goal node, and where for every (inner) AND-node contained in the solution, each successor in the original graph is also in the solution.

AO* relates to the composition technique presented here as follows. On one hand, the advantages of AO* over the branching technique presented in Section 6.1 are that it provides much better backtracking, i.e. we just continue with another candidate if there is no solution for the else-branch, and that it has the natural ability to possibly join two branches if they reach a common state. On the other hand, AO* imposes a significant complexity increase, because it now needs to maintain not only a list of paths but of graphs.

To augment the capabilities of AO* in the direction of cyclic solutions, Hansen and Zilberstein proposed an extension of the AO* algorithm in order to find solution graphs with loops [49]. The algorithm is called LAO* and is able to detect solutions with loops by computing solution costs on the basis of value functions instead of back propagation. Even though well known in the planning community, I am not aware that this approach has ever been used to tackle composition problems. It seems that this approach could be used to directly find solutions in the state transition system that constitutes the composition semantics described in Section 2.1.2.2; this would of course be a forward search. It would be interesting future work to examine this possibility, but carrying out such a study is beyond this thesis.

8.2.1.2 Planning in Non-Deterministic Environments

FOND Planning The branch of planning dedicated to find non-sequential plans (in fully observable environments) is called *fully observable non-deterministic (FOND) planning*. Here, the outcome of actions is assumed to be non-deterministic, i.e. applying an action in a state may yield not only one but any of potentially many successor state candidates. A plan then cannot be a sequence of actions, because it must be able to react to the outcome of an action unless we replan at plan execution time. To this end, a non-sequential plan is often described as a state-action table, which is called a *universal plan* [105] or *policy* in the planning context.

The first major line of research on planning in non-deterministic environments was driven by principles of model checking. Cimatti et al. presented a model-based planner called MBP that is able to find *strong* plans in non-deterministic environments [24, 27]. Strong plans are loop-free plans, i.e. policies in which every state can be reached at most once, which are guaranteed to yield a goal state no matter the outcomes of the actions; clearly, strong plans cannot exist for every query. In [27], Cimatti et al. oppose strong plans to *weak* plans, which are plans for which at least one case of success exists but that do not need to guarantee that the goal is reached. In order to solve the planning problem, MBP uses the model checker SMV developed by McMillan et al. [86].

In a next step, Cimatti et al. presented an algorithm to find *strong cyclic* plans [26]. Strong cyclic plans are policies that may contain loops if they are guaranteed to yield a goal state given that every loop is left after a finite number steps. Daniele et al. remarked that the approach lacks a proper planning-related formalization and that the algorithm may construct solutions with "hopeless loops", and proposed another algorithm for the setting [35]. In [25], Cimatti et al. give a good summary of all these developments where they also consider the objections stated in [35].

Finding strong cyclic plans did not become a too hot topic, but several approaches showed

up over time. For example, Levesque presents KPLANNER, which is a further algorithm to identify strong cyclic plans [76]. Kuter et al. presented an approach that finds strong cyclic solutions based on iterative calls of classical planners [72].

Considering the above works on model checking based composition, the relation between FOND planning and software composition is obvious. In fact, one of the main drivers for FOND planning is software composition itself [12, 63, 79, 97]. It seems that there is almost as much literature related to this type of service composition as for FOND planning in general.

However, the coverage of software composition by existing FOND or conformant planning tools such as MBP, KPLANNER, or CFF is limited to the propositional setting. This setting is either assumed from the very beginning or a PDDL-like specification is grounded. Operations that create new objects are not considered.

Moreover, strong cycles are also semantically different from the loops derived from templates as I use them in this thesis. In strong cycles, each iteration of the loop executes exactly the same (sequence of) actions. So each iteration does exactly the same as the previous one and only differs in the reaction of the environment. In the loops here, each iteration executes a *new* action. In fact, each iteration performs the same sequence of *operations* and *tests* but on different object (contents) yielding different actions. In this sense, this is much more like loops whereas strong cycles rather mimic a try-until-pattern.

Planning with (PO)MDPs Another very active research branch in planning in non-deterministic domains is in the area of (Partially Observable) Markov Decision Processes (POMDPs). Instead of a state transition relation, semantics are given by transition *probabilities*. The "quality" of the agent's behavior is modeled by a reward function. The main task is to find controllers, i.e. state-action tables, often called policies, that maximize the reward in the limit [3].

While there are possible applications of these models for the composition problem, it is unclear how MDP-based composition can be used to tackle the composition problem as posed in this thesis. In fact, MDP models have been applied to the composition problem in modeling the response behavior of operations probabilistically [50]. However, MDPs are finite models in the sense that they work on finite state spaces only; a condition that is not satisfied in the composition problem. Enforcing a finite model would probably be similar to setting up a PDDL encoding for a predefined finite set of data containers. In [50] and related publications, this problem is avoided by the assumption that operations do not have parameters.

8.2.1.3 Planning Under Uncertainty

A second branch of planning that can be used to create somewhat non-sequential compositions is *conformant planning*. Conformant planning develops plans in a belief (set of states) space and only allows to use actions that are applicable in *all* of the states of a belief. It can be solved by interpreting the belief space as a state space itself and apply standard planning [15]. The most important difference to FOND planning is that conformant planning makes no assumption about what the executing agent can sense and, hence, only produces plans whose executability does not depend on the concrete circumstances at runtime. As a consequence, the output in conformant planning is a sequence of actions instead of a policy as in FOND. A successful algorithm for conformant planning is Conformant Fast Forward (CFF) presented by Hoffmann and Brafman [61].

As discussed above, Hoffmann et al. themselves applied Conformant FF to the composition

problem [59, 64]. However, the formal semantics deviate slightly from those of conformant planning. More precisely, in [59], actions can be added to a plan even if they may not be applicable in all states of the belief in which they are executed, but it is only required that a plan guarantees the desired goal state. It is assumed that an action whose precondition is not satisfied in the actual state at runtime simply does not change the state. An implicit consqeuence of this strategy in practice is that, at plan execution time, the actual state should (if possible) be determined in order to only execute actions that actually *are* applicable.

8.2.1.4 Computing Plans with Control Flow

An extension of classical planning that also involves loops is the area of *generalized plans*. Generalized plans were first proposed by Srivastava et al. [110,111]. The idea is to create lifted plans whose actions may not be defined on actual objects of the environment but abstract objects that may or may not exist at runtime. Conditions on these abstract objects *within the plan* can be used to control the behavior of the agent. For example, the plan could be to put the top block, say c, from a specific stack on the table *until* no such c exists anymore. In fact, such a plan corresponds to a program and, hence, finding these plans can be used to solve the software composition problem.

The planning algorithm is not a classical planner but rather a learning algorithm that tries to derive a general plan from a set of sequential plans (for specific problem instances). In fact, the work is very related to the exhaustively addressed field of automata learning where a set of words is to be generalized to an automata.

While compositions in this thesis are syntactically almost equal to generalized plans, the way how these are obtained are very different. The approach in [110] is very flexible in the sense that there must be no previous knowledge about possible patterns while I adopt templates whose suitabilitiy for a specific situation must be determined. The other side of the coin is that one needs (possibly a lot of) example plans. In [111], they try to overcome this problem by generating the sequential plans themselves. But apart from this issue, a more severe question is which kind of structures can be learnt in that way. The currently used examples are rather simple (performing a sequence of actions on a set of items) and do not exceed what can be done with templates of the kind I proposed in this thesis.

An empirical comparison of the two approaches would be interesting future work. On one hand, if the loop parts of generalized plans always follows specific patterns, it is questionable why they should be learnt at all. In particular, because this learning process must been carried out for each new domain from again. On the other side, *detecting* the suitability of the templates I proposed may also be very costly, specifically in domains with many predicates. Since the number of predicates in planning domains is usually farily limited, one may believe that templates can be expected to be significantly faster, but this is speculative and requires further investigation.

A possibly fruitful approach for the future would be to use the approaches complementary. If it turns out that templates are much faster, they should be used to create the generalized plans. However, where do the templates come from? Applying the above approaches could give suggestions for the design of new templates.

A follow-up work of the above approaches was presented in [65]. The idea here is to try to find more *compact* plans when steps occur iteratively. For example, the plan $\langle a_1, a_1, a_1, a_1, a_1, a_2\rangle$, could be written compactly as $\langle a_1, \texttt{if (c++ != 5) goto 1}, a_2\rangle$ where c is a numeric variable initialized with 0 and the `goto` action is an action that changes the position of the program counter. In spite of the apparent different syntax of the second plan, it can be encoded with

standard planning techniques (with support for numeric expressions) and, hence, be solved by classical planners. In other words, the idea here is to reformulate a planning problem in order to obtain more compact plans. Clearly, these goto-statements may induce loops on the control flow.

Since each planning problem is seen as a particular *invocation* of the desired program, several such invocations are given in form of *tests*. That is, instead of solving only one such planning problem, several problems are given and compiled into one problem, which amounts to create a program that satisfies *each* of these tests.

In a way, this can be seen as a particular form of supervised learning. The given data are the example inputs and outputs of different runs of the desired program. The task is now to find the function (as a plan containing `goto`-actions) such that all these tests are satisfied.

Even though this compact plan representation looks quite appealing, the usefulness of the solutions in terms of generalization is unclear. Instead of having a *parametrized* plan that computes a sum $\sum_{k=1}^{l} k$, we get a plan that is guaranteed to compute the correct result for some particular values of l (namely the ones given in the examples), but there is no assertion about what the result will be for values of l that do not occur in the examples. There is no notion of generalization or learning like in [110].

8.2.2 Search Graph Pruning

Pruning has a long tradition in search and planning. Important strategy besides simple state (or resource) subsumption are: Branch and bound, dead end detection [55], partial-order reduction [123], dominance pruning [48, 116].

8.2.2.1 Branch and Bound

Branch and bound is a widely-used technique in the area of operations research. The idea is to partition (branch) the search space in a way such that one partition can be ruled out based on efficiently computable bounds for optimal solutions within those partitions. In mixed integer programming, this is usually done by creating one partition for each of n intervals from which a particular variable can draw values, and compute a relaxed (linear programming) solution for each such partition; these are always at least as good as the optimal integer solution. Partitions with values worse than a known integer solution can be pruned.

The potential benefit of applying branch and bound to the composition problem is rather unclear. The first question is for which of the exploration strategies we should apply the technique. For e_{fast}, we should prune areas of the search space in which solutions are generally farer away than in other areas. In e_{nf}, we should prune areas where all solutions are *Pareto*-inferior to another solution we already found. It is everything but clear whether such splits actually exist and how these bounds could be efficiently computed. Also, branch and bound directly influences the search space definition, so a reengineering of the search graph would be required unless one sees the current definition as a branch itself.

8.2.2.2 Partial-Order Reduction

Partial-order reduction is a technique from the area of verification in which the branching factor of a node is reduced without losing completeness. Typical examples are so called *stubborn sets* [118], which are sets of actions whose applicability is independent from the application

of other actions applicable in that state; i.e., actions of stubborn sets can be safely ignored in this step as they can also be executed later. Similar concepts are *sleep sets* and *persistent sets* [45], which contain actions whose preconditions can be reached *again* (not necessarily applicable in the immediaty successor, but such a successor is always reachable). Partial-order reduction has also been applied in automated planning during the last decade [123].

While partial-order reduction is an interesting pruning technique to the service composition problem in general, it is not clear to which degree it applies to the search technique described in this paper. The reason is that the approach has a somewhat natural application in forward search, but its translation to backward search or even partial order planning is not clear. Of course, the above techniques make sense if the successors of a node are determined based on the preconditions, i.e., based on *applicability*. But in both backward search and partial order planning, successors are determined based on *relevance*, and the idea to defer an action because it will still be relevant at a later point of time does not make much sense.

However, techniques like these can be important when determining the literal to be resolved in a node (flaw in partial order planning). In fact, the SelectFlaw routine for the partial order composition algorithm was designed with a similar idea mind (cf. Section 5.1.3.4). Here, type flaws are deferred because their applicability is hardly touched by the previous actions. For the backward search structure, such a choice is missing, and it would be interesting future work to see whether there is a counterpart of partial-order reduction for backward search.

8.2.2.3 Dead-End Detection

Dead-end detection tries to discover whether there is a path from a node to any solution. Clearly, the node can be safely pruned if such a path does not exist. I have considered a simple dead-end pruning in the backward search by relaxing the problem to a propositional one. If no solution exists on that level, there cannot be any on the first order level.

There are many techniques for dead-end recognition, but, like in the case of partial-order reduction, most of them are tailored for forward search. In fact, dead-end detection is often encoded into the heuristic, which penalties dead-ends with a value of ∞ [55], and, in one way or the other, most of them rely on the fast-forward heuristic that ignores all negative effects of an action [62]. The same applies to a recent approach to dead-end detection based on so called *traps*, which are formulas that, if contained in a state, will be contained in any successor state as well [77]. The idea of traps may be possibly extended to a backward setting, but the presented algorithm Trapper works for forward search graphs.

While relaxation is a common technique for dead-end detection, Helmert [55] has shown that these techniques are sometimes *too* relaxed and proposed a solution based on a translation of the problem representation. More precisely, the original problem representation (STRIPS) is translated into a multi-valued state variables representation. Using a so called causal graph, the variables are split into a high-level and several low-level variables where low-level variables are independent among each other but may influence the high level variable. It can be efficiently checked whether a required goal value for the high-level variable can still be reached based on the possible values for low-level variables, and this information can be used to prune a node. However, like the above approaches, this technique is primarily designed for forward-search spaces, and a translation to backward search or partial order planning is at least not trivial.

Moreover, the above pruning techniques work on *ground* models. That is, the set of actions is fixed in the beginning, because the available objects are known. An adaption to our object

creation setting is not trivial, and it is not even clear whether such an adaption is possible.

8.2.2.4 Dominance Pruning

Dominance pruning is an interesting generalization of the simple case of state subsumption [48, 116]. The idea is to insert nodes only if they are not dominated by other existing nodes based on a general pruning relation $\succeq$, which is similar to pruning in this thesis except that I also prune nodes if they have been already on *OPEN* before. The important difference is that their pruning also considers the g-value of the nodes (prune a node only if it has a worse g-value) while the algorithm as I present it is *independent* from the exploration strategy and prunes only based on the (strict) comparison relation. The other requirements made upon the comparison relation $\succeq$ are similar except that $\succeq$ does not need to be well-founded in [48,116] but needs to be cost-preserving instead. This is similar to my requirement of the length of a rest-path to a solution, but it is not the same, because costs may obviously differ from path length. This difference is mainly due to the fact that they do address an optimization problem and, hence, must not prune cheaper solutions while this is no problem in the setting here.

While the contribution in [48] and [116] is to automatically *find* a possibly good comparison relation, I was rather interested in the properties that such a relation must have if it is independent from the exploration strategy. As discussed in Section 3.2.3.2, ignoring the g-value may lead to an infinite pruning behavior of the algorithm, i.e. that the search algorithm chases an infinite path. The consequence is that the search algorithm is not complete in general, which gave the motivation to define the conditions on completeness-preservingness in Section 3.2.3.2 and prove that completeness is then actually achieved (cf. Section 3.2.4.2). This problem is not existent in the above mentioned approaches since a consideration of g in the pruning decision together with the knowledge that $h^*(n) \leq h^*(n')$ if n dominates n' implies that there is a particular solution within a finite subgraph of the search space that is never pruned and hence, explored after a finite number of steps.

Like the other pruning techniques, the above approaches for dominance pruning are designed to finite search spaces. For example, the above discussed approach in [116] is based on the merge-and-shrink abstraction [57], which is defined on finite state spaces, and it is not trivially clear how this technique generalizes to infinite environments. The same is true for other recent pruning techniques such as pruning in deletion free domains [44]. There, the idea is to prune the search space based on landmark ordering. The techniques suggested here are clearly interesting also for the case of composition, but it is not yet clear how they can be lifted to the non-propositional case.

8.2.3 Non-Scalar Costs in Planning and Search

What I call non-functional properties in this thesis is called *costs* in the general realm of planning and search. In classical search, costs were always considered to be scalar values, but planning and search with cost vectors is more common nowadays.

There is a whole line of research dedicated to the extension of the classical A* algorithm in order to cope with non-scalar costs. Stewart et al. made a first approach in these regards presenting Multiobjective A* (MOA*) [112]. The main idea behind MOA* is to only expand non-dominated nodes and to maintain the Pareto frontier in memory. Lawrence et al. presented a paper with elaborations on MOA* with consistent heuristics and giving a good overview over its development [80]. In the parts related to costs, MOA* is largely identical to the algorithm presented in Section 3.2.

The crucial question, however, is where the heuristics for the algorithm come from. In this thesis, I solved a relaxed problem and took the values from the relaxed solutions. The natural question is if there are others.

A natural place to search for this kind of heuristic is the planning community, because heuristic design is an important subfield here. In fact, in the last decade, the planning community has considered non-scalar costs in terms of *numeric expressions* associated with states [42]. The motivation behind this was the setting of "planning with resources", i.e. where actions "consume" or "produce" previously defined inventories of domain-specific resources. States then do not only consists of propositions but also of a numeric vector whose semantic depend on the context, e.g. resources used by the solution or domain-specific costs.

In fact, there has been considerable work on planning with numeric expressions, and they can be considered a standard element in the planning community. Hoffmann presented Metric-FF [58], the first planner able to consider numeric values not only in the validity check but even in the computation of the heuristic (even though the utility of doing so was questioned in [103]). Later algorithms such as LAMA [103] integrated action costs as a default functionality.

However, I am not aware of any implemented planner that copes with non-scalar costs and heuristics. For example, both Metric-FF and LAMA are addressed with weighted A*, which works with scalar cost values. In fact, Metric-FF even comes with the possibility to use a dedicated cost function as path costs. This is exactly the support for the exploration strategy e_{nf} where the version presented in the above cited paper corresponds to the strategy e_{fast}. But also here weighted A* is applied. For more sophisticated optimization, we would need to apply (some variant of (a weighted)) multiobjective A* as discussed above. Since also the current planning competitions work only on scalar cost functions, I am not sure whether there has been progress on this topic.

As a final remark, it is crucial to distinguish between typical integer programming optimization (or approximation) problems with huge solution spaces and planning problems where only few elements of the search space are solutions. In the first case, genetic algorithms constitute a well-known and simple paradigm. For the case of multicriterial optimization, algorithms such as NSGA-II can be used [38]. However, in search spaces with few solutions, as in the case of planning, it is a hopeless undertaking to try to get "valid" offsprings by chance. Hence, solutions for genetic algorithms do not help at all for multicriterial planning problems.

Another quite theoretic option is to try to compile the property vectors into scalars that reflect the *utility* of the vector. This is a very common technique in decision theory [67]. However, such a utility function is often not available. In fact, creating real valued utility functions is a tedious task and highly user dependent. That is, there is not even such a "general" utility function but every client has her own one. As a consequence, we cannot generally assume that we can compile the non-functional properties away into a scalar.

To summarize, the search framework for non-scalar search is old, but there is little work on finding good heuristics. In particular, I am not aware of any common technique that would allow for a specifically well estimate of h in e_{nf}. In this sense, the solving the relaxed problem with a multi objective search was the most reasonable thing to do at this point.

8.2.4 Finding Multiple Solutions

The previously discussed algorithm MULTIOBJECTIVE A* (MOA*) naturally returns a *set* of non-dominated solutions [112]. In general, this set can be exponentially large. However, since the plans in the solution set are all pareto-optimal (also with respect to solutions not detected so far), one can simply cancel the algorithm as soon as the pool reaches a previously fixed size k.

Such a k-best algorithm can obviously also been defined for classical planning problems. In fact, this is a trivial extension of the A* algorithm where one simply does not cancel on a given solution but continues until k solutions were returned.

For constraint satisfaction problems, finding multiple solutions in AI planning has been mainly tackled through the notion of *diverse plans*. The general idea of plan diversity is to define a (possibly domain independent) distance metric over plans and to create a set of diverse plans instead of returning only one solution. One of the first approaches for finding diverse plans was presented by Herbrard et al. [54] and Srivastava et al. [108].

In fact, service composition could be an interesting application for plan diversity, and it is even used as a motivation in [108]. We have seen in Section 7.2.2 that the composition algorithms return several solutions in the given time bound, and it would be an interesting endeavor to design the search process in a way that possibly diverse solutions are computed. Such a mechanic is, however, beyond the scope of this thesis.

8.2.5 Constructive Theorem Proving

The number of technical achievements in theorem proving is much too long to be treated here in an exhaustive manner. First order logic theorem proving has been a heavily studied field for decades and has elaborated several very important techniques, some of which I also used in this thesis. Most notable achievements are probably the resolution calculus with unification [104], model elimination [78], model checking [29], the tableaux technique [20], the Davis-Putnam-Logemann-Loveland (DPLL) [36], but also others.

However, finding *constructive* proofs is not at the core business of theorem proving. Constructive here means that the proof is achieved only by means of *inference* and not by *model checking* in the sense of fixing truth values; so one obtains a chain of arguments that explains the solution. An important implication of a constructive proof is that we can directly translate it into a composition; in particular, for every output variable of the query, we have an instruction of how to create an object with the respective property. Purely resolution based techniques are constructive, but most theorem provers also apply other techniques. For example, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is the basis of most provers, is not constructive but enumerates the models where no unit-resolution is possible.

The constructive theorem prover (language) that is closest to the work in this thesis is clearly Prolog. In fact, if non-functional properties are left aside, we can encode a composition problem as specified in Section 2.1 in the Prolog language (operation outputs become function symbols that depend on the inputs) and solve it with the Prolog solver. So, similar to the case of deductive synthesis, the connection between the formal models is very close. Prolog is constructive in the sense that it can return an SLD refutation as a proof, which is a clear chain of arguments corresponding to the operations and the rules of the background knowledge.

Considering Green's work [47] and the first approaches on deductive synthesis [81, 82], the question arises whether program synthesis is just one case of application of theorem proving.

The observation that we can use Prolog to encode and solve the composition problem (without non-functional properties) seems to confirm this suspicion.

However, a closer look suggests that the answer is probably negative and that software composition can be justified to be a field that is emancipated from theorem proving:

1. First, currently no theorem prover is applicable to the composition setting, and most are not even close to it. Concrete problems are:
   (a) very few theorem provers provide (translatable) constructive proofs. Note that providing a proof in general is not sufficient here. For example, I applied the Z3 solver [37] to the composition problem in order to verify the experiments. Z3 allows to retrieve a proof, but the proof cannot be translated into a composition (as for example in the case of an SLD resolution obtained in Prolog).
   (b) theorem provers cannot treat non-functional properties, i.e. costs of proofs.
   (c) in the case of interleaved composition as done by Waldinger [120], SNARK is the only theorem prover that does the job.
2. unless we have a highly customizable theorem prover, it only implements one particular type of search strategy. For example, we could use Prolog, but then we would be bound to a depth-first search with backtracking, which is not even a complete search technique let alone the lack of heuristic support.
3. theorem provers cannot distinguish between operations and rules from the background knowledge. In other words, we just give away information that we may want to use in the search process.

Of course, one can extend every existing theorem prover in a way that it fixes the above shortcomings in one way or the other. But then the question is why we use a theorem prover at all and do not develop a new technique. Or put differently: Why should we call the result of this modification process still a general theorem prover if it has been specialized for this class of problems?

Summarizing, software composition can be seen through the glasses of theorem proving, but it is currently clearly too much to say that existing theorem proving techniques make software composition superfluous. There are significant intersections between theorem proving and software composition, but software composition seems to come with properties that are proprietary for synthesis and irrelevant for other applications of theorem proving.

9. Conclusion and Outlook

I close this thesis by summarizing the results from the previous chapters in form of answers to the research questions posed in the introduction, giving a brief conclusion statement, and finally making some suggestions for next research steps.

9.1 Summary of Contributions

RQ 1. *How can we solve the sequential composition problem under quality constraints?*
I have answered this question by modifying algorithms for classical backward search and partial order planning in order to cope with the special conditions of service composition. The main modifications I made on the traditional techniques were (i) to introduce a mechanism that copes with the fact that the objects used as parameters for operations are not global constants but have a point of time in which they are created, (ii) the consideration not only of types in general but of type *systems* (i.e. with sub-type relation), and (iii) the consideration of non-functional properties that are neither scalars nor aggregated in an additive fashion.

Adopting these modifications adequately imposes some challenges. First, using backward techniques for composition, one needs additional soundness checks that are not existent in classical planning. For example, we must make sure that there are no assertions about objects that do not exist yet at the respective point of time and that a variable is not written by two different operation outputs. These conditions are easily satisfied in forward search by using every constant at most once as an output but are more difficult to treat in backward search and partial order composition. Second, using a type heterarchy, we must make sure that an operation can be executed also if the type predicate in the precondition of an operation does not match exactly the type predicate in the state but if it is at least as general.

The developed solutions are slightly limited in two ways. First, the theorems on completeness only hold for positive domains, i.e. where operations only have positive preconditions and postconditions and the background knowledge is a definite Horn formula. Both algorithms are also applicable in non-positive scenarios but cannot guarantee to find a solution even if one exists. Second, both developed approaches assume that the assumption of Invocation and Reasonable Persistence (IRP) holds, i.e. that knowledge obtained once never becomes invalid. This assumption strongly facilitates the algorithmic solutions; for example, we do not need threat treatment in the partial order composition. While there is a reasonable set of applications where this assumption holds, in particular read-only environments, it also excludes many relevant composition setups. However, the meaning of this limitation is quite unclear. In fact, we could just apply the algorithms even if the IRP assumption does not hold and this might just work out fine in some cases. Nevertheless, an appropriate treatment of problems where IRP does not holds would be desirable in the future.

RQ 2. *How can we find compositions with branches and, in particular, with loops?*

For this question, I have focused mainly on the case of composition with loops. Given an algorithm for sequential composition, there is a straight forward extension to find compositions with branches. One can apply a relaxed version of sequential composition that may add conditional statements, which impose *if-then*-branches, and then recursively finds solutions to the *else*-branches. Clearly, this is only one way to find compositions with branches; another would be to modify the search graph to an AND/OR-graph and conduct a respective search.

In order to find compositions with loops, I have applied a template based technique. Loops are not built arbitrarily but inserted as closed building blocks by the previously developed sequential composition algorithms. These building blocks are instantiations of *templates* and are derived during search. The templates are domain independent loop patterns that frequently occur in programming such as computing a subset of a given set with certain properties or finding the item of a set that is maximal or minimal with respect to a particular property. Given a particular rest problem in the search graph of the backward composition or partial order composition, we can apply a mechanism that checks on the fly whether one of the available templates can be instantiated to resolve a flaw of the current rest problem. Such an instantiation can then be treated by the search algorithm as if it was an atomic operation; i.e. the actual composition algorithm is not even aware that it is inserting a loop block.

This technique seems to be a reasonable solution to the problem of finding compositions with loops but is obviously limited in that it can only find loops that are instances of a pattern we have stored in our database. Hence, the approach has a very high leverage in the number of templates that are available but there is no flexibility to find loops outside of these templates. However, it is not clear up to which degree of freedom in the creation of loops we can still find solutions in reasonable time. Here, we must expect a trade off between flexibility and feasibility. In this regard, the template technique presented in this thesis seems to be a good fit.

RQ 3. *Is automated service composition computationally feasible?*

I answer this question from the experimental viewpoint. From the theoretical viewpoint, the composition problem is undecidable and, hence, obviously not feasible. Already our results show that one can easily create very hard problems that cannot be solved automatically in reasonable time even though there are relatively short solutions. However, this type of discussion does not give us relevant insights about the applicability in real world setups, so I answer the question from the viewpoint of *practical* feasibility.

As already alluded in the introduction, it is hard to answer this question without any real world problems, which are currently not existent. For some problems, like the traveling salesman problem, one can easily create realistic problems that allow to make assertions about the performance of a solution in the real world. However, automated service composition relies on the assumption of semantic descriptions, which are currently not there, and which will not emerge unless there is a credible promise that making such specifications can yield a medium-term advantage. The algorithms presented in this thesis are an attempt to give such a promise, but I cannot rely on "real" problems in order to demonstrate the practical applicability of my solutions.

To make a possibly reasonable assertion about feasibility in spite of this problem, I conducted an exhaustive evaluation. Even though the whole evaluation setup is artificial, we tried to make it reflect as many properties found in real programming environments as possible.

Based on the results found in this evaluation, I would answer the above question with yes.

The result is that the presented algorithms are likely to find a solution within a few minutes or even some few seconds in realistic settings. Settings that I consider realistic here contain up to some hundreds (Horn) clauses describing the background knowledge, up to 10 different operations that can be used to achieve the same predicate, and queries whose solution length ranges between 2 and 10. Finding compositions with loops may increase the effort from some seconds to several minutes, but a reliable assertion of the runtime is not possible here, because the template instantiation seems to be highly sensitive to domain properties, which we cannot reasonably simulate in this artificial environment.

RQ 4. *Is one of the composition algorithms superior to the others?*
Yes. At least in our experiments, partial order composition highly outperforms backward composition. In many setups, in particular the more difficult ones, partial order composition was faster by a factor of 10 or more. Since the two strategies are highly similar except the point of time when decisions are made (partial order composition applies least-commitment), this intuitively means that the advantage of a state space search, which enables a highly effective pruning, does not compensate the complexity that arises from explicitly enumerating the possible *permutations* of (sub)compositions.

However, this does not mean that we should not consider backward composition in future developments. One reason for the advantage of partial order composition is perhaps the fact that the "partiality" of solutions is very high due to the IRP assumption. If we would explicitly consider world-altering effects of operations, it could happen that the advantage of partial order composition decreases or even vanishes.

The research question must currently remain unanswered with respect to a comparison to the previous approaches in [84, 107, 122]. On one hand, the conditions of the available evaluations of the previous approaches were too different from the ones I adopted here in order to derive reliable assertions. On the other hand, I was not able to run the experiments conducted in this thesis with the previous algorithms as their implementation was either not available or not sufficiently documented in order to run the experiments. In any case, the results would have been somewhat biased by the fact that they cannot cope with non-functional properties, and I would have to leave them out of question. Of course, it would be interesting to conduct a comparative analysis between them in a competition, but this remains future work.

9.2 Conclusion

Together with previous approaches, this thesis has shown that, in spite of the enormous problem complexity, automated service composition seems to be feasible in the expected area of application already now. Remember that our goal is not to substitute but to complement traditional programming. While it is reasonable that *some* (probably most) parts of the program remain imperative and implementation-tied, we would prefer to write *some* parts of the code in form of declarative and implementation-independent statements that can then be "ground" by composition algorithms as the one presented in this thesis. It is common sense in the community of automated composition that the subprograms entailed by these declarative statements will be rather small [122], which also justifies our evaluation setup. In other words, already the techniques developed so far are apt to cope with the degree of complexity of problems we would expect in real applications.

This is a quite encouraging insight and also animates to try the technique in real applications. In fact, we already developed an extension to the Java programming language that allows to write augmented code containing both classical Java code and the declarative statements defining the postconditions of the desired composition. However, we still do not know anything about the type of applications in which automated composition is beneficial. While this can of course be tried manually, a scientific study that analyzes these conditions, similar to design patterns in object oriented programming, would be more appropriate.

9.3 Future Research Directions

This work has complemented previous approaches by very fundamental yet rudimentary techniques. Now that we have a basic portfolio of techniques to tackle the problem, next steps should aim at application studies, generalization in the sense of dropping assumptions, and refinement in the sense of improving particular properties.

Without any doubt, as sketched above, the most urgent next step would be a case study that analyzes in which areas and which type of applications, automated composition can be applied. A possible research question here could be: How much can we reduce the code of existing applications by using declarative statements combined with automated composition? This question aims at re-engineering existing applications in order to get an estimate of the gain of automation techniques. Of course, code length is only one easy "performance" measure, which is foremost interesting because it reflects readability. Other, even more important but less evaluable measures would be the maintainability of such re-factored programs in comparison to the fully-tied versions. Apart from these considerations related to applicability, the answers I gave to the above research questions allow to immediately derive a couple of rather technical research questions.

The developed algorithms rely on quite some more or less heavy assumption, which can be dropped. One is the IRP assumption, which is particular for the approaches I presented in this thesis, and which should the first to be dropped in future research and replaced by some kind of belief revision. Also, adding complete support for non-positive setups, in particular knowledge bases in CNF, would be desirable. Third, non-functional properties are currently limited to be numbers but should rather be expected to be *functions* that must be evaluated with respect to the input (sizes) of the operation. This is particularly important for compositions with loops and a highly non-trivial extension. Finally, we could allow for more flexible operation descriptions containing quantifiers; note that allowing for disjunctions would only be syntactic sugar since these can already be simulated using the background knowledge.

Another point of possible research is dedicated to an improvement of the techniques developed so far. The pruning applied so far can be significantly improved for both techniques; it would a great achievement to automatically identify dominance relations used in pruning as done in planning [48, 116]. Also, once the IRP assumption is dropped, it would make sense to modify the backward composition algorithm into the direction of the partial order composition in that decisions (e.g. on bindings) are deferred. This would yield a much smaller branching factor in the search graph and possibly better performance than partial order composition. However, it seems that weakest commits are a dominant strategy unless we one must consider threats between the operations, which is not the case under the IRP assumption. Third, a more efficient integration of template-instantiation technique into the sequential search would be desirable. It will not be practicable to run the instantiator for every node of the search graph, which calls for a sophisticated technique to control these invocations.

A. Detailed Versions of Sketched Proofs

This appendix contains the detailed versions of proofs that are only sketched or even completely omitted in the main matter.

Undecidability of The Composition Problem

Theorem 2.2. *Let $p = \langle\langle \mathcal{T}, \Omega, \mathcal{N}\rangle, O, q\rangle$ be the instance of a sequential composition problem. The decision problem whether a solution to p exists is undecidable.*

Proof. The proof is almost a copy of the one for undecidability of $\mathcal{WSC}|_{fast}$ in [59]. Except the repair of a tiny mistake[1], the only difference is the necessity to assign types to the operation inputs and outputs. Nevertheless, I present it again with the respective adaption for the sake of self-containedness.

The idea is to reduce the halting problem of Abacus machines (register machines), which is undecidable [16], to a sequential composition problem. An Abacus machine consists of a set of registers $r_1, .., r_m$ each of which takes non-negative integer values initialized with 0 and a set of states $q_0, .., q_n$ where q_0 is the initial state and q_n is the halting state. Each state q except q_n is associated with an instruction $INC_{r,q'}$ or $DEC_{r,q'_+,q'_=}$, which is executed in the respective state. $INC_{r,q'}$ increases the value of register r by 1 and switches to q'. $DEC_{r,q'_+,q'_=}$ decreases r by 1 and switches to q'_+ if $r > 0$; if $r = 0$, it switches to $q'_=$ and leaves r untouched. The proof is achieved by translating a concrete Abacus machine into an instance of the sequential composition problem that simulates running the machine with the query postcondition corresponding to the fact that the machine has stopped.

Given an Abacus machine, we derive the following composition problem instance:

- *Type Heterarchy.* $\mathcal{T}$ consists of only one type t.
- *Knowledge Base.* We do not need background knowledge, i.e. $\Omega = \emptyset$
- *Non-Functional Properties* $\mathcal{N}$. No non-functional properties are required, i.e. $\mathcal{N} = \emptyset$
- *Operations* O. I write an operation o as $\langle X_o, Pre_o, Y_o, Post_o, Z_o\rangle$. The operations are
  - a successor operation $\langle\{n\}, t(n) \wedge nat(n), \{n'\}, t(n') \wedge nat(n') \wedge succ(n, n'), ()\rangle$.

[1] In the proof in [59], the new value valid after the increase and decrease instruction, which is denoted v', needs to be an *input* of the respective two operations; in the third operation, which does not change any register, this repair is not required.

– for each state q with instruction $INC_{r_j,q'}$ an operation

$$\begin{array}{ll}\langle\{v_1,..,v_m,s,v'\},&\\ (\bigwedge_{i=1}^{m} t(v_i)) \wedge t(s) \wedge t(v') \wedge & pc_q(s) \wedge (\bigwedge_{i=1}^{m} val_i(s,v_i)) \wedge succ(v_j,v'),\\ \{s'\},&\\ t(s') \wedge & pc_{q'}(s') \wedge \left(\bigwedge_{i=1,i\neq j}^{m} val_i(s',v_i)\right) \wedge val_j(s',v'),\\ ()\rangle&\end{array}$$

– two operations for each state q with an instruction $DEC_{r_j,q'_+,q'_=}$, which are

$$\begin{array}{ll}\langle\{v_1,..,v_m,s,v'\},&\\ (\bigwedge_{i=1}^{m} t(v_i)) \wedge t(s) \wedge t(v') \wedge & pc_q(s) \wedge (\bigwedge_{i=1}^{m} val_i(s,v_i)) \wedge succ(v',v_j),\\ \{s'\},&\\ t(s') \wedge & pc_{q'_+}(s') \wedge \left(\bigwedge_{i=1,i\neq j}^{m} val_i(s',v_i)\right) \wedge val_j(s',v'),\\ ()\rangle&\end{array}$$

and

$$\begin{array}{ll}\langle\{v_1,..,v_m,s\},&\\ (\bigwedge_{i=1}^{m} t(v_i)) \wedge t(s) \wedge & pc_q(s) \wedge (\bigwedge_{i=1}^{m} val_i(s,v_i)) \wedge zero(v_j),\\ \{s'\},&\\ t(s') \wedge & pc_{q'_=}(s') \wedge (\bigwedge_{i=1}^{m} val_i(s',v_i)),\\ ()\rangle&\end{array}$$

- *Query q.* The query is $\langle o_q, one, seq, \odot\rangle$ where $\odot(c) = ()$ for every composition c and

$$o_q = \begin{array}{ll}\langle\{s_0,v_0\},&\\ t(s_0) \wedge t(v_0) \wedge & pc_{q_0}(s_0) \wedge (\bigwedge_{i=1}^{m} val_i(s_0,v_0)) \wedge nat(v_0) \wedge zero(v_0),\\ \{s^*\},&\\ t(s^*) \wedge & pc_{q_n}(s^*)\\ ()\rangle&\end{array}$$

The semantics of the predicates is the intuitive one: $t(x)$ is the (only) type predicate, $pc_{q_i}(s)$ asserts that the program counter at step s is q_i for $0 \leq i \leq n$, $val_i(s,v)$ asserts that register r_i has the value represented by v at step s, $succ(v,v')$ asserts that v' is v plus one, and $zero(x)$ says that x encodes the number 0. While there may be arbitrarily many objects encoding each positive integer, which is not a problem, v_0 is the only object for which $zero$ holds.

It can then be easily shown that there is a solution to the compiled (sequential) composition problem instance iff the Abacus machine halts. Each computation step of the Abacus corresponds to one or two operation invocations: One in the case of decrease and two in the case of increase because we first need to apply the successor operation. This defines a canonical sequential composition that minimally transforms the initial state into the state that corresponds to the state of the Abacus at a certain point of time. Likewise, each state reachable by such a canonical composition is reached by the Abacus after a number of steps, which is upper bounded by the size of that composition. In particular, if the Abacus halts, it has reached state q_n, and the canonical composition minimally transforms the initial state into one where $pc_{q_n}(t^*)$ holds, which makes it a solution to the query. Vice versa, if such composition of length l exists, we know by the above that the Abacus will reach q_n after at most l steps. □

Proofs Related to The Search Algorithm

Lemma 3.3. *Let $\mathcal{S}$ be a search structure with a goal node, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then, at each point of time before* $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *returns a solution, there is a path $(n^0_\mathcal{S}, .., n, .., n^*) \in P_\mathcal{S}$ such that $\star_\mathcal{S}(n^*) = true$ and $(n^0_\mathcal{S}, .., n) \in OPEN$.*

Proof. Suppose that the search graph $G_\mathcal{S}$ contains a goal node n^*; then there is also a goal path $(n^0_\mathcal{S}, .., n^*)$ in the set of paths $P_\mathcal{S}$ of $G_\mathcal{S}$. The proof now goes by induction over the number k of iterations of the main loop.

Basis. If $k = 0$, then $OPEN = \{(n^0_\mathcal{S})\}$, and the claim is trivially true.

Inductive Step. Let $k > 0$, and suppose that in iteration $k - 1$ there was a path $p = (n^0_\mathcal{S}, .., n)$ on *OPEN* and a path from n to a goal node exists in $G_\mathcal{S}$. Two cases are possible:

1. $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *expands* p. At least one successor n' of n is on the path to a solution. Let $p' = (n^0_\mathcal{S}, .., n, n')$ be the extension of p. Three cases are possible:
   (a) n' is a solution itself (then we are done).
   (b) n' is pruned. Since no solution was returned and since p is not a dead end, this case can only occur if another path $p'' = (n^0_\mathcal{S}, .., n'')$ is on *OPEN* such that $n'' \succ n'$. But then, there must be also a path from n'' to a goal node, so the path p'' satisfies the required condition.
   (c) p', and, hence, a new subpath of a solution path, is on *OPEN*.
2. $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *expands* $p' \neq p$. Two cases are possible:
   (a) n is not pruned during the expansion of p'. Then p and, hence, a subpath to a goal node remains on *OPEN*.
   (b) n is pruned by the expansion of p'. This means that, when $p' = (n^0_\mathcal{S}, .., n')$, there is a successor n'' of n' such that $n'' \succ n$, and the path $(n^0_\mathcal{S}, .., n', n'')$ is put on *OPEN*. Since $n'' \succ n$ implies that there is a path from n'' to a goal node, there is a subpath of a solution path on *OPEN*.

So there is always a subpath of a path that leads to a goal node on *OPEN*. □

Lemma 3.5. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$, and let $p = (n^0_\mathcal{S}, .., n) \in OPEN$ such that there is a path of length r from n to a goal node. Then, after a finite number of steps,* $\text{SEARCH}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *does one of the following:*

1. *it returns a solution,*
2. *it puts a path $p' = (n^0_\mathcal{S}, .., n')$ on OPEN such that there is a path from n' to a goal node that is shorter than r, or*
3. *it puts a path $p' = (n^0_\mathcal{S}, .., n')$ on OPEN such that $n' \succ n$.*

Proof. Let $N^n_\mathcal{S} = \{p' \mid p' \in P_\mathcal{S}, f(p') \leq f(p)\}$ be the set of all paths with an evaluation value of at most the value of p. This set is finite by Observation 3.2.

Now suppose that $|N_{\mathcal{S}}^n|$ iterations are performed and condition 1 and 2 are not true (otherwise we are done). We know that every path p' with $f(p') \leq f(p)$ that was not pruned has been expanded. In particular, this holds for p itself. Two cases are possible:

1. p has not been pruned. Then it has been expanded, and there is a successor node n' with a path from n' to a goal node with length $r' < r$. Either n' is pruned due to the existence of another path $p'' = (n_{\mathcal{S}}^0, .., n'') \in \mathit{OPEN}$ with $n'' \succ n'$. But then we know that there is a path from n'' to a goal node with length $r'' \leq r' < r$, and p'' is the respective path on OPEN. Or n' is not pruned. But then $(n_{\mathcal{S}}^0, .., n, n')$ is put on OPEN, and the length from n' to a goal node is $r' < r$.

2. p has been pruned. This could only happen by finding a path $p' = (n_{\mathcal{S}}^0, .., n')$ with $n' \succ n$, which was put on OPEN after pruning.

□

Lemma 3.6. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then* $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *returns a solution after a finite number of steps if one exists.*

Proof. Suppose that a solution to the original problem exists. By the completeness of $\mathcal{S}$, $G_{\mathcal{S}}$ contains a path to a goal node. Lemma 3.4 tells us that the algorithm will not halt with "fail".

Now consider an arbitrary point of time during execution. By Lemma 3.3, we know that OPEN has a path $p = (n_{\mathcal{S}}^0, .., n)$ that can be completed to a solution. Let r be the length of the (remaining) path from n to a goal node.

Applying Lemma 3.5 to p tells us that, after a finite number of steps, say t_0, the algorithm will have returned a solution (then we are done), or it has put a path $p' = (n_{\mathcal{S}}^0, .., n')$ on OPEN such that either the distance from n' to a goal node is smaller than r or $n' \succ n$.

Let us first focus on the case that $n' \succ n$. Applying Lemma 3.5 recursively to p', after a finite number of steps, say t_1, we either returned a solution, found a path with a shorter rest-length to a goal node, or identified another n'' such that $n'' \succ n'$. Since $\succ$ must be well-founded on the sets of nodes $N_{\mathcal{S}}$, there is a node $n_{min} \succ \ldots \succ n' \succ n$ such that no n'' with $n'' \succ n_{min}$ exists. Hence, the third case can occur only finitely often, say k times starting from n. So, after at most $\sum_{i=0}^{k-1} t_i$ iterations, there will be a path $(n_{\mathcal{S}}^0, .., n_{min})$ on OPEN with rest-length to a goal node at most r and n_{min} will never be pruned; here t_i was the time horizon for the respective path until one of the cases of Lemma 3.5 occurred.

Then we know that after a finite number of steps, if no solution has been returned, p' must be a subpath of a path to a goal node with a distance of $r' < r$ to the goal node. More precisely, after at most $\sum_{i=0}^{k-1} t_i$ many iterations, there is a path on OPEN whose head node cannot be pruned, and within t_k many steps, either the first or the second case of Lemma 3.5 becomes true. Hence, after at most $\sum_{i=0}^{k} t_i$ iterations and unless a solution has been returned, a path with a rest distance to a goal smaller than r is on OPEN. In other words, the distance to a goal node must strictly decrease within a finite time horizon.

But then the algorithm must eventually put a path p^* on OPEN for which $r = 0$. When this happens, the head of the path is a goal node and the algorithm returns a solution. □

Lemma 3.7. *Let $\mathcal{S}$ be a complete search structure, $\mathcal{E}$ be a strictly increasing exploration strategy, and let $\succeq$ be a completeness-preserving SR-dominance relation for $\mathcal{S}$. Then* $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_\succeq}$ *outputs every solution for which ξ holds after a finite number of steps.*

Proof. Let $s \in \mathscr{S}^*$ be a solution that satisfies ξ. By completeness of $\mathcal{S}$, there is a path $p^* = (n^0_{\mathcal{S}}, .., n^*)$ such that $s \in \textsc{Trans}_{\mathcal{S}}(p^*)$. The pruning mechanism $\mathcal{P}_{\succeq}$ will never prune a path to a node n that produces s.

Before termination of $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_{\succeq}}$, for every path, there is either (exactly) one of its subpaths on *OPEN* or all of its nodes have been expanded. In particular, this holds for p^*, so either n^* has been expanded (solution announced) or Lemma 3.3 ensures that there is a subpath of p^* in *OPEN*. Note that this condition also implies that the algorithm does not terminate with "no" before a solution is returned, because *OPEN* is not empty.

Now let $M = max_{p' \subseteq p^*} f(p')$ be the maximum evaluation value for any of the subpaths of the solution path p^*. $\textsc{Search}_{\mathcal{S},\mathcal{E},\mathcal{P}_{\succeq}}$ will not select any path p with $f(p) > M$ for expansion unless every path p with $f(p) \leq M$ has been expanded or pruned.

Moreover, let ${P_{\mathcal{S}}}^M = \{p \mid p \in P_{\mathcal{S}}, f(p) \leq M\}$ be the set of paths with an evaluation value of at most M. By Observation 3.2, this set is finite.

But then, after at most $|{P_{\mathcal{S}}}^M|$ steps, every subpath of p^* has been expanded; in particular, a path pointing to the parent node of n^*. But then, n^* was generated through the expansion process and announced as a solution. By the completeness of $\mathcal{S}$, the goal check in line 9 yields *true*, and the solution(s) in $\textsc{Trans}_{\mathcal{S}}(n^*)$, in particular s, are out-streamed. □

Proofs Related to BW

Lemma 4.1. *Let q be a query and $p = (n^0_{BW}, .., n)$ a path in G_{BW}. Then $\textsc{Trans}_{BW}(p)$ computes in linear time w.r.t. $|p|$ a sequential composition that transforms $\lambda(n)$ into $Post_q$.*

Proof. It is obvious that $\textsc{Trans}_{\text{BW}}(p)$ computes a single sequential composition within linear time in the length of p since it walks along p and performs constant-time operations. What needs to be shown is that the resulting composition $c(n)$ minimally transforms $\lambda(n)$ into $Post_q$. The proof is by induction over all paths of length k in the graph.

Induction Basis. Let $k = 0$. There is only one path of length 0, which is the path (n^0_{BW}) containing only the root with label $\lambda(n^0_{\text{BW}}) = Post_q$. Obviously, the empty composition transforms $\lambda(n^0_{\text{BW}}) = Post_q$ minimally into $Post_q$.

Inductive Step. Let $p = (n^0_{\text{BW}}, .., n', n)$ be a path of length $k > 0$. By the induction hypothesis, we know that $c(n')$ that minimally transforms $\lambda(n')$ into $Post_q$. Let ${s^0}'$ and ${s^f}'$ be the initial and final state of $c(n')$ respectively, and let λ' be a valid labeling of $c(n')$ such that $\lambda'({s^0}') = \lambda(n')$ and $\lambda'({s^f}') = Post_q$. Note the conceptual difference between λ, which defines labels of *nodes* in the search graph, and λ', which is a labeling for *composition states*.

We now define a valid labeling λ'' for $c(n)$ that shows that $c(n)$ transforms $\lambda(n)$ into $Post_q$. First, for every state s of $c(n')$ *except the initial state* ${s^0}'$, we set $\lambda''(s) = \lambda'(s)$. The remaining state labeling λ'' depends on the edge (n, n'), which can be labeled in two ways:

1. it is labeled with an implication $\left(\bigwedge_{j \neq i} \alpha_j \rightarrow \alpha_i\right)[\sigma]$ corresponding to a clause from Ω. Then $c(n) = c(n')$ and $\lambda(n) = (\lambda(n') \setminus \alpha_i[\sigma]) \cup \bigcup_{j \neq i} \alpha_j[\sigma]$, and, in particular, $\lambda(n) \wedge \Omega \wedge \mathcal{T} \models \lambda(n')$. Now we simply define $\lambda''({s^0}') = \lambda(n)$, which directly follows $\lambda''({s^0}') \wedge \Omega \wedge \mathcal{T} \models \lambda'({s^0}')$. So λ'' only changes the labeling of the initial state ${s^0}'$ in a way that every successor state has still a valid labeling. Since λ' was a valid labeling, also λ'' is a valid labeling, and, since $\lambda''(s^0) = \lambda(n)$, $c(n)$ transforms $\lambda(n)$ into $Post_q$.

2. it is labeled with an operation invocation $o[\sigma]$. Then $c(n)$ has one more state than $c(n')$, which is the new initial state s^0. We first set $\lambda''(s^{0\prime}) = \lambda'(s^{0\prime})$, so the labeling of the initial state of the now extended composition remains unchanged. We only need a labeling for the new initial state s^0. But here we can simply use $\lambda(n)$, i.e. $\lambda''(s^0) = \lambda(n)$. The node label definition $\lambda(n) = (\lambda(n') \setminus (Post_o[\sigma] \cup T)) \cup Pre_o[\sigma_{in}]$ implies that $\lambda(n) \wedge Post_o[\sigma] \wedge \Omega \wedge \mathcal{T} \models \lambda(n')$. Plugging in the above definitions, this equals the assertion that $\lambda''(s^0) \wedge Post_o[\sigma] \wedge \Omega \wedge \mathcal{T} \models \lambda''(s^{0\prime})$, which implies that λ'' is a valid labeling for $c(n)$. Since $\lambda''(s^0) = \lambda(n)$, λ'' itself is the witness that proves $c(n)$ to transform $\lambda(n)$ into $Post_q$.

So we conclude that $c(n)$ transforms $\lambda(n)$ into $Post_q$ minimally for any node n in G_{BW}. □

Theorem 4.2. *The search structure BW is correct.*

Proof. Recall that correctness of a search structure means the following (cf. Section 3.1.3.1): Let q be a query and $p = (n^0_{\mathrm{BW}}, .., n)$ be a path of the search structure graph G_{BW} with $\star_{\mathrm{BW}}(n) = \mathit{true}$. Then $\textsc{Trans}_{\mathrm{BW}}(p)$ is not empty and each of its elements is a solution to q.

Let p be such a path and $\star_{\mathrm{BW}}(n) = \mathit{true}$. By construction, $\textsc{Trans}_{\mathrm{BW}}(p) = \{c(n)\}$ maps each node to a set with *exactly* one composition, so we only need show that $c(n)$ is a solution to the query q, which means that $c(n)$ must transform Pre_q into $Post_q$ and $\odot(c) \leq Z_q$.

Both conditions follow from the definition of $\star_{\mathrm{BW}}$ and the previous Lemma. First, $\star_{\mathrm{BW}}(n) = \mathit{true}$ implies that $Pre_q \wedge \mathcal{T} \models \lambda(n)$. By Lemma 4.1, $c(n)$ transforms $\lambda(n)$ into $Post_q$. The only problem could be that some of the data containers in the query inputs have other type definitions in $\lambda(n)$ than in Pre_q. We can solve this problem as follows. Let λ' be a valid state labeling for $c(n)$ that shows how it transforms $\lambda(n)$ into $Post_q$. We adjust λ' for the initial state of $c(n)$ by replacing the types defined for query inputs by the actual types in Pre_q. Obviously, the adjusted λ' is still valid and a witness that $c(n)$ transforms Pre_q into $Post_q$. The condition $\odot(c) \leq Z_q$ is directly implied by the definition of $\star_{\mathrm{BW}}$. □

Lemma 4.4. *Let β be a finite conjunction of ground literals and n be a node in G_{BW} such that $\beta \wedge \Omega \wedge \mathcal{T} \models \lambda(n)$. Then there is a finite path from n to a node n' with $\beta \wedge \mathcal{T} \models \lambda(n')$ and $\lambda(n') \wedge \Omega \wedge \mathcal{T} \models \lambda(n)$.*

Proof. First observe that, by $\beta \wedge \Omega \wedge \mathcal{T} \models \lambda(n)$, we know that $\alpha = \beta \wedge \Omega \wedge \mathcal{T} \wedge \neg\lambda(n)$ is contradictory. Since $\lambda(n)$ contains only positive literals, α is a Horn formula and $\neg\lambda(n)$ is a negative clause. In particular, there is a linear input resolution refutation for α with a negative top clause and only definite side clauses [5]. Since $\neg\lambda(n)$ must be part of the proof, it must be the top clause.

The proof goes then by induction over the number k of Ω-clauses (clauses from Ω) used by linear input resolution to infer *any* state from β using Ω and $\mathcal{T}$. In other words, given a conjunction of ground literals β, then for any $k \in \mathbb{N}$ and for any node n in G_{BW} such that there is a (minimal) linear input resolution refutation for $\beta \wedge \Omega \wedge \mathcal{T} \wedge \neg\lambda(n)$ using k Ω-clauses, there is a path of length k in G_{BW} from n to a node n' and $\beta \wedge \mathcal{T} \models \lambda(n')$.

Induction Basis. Let $k = 0$. Then no Ω-clause is used for the refutation. But then Ω is not necessary for the refutation, and $\beta \wedge \mathcal{T} \models \lambda(n)$.

Inductive Step. Let $k > 0$. Then there is a side clause of the form $\neg P_1 \vee .. \vee \neg P_m \vee L$ from Ω involved in the refutation where L is the literal over which we resolve. Note that L is not a type literal, because clauses may contain type literals only with negation.

We can assume that this is the first side clause in the refutation. Since L is not a type predicate, no clause from $\mathcal{T}$ is necessary to derive L. The only other side clauses not from Ω could be unit clauses from β or $\neg\lambda(n)$ itself. Even if $\neg\lambda(n)$ was a unit clause, it would be non-definite Horn and not be used as a side clause. For unit clauses from β, we can assume that they occur at the end of the refutation without loss of generality.

But then we know that there exists and edge from n to some n'' such that $\lambda(n'') = (\lambda(n) \setminus \{L\}) \cup \{\neg P_1, .., \neg P_m\}$. Since we resolve $\neg P_1 \vee .. \vee \neg P_m \vee L$ and $\neg\lambda(n)$ over L, $\neg L$ must be in $\neg\lambda(n)$, which means that $L \in \lambda(n)$. This satisfies condition (3b) of Def. 16 (Section 4.2.1), and the above edge exists.

Then the existence of a node n' with $\beta \wedge \mathcal{T} \models \lambda(n')$ follows directly from the induction hypothesis. We know that $\neg\lambda(n'')$ corresponds to the resolvent of the above resolution step with clause $\neg P_1 \vee .. \vee \neg P_m \vee L$. So we know that there exists a refutation of $\beta \wedge \Omega \wedge \mathcal{T} \wedge \neg\lambda(n'')$ that uses $k-1$ Ω-clauses (namely exactly the rest of the above refutation). So, the Lemma holds for $\lambda(n'')$ by the induction hypothesis, and we can conclude that there is a path from n'' to a node n' such that $\beta \wedge \mathcal{T} \models \lambda(n')$. Since n'' is a direct successor of n, there is also a path from n to n', which yields the Lemma.

□

Lemma 4.5. *Let q be a query, β be a finite conjunction of ground literals, and $c \in \mathscr{S}$ be a sequential composition that transforms β into $Post_q$ minimally (cf. Def. 12 in Section 2.1.2.2). Then there is a node n in G_{BW} such that $c = c(n)$ and $\beta \wedge \mathcal{T} \models \lambda(n)$.*

Proof. The proof is by induction over the number of transitions k of c.

Induction Basis. Let $k = 0$. So there is a labeling $\widehat{\lambda}$ such that, for the only state s of c, we have $\beta \models \widehat{\lambda}(s)$ and $\widehat{\lambda}(s) \models Post_q$; in particular $\beta \models Post_q$. Then we can simply set $c = c(n^0_{\mathrm{BW}})$. By construction, $\lambda(n) = Post_q$; so β implies $\lambda(n)$, and, in particular, $\beta \wedge \mathcal{T} \models \lambda(n)$.

Inductive Step. Suppose that $c = (S, \Sigma, \delta, s^0, F)$ has k transitions (operation invocations) with $k > 0$. Since c transforms β into $Post_q$ minimally, there is a valid state labeling $\widehat{\lambda}$ for c such that $\beta \models \widehat{\lambda}(s^0)$.

By the induction hypothesis, for the composition c' induced by c without the first transition, there is a node n' such that $c' = c(n')$. Let $\delta(s^0, o[\sigma]) = s'$ be the transition starting from the initial state of c. Now let $c' = \langle S \setminus \{s^0\}, \Sigma \setminus \{o[\sigma]\}, \delta \setminus \{((s^0, o[\sigma]), s')\}, s', F\rangle$ denote the composition without this initial transition. c' is a sequential composition with $k-1$ transitions that minimally transforms $\widehat{\lambda}(s')$ into $Post_q$. Then, by the induction hypothesis, we know that there is a node $n' \in N_{\mathrm{BW}}$ with $c' \in \textsc{Trans}_{\mathrm{BW}}(n')$, i.e. $c' = c(n')$, and $\widehat{\lambda}(s') \wedge \mathcal{T} \models \lambda(n')$.

We can infer that the *node* label of n' is implied by the *state label* of the initial state of c and the postcondition of $o[\sigma]$, i.e. $\widehat{\lambda}(s^0) \wedge \Omega \wedge \mathcal{T} \wedge Post_o[\sigma] \models \lambda(n')$. The presence of the first transition $\delta(s^0, o[\sigma]) = s'$ in c implies $\widehat{\lambda}(s^0) \wedge \Omega \wedge \mathcal{T} \wedge Post_o[\sigma] \models \widehat{\lambda}(s')$. This is because $\widehat{\lambda}(s')$ is a valid label of the successor state s' of s^0 (cf. Def. 11 in Section 2.1.2.2). Combining this with $\widehat{\lambda}(s') \wedge \mathcal{T} \models \lambda(n')$ from the induction hypothesis, the claim holds by transitivity of $\models$.

Then we can follow that there is a node n'' reachable from n' such that $\widehat{\lambda}(s^0) \wedge Post_o[\sigma] \wedge \mathcal{T} \models \lambda(n'')$. This follows directly from Lemma 4.4 using $\widehat{\lambda}(s^0) \wedge Post_o[\sigma]$ as the respective β.

We can now show that there is an operation invocation edge from n'' to a node n''' labeled $o[\sigma]$ in E_{BW} such that $\lambda(n''') = (\lambda(n'') \setminus Post_o[\sigma]) \cup Pre_o[\sigma_{in}]$. To prove this, I show that the three conditions posed on operation invocation edges in (2) of Def. 16 (cf. Section 4.2.1) are satisfied for $o[\sigma]$ in n''.

1. *Show Relevance (2c).* The state associated with n'' contains at least one literal that can be derived from $Post_o[\sigma] \wedge \mathcal{T}$. Suppose the opposite, then $\widehat{\lambda}(s^0) \wedge \mathcal{T} \models \lambda(n')$. Then c' achieves $Post_q$ minimally not only from $\lambda(n')$ but even from $\widehat{\lambda}(s^0)$, and, hence, from β; then the initial transition is not necessary and c is not minimal, which contradicts our assumption.

2. *Show Type Compatibility (2d).* What we need is that $Post_o[\sigma] \wedge \mathcal{T} \models T$ where T are the literals $t(v)$ in $\lambda(n'')$ such that an output y of o exists with $\sigma_{out}(y) = v$. Now consider a literal $t(v) \in T$. Since $t(v) \in \lambda(n'')$, by the above, we know that $\widehat{\lambda}(s^0) \wedge Post_o[\sigma] \wedge \mathcal{T} \models t(v)$. But we know that the constant v does *not* occur in $\widehat{\lambda}(s^0)$; otherwise $o[\sigma]$ would not have been applicable in s^0 under $\widehat{\lambda}$. But then, it must hold that $Post_o[\sigma] \wedge \mathcal{T} \models t(v)$.

3. *Show Consistency (2e).* The last requirement is that the resulting state does not contain any data container targeted by the output mapping. Suppose that there is a literal L in $\lambda(n''')$ with a data container v and $\sigma_{out}(y) = v$ for some output y of o. By definition, $\lambda(n''') = (\lambda(n'') \setminus (Post_o[\sigma] \cup T)) \cup Pre_o[\sigma_{in}]$. Obviously, L is not in $Pre_o[\sigma_{in}]$, so it must be in $\lambda(n'')$ and neither in $Post_o[\sigma]$ nor in T. In particular, $\widehat{\lambda}(s^0) \wedge Post_o[\sigma] \wedge \mathcal{T} \models \lambda(n'') \models L$ then holds. Also, we know that v does not occur in $\widehat{\lambda}(s^0)$; otherwise $o[\sigma]$ would not be applicable in s^0 under $\widehat{\lambda}$. Hence, $Post_o[\sigma] \wedge \mathcal{T} \models L$. If L is not a type literal, which would make it member of T, it must be in $Post_o[\sigma]$. In any case, $L \notin \lambda(n''')$.

This completes the proof in so far that we know that there is a node n''' such that $c = c(n''')$, i.e. composition c is covered[2]. However, we must still show that there is a path from n''' to a node n such that also $\beta \wedge \mathcal{T} \models \lambda(n)$, which is not necessarily n'''.

I first show that $\beta \wedge \Omega \wedge \mathcal{T} \models \lambda(n''')$. Recall again that $\lambda(n''') = (\lambda(n'') \setminus (Post_o[\sigma] \cup T)) \cup Pre_o[\sigma_{in}]$. Now let L be a literal in $\lambda(n''')$. Either $L \in Pre_o[\sigma_{in}]$. In this case, by applicability of $o[\sigma]$ in s^0 under $\widehat{\lambda}$, the above follows because of $\beta \models \widehat{\lambda}(s^0)$ and $\widehat{\lambda}(s^0) \wedge \Omega \wedge \mathcal{T} \models Pre_o[\sigma_{in}]$. Or $L \notin Pre_o[\sigma_{in}]$, which implies that $L \in \lambda(n'')$ and $L \notin T$ and $L \notin Post_o[\sigma]$. Combining this with the fact that $\widehat{\lambda}(s^0) \wedge Post_o[\sigma] \wedge \mathcal{T} \models \lambda(n'')$, we also know that $\widehat{\lambda}(s^0) \wedge Post_o[\sigma] \wedge \mathcal{T} \models L$; using $L \notin Post_o[\sigma]$ yields $\widehat{\lambda}(s^0) \wedge \mathcal{T} \models L$. By $\beta \models \widehat{\lambda}(s^0)$, which follows from the validity of c under $\widehat{\lambda}$, we even have $\beta \wedge \mathcal{T} \models L$; in particular $\beta \wedge \Omega \wedge \mathcal{T} \models L$.

Then, applying again Lemma 4.4 gives us that there is a node n such that $c = c(n)$ and $\beta \wedge \mathcal{T} \models \lambda(n)$. There is a path from n' to n'', an edge labeled $o[\sigma]$ from n'' to n''', and a path from n''' to n. We know that $\textsc{Trans}_{\text{BW}}((n^0_{\text{BW}}, .., n))$ prepends $o[\sigma]$ to the result of $\textsc{Trans}_{\text{BW}}((n^0_{\text{BW}}, .., n'))$, which yields c.

□

Theorem 4.6. *The search structure BW is complete.*

Proof. Let q be a query. Two conditions need to be shown:

1. for every class C of equivalent solutions in in $\mathscr{S}^*$, there must be a path p such that $\textsc{Trans}_{\text{BW}}(p) \cap C \neq \emptyset$. We defined the equivalence classes for compositions based on the data container renaming (cf. Section 2.1.2), so this amounts to say that for every solution composition c, there must be a path p such that $c' \in \textsc{Trans}_{\text{BW}}(p)$ where c

[2]In fact, we have assumed that the naming of new variables in σ_{in} is identical in G_{BW}, which is not necessarily the case. However, we would still obtain an equivalent composition. For simplicity, I left this aspect out of the proof.

and c' are equivalent modulo data container renaming. The existence of a node n with $c' = c(n) = \textsc{Trans}_{\text{BW}}((n^0_{\text{BW}}, .., n))$ is precisely the assertion of Lemma 4.5.

2. At least one of the nodes that are reached by a path $p = (n^0, .., n)$ where $\textsc{Trans}(p)$ is a solution must recognize the solution by $\star(n) = \mathit{true}$. Now consider for the above query q and solution c the path to one node n we obtained by Lemma 4.5 for setting $\beta = \mathit{Pre}_q$. The Lemma tells us that $\mathit{Pre}_q \wedge \mathcal{T} \models \lambda(n)$, which is the first condition of $\star_{\text{BW}}$. Since c is a solution and $c(n)$ is equivalent to c, $\odot(c) = \odot(c(n)) = \odot(n) \leq Z_q$, which is the second condition; hence, $\star_{\text{BW}}(n) = \mathit{true}$.

□

Lemma 4.7. *Let $n_1 \succeq_{BW} n_2$ be true and let (n_2, n'_2) be an edge labeled $o[\sigma]$ in G_{BW}. Then $n_1 \succeq_{BW} n'_2$ or there is a direct successor n'_1 of n_1 in G_{BW} such that $n'_1 \succeq_{BW} n'_2$.*

Proof. Assume that $n_1 \succeq_{\text{BW}} n'_2$ does not hold; otherwise we are done. I now show that there is an edge from n_1 that is analogous to the one from n_2 to n'_2, i.e. we can apply the same operation backwards (with a different input and output mapping) such that the resulting node satisfies the desired property. The rest of the proof focuses on explaining how the alternative mapping σ' and the projection from data containers in $\lambda(n'_1)$ to those in $\lambda(n'_2)$ is created.

By $n_1 \succeq_{\text{BW}} n_2$, we know that there exists an injective φ such that $\lambda(n_1)[\varphi] \subseteq \lambda(n_2)$ holds. I show that there are an operation invocation $o[\sigma']$ and a mapping $\varphi' : \Gamma_{data}(n'_1) \to \Gamma_{data}(n'_2)$ such that there is a an edge from n_1 to some n'_1 labeled $o[\sigma']$ and $\lambda(n'_1)[\varphi'] \subseteq \lambda(n'_2)$ holds.

Preliminaries. First, let us group the data containers that occur in the label of the successor node of n_2, hence in $\lambda(n'_2)$ with respect to their occurrence in the label of n_1, hence $\lambda(n_1)$. To this end, define the following three sets:

- $\mathit{NEW} = \{v \mid v \in \Gamma_{data}(n'_2), v \notin \Gamma_{data}(n_2)\}$
- $\mathit{OLD\_KNOWN} = \{v \mid v \in \Gamma_{data}(n_2) \wedge v \in \varphi(\Gamma_{data}(n_1))\}$
- $\mathit{OLD\_UNKNOWN} = \{v \mid v \in \Gamma_{data}(n_2) \wedge v \notin \varphi(\Gamma_{data}(n_1))\}$

Intuitively, based on the input mapping σ_{in} of the operation invocation $o[\sigma]$ used to label the edge from n_2 to n'_2, *NEW* is the set of *newly inserted* data containers, *OLD_KNOWN* is the set of data containers that already existed in the label of *both* n_2 and n_1 (modulo φ), and *OLD_UNKNOWN* is the set of data containers that existed in the label of n_2 *but not* in n_1 (modulo φ).

Construction of σ' and φ'. Based on these sets, we can define the mapping σ' of the operation invocation for the edge outgoing from n_1 and the extended data container mapping φ'. φ' extends φ by a mapping of the data containers that are newly introduced by σ'_{in}, so we first define $\varphi'(v) = \varphi(v)$ for each data container v for which φ is defined. Moreover, let φ^{-1} be the inverse function of φ; φ^{-1} is defined since φ is injective.

Now we define the input mapping σ'_{in} and the remaining part of φ'. Let $x \in X_o$ be any input of operation o. Three cases are possible:

1. $\sigma(x) \in \mathit{NEW}$. So $o[\sigma]$ introduced a new data container $v_x^{n'_2}$ and $o[\sigma']$ will do the same. Hence, define $\sigma'_{in}(x) = v_x^{n'_1}$ and $\varphi'(v_x^{n'_1}) = v_x^{n'_2}$.
2. $\sigma(x) \in \mathit{OLD\_KNOWN}$. So $o[\sigma]$ binds x to a data container that is already targeted in φ. Then $\varphi'(\sigma'_{in}(x))$ is already defined as $\varphi(\sigma'_{in}(x))$, and we only set $\sigma'_{in}(x) = \varphi^{-1}(\sigma(x))$.

3. $\sigma(x) \in \mathit{OLD\_UNKNOWN}$. So $o[\sigma]$ binds x to a data container known in $\lambda(n_2)$ and *not* targeted by φ; hence, we need a new one in $\lambda(n_1')$. Then set $\sigma'_{in}(x) = v_x^{n_1'}$ and $\varphi'(v_x^{n_1'}) = \sigma(x)$.

Given this complete mapping for inputs and data containers, we define σ'_{out}. Let $y \in Y_o$ be an output of the operation o. Either $\sigma(y) \in \mathit{OLD\_KNOWN}$ or $\sigma(y) \in \mathit{OLD\_UNKNOWN}$. In the first case, we define $\sigma'_{out}(y) = \varphi^{-1}(\sigma(y))$. In the second case, we do not define $\sigma'_{out}(y)$. Note that the mapping φ' is already defined for the data containers involved here.

The above definition of σ' and φ' implies that the concatenated substitution of σ' and φ' *equals* σ for every input $x \in X_o$ and every bound output $y \in Y_o$ for which σ'_{out} is defined. That is, $\varphi'(\sigma'(z)) = \sigma(z)$ holds for every $z \in X_o$ and every $z \in Y_o$ for which $\sigma'_{out}(z)$ is defined. In particular, $L[\sigma'][\varphi'] = L[\sigma]$ holds for every literal in the preconditions of o and every literal in the postcondition of o that contains only outputs that are bound in σ'_{out}.

Proof of Desired Property. Given σ' and φ', it remains to show that G_{BW} contains an edge from n_1 to a node n_1' with label $o[\sigma']$ and $\lambda(n_1')[\varphi'] \subseteq \lambda(n_2')$. To this end, we consider the label $\beta = (\lambda(n_1) \setminus (\mathit{Post}_o[\sigma'] \cup T') \cup \mathit{Pre}_o[\sigma'_{in}])$ that *would* belong to a node n_1' obtained through an edge $o[\sigma]$ from n_1. According to the search graph definition (cf. (2a) of Def. 16 in Section 4.2.1), T' is the set of type predicates in $\lambda(n_1)$ of data containers that are bound to outputs of the prepended action, i.e. $T' = \{t(v) \mid \exists y \in Y_o : \sigma'_{out}(y) = v, t(v) \in \lambda(n_1)\}$. The remaining proof consists in showing that $\beta[\varphi'] \subseteq \lambda(n_2')$ and that there actually *is* an edge (n_1, n_1') labeled $o[\sigma']$.

1. *Show that* $\beta[\varphi'] \subseteq \lambda(n_2')$.

   Let $L \in \beta$. We must show that $L[\varphi'] \in \lambda(n_2')$.

   By definition of β, we know that $L \in \mathit{Pre}_o[\sigma'_{in}]$ or $L \notin (\mathit{Post}_o[\sigma'] \cup T')$, and by Def. 16, we know that $\lambda(n_2') = (\lambda(n_2) \setminus (\mathit{Post}_o[\sigma] \cup T)) \cup \mathit{Pre}_o[\sigma_{in}]$ where $T = \{t(v) \mid \exists y \in Y_o : \sigma_{out}(y) = v, t(v) \in \lambda(n_2)\}$ are the type predicates in $\lambda(n_2)$ of data containers bound to outputs of o through σ. Two cases are possible:

   (a) $L \in \mathit{Pre}_o[\sigma'_{in}]$. Then there is a literal $L' \in \mathit{Pre}_o$ and $L'[\sigma'_{in}] = L$. Mapping L' with σ_{in} instead of σ'_{in}, we have $L'[\sigma_{in}] \in \mathit{Pre}_o[\sigma_{in}]$. By construction of σ'_{in} and φ', we have $L'[\sigma_{in}] = L'[\sigma'_{in}][\varphi']$. But then, $L'[\sigma_{in}] = L[\varphi']$ and, thereby, $L[\varphi'] \in \mathit{Pre}_o[\sigma_{in}]$ holds. Every literal of $\mathit{Pre}_o[\sigma_{in}]$ is in $\lambda(n_2')$, so in particular $L[\varphi'] \in \lambda(n_2')$.

   (b) $L \notin \mathit{Pre}_o[\sigma'_{in}]$. Since $L \in \beta$, it must be the case that $L \in \lambda(n_1)$, that $L \notin \mathit{Post}_o[\sigma']$, and that $L \notin T'$. By $L \in \lambda(n_1)$ and $n_1 \succeq_{\mathrm{BW}} n_2$, we know that all data containers in $L[\varphi']$ are in $\mathit{OLD\_KNOWN}$, which implies $L[\varphi'] = L[\varphi]$, and that $L[\varphi] \in \lambda(n_2)$. Since $L[\varphi]$ is in $\lambda(n_2)$, we must only show that it is not removed by the backward application of the operation invocation, i.e. we must show that $L[\varphi] \notin \mathit{Post}_o[\sigma]$ and $L[\varphi] \notin T$.

      i. *Show that* $L[\varphi] \notin \mathit{Post}_o[\sigma]$. Suppose the contrary, i.e. $L[\varphi] \in \mathit{Post}_o[\sigma]$. Then there is a literal $L' \in \mathit{Post}_o$ such that $L'[\sigma'][\varphi] = L[\varphi]$, and, by injectivity of φ, $L'[\sigma'] = L$. But this would imply $L \in \mathit{Post}_o[\sigma']$, which we know to be false.

      ii. *Show that* $L[\varphi] \notin T$. The proof is again by contradiction, so suppose that $L[\varphi] \in T$. Then $L[\varphi] = t(\varphi(v))$ and $L = t(v)$ where v is the data container in β. By definition of T, we know that there is an output $y \in Y_o$ of operation o that is bound to $\varphi(v)$, i.e. $\sigma_{out}(y) = \varphi(v)$. Now recall that $\varphi(v) \in \mathit{OLD\_KNOWN}$, so applying the definition of σ'_{out} to y, we get $\sigma'_{out}(y) = \varphi^{-1}(\sigma_{out}(y)) =$

$\varphi^{-1}(\varphi(v)) = v$. So σ'_{out} binds the output y of operation o to the data container v. But then, $t(v) = L$ would, due to its membership in $\lambda(n_1)$, also be in T', which we know to be false.

This completes the first part of the proof, i.e. we know that $\beta[\varphi'] \subseteq \lambda(n'_2)$; in particular, $\lambda(n'_1)[\varphi'] \subseteq \lambda(n'_2)$ if the node exists. Now we only need to show that n'_1 is a successor of n_1 over an edge labeled $o[\sigma']$.

2. *Show that there is* (n_1, n'_1) *with label* $o[\sigma']$ *such that* $\lambda(n'_1) = \beta$. To this end, I show that the conditions for the existence of an edge defined in (2) of Def. 16 hold. The mapping σ' satisfies the signature conditions, since $\sigma'_{in} : X_o \to \Gamma_{data}(n_1) \cup \{v_{x1}^{n'_1}, \ldots, v_{x|X_o|}^{n'_1}\}$ and $\sigma'_{out} : Y_o \rightsquigarrow \Gamma_{data}(n_1)$. $\lambda(n'_1) = \beta$ holds directly by the state definition (2b). It remains to show the three conditions:

   (a) *(2d) the types of produced data containers are consistent with the type hierarchy, i.e.* $Post_o[\sigma'_{out}] \wedge \mathcal{T} \models \bigwedge_{t \in T'} t$.
   I show that $Post_o[\sigma'_{out}] \wedge \mathcal{T} \models t(v)$ holds for every $t(v) \in T'$. Since the edge (n_2, n'_2) exists with label $o[\sigma]$, we know that $Post_o[\sigma] \wedge \mathcal{T} \models \bigwedge_{t \in T} t$; in particular, $Post_o[\sigma_{out}] \wedge \mathcal{T} \models t(\varphi(v))$ for every $t(\varphi(v)) \in T$. Obviously, the consequence holds also for a sub-formula ${Post_o}^U[\sigma]$ of $Post_o[\sigma]$ that only contains the literals with data containers of *OLD_KNOWN*, i.e. ${Post_o}^U[\sigma] \wedge \mathcal{T} \models t(\varphi(v))$. But for this sub-formula, we also know that ${Post_o}^U[\sigma] = {Post_o}^U[\sigma'][\varphi]$ holds, so ${Post_o}^U[\sigma'][\varphi] \wedge \mathcal{T} \models t(\varphi(v))$. But then (since φ is injective), it also holds that $Post_o[\sigma'] \wedge \mathcal{T} \models {Post_o}^U[\sigma'] \wedge \mathcal{T} \models t(v)$.

   (b) *(2e) Produced data containers do neither occur in the rest problem nor in the query inputs, i.e.* $\sigma'_{out}(Y_o) \cap (\Gamma_{data}(n'_1) \cup X_q) = \emptyset$. The proof is by contradiction; that is, suppose that an element is in $\sigma'_{out}(Y_o)$ and one of the sets $\Gamma_{data}(n'_1)$ or X_q at the same time.
   Let $v \in \sigma'_{out}(Y_o)$, i.e. there is a $y \in Y_o$ such that $v = \sigma'_{out}(y)$. Recall that by the definition of σ'_{out}, $\sigma'_{out}(y) = \varphi^{-1}(\sigma_{out}(y))$ and, hence, $v = \varphi^{-1}(\sigma_{out}(y))$. Moreover, recall that $\sigma_{out}(Y_o) \cap \Gamma_{data}(n'_2) = \emptyset$ and $\sigma_{out}(Y_o) \cap X_q = \emptyset$ holds, because condition (2e) is true for the edge (n_2, n'_2).
      i. Suppose that $v \in \Gamma_{data}(n'_1)$. Then $\varphi'(v) \in \Gamma_{data}(n'_1)[\varphi']$ and, since $\lambda(n'_1)[\varphi'] \subseteq \lambda(n'_2)$, $\varphi'(v) \in \Gamma_{data}(n'_2)$. Since $v = \sigma'_{out}(y) = \varphi^{-1}(\sigma_{out}(y))$, we also know that $\varphi'(\varphi^{-1}(\sigma_{out}(y))) = \sigma_{out}(y) \in \Gamma_{data}(n'_2)$ holds, which contradicts $\sigma_{out}(Y_o) \cap \Gamma_{data}(n'_2) = \emptyset$.
      ii. Suppose that $v \in X_q$. Then $\varphi'(v) \in X_q[\varphi'] = X_q$; the equation holds because φ' replaces only introduced variables, which do not occur in X_q. But then $\varphi'(\varphi^{-1}(\sigma_{out}(y))) = \sigma_{out}(y) \in X_q$, which contradicts $\sigma_{out}(Y_o) \cap X_q = \emptyset$.

   (c) *(2f). If* σ' *maps an input* x *to a new data container, it must choose* $v_x^{n'_1}$ *as a name.* This condition is obviously true by construction.

□

Lemma 4.8. *Let* $n_1 \succeq_{BW} n_2$ *be true and let* (n_2, n'_2) *be an edge in* G_{BW} *with the label* $\left(\left(\bigwedge_{j \neq i} \alpha_j\right) \to \alpha_i\right)[\sigma]$. *Then* $n_1 \succeq_{BW} n'_2$ *or there is a direct successor* n'_1 *of* n_1 *in* G_{BW} *such that* $n'_1 \succeq_{BW} n'_2$.

Proof. Assume that $n_1 \succeq_{\mathrm{BW}} n_2'$ does not hold; otherwise we are done. Like in Lemma 4.7, the idea is to show that we can apply the same clause backwards in n_1.

What we must do is to show that there is a clause application $\left(\left(\bigwedge_{j\neq i}\alpha_j\right)\rightarrow\alpha_i\right)[\sigma']$ ($\alpha[\sigma']$ for short) and a mapping $\varphi' : \Gamma_{data}(n_1') \rightarrow \Gamma_{data}(n_2')$ such that there is a an edge from n_1 to some n_1' labeled $\alpha[\sigma']$ and $\lambda(n_1')[\varphi'] \subseteq \lambda(n_2')$ holds.

Like in the proof of Lemma 4.7, let us consider sets of the data containers in $\lambda(n_2')$. I.e. suppose that *NEW*, *OLD_KNOWN*, *OLD_UNKNOWN* are defined as in the above proof of Lemma 4.7.

Based on these sets, we can define σ' and φ' stepwise. Let $\varphi'(v) = \varphi(v)$ for each data container v for which φ is defined. In the following, let φ^{-1} be the inverse function of φ; φ^{-1} is defined since φ is injective.

Now we define σ' and the rest of φ'. Let $x \in vars(\alpha)$ be any variable that occurs in the clause α. We construct σ' in a way such that for any $x \in vars(\alpha)$ it holds that $\varphi'(\sigma'(x)) = \sigma(x)$. Three cases are possible:

1. $\sigma(x) \in NEW$. Then $\sigma(x) = v_x^{n_2'}$. Define $\sigma'(x) = v_x^{n_1'}$ and $\varphi'(v_x^{n_1'}) = v_x^{n_2'}$. Then $\varphi'(\sigma'(x)) = \varphi'(v_x^{n_1'}) = v_x^{n_2'} = \sigma(x)$ holds.
2. $\sigma(x) \in OLD\_KNOWN$. Define $\sigma'(x) = \varphi^{-1}(\sigma(x))$. Then $\varphi'(\sigma'(x)) = \varphi'(\varphi^{-1}(\sigma(x))) = \sigma(x)$ holds obviously.
3. $\sigma(x) \in OLD\_UNKNOWN$. Define $\sigma'(x) = v_x^{n_1'}$ and $\varphi'(v_x^{n_1'}) = \sigma(x)$. Then $\varphi'(\sigma'(x)) = \sigma(x)$ holds.

Now define $\beta = (\lambda(n_1) \setminus \{\alpha_i[\sigma']\}) \cup \left(\bigcup_{j\neq i}\alpha_j[\sigma']\right)$.

We must show that $\beta[\varphi'] \subseteq \lambda(n_2')$ and that there exists an edge (n_1, n_1') labeled $\alpha[\sigma']$ such that $\beta = \lambda(n_1')$.

1. *Show* $\beta[\varphi'] \subseteq \lambda(n_2')$. Let $L \in \beta$. We must show that $L[\varphi'] \in \lambda(n_2')$ where $\lambda(n_2') = \lambda(n_2) \setminus \{\alpha_i[\sigma]\} \cup \left(\bigcup_{j\neq i}\alpha_j\right)$. Two cases are possible:
   (a) $L = \alpha_j[\sigma']$. Then $L[\varphi'] = \alpha_j[\sigma'][\varphi'] = \alpha_j[\sigma] \in \lambda(n_2')$.
   (b) $L \neq \alpha_j[\sigma']$. Then $L \in \lambda(n_1)$ and $L \neq \alpha_i[\sigma]$, and, in particular, $L[\varphi] = L[\varphi']$ and $L[\varphi'] \neq \alpha_i[\sigma'][\varphi'] = \alpha_i[\sigma]$; hence, $L[\varphi'] \neq \alpha_i[\sigma]$. By $n_1 \succeq_{\mathrm{BW}} n_2$, we know that $L[\varphi]$, and, hence, $L[\varphi']$, is in $\lambda(n_2)$. But then $L[\varphi']$ is in $\lambda(n_2')$.
2. *Show that there is* (n_1, n_1') *with label* $\alpha[\sigma']$ *in* G_{BW} *such that* $\lambda(n_1') = \beta$. To this end, consider (3) of Def. 16. The mapping σ' satisfies the signature conditions, since $\sigma' : vars(\alpha) \rightarrow \Gamma_{data}(n_1) \cup \{v_1^{n_1'}, \ldots, v_{|vars(\alpha)|}^{n_1'}\}$. $\lambda(n_1') = \beta$ holds directly by the state definition (3a). It remains to show the following conditions:
   (a) *(3b) Show that* $\alpha_i[\sigma'] \in \lambda(n_1)$. We know that $n_1 \succeq_{\mathrm{BW}} n_2'$ does *not* hold, which implies that there exists a literal $L \in \lambda(n_1)$ such that there is no mapping $\hat{\varphi}$ with $L[\hat{\varphi}] \in \lambda(n_2')$. However, by $n_1 \succeq_{\mathrm{BW}} n_2$, there *is* a mapping φ such that $L[\varphi] \in \lambda(n_2)$. Since $\lambda(n_2)$ has exactly one literal that is not in $\lambda(n_2')$, namely $\alpha_i[\sigma]$, we know that $L[\varphi] = \alpha_i[\sigma]$. Since $\alpha_i[\varphi] \in \lambda(n_2)$, we know that $\alpha_i[\varphi] = \alpha_i[\varphi']$ and, thus, $L[\varphi] = L[\varphi']$. Then $L[\varphi'] = \alpha_i[\sigma] = \alpha_i[\sigma'][\varphi']$; hence, $L = \alpha_i[\sigma']$. Since $L \in \lambda(n_1)$, also $\alpha_i[\sigma'] \in \lambda(n_1)$.

(b) *(3c) If σ' maps a variable x to a new container, then $\sigma'(x) = v_x^{n'}$.*
This condition is obviously true by construction.

□

Proofs Related to PO

For the proofs regarding PO, recall the definition of the action labeling (cf. Section 5.3). Given a serialized partial composition $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$, this labeling is defined as follows:

$$\tilde{\lambda}(a_i) = \begin{cases} Post_{a_0}[\psi] & \text{if } i = 0 \\ \tilde{\lambda}(a_{i-1}) \cup Post_{a_i}[\psi] & \text{if } 1 \leq i < l \\ \textit{undefined} & \text{else} \end{cases}$$

Lemma 5.2. *Let G be an instance of G_{PO} for a query, n be a node in G with an empty agenda, and let $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$ be a serialized partial composition of $c(n)$ with $a_i, .., a_j$ being a sequence of clause actions in it. Then $\tilde{\lambda}(a_{i-1}) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_j)$.*

Proof. The proof is by induction over the length k of this chain.

Induction Basis. Let $k = 0$. Then $a_j = a_{i-1}$, and the claim is trivially true.

Inductive Step. Let $k > 0$. Then we can say that $k = j - i + 1$ and, by the induction hypothesis, we know that $\tilde{\lambda}(a_{i-1}) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_{j-1})$.

Now consider a particular $L \in \tilde{\lambda}(a_j)$. We must show that $\tilde{\lambda}(a_{j-1}) \wedge \Omega \wedge \mathcal{T} \models L$. Suppose that $L \notin \tilde{\lambda}(a_{j-1})$; otherwise the claim holds. Furthermore, suppose that $L = L'[\psi]$ where ψ is the grounding obtained through GETPARAMETERMAP.

Then we know that $L' \in Post_{a_j}$. By $L \in \tilde{\lambda}(a_j) = \tilde{\lambda}(a_{j-1}) \cup Post_{a_j}[\psi]$, by our above assumption that $L \notin \tilde{\lambda}(a_{j-1})$, and by $L = L'[\psi]$ this must be the case.

But then $\tilde{\lambda}(a_{j-1}) \wedge \Omega \wedge \mathcal{T} \models L$. Since a_j corresponds to a clause, it must be of the form $P_1 \vee .. \vee P_r \vee L$. In particular, $\neg P_1, .., \neg P_r$ are in Pre_{a_j}. By (2) of Lemma 5.1, we know that each of these literals $\neg P_i$ is in $\tilde{\lambda}(a_{j-1})$. But then, it can easily be seen, e.g. using unit resolution, that $\tilde{\lambda}(a_{j-1}) \wedge (P_1 \vee .. \vee P_r \vee L) \models L$; in particular $\tilde{\lambda}(a_{j-1}) \wedge \Omega \wedge \mathcal{T} \models L$. □

Lemma 5.3. *Let G be an instance of G_{PO} for query q, and let $p = (n_{PO}^0, .., n)$ be a path in G such that n has an empty agenda. Then every composition $c \in \text{TRANS}_{PO}(p)$ transforms Pre_q into $Post_q$.*

Proof. Let G be an instance of G_{PO} for query q, and let n be a node in G with an empty agenda. Moreover, let $c = \langle o_1[\sigma_1], .., o_m[\sigma_m]\rangle$ be a composition in $\text{TRANS}_{\text{PO}}(n)$ and $\langle a_0, a_1, .., a_l\rangle|_{BI,CL}$ be the serialized partial composition from which c was derived.

The basis of the proof is the labeling λ, which we define as follows:

$$\lambda(s_i) = \begin{cases} \tilde{\lambda}(a_0) & \text{if } i = 0 \\ \tilde{\lambda}(a_{l-1}) & \text{if } i = m \\ \tilde{\lambda}(\kappa_{BI}^{-1}(o_i[\sigma_i])) & \text{else} \end{cases}$$

where $\kappa_{BI}(a_i)$ is the operation invocation $o_i[\sigma_i]$ of the serialized partial composition corresponding induced by the action a_i; i.e. $\kappa_{BI}^{-1}(o_i[\sigma_i])$ is the action in the serialized partial composition that induced the operation invocation $o_i[\sigma_i]$ (cf. Section 5.2.2).

The proof consists of (i) showing that λ is a valid labeling for c and (ii) showing that λ even is a witness for the transformation of Pre_q into $Post_q$ by c. As before, ψ is used to denote the output of getParameterMap.

Validity of λ. To show that λ is a valid labeling for c, I first show that every operation invocation is executable in its respective state and then that the labeling of a state can be inferred from the labeling of the predecessor state, the postcondition of the applied operation invocation, and the background knowledge.

In the following, we will focus on an arbitrary but fixed operation invocation $o_i[\sigma_i]$ and its executor state s_{i-1}. Let $a_k = \kappa_{BI}^{-1}(o_i[\sigma_i])$ be the action belonging to operation invocation $o_i[\sigma_i]$, and a_j be the initial action a_0 if $o_i[\sigma_i]$ is the first operation invocation in the partial composition ($i = 1$) and $\kappa_{BI}^{-1}(o_{i-1}[\sigma_{i-1}])$ otherwise ($i > 1$). Then $\lambda(s_{i-1}) = \tilde{\lambda}(a_j)$. In any case $j < k$, and all actions between a_j and a_k are clause applications.

1. *Show that $o_i[\sigma_i]$ is applicable in s_{i-1} under λ.* In other words, we need to show that $\lambda(s_{i-1}) \wedge \Omega \wedge \mathcal{T} \models Pre_{o_i}[\sigma_i]$.

   The claim follows directly from the previous Lemmas. First, by Lemma 5.2, we know that $\tilde{\lambda}(a_j) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_{k-1})$, because there are only clause actions between a_j and a_k. And by (2) of Lemma 5.1, we know that $\tilde{\lambda}(a_{k-1}) \models Pre_o[\sigma]$ holds. Putting these together with the definition of $\lambda(s_{i-1})$ gives us that $\lambda(s_{i-1}) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_{k-1}) \models Pre_{o_i}[\sigma_i]$.

2. *Show that $\lambda(s_{i-1}) \wedge \Omega \wedge \mathcal{T} \wedge Post_{o_i}[\sigma_i] \models \lambda(s_i)$.*

   Let $L \in \lambda(s_i)$ and $L \notin Post_{o_i}[\sigma_i]$; otherwise the claim is trivially true.

   Since $L \notin Post_{o_i}[\sigma_i]$, L must be in $\tilde{\lambda}(a_{k-1})$. This is because $L \in \lambda(s_i) = \tilde{\lambda}(a_k) = \tilde{\lambda}(a_{k-1}) \cup Post_{a_k}[\psi]$ and $Post_{a_k}[\psi] = Post_{o_i}[\sigma_i]$.

   Then the claim again follows from Lemma 5.2. Since all actions between (and excluding) a_j and a_k are clause applications, we know that $\tilde{\lambda}(a_j) \wedge \Omega \wedge \mathcal{T} \models \tilde{\lambda}(a_{k-1})$. Since $\lambda(s_{i-1}) = \hat{\lambda}(a_j)$ and since $L \in \tilde{\lambda}(a_{k-1})$, we know that $\lambda(s_{i-1}) \wedge \Omega \wedge \mathcal{T} \models L$; in particular, $\lambda(s_{i-1}) \wedge \Omega \wedge \mathcal{T} \wedge Post_{o_i}[\sigma_i] \models \lambda(s_i)$.

This completes the proof of validity of λ.

Transformation from Pre_q into $Post_q$. In order to make λ a witness for the transformation, we need to show that $Pre_q \models \lambda(s_0)$ and $\lambda(s_m) \models Post_q$.

$Pre_q \models \lambda(s_0)$ holds by construction. This is because $\lambda(s_0) = \tilde{\lambda}(a_0) = Post_{a_0}[\psi] = Pre_q$.

Now consider the only final state s_m. By definition, $\lambda(s_m) = \tilde{\lambda}(a_{l-1})$, and a_{l-1} is the last action before a^* in the serialized partial composition. Now let $L \in Post_q$ be a literal that needs to be achieved; we must show that $L \in \lambda(s_m)$. By definition of a^*, there is a literal $L' \in Pre_{a^*}$ such that $L = L'[\psi]$. The agenda is empty, so there is a causal link $\langle a_j, L', a^* \rangle$ in $CL(n)$ with $j \leq l - 1$. By (1) of Lemma 5.1, we know that $L'[\psi] \in \tilde{\lambda}(a_j)$, and, by $j \leq l - 1$, we know that $L'[\psi] \in \tilde{\lambda}(a_{l-1})$. Using that $L = L'[\psi]$ and $\lambda(s_m) = \tilde{\lambda}(a_{l-1})$, we obtain that $L \in \lambda(s_m)$, which yields the claim.

□

Lemma 5.6. *Let q be a query such that $Pre_q \wedge \Omega \wedge \mathcal{T} \models Post_q$ and let G be an arbitrary but fixed search graph instance of G_{PO} for a run on q. Then there is a path p from n^0 to a node n in G such that $AG(n) = \emptyset$ and $AS(n)$ does not contain any operation actions.*

Proof. For every unsatisfiable Horn formula with exactly one negative clause, there is a finite SLD-refutation [5]. I show that there is a path from n^0 to n that mimics an SLD-refutation for $Pre_q \wedge \Omega \wedge \mathcal{T} \wedge \neg Post_q$, which, since Pre_q and $Post_q$ are positive, is a Horn formula and has only one negative clause $\neg Post_q$.

The proof for the Lemma is then by induction over the number of side-clauses (usages) from Ω in these proofs. In other words, for every $k \in \mathbb{N}$, for every query q, and for every search graph G on q it holds that there is a node n with the above properties in G if an SLD-refutation with k side-clauses from Ω exists for $Pre_q \wedge \Omega \wedge \mathcal{T} \wedge \neg Post_q$.

For the following, suppose that the flaw selected by SelectFlaw(n^0) is $\langle L, a\rangle$.

Induction Basis. Let $k = 0$. Then no side clause in the proof is from Ω.

Then $Pre_q \wedge \mathcal{T} \models Post_q$ and, in particular, $Post_{a_0} \wedge \mathcal{T} \wedge BI(n^0) \models L$. Then, by (2) of Def. 18, there is a successor n' of n^0 such that $AS(n') = AS(n^0)$ and with $CL(n') = \{\langle a_0, L, a^*\rangle\}$. Since no new action is inserted, the next addressed flaw will be also from a^*, so the same argument applies. This can only continue $|Pre_{a^*}|$ times; then we will reach a node n with $AG(n) = \emptyset$. No action was added, so $AS(n)$ cannot contain any action belong to an operation.

Inductive Step. Let $k > 0$. Then at least one side-clause is from Ω.

First, n^0 has a successor node n' for an arbitrary side-clause from the SLD refutation used to resolve L. Suppose that bw is an SLD refutation of length k for $Pre_q \wedge \Omega \wedge \mathcal{T} \wedge \neg Post_q$. We can reorder the side clauses of bw in a way that any used side-clause of the form $\neg P_1 \vee .. \vee \neg P_m \vee L$ is the first one to be used without changing the proof size; let this proof be bw'. For this clause, by (2) of Def. 18, we have an edge to a new successor n' of n^0 with a new action $\bar{a}$ that has precondition $Pre_{\bar{a}} = P'_1 \wedge .. \wedge P'_m$ and postcondition $Post_{\bar{a}} = L'$ such that L' and L unify.

Now we consider a subquery $\hat{q}$ with $Pre_{\hat{q}} = Pre_q$ and $Post_{\hat{q}} = (Post_q \setminus \{L\}) \cup \{P_1, .., P_m\}$. Let $\hat{G}$ be an arbitrary but fixed instance of G_{PO} of a run on $\hat{q}$.

By the induction hypothesis, there is a node $\hat{n}$ in $\hat{G}$ with the claimed properties with respect to $\hat{q}$. This is because we know that there exists an SLD-refutation of length $k-1$ for $Pre_q \wedge \Omega \wedge \mathcal{T} \wedge \neg Post_{\hat{q}}$, namely the rest of the proof of bw' starting from its first resolvent, which is precisely $\neg Post_{\hat{q}}$. So we know that $AS(\hat{n})$ does not contain operation actions and $AG(\hat{n}) = \emptyset$.

We will now iteratively create a path in G with n_c being the currently considered head node of that path. We use an auxiliary function γ to describe the relations between actions in $AS(n_c)$ and $AS(\hat{n})$, and δ to reference an action-literal pair. We set $\gamma(a_0) = \hat{a}_0$ and $\gamma(a^*) = \gamma(\bar{a}) = \hat{a}^*$. Initially, we consider n_c being n', i.e. the child of n^0 described previously. Given a flaw $\langle L, a\rangle$, I define $\delta(\langle L, a\rangle) = \delta(\langle L(a.v_1, .., a.v_m), a\rangle) = \langle L(\gamma(a).v_1, .., \gamma(a).v_m), \gamma(a)\rangle$ as the version of the flaw addressed in n_c in the partial composition of $\hat{n}$. Obviously, if $\langle L, a\rangle$ is a flaw, then $L(\gamma(a).v_1, .., \gamma(a).v_m)$ is a literal in $Pre_{\gamma(a)}$.

If $AG(n_c)$ is empty, we are ready, so consider the case that there is still a flaw and suppose that $\langle L, a\rangle$ is the one selected in n_c. We know that $\gamma(a)$ is in $AS(\hat{n})$ and, since $AG(\hat{n}) = \emptyset$, there is a causal link $\langle \hat{a}', \delta(\langle L, a\rangle), \gamma(a)\rangle \in CL(\hat{n})$.

Then there is at least one resolver for $\langle L, a\rangle$ that yields a new consistent partial composition. Since $AS(\hat{n})$ contains no operation actions, $\hat{a}'$ can only be a clause application or the init action. If $\hat{a}'$ corresponds to $\hat{a}_0$, then we can also use a_0 as a resolver for $\langle L, a\rangle$. Otherwise, suppose that α is the clause belonging to $\hat{a}'$. If there is no action $a' \in AS(n_c)$ such that $\gamma(a') = \hat{a}'$, we can use an edge that defines a new action a' from NewActions(n_c) for α with $\gamma(a') = \hat{a}'$. Otherwise, we can reuse a' to resolve this flaw and go along a respective edge. In both cases, we obtain a causal link of the form $\langle a', L, a\rangle$ for the new node n_c.

Walking along a path in G this way must eventually yield a node n with an empty agenda. In every step we use a causal link of $CL(\hat{n})$ for guidance. But we also use each of these links at most once; since γ is injective, we would otherwise resolve the same flaw $\langle L, a\rangle$ twice, which cannot be the case. Hence, $AG(n_c)$ must be empty after at most $|CL(\hat{n})|$ many steps.

Since we only introduced clause application actions, $AS(n_c)$ has no operation action. □

Lemma 5.8. *Let q be a query and c be a composition that solves q minimally. Moreover, let G be an instance of a search graph of G_{PO} for q. Then G contains a path $p = (n^0_{PO}, .., n)$ such that $c \in \text{TRANS}_{PO}(p)$ and $\star_{PO}(n) = true$.*

This proof is inspired by the proof of completeness of UCPOP [96]. Apart from the difference that we have no delete lists, I fixed a flaw of their proof that is based on the assumption that the solution path of the subquery is a subpath of the solution path of the actual query. This is not the case in general, because the flaw selected in the root node of the subquery may be a different one than in the root node of the original query, which, of course, propagates down to every other node. This becomes particularly drastic in the case of a randomized SELECTFLAW, where the selected flaw differs even for the *same* query in two different runs. However, the core idea of the proof is analogous.

Proof. The proof is by induction over all queries and all solutions of length k to those queries.

Induction Basis. Let $k = 0$. Let q be a query and the empty composition c_ε (length 0) be a solution to q. Then $Pre_q \wedge \Omega \wedge \mathcal{T} \models Post_q$, and, by the previous Lemma 5.6, there is a path $p = (n^0_{\text{PO}}, .., n)$ in G such that $AG(n) = \emptyset$ and $c_\varepsilon \in \text{TRANS}_{\text{PO}}(p)$. Since $\odot(n) = 0 \leq Z_q$, we also have $\star_{\text{PO}}(n) = true$.

Inductive Step. Let $k > 0$. That is, let q be a query with a solution $c = \langle o_1[\sigma_1], .., o_k[\sigma_k]\rangle$.

Now define two subqueries $\hat{q}_1$ and $\hat{q}_2$ as follows. First, we define $\hat{q}_1$ with $Pre_{\hat{q}_1} = Pre_q$, $Post_{\hat{q}_1} = Pre_{o_1}[\sigma_1]$, $Z_{\hat{q}_1} = 0$. Second, we define $\hat{q}_2$ with $Pre_{\hat{q}_2} = Pre_q \cup Post_{o_1}[\sigma_1]$, $Post_{\hat{q}_2} = Post_q$, and $Z_{\hat{q}_2} = Z_q$ respectively[3]. Let $\hat{G}_1$ and $\hat{G}_2$ be arbitrary but fixed instances of G_{PO} induced by these queries for a particular run.

By the induction hypothesis, we know that $\hat{G}_1$ and $\hat{G}_2$ contain solutions nodes for $\langle\rangle$ and $\langle o_2[\sigma_2], .., o_k[\sigma_k]\rangle$ respectively. Let $\hat{n}_1$ and $\hat{n}_2$ denote such nodes respectively, and let $\langle \hat{a}^1_0, \hat{a}^1_1, .., \hat{a}^{1*}\rangle|_{BI(\hat{n}_1), CL(\hat{n}_1)}$ and $\langle \hat{a}^2_0, \hat{a}^2_1, .., \hat{a}^{2*}\rangle|_{BI(\hat{n}_2), CL(\hat{n}_2)}$ be the serialized partial compositions created by TRANS_{PO} before returning the compositions.

Using the solutions of these subqueries, I now show that there exists a path $p = (n^0_{\text{PO}}, .., n)$ in G such that $\star_{\text{PO}}(n) = true$ and $c \in \text{TRANS}_{\text{PO}}(p)$. To this end, we iteratively consider a *current* node n_c, which is initially n^0. In addition, we consider a mapping γ of actions in the partial composition of n_c and actions in $\{\hat{a}^1_0, \hat{a}^1_1, .., \hat{a}^{1*}, \hat{a}^2_0, \hat{a}^2_1, .., \hat{a}^{2*}\}$. Initially, we set $\gamma(a_0) = \hat{a}^1_0$ and $\gamma(a^*) = \hat{a}^{2*}$. Considering the following routine, it is easy to see that γ is defined at each time for all actions in the partial composition of n_c. In addition, I will use δ as in the previous Lemma to denote the flaw correspondence.

If the agenda of n_c is not empty, then we know an action $\hat{a}'$ that can be used to compute a resolver for flaw selected in n_c. Since $AG(n_c) \neq \emptyset$, such a flaw $\langle L, a\rangle$ exists. We know that $\gamma(a)$, and, hence, $\langle L', \gamma(a)\rangle = \delta(L, a)$ are defined. Also, we know that $L' \in Pre_{\gamma(a)}$ and, since the agenda of $\hat{n}_1$ and $\hat{n}_2$ is empty, there is a causal link $\langle \hat{a}', L', \gamma(a)\rangle \in CL(\hat{n}_1) \cup CL(\hat{n}_2)$ for some action $\hat{a}'$; here, $\hat{a}'$ is either an action of the first or the second partial composition. $\hat{a}'$ itself is neither in nor added to the plan of n_c, but we can derive an action a' based on it.

[3] One could also define $(Z_{\hat{q}_2})_i = (Z_q)_i \ominus_i (Z_{o_1})_i$, but this not necessary since we only need to make sure that the desired sub-solution can be found.

Considering the link $\langle \hat{a}', L', \gamma(a) \rangle$, I now show that there is a successor $n_c{}'$ of n_c that has a correspondence to this link. In the following, ψ_2 is the parameter mapping obtained from GETPARAMETERMAP in TRANS_{PO} when invoked on the path to $\hat{n}_2$. Three cases are possible:

1. $\hat{a}' = \hat{a}_0^2$ *and* $L'[\psi_2] \in \mathit{Pre}_q$. Then L can be unified with a literal of Post_{a_0}, and, hence, there is an edge from n_c to a child node $n_c{}'$ using a_0 as resolver.

2. $\hat{a}' = \hat{a}_0^2$ *and* $L'[\psi_2] \in \mathit{Post}_{o_1}[\sigma]$. If we run into this case for the first time, then we know that there is an edge from n_c that uses a new action corresponding to a new instance of the operation belonging to o_1. Let $\bar{a}$ be this action, and define $\gamma(\bar{a}) = \hat{a}^{1*}$. If we run into this case again, we can *reuse* the action $\bar{a}$ as a resolver, i.e. there is an edge from n_c that uses $\bar{a}$ as a resolver.

3. *Any other case.* Either the plan of n_c does not yet contain any action a' such that $\gamma(a') = \hat{a}'$. Then $\hat{a}'$ is either an operation or Horn implication such that a literal in $\mathit{Post}_{\hat{a}'}$ unifies with L, so there is an edge from n_c that inserts a new action a' corresponding to that operation or Horn implication used as a resolver for L; we set $\gamma(a') = \hat{a}'$. Or there is already an action a' such that $\gamma(a') = \hat{a}'$, then there is an edge that uses a' as a resolver without adding a new action.

Using this technique, we must eventually obtain a node n whose agenda is empty. In each step, we create a successor whose computation is driven by a causal link of $CL(\hat{n}_1)$ or $CL(\hat{n}_2)$. But each link is used at most once (otherwise we would resolve the same flaw twice), and since both sets are finite, there must be a last node n. Since the above routine only stops if the agenda of n is empty, we know that this must be the case for some n_c.

It can be easily seen that $c \in \text{TRANS}_{\text{PO}}(p)$. Suppose that, following the advices from $\hat{G}_1$ and $\hat{G}_2$, we obtain the action set $AS(n) = \{a_0, \gamma^{-1}(\hat{a}_1^1), .., \gamma^{-1}(\hat{a}_l^1), \bar{a}, \gamma^{-1}(\hat{a}_1^2), .., a^*\}$. The ordering $\langle \gamma^{-1}(\hat{a}_1^2), .., a^* \rangle$ induced the composition $\langle o_2[\sigma_2], .., o_k[\sigma_k] \rangle$ and is compatible with the orders we inserted (since we defined exactly the same orders on that subset of actions). The action $\bar{a}$ responsible for the operation invocation $o_1[\sigma_1]$ is ordered before at least one of the actions inducing these operation invocations; in particular, it is not order *after* any of these. Hence, every topological ordering that induces the composition $\langle o_1[\sigma_1], .., o_k[\sigma_k] \rangle$ is considered by $\text{TRANS}_{\text{PO}}(p)$; in particular, c is returned.

The previous argument implicitly assumes that $\bar{a}$ actually *is* in the sequence. But this always holds due to the minimality of c. The action $\bar{a}$ is contained if and only if the second of the above three cases occurs at least once. By minimality of c, this must be the case, because if no flaw was resolvable using $\bar{a}$, and hence o_1, that operation would not be necessary, and $\langle o_2[\sigma_2], .., o_k[\sigma_k] \rangle$ would also be a solution to q.

The positive test of the goal function $\star_{\text{PO}}(n) = \mathit{true}$ follows trivially. By definition, $\odot(n) = \odot(c(n))$ where $c(n)$ is the partial composition associated with n. Since c is a serialization of $c(n)$, its non-functional properties are equivalent. Hence, $\odot(n) = \odot(c(n)) = \odot(c) \leq Z_q$. Since the agenda of n is empty, $\star_{\text{PO}}(n)$ evaluates to *true*. □

BIBLIOGRAPHY

[1] University of Paderborn. CRC 901 - On-The-Fly Computing. http://sfb901.uni-paderborn.de/. Accessed: 2016-08-15.

[2] L. Ai and M. Tang. Qos-based web service composition accommodating inter-service dependencies using minimal-conflict hill-climbing repair genetic algorithm. In *Proceedings of the IEEE Fourth International Conference on eScience*, pages 119–126, 2008.

[3] C. Amato, B. Bonet, and S. Zilberstein. Finite-state controllers based on mealy machines for centralized and decentralized POMDPs. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1052–1058. AAAI Press, 2010.

[4] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[5] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, July 1982.

[6] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, 2007.

[7] L. Aversano, G. Canfora, and A. Ciampi. An algorithm for web service discovery through their composition. In *Proceedings of the IEEE International Conference on Web Services*, pages 332–339, 2004.

[8] O. Aydın, N. K. Cicekli, and I. Cicekli. Automated web services composition with the event calculus. In *Engineering Societies in the Agents World VIII*, pages 142–157. Springer, 2008.

[9] P. Bartalos and M. Bieliková. Semantic web service composition framework based on parallel processing. In *Proceedings of the Conference on Commerce and Enterprise Computing*, pages 495–498. IEEE, 2009.

[10] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proceedings of the International Conference on Service-Oriented Computing*, pages 43–58. Springer, 2003.

[11] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz. Heuristics for qos-aware web service composition. In *Proceedings of the International Conference on Web Services*, pages 72–82. IEEE, 2006.

[12] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3):316–361, 2010.

[13] M. B. Blake and D. J. Cummings. Workflow composition of service level agreements. In *Proceedings of the International Conference on Services Computing*, pages 138–145. IEEE, 2007.

[14] R. Bodik and B. Jobstmann. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer*, 15(5):397–411, 2013.

[15] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 52–61, 2000.

[16] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and logic*. Cambridge University Press, 2002.

[17] J. R. Buchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[18] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. Qos-aware replanning of composite web services. In *Proceedings of the International Conference on Web Services*, pages 121–129. IEEE, 2005.

[19] J. Cardoso and A. Sheth. Semantic e-workflow composition. *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.

[20] W. A. Carnielli. Systematization of finite many-valued logics through the method of tableaux. *The Journal of Symbolic Logic*, 52(02):473–493, 1987.

[21] N. Channa, S. Li, A. W. Shaikh, and X. Fu. Constraint satisfaction in dynamic web service composition. In *Proceedings of the Sixteenth International Workshop on Database and Expert Systems Applications*, pages 658–664. IEEE, 2005.

[22] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[23] A. Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962.

[24] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *European Conference on Planning*, pages 130–142. Springer, 1997.

[25] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1):35–84, 2003.

[26] A. Cimatti, M. Roveri, and P. Traverso. Automatic obdd-based generation of universal plans in non-deterministic domains. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference*, pages 875–881, 1998.

[27] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *AIPS*, volume 98, pages 36–43, 1998.

[28] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[30] D. B. Claro, P. Albers, and J.-K. Hao. Selecting web services for optimal composition. In *ICWS international workshop on semantic and dynamic web processes*, 2005.

[31] I. Constantinescu, W. Binder, and B. Faltings. Service composition with directories. In *Software Composition*, pages 163–177. Springer, 2006.

[32] I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *Proceedings of the International Conference on Web Services*, pages 506–513. IEEE, 2004.

[33] I. Constantinescu, B. Faltings, and W. Binder. Type based service composition. In *Proceedings of the 13th International World Wide Web conference on Alternate track papers & posters*, pages 268–269. ACM, 2004.

[34] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial*, pages 447–454, 2002.

[35] M. Daniele, P. Traverso, and M. Y. Vardi. Strong cyclic planning revisited. In *Recent Advances in AI Planning, 5th European Conference on Planning*, pages 35–48. Springer, 1999.

[36] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[37] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[38] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[39] J. El Haddad, M. Manouvrier, and M. Rukoz. TQoS: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing*, 3(1):73–85, 2010.

[40] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning, technical report cs-tr-2797, umiacs-tr-91-154, src-tr-91-96, 1991.

[41] H. W. Fowler and J. Butterfield. *Fowler's Dictionary of Modern English Usage*. Oxford University Press, 2015.

[42] M. Fox and D. Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[43] M. Garey and D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. San Francisco: Freeman, 1979.

[44] A. Gefen and R. I. Brafman. Pruning methods for optimal delete-free planning. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

[45] P. Godefroid, J. Van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[46] J. C. González-Moreno, M. T. Hortala-Gonzalez, F. J. Lopez-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40(1):47–87, 1999.

[47] C. Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–240, 1969.

[48] D. L. W. Hall, A. Cohen, D. Burkett, and D. Klein. Faster optimal planning with partial-order pruning. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

[49] E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1):35–62, 2001.

[50] J. Harney and P. Doshi. Selective querying for adapting web service compositions using the value of changed information. *IEEE Transactions on Services Computing*, 1(3):169–185, 2008.

[51] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[52] S. V. Hashemian and F. Mavaddat. A graph-based approach to web services composition. In *Proceedings of the Symposium on Applications and the Internet*, pages 183–189. IEEE, 2005.

[53] A. B. Hassine, S. Matsubara, and T. Ishida. A constraint-based approach to horizontal web service composition. In *The Semantic Web - ISWC 2006*, pages 130–143. Springer, 2006.

[54] E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In *AAAI*, volume 5, pages 372–377, 2005.

[55] M. Helmert. A planning heuristic based on causal graph analysis. In *ICAPS*, volume 4, pages 161–170, 2004.

[56] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[57] M. Helmert, P. Haslum, J. Hoffmann, and R. Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM (JACM)*, 61(3):16, 2014.

[58] J. Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[59] J. Hoffmann, P. Bertoli, M. Helmert, and M. Pistore. Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection. *Journal of Artificial Intelligence Research*, pages 49–117, 2009.

[60] J. Hoffmann, P. Bertoli, and M. Pistore. Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, volume 22, page 1013, 2007.

[61] J. Hoffmann and R. I. Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6):507–541, 2006.

[62] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[63] J. Hoffmann, I. Weber, and F. M. Kraft. SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management. *Journal of Artificial Intelligence Research*, pages 587–632, 2012.

[64] J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining scalability and expressivity in the automatic composition of semantic web services. In *Proceedings of the Eighth International Conference on Web Engineering*, pages 98–107. IEEE, 2008.

[65] S. Jiménez and A. Jonsson. Computing plans with control flow and procedures using a classical planner. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, pages 62–69, 2015.

[66] A. Jungmann and F. Mohr. An approach towards adaptive service composition in markets of composed services. *Journal of Internet Services and Applications*, 6(1):1–18, 2015.

[67] R. L. Keeney and H. Raiffa. *Decisions with multiple objectives: preferences and value trade-offs*. Cambridge University Press, 1993.

[68] M. Klusch. Semantic web service coordination. In *CASCOM: Intelligent Service Coordination in the Semantic Web*, pages 59–104. Springer, 2008.

[69] M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with OWLS-XPlan. In *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pages 55–62, 2005.

[70] S. Kona, A. Bansal, M. B. Blake, and G. Gupta. Generalized semantics-based service composition. In *Proceedings of the International Conference on Web Services*, pages 219–227. IEEE, 2008.

[71] J. R. Koza and J. P. Rice. Automatic programming of robots using genetic programming. In *Proceedings of the 10th National Conference on Artificial Intelligence*, volume 92, pages 194–207, 1992.

[72] U. Kuter, D. Nau, E. Reisner, and R. P. Goldman. Using classical planners to solve nondeterministic planning problems. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, 2008.

[73] S. Lämmermann. Runtime service composition via logic-based program synthesis, 2002.

[74] F. Lécué. Optimizing qos-aware semantic web service composition. In *The Semantic Web - ISWC*, pages 375–391. Springer, 2009.

[75] F. Lécué and A. Delteil. Making the difference in semantic web service composition. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1383–1388, 2007.

[76] H. J. Levesque. Planning with loops. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 509–515, 2005.

[77] N. Lipovetzky, C. J. Muise, and H. Geffner. Traps, invariants, and dead-ends. In *ICAPS*, pages 211–215, 2016.

[78] D. W. Loveland. Mechanical theorem-proving by model elimination. In *Automation of Reasoning*, pages 117–134. Springer, 1968.

[79] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. *International Journal on Software Tools for Technology Transfer*, 15(5):603–618, 2013.

[80] L. Mandow and J. L. P. De La Cruz. Multiobjective A* search with consistent heuristics. *Journal of the ACM*, 57(5):27, 2010.

[81] Z. Manna and R. Waldinger. Synthesis: Dreams - programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979.

[82] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[83] D. McAllester and D. Rosenblatt. Systematic nonlinear planning. Technical report, Massachusetts Institute of Technology, 1991.

[84] D. V. McDermott. Estimated-regression planning for interactions with web services. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, volume 2, pages 204–211, 2002.

[85] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning*, pages 482–493, 2002.

[86] K. L. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[87] F. Mohr. Issues of automated software composition in ai planning. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 895–898. ACM, 2014.

[88] F. Mohr. *Automated Software and Service Composition*. Springer, 2016.

[89] F. Mohr. Non-sequential automated service composition via templates, 2017 (submitted).

[90] F. Mohr, A. Jungmann, and H. Kleine Büning. Automated online service composition. In *Proceedings of the IEEE International Conference on Services Computing*, pages 57–64, 2015.

[91] F. Mohr and H. Kleine Büning. Semi-automated software composition through generated components. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, page 676. ACM, 2013.

[92] F. Mohr and S. Walther. Template-based generation of semantic services. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 188–203. Springer, 2015.

[93] N. J. Nilsson. *Principles of Artificial Intelligence*. Symbolic Computation. Springer Berlin, 1982.

[94] J. Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984.

[95] J. Peer. A PDDL based tool for automatic web service composition. In *Principles and practice of semantic web reasoning*, pages 149–163. Springer, 2004.

[96] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, 1992.

[97] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Artificial Intelligence: Methodology, Systems, and Applications*, pages 106–115. Springer, 2004.

[98] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1252–1259, 2005.

[99] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, volume 1, pages 479–486, 2001.

[100] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, pages 2–11, 2005.

[101] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *Proceedings of the International Conference on Web Services*, pages 293–301. IEEE, 2005.

[102] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[103] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010.

[104] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[105] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[106] M. Sheshagiri, M. DesJardins, and T. Finin. A planner for composing services described in DAML-S. *Web Services and Agent-based Engineering-AAMAS*, 3:1–5, 2003.

[107] A. Sirbu and J. Hoffmann. Towards scalable web service composition with partial matches. In *Proceedings of the International Conference on Web Services*, pages 29–36. IEEE, 2008.

[108] B. Srivastava, T. A. Nguyen, A. Gerevini, S. Kambhampati, M. B. Do, and I. Serina. Domain independent approaches for finding diverse plans. In *IJCAI*, pages 2016–2022, 2007.

[109] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. *ACM Sigplan Notices*, 45(1):313–326, 2010.

[110] S. Srivastava, N. Immerman, and S. Zilberstein. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2):615–647, 2011.

[111] S. Srivastava, N. Immerman, S. Zilberstein, and T. Zhang. Directed search for generalized plans using classical planners. In *ICAPS*, 2011.

[112] B. S. Stewart and C. C. White III. Multiobjective A. *Journal of the ACM (JACM)*, 38(4):775–814, 1991.

[113] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Automated Deduction—CADE-12*, pages 341–355. Springer, 1994.

[114] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, 2002.

[115] R. Thiagarajan and M. Stumptner. Service composition with consistency-based matchmaking: a csp-based approach. In *Proceedings of the European Conference on Web Services*, pages 23–32. IEEE, 2007.

[116] A. Torralba and J. Hoffmann. Simulation-based admissible dominance pruning. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 1689–1695, 2015.

[117] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *The Semantic Web*, pages 380–394. Springer, 2004.

[118] A. Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, pages 491–515. Springer, 1989.

[119] M. Vuković, E. Kotsovinos, and P. Robinson. An architecture for rapid, on-demand service composition. *Service Oriented Computing and Applications*, 1(4):197–212, 2007.

[120] R. Waldinger. Web agents cooperating deductively. In *Formal Approaches to Agent-Based Systems*, pages 250–262. Springer, 2001.

[121] S. Walther and H. Wehrheim. Verified service compositions by template-based construction. In *Proceedings of the 11th Symposium on Formal Aspects of Component Software*, pages 31–48, 2014.

[122] I. M. Weber. *Semantic Methods for Execution-level Business Process Modeling: Modeling Support Through Process Verification and Service Composition.* Springer, 2009.

[123] M. Wehrle and M. Helmert. About partial order reduction in planning and computer aided verification. In *ICAPS*, 2012.

[124] T. Weise, S. Bleul, M. Kirchhoff, and K. Geihs. Semantic web service composition for service-oriented architectures. In *Proceedings of the fifth IEEE Conference on Enterprise Computing*, pages 355–358, 2008.

[125] M. Winslett. Reasoning about action using a possible models approach. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 89–93, 1988.

[126] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. *Automating DAML-S web services composition using SHOP2*. Springer, 2003.

[127] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *Proceedings of the 12th international conference on World Wide Web*, pages 411–421. ACM, 2003.

[128] G. Zou, Y. Gan, Y. Chen, and B. Zhang. Dynamic composition of web services using efficient planners in large-scale service repository. *Knowledge-Based Systems*, 62:98–112, 2014.