ISBN 978-90-365-3041-5

Graph-based Specification and Verification for Aspect-Oriented Languages

Tom Staijen

Graph-based Specification and Verification for Aspect-Oriented Languages

Tom Staijen

U i t n o d i g i n g

Op donderdag 3 juni 2010
om 15:00 uur verdedig ik
mijn proefschrift, getiteld

Graph-based Specification and Verification for Aspect-Oriented Languages

in Collegezaal 4 in
gebouw 'Waaier' van de
Universiteit Twente.

Voorafgaand aan de verdediging
zal ik om 14:45 een korte
toelichting geven op mijn
onderzoek.

U bent van harte welkom bij de
verdediging en de receptie die
aansluitend wordt gehouden.

Vanaf 20:30 borrelen bij
café Central Park in Enschede
(RSVP voor 28 mei)

Tom Staijen
t.staijen@gmail.com
+31 6 10027649

Paranimfen:
Jackelien Barelds en Rein Staijen

Graph-Based Specification and Verification for Aspect-Oriented Languages

Tom Staijen

CTIT

CTIT Ph.D. thesis series no. 10-171.
Centre for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede, The Netherlands.

IPA Dissertation Series 2010-04
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-3041-5
ISSN 1381-3617 (CTIT Ph.D. thesis series no. 10-171)
http://dx.doi.org/10.3990/1.9789036530415

Cover design by Tom Staijen
Printed by Ipskamp Drukkers, Enschede, The Netherlands

Graph-Based Specification and Verification for Aspect-Oriented Languages

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 3 juni 2010 om 15:00 uur

door

Tom Staijen

geboren op 27 januari 1978
te Stad Delden

Dit proefschrift is goedgekeurd door

Prof. Dr. Ir. M. Akşit (promotor)
Dr. Ir. A. Rensink (assistent-promotor)

"Common sense, do what it will, cannot avoid being surprised occasionally. The object of science is to spare it this emotion and create mental habits which shall be in such close accord with the habits of the world as to secure that nothing shall be unexpected."

– Bertrand Russell

Acknowledgements

The first person I would like to thank is my daily supervisor and assistant-promoter Arend Rensink. During the past five years, Arend has taught me a great deal about formal methods in general and about graph transformation, about writing proofs and to not fear Greek symbols. I also want to thank him for listening and understanding when things did not go too well, and for triggering me when things did not go very fast. However, the most precious thing I need to thank Arend for, is the enthusiasm and joy I have in specifying systems formally, particularly for the purpose of solving puzzles.

I have carried out my Ph.D. studies at the software engineering group lead my Mehmet Akşit. The sprout of his mind "Composition Filters" has found its way into this thesis. His endless enthusiasm has been a continuous motivation, his presence a motive for generalisation, always to look for a "bigger picture". His feedback, advice and guidance can be found throughout this thesis.

I like to thank the members of my Ph.D. committee: Shmuel Katz, Dark Janssens, Jaco van de Pol and Pieter Hartel. Their useful comments enabled me to dramatically improve this thesis.

I would like to thank all members of the software engineering group and the formal methods and tools group. They have provided me with useful feedback during our regular seminars and provided me with a pleasant working environment and of course a number of unforgettable social events. In particular, I want to thank ex-fellow-Ph.D. student Pascal Dürr with whom I went on some awesome hiking trips, Ivan Zapreev, we "recognised" each other in Marktoberdorf and have had the best times since, and Wilke Havinga, with whom I worked in the Network of Excellence on Aspect-Oriented Software Development.

I also want to thank the many people in the European Network of Excellence on Aspect-Oriented Software Development for providing a forum of enthusiasts

about AOSD. In particular, I would like to thank fellow participants of the Formal Methods Lab: Shmuel Katz, Emilia Katz, Mario Südholt, and Remi Douance.

I would like to thank Ellen Roberts-Tieke, Joke Lammerink and especially Jeanette Rebel-de Boer for their invaluable administrative support.

Finally, I want to thank my family:

My dear girlfriend and paranymph Jackelien, whom I created a lovely home with and who has supported me through most of the writing phase. The foresight of living with you without the burden of writing has motivated me through the final year.

My brother Rein, with whom I enjoyed so many relaxing evenings and weekends, setting the PhD work aside for a while. Charge!

And last, but most definitely not the least, my father and paranymph Rein and my mother Thea, who have encouraged and supported me in practically any way though the good and bad times of the past ten years of study. Dank jullie wel.

Abstract

Aspect-oriented software development aims at improving separation of concerns at all levels in the software development life-cycle, from architecture to code implementation. In particular, aspect-oriented programming addresses separation of concerns at the implementation level, by allowing the modular expression of crosscutting concerns.

In this thesis we strive to develop verification methods specifically for aspect-oriented languages. For this purpose, we model the behaviour of these languages through an operational semantics. We use graph transformations to specify these semantics. Graph transformation has mathematical foundation, and provides an intuitive way to describe component-based systems, such as software systems. In addition, graph transformations have an executable nature, and can be used to generate the execution state space of programs. We use these state spaces for the verification of programs with aspects.

We start by defining an improvement of specification by rule-based systems. Pure rule-based systems typically consist of a single, unstructured set of rules. The behaviour of such systems is that all rules are applicable in every state. Rules can then only be forced into a certain order of application by adding special elements to the states, which are tested for within the rules. In other words, control over rule applicability is not explicit but has to be encoded in the state, which reduces understandability and maintainability of the rule-based system as a whole. We propose so-called *control automata*, which can be added on top of pure rule-based systems. The resulting behaviour is defined as the product of the original state space and the control automaton. Our control automata include so-called failure transitions, representing the observation of the non-applicability of one or more rules. The result is a reactive semantics for control expressions, which is distinct from the usual input-output semantics. Control automata may introduce artificial non-determinism into the behaviour, which is an undesirable effect. We introduce *guarded control automata* to get rid of this effect, and we define a semantics-preserving transformation from ordinary control automata to guarded ones.

In the next part of the thesis, we specify the run-time semantics of a number of aspect-oriented languages, namely of (1) Composition Filters, (2) Featherweight Java with assignments with an aspectual extension, and (3) a subset of multi-threaded Java extended with a subset of AspectJ. We illustrate how such a graph-based semantics can aid in understanding the run-time behaviour of a language. Moreover, we show that such a semantics can be used to simulate a (partial) program and we expose the steps involved in the execution of such a program. We illustrate that this executable nature benefits the rigour of the specification method; mistakes are easily detected by simply testing the simulation. Finally, we show that the resulting labelled transition systems can be used for existing verification methods.

Then, we propose two novel approaches that address complications caused by the use of aspect-oriented programming.

The first approach addresses a problem that can occur in many aspect-oriented languages. Aspects that in isolation behave correctly may interact when being combined. When interaction changes an aspect's behaviour or disables an aspect, we call this interference. One particular type of interference occurs when aspects are applied to shared join points, since then the ordering of the aspects can also influence the behaviour of the composition. We present an approach to detect aspect interference at shared join points. The approach is based on simulation of all orderings of the advices that are scheduled for execution at a shared joinpoint. A confluence analysis is performed on the resulting state space to detect whether the execution order has affected the resulting states.

The second approach addresses the problem of verification of dynamic properties of systems. These properties can only be verified by simulating the execution of the system: an execution semantics is required. More specifically, we deal with properties that require tracking of individual objects over time. We stress the need for automatic verification of properties (or constraints) for aspect-oriented programming because of the obliviousness properties of such languages: a developer cannot tell from looking at the base code that aspects are executed. When software evolves, existing functionality may break unintentionally. We propose to augment an existing graph-based execution semantics with special verification rules. These rules can, when needed, add information to the graphs for tracking of objects. The properties are specified as events (or interactions) related to roles. Once these roles are identified in the syntax, the program can be verified regardless of implementation details. We show that the approach can be applied to both object-oriented and aspect-oriented implementations.

Contents

Chapter 1

Introduction

Aspect-oriented programming (AOP) languages aim to improve the modularisation of concerns in the specification of software systems. In this thesis we strive to develop verification methods specifically for such languages. For this purpose, we model the behaviour of these languages using an operational semantics. We use graph transformations to specify these semantics. Graph transformation has mathematical foundation, and provides an intuitive way to describe component-based systems, such as software systems.

This introductory chapter is organised as follows: in Section 1.1 we elaborate on aspect-oriented programming, and the phenomenon of crosscutting to which it offers a solution. In Section 1.2, we motivate the work in this thesis: we elaborate on the disadvantages of aspect-oriented programming and what kind of problems are caused specifically by using AOP. In Section 1.3 we give some background on verification, and the choice of using graph transformation for this purpose. Finally, in Section 1.4 we provide an outline of the rest of this thesis.

1.1 Aspect-Oriented Programming

Aspect-oriented software development (AOSD) aims at improving separation of concerns (SoC) at all levels in the software development life-cycle, from requirements engineering and architecture to code implementation. In particular, aspect-oriented programming addresses SoC at the implementation level, by allowing the modular expression of crosscutting concerns. Such a modular representation of a crosscutting concern is called an *aspect*.

1.1.1 Separation of Concerns

Separation of concerns is one of the key goals in the field of software and system engineering. The term was probably coined by Dijkstra in 1974 [Dij74] — later published in [Dij82] — as "the only effective way of ordering one's thoughts".

In computer science, separation of concerns is the process of separating a computer program into distinct features that overlap in functionality as little as possible. All programming paradigms aim at improving SoC. For example, object-oriented programming languages can separate concerns into classes. Possible motivations of SoC are to reduce complexity, and to enhance adaptability and evolvability.

A concern is defined as *"a focus of interest pertaining to the development of a system under design, its operation or any other matters that are important to one or more stakeholders"* [vdBCC05]. In this context, separation of concerns is considered *"the study and realisation of each concern in isolation for the sake of its own consistency"* [vdBCC05].

Concerns are typically identified in the early phases of the development life-cycle from the requirements of all stakeholders. Later on, a decomposition into modules is chosen for the implementation. If the identified concerns do not match the decomposition structure, such a mismatch between the design (concerns) and the implementation (modules) may cause one or more of the following issues:

1. **Tangling** occurs when concerns are mixed together in one module [vdBCC05]. In a modular implementation, each module (or group of modules) exactly describes a single concern. Tangling negatively influences the *reusability* of a module. A component mixing concerns is not properly reusable because the composition of concerns in this tangled code is likely to be application specific.

2. **Scattering** is the occurrence of elements that belong to one concern in modules encapsulating other concerns [vdBCC05]. In other words, code corresponding to one concern is spread out over several modules. This negatively influences the *maintainability* of a concern; making changes and finding errors is much easier when all related code is in one place. *Code Replication* is the occurrence of (nearly) identical code in multiple places, often resulting from a pattern of copying and editing existing source fragments.

3. **Crosscutting** is the combination of tangling and scattering. A concern that cannot be modularly represented within the chosen decomposition structure

due to tangling and scattering is called a *crosscutting concern.*

Aspect-oriented programming aim to address the issues identified above. The typical demonstrative example of AOP is the *Logging* concern. Typically, calls to a logging method are added throughout the code to write certain information to a file or screen. With AOP, this method and a specification of *what to log* are specified in a single module, called an *aspect.* In the remainder of this section, we explain the basic ideas of AOP.

1.1.2 AOP = Quantification & Obliviousness

Aspect-oriented programming languages enable the modular expression of crosscutting concerns by means of supporting *quantification* over a program. They allow programming by making quantified assertions over programs written by programmers *oblivious* to such assertions [FF05].

Quantification

Quantification can be understood as follows: "in programs P, whenever a condition C arises, perform action A" over conventionally coded programs P. Aspects may perform actions at one or more points during the execution of a program. Such *"well-defined places during the execution of a program where additional behaviour can be attached"* [vdBCC05] are called *joinpoints.* Expressing quantification is done by so-called *pointcuts.* A pointcut describes a set of joinpoints. The *kind* of points (i.e. event types) that can be selected in the program depends on the *joinpoint model* of the aspect language. Typically, pointcuts can express predicates over the syntax(-tree) of the program; most aspect languages also allow predicates over the run-time state of a program. Besides a set of joinpoints, a pointcut also describes the scope of extension at the joinpoints. The additional behaviour that is executed at a joinpoint is called an *advice.* Advice is traditionally specified to be executed *before*, *after* or *around* the selected joinpoint. A common feature of the advice language — commonly used in around advice — is a so-called *proceed* statement; executing this statement continues the execution of the intercepted joinpoint (i.e. the event that triggered the execution of the advice).

Obliviousness

Another property of AOP languages we find important is *obliviousness.* *"Obliviousness states that you can't tell that the aspect code will execute by examining the body of the base code"* [FF05]. Better aspect languages minimise the degree to which developers have to change their behaviour (i.e. implementation style) to realise the use of aspects. In other words: when there are no constraints on the way the base system is coded, aspects can always be added later.

1.1.3 Symmetric vs. Asymmetric AOP

The distinction between *base system* and *aspects* comes from the traditional design of aspect languages as an extension to "normal" object-oriented languages, the so-called base language. We refer to the base system as the part of the system under advice of an aspect, whereas the "aspects" are the part of the system with an advising role. Typically, the base system is the part that is implemented in the base language, whereas the aspects are the parts implemented using the aspect language (i.e. aspect modules containing pointcut and advice declarations).

This distinction is not always straightforward; certain languages allow pointcuts to select joinpoints in the execution of an advice. In that case, the aspect code under advice has the role of the base system.

Languages that support this distinction fall in the category of *asymmetric* aspect languages. There are also *symmetric* aspect languages, that make no distinction between base and aspect, for example by declaring everything as an *aspect*. However, these languages do not fall in the scope of this thesis. In this thesis, we assume that the base system is the part that is implemented in the base language. Symmetric AOP tries to come close to the "grand vision" of AOP, but asymmetric has simply turned out to be more pragmatic.

Also, the properties of quantification and obliviousness are not limited to object-oriented programming languages. Aspect-oriented languages can just as well be designed as an extension to procedural languages. In this thesis, we only consider aspect languages that are extensions to object-oriented languages.

Exporting Behaviour rather then Importing

Another way to look at the difference between object-oriented and aspect-oriented languages is the way concerns are composed. In object-oriented programming languages, functionality of other concerns is imported (or invoked) by using method calls. In AOP however, functionality of a concern is exported to other concerns (e.g. specific locations in the program).

Interface AOP vs. fine-grained AOP

Another distinction that can be made between aspect languages is by the level at which they extend the base program. Some languages have a very fine-grained joinpoint model, which are typically suitable for crosscutting concerns like *logging* and *debugging*. Other languages try to employ AOP on the interface of objects, i.e. as a composition mechanism. They obey the encapsulation of objects. In this thesis we will see both types of languages.

Weaving

Where AOP languages allow crosscutting concerns to be expressed as isolated modules through special language extensions, they are typically executed as traditional programs, i.e. by using no other instructions than those available in the extended base language. The process of composing the base system and the aspects is called *weaving*. Weaving is defined as *"the process of composing core functionality modules with aspects, thereby yielding a working system"* [vdBCC05]. Weaving is typically performed at compile-time — e.g. by in-lining aspect code or invocations to this code — but other approaches exist that weave at run-time. We use *weave-time* to indicate the moment of weaving, regardless of it being at compile-time or run-time.

1.2 Motivation: The disadvantages of Obliviousness

Since AOP increases separation of concerns at the implementation level, the benefits of AOP are the same as the advantages of SoC. Qualities like *reusability*, *adaptability*, *maintainability* and *evolvability* are all properties that benefit directly from a successful mapping of concerns to implementation modules; they benefit from SoC and thus from AOP, which aim to achieve this at the implementation level.

SoC also lowers the complexity of concerns (by untangling tangled code and isolating scattered code), thereby increasing the *readability* and *understandability* of an isolated concern. This isolation, however, is the cause of the obliviousness property of AOP. When there is no sign of the aspect at the code-locations where it "applies", it becomes harder to understand the composed behaviour of the base program and the aspects as a whole.

Although some may claim that it should not be necessary to know of the existence of aspects, not knowing about them may quickly result in a programmer breaking the application. Many aspect languages are syntax-oriented; small changes in the base code may cause an aspect to disengage from a location in the base program, or match a previously unmatched location. This problem is called the “fragile pointcut problem”.

The actual problem here is that the complexity of the system as a whole is related to the complexity of the used composition mechanisms. AOP introduces new composition mechanisms, which may not be understood by all developers. This is directly related to the obliviousness property that characterises AOP languages. This property tends to be a disadvantage when reading code and thereby understanding the system as a whole.

Another problem of obliviousness is that, although it does not imply that developers are unaware of the aspect that advises a certain piece of base code, they may very well be. Obliviousness does not guarantee that changes to the base system will not cause an aspect to behave differently than intended. In large development teams, developers may be responsible for different software artifacts. Certain developers may be unaware of the aspects advising “their” base code, or the usage of aspects in the system in general. They may change the code and thereby manipulate the behaviour of the aspect. The resulting behaviour would be correct w.r.t. the specification of the aspect, which was made under the assumptions made for the old base system, but might not be the intended result.

The distinction between (any particular) base language and aspect language may result in developers becoming experts in one or the other. Base system developers that should be aware of the aspects advising their base code, may not even understand the execution semantics of these aspects, and the effects resulting from possible changes to the base system.

To be able to verify the correctness of a system with aspects requires the development of verification techniques for these languages. In this thesis, we focus on such verification techniques. In particular, we want to be able to detect unintended behaviour of aspects, which can be caused by aspect interaction.

1.2.1 Aspect Interference

We have already explained how the introduction of new composition mechanisms is a disadvantage for understanding the behaviour of the system as a whole. This becomes even worse when multiple aspects are applied to a system. Unexpected results can emerge: two or more aspects behaving correctly when applied in isolation, may interact in an undesired matter when applied together. This phenomenon is called *aspect interference*. Interference between aspects occurs when one aspect disables or changes the behaviour or applicability (i.e. the composition with the base system) of another aspect. There are different causes for aspect inference:

- At weave-time, the selected set of join points of one aspect can be changed by another aspect;

- At weave-time, aspects that change the static structure of a program (introductions) can cause ambiguous weaving - resulting in different programs - depending on the weaving order [HNBA07].
- At run-time, one aspect can modify fields or variables, affecting the behaviour of another aspect;
- At run-time, one aspect can change the control-flow of the system, causing a join point of another aspect to never be reached.

Here, the problems of obliviousness regarding the unawareness of the existence of aspects is complicated even further; not only should a developer realise that multiple aspects advice the same join point, but also that the composed behaviour may not be the desired behaviour. There are already some tools that verify whether overlapping pointcuts exist in a program. However, understanding the composed behaviour is still very difficult.

1.3 Verification and Graph Transformation

In this thesis, we introduce verification techniques for problems that are unique to aspect-oriented programming languages.

System verification aims at verifying whether a system satisfies a set of requirements. This can be done in many different ways. In this thesis, we focus on *formal verification*.

Formal verification techniques provide means to determine whether a system is correct with respect to a set of requirements, often called properties, based on a *model* of the system. One such technique is model checking, where the central idea is to verify all possible executions of a model of the system and check whether they satisfy the required properties.

Model checking [CGP99] is based on a modal extension of propositional logic. The properties are specified on the base level by propositions that are satisfied by a subset of the model being checked. The information in each of the states is abstracted to the subset of properties that is satisfied there. Only the information remains that is considered interesting for verification. On top of that we define a modal logic, in which the properties of the lower level are treated as propositions.

To obtain a model of the system, we specify the behaviour of the system using a formal specification technique. In other words, we specify a formal semantics of the language. Research on formal semantics started in the 1960s. Most resulting methods are based on either (1) *denotational semantics*, where the meanings of expressions are described as mathematical objects called *denotations*, or (2) *operational semantics*, which describes how a program is interpreted as a sequence of computational steps.

The traditional approach for defining an operational semantics for a programming languages is *Strucural Operation Semantics* (SOS), originally introduced by Gordon Plotkin in [Plo81]. SOS specifications take the form of a set of inference rules which define the valid transitions of a composite piece of syntax in terms of the transitions of its components. These formal definitions can be used for the understanding and analysis of the behaviour of programs. However, the mainly textual notation and the underlying mathematics does not easily let SOS be used by "regular" software developers.

In this thesis, we use graph transformation for the specification of operational semantics of programming languages. That is, we specify the run-time behaviour of programming languages using graph transformation rules; these can then be used to simulate the execution of a program when specified as a graph that corresponds to this program.

Graph transformation was first introduced in the early 1970s [EPS73] to generalise Chomsky's string grammars. Graph transformations provide an intuitive and formal way of specifying local graph changes — the creation and deletion of graph elements — in a rule-based manner.

We believe that graph transformation provides a natural way for defining the operational semantics of programming languages. Especially for object-oriented languages, graphs provide a natural way to represent program states. States based on objects can easily be represented as graphs, where the objects are nodes, and edges represent the relationships between objects. State changes are naturally represented as graph transformations. The visual notation of graph transformations may be more compelling and intuitive to programmers.

Another advantage is that, with the long history of graph transformation, tool support exists for specifying graph transformation rule systems and using it for analysis. One such tool is the GROOVE Tool Set [Ren04]. This tool is unique in its ability to perform an exhaustive exploration on the possible applications of graph transformation rules given a certain *start graph*. This is then represented as a *graph transition system*, a special *labelled transition system* (LTS) with graphs as states and rule names as labels.

When we use graph transformation rules to specify the run-time behaviour of a programming language, this "state space" represents all possible executions of a program specified in that language when it is represented as a graph, and can be used for the analysis and verification of that program. For example, the LTS lends itself directly for model checking (see [KR06]).

1.4 Outline of the Thesis

This thesis is structured as follows.

Chapter 2 serves two purposes. First, we introduce the theory of graph transformation, to give an intuition and to be able to understand the meaning of the graph transformation rules in this thesis. Then, we extend the exploration of rule applications with a notion of control. We define a control language over rules, that specifies the order in which rules can be applied. This way, control is specified as a separate artifact instead of having control information in the graphs and rules. This benefits the simplicity and readability of the rules.

Chapter 3 gives an introduction to the aspect-oriented programming language Composition Filters. The design of this language lends itself very well for verification. In this chapter we define a graph transformation-based operational semantics for the run-time behaviour of the language, and show how this can be used to simulate joinpoint execution. The semantics can serve as a reference for the language semantics.

Chapter 4 defines an operational semantics of *Featherweight Java* with assignments and a simple aspectual extension using method-call interception and around advice with proceed. We use this semantics to illustrate the advantages of using graph transformation for defining language semantics operationally. We show that a graph transformation based semantics can be consistent with a structural operation semantics of the same base language, and illustrate the advantages of our semantics compared to SOS: the directly executable nature of graph transformations and its visual nature aiding in the completeness, readability and understandability of the semantics. These advantages indirectly benefit the rigour of the approach: fewer mistakes are made, and any mistakes that are made are easily found during simulation. The work has been published in [SR09].

Chapter 5 discusses our approach to detect a special kind of aspect interference, namely when it happens at so-called *shared joinpoints*. Shared join points are joinpoints that are selected by pointcuts of more then one aspect. We show that, by simulating all possible orderings of advices scheduled for execution at the joinpoint, we can detect whether the advices interfere. When no interference occurs, the resulting states after execution of the advices are equal. Because we use graph transformation, we can use isomorphism as a criterion for the equality of states. The work described in this Chapter — combined with the work in Chapter 2 — has been published in [ARS09], where it received a *Best Paper Award*.

Chapter 6 discusses our approach to verify system properties on systems with aspects. Because aspects can be introduced during the design of a system, but also by refactoring the implementation, we propose a verification approach that is *oblivious to implementation details* (e.g. whether OOP or AOP is used). We

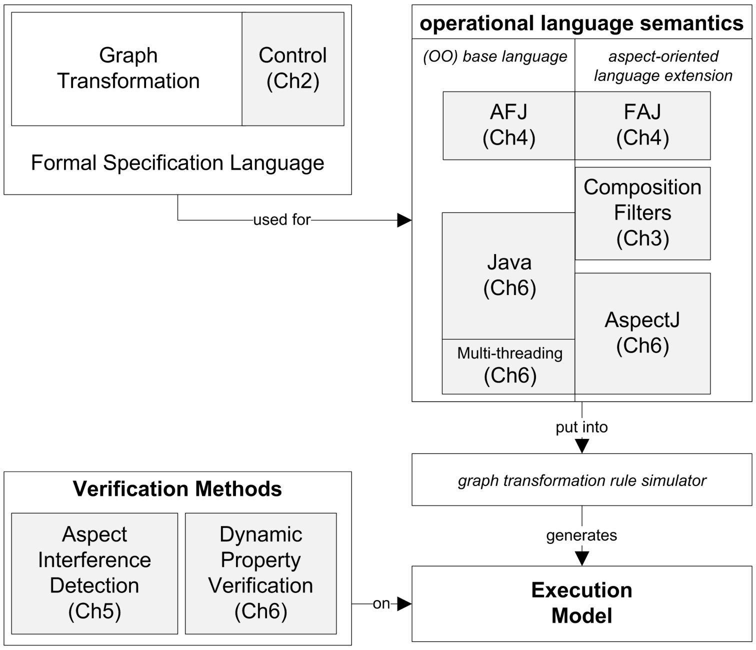

Graph Transformation
Control (Ch2)
Formal Specification Language
used for
operational language semantics
(OO) base language
aspect-oriented language extension
AFJ (Ch4)
FAJ (Ch4)
Composition Filters (Ch3)
Java (Ch6)
Multi-threading (Ch6)
AspectJ (Ch6)
put into
graph transformation rule simulator
generates
Execution Model
Verification Methods
Aspect Interference Detection (Ch5)
Dynamic Property Verification (Ch6)
on

Figure 1.1: Relations among the Contents of the Thesis

illustrate the approach with different implementations of the observer pattern in Java and AspectJ, and define an ad-hoc operational semantics to simulate multi-threaded usage of the given implementations. For the system properties that need to be verified, additional rules are used. These rules typically match the graph-based *run-time state representation* — objects with a certain role, relationships between objects, and interactions between objects. By analysing the occurrence of these rules in the generated graph transition system, we can verify if the system properties hold.

Chapter 7 concludes this thesis by shortly summarising the main results and contributions of our work. It discusses some of the limitations of our approach and looks ahead at some research topics that are useful extensions to the work.

1.4.1 Overview

The different subjects in this thesis and their correlation are shown in Figure 1.1. We extend the graph transformation formalism with a notion of Control. Then, we use graph transformations as a formal specification language to define the operational semantics of a number of different (object- and aspect-oriented) languages. Then, these semantics are used to generate an execution model of an input program. Finally, we define two different verification methods that can be applied to these execution models.

Chapter 2

Controlled Graph Transformation

2.1 Introduction

The use of rule-based specification and programming languages to model complex systems is a commonly accepted approach in the field of computer science. Examples are term rewrite systems [Der05], Petri nets [GR82] and graph rewrite systems [Roz97]. In rule-based systems, applying a rule in a state typically results in another state, and all rules are scheduled — allowed to be matched and applied — in every state at all times. The result of exploring all possible rule applications is the full state space.

Rules are stand-alone entities; they require to specify exactly the conditions required for the rule to be applied. Such conditions may become very complex and may use control information that is added to the state by other rules. Such information quickly complicates the comprehensibility of the entire rule system, and introduces hidden dependencies between rules.

We work in the setting of graph transformations. Graph transformation is a formal specification technique that supports rule based specification as well as an intuitive visual representation of states and rules. We use this approach for the specification of systems. We use graph grammars to generate a *reactive* view on the system. Graph grammars are used to generate the full state space of the system, which is used for verification by, for instance, model checking. We use the GROOVE [Ren04] toolkit for the generation of the full state space, as it is unique in its capability to do so.

This chapter serves two purposes. First, it provides an introduction to graph transformations. It gives formal definitions of graphs, rules, and transformations as well as the visual notation for rules used in GROOVE. The other part describes a contribution to extend GROOVE with control expressions. The results of this work are not restricted to GROOVE or graph transformation in general, however, but extend to arbitrary rule-based languages.

In the context of graph transformation, extending the rule system with a mechanism for controlling rule application is very popular. Such a mechanism, often called *control expressions*, can increase the ease of specification a great deal. One of the main advantages of using explicit control expressions is that it reduces the amount of control information required in the states and rules. Also, it provides an explicit view on dependencies between rules.

Typically, control expressions in the field of graph transformations are equipped with an *input-output* semantics, motivated by an interest in the transformational behaviour of a system. This implies that the meaning of a graph grammar is taken to be a binary relation between input graphs and output graphs. As mentioned earlier, we want to use graph transformations to generate a reactive view of the system. Therefore, we need control expressions with a reactive semantics.

We propose an approach to specify control expressions that satisfies the following main requirements:

- The control expressions must have a reactive semantics to preserve the reactive view of the system, because we are interested in verification based on this view.
- The control expression should not introduce spurious non-determinism. Any existing assumptions about the generated state space should be preserved. We explain later how introducing non-determinism can influence the semantics of a rule system (i.e., the trace language of the state space).
- The control expressions should be specified without reference to any particular rule system. This also enables the approach to be usable for different kinds of rule-based systems.

In this work, we introduce a control mechanism with a reactive semantics. A control language is defined for the purpose of specifying control expressions over rules in a simple and intuitive manner. These control expressions are translated to so-called *control automata*, which can be added on top of pure rule systems (rule systems where allowing the application of a rule is purely based on the rule itself). Control automata include transitions that specify the scheduling of a rule, and so-called *failure transitions* that describe the observation of a set of rules being non-applicable. The resulting behaviour is defined as the product of the original state space and the control automaton. Because the product with ordinary control automata can introduce spurious non-determinism, we also define a variation called *guarded control automata*, which do not have this undesirable effect.

In the next section we introduce the basic concepts underlying the graph transformation technique. In Section 2.3, we define control automata and show the result of combining such an automaton with system automata. In Section 2.4 we present a control language, and show how programs written in this language can be translated to control automata. In Section 2.5 we present guarded control automata, and we prove them to be equivalent to the normal automata. In Section 2.6 we illustrate the usage of the approach. Finally, in Section 2.7 we discuss our contribution, related work, and future work.

2.2 Graph Transformations

In this section, we introduce the basic concepts and underlying techniques of graph transformation. First, we introduce the concept of *graphs* and morphisms. Then we explain *graph transformation rules* and the transformation mechanism. Also, we explain the visual notation for rules, which comes from the GROOVE tool, and is used for the rules throughout this thesis.

2.2.1 Graphs and Morphisms

Graphs essentially consist of boxes — called nodes or vertices — and arrows — called edges. Graphs can be used to model practically any structure. Graphs are especially suitable for modelling structures that are dynamic in size, because they can have any number of nodes and vertices. In the next chapter, we show that graphs can very well be used to represent software systems. Graph transformations are changes to a graph. For rule-based systems in general, transformations are described by rules. For graphs, such rules are often referred to as *graph production rules.*

We now define the concept of a graph. We assume a known set of labels Label. A label is a name used to label the edges of our graphs.

Definition 2.1 (graph)**.** *A graph over* Label *is a tuple* (N, E)*, where* N *is a finite set of nodes and* $E \subseteq N \times \mathsf{Label} \times N$ *is a finite set of edges.*

An edge $e \in E$ is a triple (n_1, l, n_2) consisting of a source node $src(e) = n_1$, a label $lab(e) = l$, and a target node $tgt(e) = n_2$. An edge with an identical source and target edge (i.e. $src(e) = tgt(e)$) is called a *self-edge.* The set of all graphs is indicated with $\mathcal{G}$.

For the definition of a graph production rule and the transformation generated by such a rule, we require the concept of a *graph morphism*, which is essentially a mapping between the components of two graphs.

Definition 2.2 (graph morphism)**.** *Given two graphs G, H, a graph morphism $f = (f_N, f_E)$ consists of two partial functions $f_N : N_G \to N_H$ and $f_E : E_G \to E_H$, such that $f_E(e) = e'$ implies $f_N(src(e)) = src(e')$, $lab(e) = lab(e')$, and $f_N(tgt(e)) = tgt(e')$. A graph morphism f is said to be* total *if both functions f_N and f_E are total.*

Thus, the nodes and edges of G are mapped onto the nodes and edges of H. For a graph morphism $m : G \to H$, we define its domain $dom(m)$ and its codomain $cod(m)$, such that:

$$\begin{aligned} dom(m) &= \{x \in (N_G \cup E_G) \mid \exists x' \in (N_H \cup E_H) : m(x) = x'\} \\ cod(m) &= \{x \in (N_H \cup E_H) \mid \exists x' \in (N_G \cup E_G) : m(x') = x\} \end{aligned}$$

2.2.2 Graph Production Rules

A graph production rule specifies a transformation to a graph, as well as when this transformation may be applied, i.e. when the rule is applicable.

Definition 2.3 (graph production rule)**.** *A graph production rule $r = (L, R, p, \mathcal{N})$ consists of a left-hand-side graph L, a right-hand side graph R, a graph morphism $p : L \to R$ mapping elements from the left-hand-side to elements of the right-hand-side, and a set of* negative application conditions *(NAC) $\mathcal{N}$. The rule is* applicable *to a graph G if there is a* total graph morphism *$f : L \to G$ that satisfies all negative application conditions in $\mathcal{N}$.*

The total graph morphism required for the applicability of the rule is also called the *graph matching.* To simplify the definitions in this section, we assume that the rule morphism is a partial identify function; this implies that we cannot *merge* nodes using the transformation definitions given below. However, merge operations are not used in the graph transformation rules used in this thesis. We now explain negative application conditions and define NAC satisfaction.

Application Conditions

Negative application conditions, first introduced in [HHT96] specify elements that may not be in the graph for the rule to match. For a rule $(L, N, p, \mathcal{N})$, $\mathcal{N}$ consists of a set of negative application conditions, which are total graph morphisms $n : L \to N$. N is a graph that consists of the left-hand-side graph extended with elements that are prohibited by the NAC. In the rules in this thesis, all NACs of a rule must be satisfied (o.e., none of the conditions can be in the graph) for the rule to be applicable.

Definition 2.4 (NAC satisfaction)**.** *Let r be a graph production rule $(L, R, p, \mathcal{N})$. For a NAC $n : L \to N \in \mathcal{N}$, a total graph morphism $m : L \to G$ is said to satisfy n if there does not exist a total graph morphism $m_N : N \to G$ such that $m_N \circ n = m$. The graph poduction rule r is said to be applicable to graph G if there exists a matching $m : L \to G$ that satisfies all negative application conditions in $\mathcal{N}$.*

Rule Application

When we apply a rule to G, the elements that are in the LHS but not in the RHS are deleted from G, and elements that are in the RHS but not in the LHS are added. Given a rule $r = (L, R, p, \mathcal{N})$, we identify the sets of nodes and edges to be deleted and those to be created as follows:

$$\begin{aligned}
N_{DEL} &= \{n \mid n\ inN_L, n \notin dom(m_N)\} \\
E_{DEL} &= \{e \mid e \in E_L, e \notin dom(m_N)\} \\
N_{NEW} &= \{n \mid n\ inN_R, n \notin cod(m_E)\} \\
E_{NEW} &= \{e \mid e \in E_R, e \notin cod(m_E)\}
\end{aligned}$$

Example 2.5. *Figure 2.1 shows a rule consisting of an left-hand-side graph, a right-hand-side graph, a morphism between the nodes, and an empty set of NACs. The edge morphism for the* next *edge between the two* Station *nodes has been omitted. The labels on the nodes are in fact self-edges. The* at *edge connecting the* Train *and the* Station *in the left-hand-side graph is not in the domain of the morphism. When we apply the rule in Figure 2.1, the* at *edge between the* Train *and the top* Station *is deleted, and a new* at *edge is created between the* Train *and the bottom (next)* Station*.*

We will show examples of rules with NACs later in this section, when we have introduced a handy visual notation. So far, applying a rule seems quite straightforward. However, things are complicated by two effects:

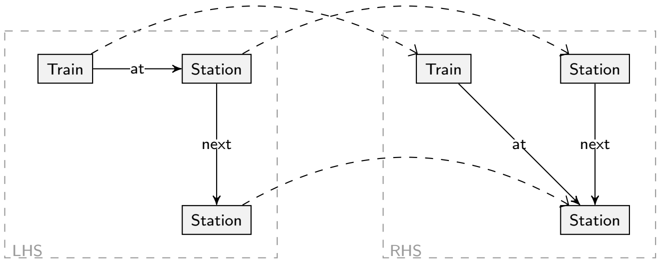


Figure 2.1: An example graph production rule with a left-hand-side, right-hand-side, and a graph morphism.

- The matching may be non-injective; there may be two nodes $n_1 \in N_{DEL}$ and $n_2 \in L \setminus N_{DEL}$ both mapped onto the same node in the graph;
- The matching may be non-surjective on the incident edges of a node scheduled to be deleted, i.e. a source or target node is scheduled to be deleted of an edge that is *not* scheduled to be deleted.

To solve these complications, we specify that deletion always wins. This means that the node in the first complication and the *dangling edge* in the second complication are both deleted.

We now define how to apply a graph production rule $r = (L, R, m)$ in G with a graph matching $f = (f_N, f_E)$ to a resulting graph H.

1. Extend the matching f to a total function f' from $N_L \cup N_R$ by adding fresh images — nodes that are not in N_G — for all the elements of N_{NEW};
2. Construct a graph $J = (f'(N_R), f'(E_R))$;
3. Construct $H = (N_H, E_H)$ with $N_H = N_J \setminus f(N_{r,DEL})$ and $E_H = E_J \setminus f(E_{r,DEL}) \cap K.src^{-1}(N_H) \cap K.tgt^{-1}(N_H)$.

In the first step, we make sure that fresh nodes and edges are created and mapped by those elements that are in R but not in L. The second step results in a graph K in which all new elements are already included. The final step deletes those elements that are in L but not in R, and makes sure no dangling edges remain. This mechanism also “resolves” the issues described above.

In fact, the described solution is the one used in the so-called *single pushout approach* (SPO). This is the approach already chosen by GROOVE [Ren04] — a toolkit for graph transformation — the implementation environment of the control expressions that are introduced in this chapter. GROOVE is unique for its ability to simulate all possible rule applications, and representing these as a labelled transition system. There are other approaches — such as the *double pushout approach* — that solve the problems described above in a different manner. For more information and a detailed comparison see [LCC+96, EHK+97]. The choice of using SPO does not make a difference for the development of the control expressions.

2.2.3 The GROOVE Notation

The rules in this thesis have a special visual notation, that comes from the GROOVE [Ren04] tool suite. In this notation, the elements of a rule (i.e. LHS, RHS) are combined into a single graph representation using different line styles. The elements are represented as follows:

- N_{DEL} and E_{DEL} are depicted by thin, dashed (blue) lines;
- N_{NEW} and E_{NEW} are depicted by the thick (green) lines;
- Elements of NACs that are not in the left-hand-side (i.e. only the part that is "prohibited") are depicted by thick striped (red) lines;
- All other elements are depicted with thin, (black) lines and represent the intersection between the left-hand-side and the right-hand-side.

In other words, this means that the LHS consists of the normal and dashed thin lines, and the RHS consists of the normal thin lines and the thick lines. To ease the comprehensibility of the visual notation, we can summarise the different styles as follows:

- normal elements represent reader elements, which are required to be matched for the rule to be applicable;
- dashed elements represent eraser elements; these elements are required to be matched for the rule to be applicable, and are erased when the rule is applied;
- thick elements represent creator elements; they are not required for the rule to be matched, but are added when the rule is applied.

Example 2.6. *Figure 2.2 shows the same rule as is shown in Figure 2.1, but instead the GROOVE notation is used. It shows the edge that is deleted using a dashed line, and a thick line for the new* `at` *edge that is created. The normal lines represent the* reader *part, that is merely required to be present.*

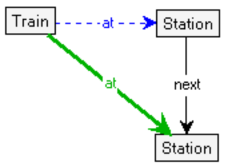


Figure 2.2: The *move* rule of Fig. 2.1 using GROOVE notation.

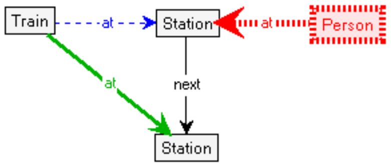
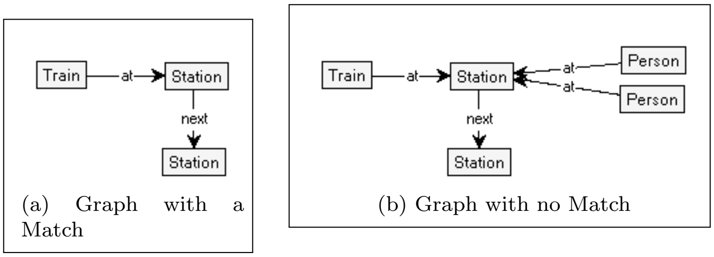


Figure 2.3: A *move* rule extended with a NAC.



(a) Graph with a Match

(b) Graph with no Match

Figure 2.4: Example graphs for the rule in Fig. 2.3

Example 2.7. *Figure 2.3 shows an extension of the* move *rule with a negative application condition. First, the rule only matches when there is a total morphism of the LHS to the graph (i.e. there is a train at a station and — from this station — another station can be reached). Then, there may not exist an extended morphism for the NAC. In the shown rule, this means that there may not be a person at the station the train is currently at. Only then the train may be "moved" to the next station. In Figure 2.4a a rule is shown where the rule does have a match. The rule, however, does not have a match in the graph shown in Figure 2.4b; the matching of a train at a station (and a next station) can be extended for a person at the station. The rule does not care how many persons are at the station.*

2.2.4 Attributed Graphs

In GROOVE it also possible to specify and manipulate data values, such as integers, booleans, and strings. Data can be specified in graphs and rules in the form of attributes, which are essentially edges to special data nodes which represent the actual data values.

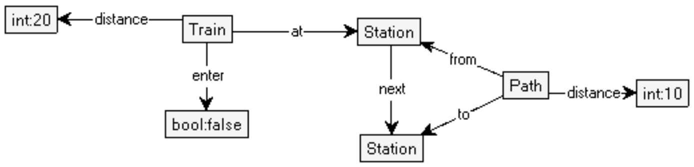


Figure 2.5: Example graph with attributes.

Example 2.8. *Figure 2.5 shows a graph with attributes. In GROOVE, attributes are displayed on the node, thereby hiding the attribute type. A* Path *node represents the distance between two stations with an attribute* distance *with integer value* 10*. The train also has a* distance *attribute with integer* 20*, representing the total distance travelled. The boolean attribute* enter *represents whether or not passengers can enter or leave the train.*

For attribute nodes, the node identity is related to the data value. This means that it makes no difference if two integer attributes with the same value are specified using the same or distinct nodes. Data values can never be created or deleted; they are always (virtually) present and are merely referred to in the graphs.

In addition to specifying data values, it is also possible to manipulate them (e.g. performing calculations in the integer domain). This is done by specifying so-called *product nodes* in a rule. A product node essentially represent a set of data values and an operation. For example, $1 + 2 = 3$ represents input values 1 and 2, output value 3 and the *add* operation. A product node is connected to its input values using π_{num}-labelled *argument edges.* Given the input values, an outgoing operator-edge (an edge labelled with the name of an operation) points to a node with the output value of the operation. For the above operation, this means that nodes with values 1 and 2 are connected to a product node as (numbered) arguments; from the product node, an outgoing *add*-edge points to a node with value 3. The data types of the targets of argument and operator edges must be consistent with the signature of the operation on the product node.

In specifying these operations, a data value can either be specified with a concrete value or an unknown value consistent with the required type of the operation it is used in. We explain this by means of an example.

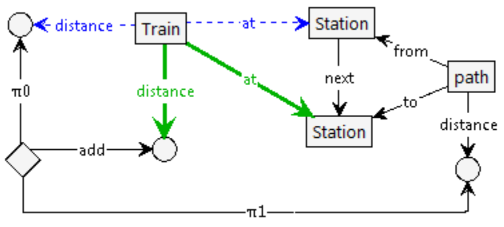


Figure 2.6: Exampe rule with an attribute operation.

Example 2.9. *Figure 2.6 represents the moving of a train from one station to the next as we have seen before. On top of that, a path with a distance has been specified for the two stations. A train is also equipped with a* distance *attribute, representing the total distance travelled. The diamond shaped node denotes a product node. It has two argument edges, one to the current distance travelled by the train, and one to the distance between the two stations. The "add"-labelled operation edge (i.e., it specifies the integer addition operation) points to the result of the operation. When matching the rule, the unvalued operation arguments are matched with concrete integer values in the graph (graphs only have concrete values). When applying the rule for this match, the result of the operation can be calculated, which is represented by the target node of the operation edge. The distance attribute of the train is updated to the calculated value by deleting the old edge and creating a new edge to this value.*

More information — including formal definitions of attributed graphs, rule, and graph transformation in GROOVE — can be found in [Kas08].

2.2.5 Nested Rules

In some of these graph production rules in this thesis, we use another extension to the graph transformation formalism, namely nested rules. Nested rules are used to make changes to sets of sub-graphs at the same time, rather then just at an existentially matched left-hand-side. We just explain them informally here. More information — including the formal definitions of matching and applying nested rules — can be found in [Ren09].

In essence, instead of only being able to select a particular sub-graph by expressing "there exists a node there that is connected to a node such", with nested rules we can for example express rules that express: "if there exists a node A, then for a nodes B connected to this node A, if there exists a node C connected this node B, then delete node B".

Specification of a nested rule in GROOVE is done through nesting levels, that use either universal or existential quantification. In fact, the graph production rules defined so far can be seen as having a single existential matching level. The definition of these levels is done be adding special nodes, labelled with an ∃ or ∀ symbol, and connected using in-labelled edges pointing "upwards". Moreover, existential and universal rules are alternating; the first (implicit) level is existential, the second universal, and so on. The matching of a rule is specified by assigning nodes to a nesting level using at-labelled edges.

Matching is done starting from the top level. For an existential level, a matching is required for the nodes and edges in this level. For a universal level, the morphism is extended to include all elements that are matched by the universal level. Edges between nodes in one level are implicitly part of this level; edges between levels belong to the lowest level.

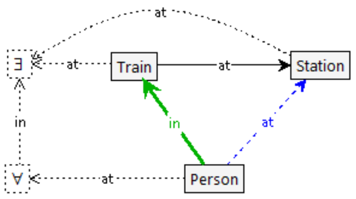


Figure 2.7: An example of a nested rule.

Example 2.10. *Figure 2.7 shows an example of a nested rule. The dotted nodes and edges represent the hierarchical nesting levels: the top ∃-node represents the first, existential level, and the bottom ∀-node represents the second, universal level. The hierarchy is represented by the dotted in-edge. The* Train *and* Station *nodes belong to the existential level, the* Person *belongs to the nested universal level. This is specified by the dotted at-edges. First, a* Train at *a* Station *is matched. Then, the* Person *node matches* all Person *nodes in the host graph that are connected to the (already matched)* Station *by a* at*-labelled edge. When applied, the rule will remove all* at *edges between the matching* Person *nodes and the matched* Station *node, and creates* in *edges between these* Person *nodes and the matched* Train *node. Intuitively, the rule matches a train at a station, and lets all people at this station board the train.*

2.2.6 Exploration Strategies

Given a start graph and a set of rules, this rule system can be represented as a labelled transition system (LTS), where the states are graphs, and the transitions are in fact rule applications. The GROOVE simulator supports different exploration strategies. These strategies specify in what order the state space is explored.

Full Exploration

A full exploration results in a state space where all states are completely explored; outgoing transitions exist for all matches of all rules in all graphs. Generating this full state space can be done either breadth-first or depth-first, but this makes no difference for the result, which is defined by the input (i.e. the start graph and the rules).

Linear Exploration

In a linear exploration, a single match is selected from the possible matches of the rules in a graph. The application results in a new graph where again a single match is selected. This is repeated until a known graph is reached (i.e. a cycle is formed) or when a graph is reached without any possible matches — a so-called final state. This strategy is typically suitable for finding a random final state in finite systems.

2.3 Rule Systems and Automata

Since this work is placed in the general context of rule systems, we first define our conception of such systems. Throughout this chapter we denote the universe of all rules as Rule.

Definition 2.11 (rule system)**.** *A rule system is a set of rules $R \subseteq \mathsf{Rule}$, which act on a universe of data structures, Data, in a manner captured by a partial* derivation *function $\delta : \mathsf{Data} \times R \times \mathsf{Id} \rightharpoonup \mathsf{Data}$, where Id is a set of* application identifiers.

The fact that $\delta(d, n, i) = d'$, which we will henceforth denote $d \xrightarrow{n,i} d'$, expresses that rule n can be applied to structure d, resulting in a new structure d'. The identifier i provides information on how n was applied precisely; this has the effect of making the result d' unambiguous.

Without going into details, we note that this definition encompasses a wide spectrum of systems, including Turing machines, Petri nets, and various kinds of rewrite systems.

Example 2.12. *In a graph production rule system, R elements are graph production rules, Data elements are graphs, and Id is the set of matchings.*

The meaning of a rule-based system is usually taken to be the *normal forms* reachable from a given input structure d; that is, those structures $d' \in \mathsf{Data}$ such that there exists a chain of derivations $d \xrightarrow{r_1,i_1} \xrightarrow{r_2,i_2} \cdots \xrightarrow{r_n,i_n} d'$ and there are no further derivations possible from d'. In other words, one is interested in the *transformational* or *input-output* behaviour of the rule system. In this work, on the other hand, we consider the *temporal* or *reactive* behaviour of rule systems, which can only be captured by taking intermediate steps into account. To formalise the reactive behaviour, we use *automata.* For the sake of simplicity, in this chapter we identify rules with their names; hence, we will use rules as (part of) transition labels. Throughout the chapter, we will use R to stand for the rule system under consideration, with associated sets Data of data structures and Id of application identifiers.

2.3.1 Automata

We distinguish system automata and control automata, with the same structure. System automata represent the behaviour of a system, whereas control automata specify the controlling behaviour that can be added to a system. First we introduce the general concept of automaton.

Definition 2.13 (automaton). *An automaton $\mathcal{A}$ is a tuple $(Q, \Sigma, \rightarrow, q_0, S)$ where*

- *Q is a set of states;*
- *Σ is a finite alphabet, which may include the special symbol λ;*
- *$\rightarrow \subseteq Q \times \Sigma \times Q$ is a set of transitions;*
- *$q_0 \in Q$ is the start state;*
- *$S \subseteq Q$ is a set of success states.*

A is called deterministic *if for all $q \in Q$ and $\ell \in \Sigma$, $\ell \neq \lambda$ and $q \xrightarrow{\ell} q_1$ and $q \xrightarrow{\ell} q_2$ implies $q_1 = q_2$.*

Intuitively, a success state is a state in which it is correct to halt execution. The special symbol λ stands for an *invisible* step. We use the following notations:

$$\begin{aligned} q \xrightarrow{\ell_1 \cdots \ell_n} q' \quad &:\Leftrightarrow \quad q \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n} q' \\ q \xRightarrow{\epsilon} q' \quad &:\Leftrightarrow \quad \exists n : q \xrightarrow{\lambda^n} q' \\ q \xRightarrow{a_1 \cdots a_n} q' \quad &:\Leftrightarrow \quad q \xRightarrow{\epsilon} \xrightarrow{a_1} \cdots \xRightarrow{\epsilon} \xrightarrow{a_n} q' \ . \end{aligned}$$

Moreover, we use $q \xrightarrow{a}$ to denote $\exists q' : q \xrightarrow{a} q'$, etc. Finally, we define the *language* of an automaton as the set of all traces $\mathcal{T}(\mathcal{A})$, with a distinguished subset of *successful* traces $\mathcal{T}^{\surd}(\mathcal{A})$. Both are needed to examine the trace-equivalence of automata; not every trace ends in a success state. For all $\mathcal{A} \in \mathsf{Aut}$:

$$\mathcal{L}(\mathcal{A}) = (\mathcal{T}(\mathcal{A}), \mathcal{T}^{\surd}(\mathcal{A})), \text{ where } \begin{array}{ll} \mathcal{T}(\mathcal{A}) & = \{w \in (\Sigma \setminus \lambda)^* \mid q_0 \overset{w}{\Longrightarrow} \} \\ \mathcal{T}^{\surd}(\mathcal{A}) & = \{w \in (\Sigma \setminus \lambda)^* \mid q_0 \overset{w}{\Longrightarrow}\overset{\epsilon}{\Longrightarrow} q' \in S\} \end{array}$$

It follows from standard theory for these acceptance conditions, that every language (to be precise, every pair $(\mathcal{T}, \mathcal{T}^{\surd})$ with $\mathcal{T} \subseteq (\Sigma \setminus \lambda)^*$ non-empty and prefix-closed and $\mathcal{T}^{\surd} \subseteq \mathcal{T}$) is (uniquely up to bisimilarity) represented by a deterministic (though generally infinite) automaton.

2.3.2 System Automata

A system automaton is essentially a state-transition system describing the step-by-step derivations of a rule system.

Definition 2.14 (system automaton)**.** *A* system automaton *is an automaton with* $\Sigma = (\mathit{Rule} \times \mathit{Id}) \cup \{\lambda\}$, *such that every* $q \in Q$ *has an associated data structure* $d_q \in \mathit{Data}$, *satisfying the following consistency properties:*

$$\begin{array}{lcl} q \xrightarrow{\lambda} q' & :\Leftrightarrow & d_q = d_{q'} \\ q \xrightarrow{r,i} q' & :\Leftrightarrow & d_q \xrightarrow{r,i} d_{q'} \end{array} .$$

For arbitrary $d \in \mathit{Data}$, *the* free automaton $\mathcal{A}_d$ *is a system automaton with data structures as states and derivations as transitions, which is the smallest such that:*

- $Q \subseteq \mathit{Data}$,
- $q_0 = d$, $q_0 \in Q$,
- *for all* $d' \in Q$ *and* $\delta(d', n, i) = d''$, $d'' \in Q$ *and* $d' \xrightarrow{n,i} d''$
- $S = Q$.

The set of system automata is denoted SAut. The (r, i)-labelled transitions are essentially rule applications, whereas in a λ-transition no data transformation occurs. Given that states are data structures combined with some extra information, a λ-transition represents a change in this extra information. The free automaton is the result of uncontrolled rule application at every state. Since no constraints are specified regarding the correctness of halting executions, every state is a successful state.

Example 2.15. *Figure 2.8 shows a system automaton based on the start graph shown in Figure 2.9 and two rules:*

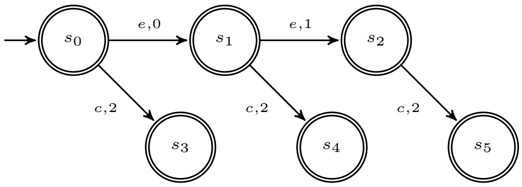


Figure 2.8: Example System Automaton

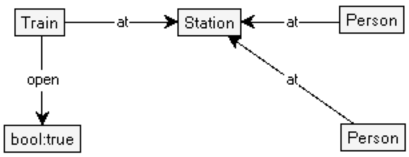
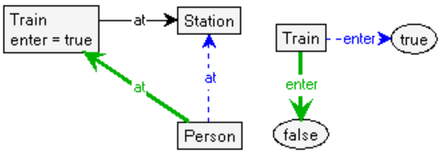


Figure 2.9: Start graph of the example rule system.



(a) enter (b) close

Figure 2.10: Rules of the example rule system.

e : This rule (Fig. 2.10a) represents a passenger entering *a train.*

c : This rule (Fig. 2.10b) represents the closing *of the train door.*

This simple process is shown for a start state where two passengers want to enter the train. The e-rule can be applied twice. The door can be closed once, regardless of the passengers being inside or outside the train. All states are successful in the free automaton.

Determinism.

We follow standard automata theory in equating all automata with their languages; or in other words, every automaton is considered to be essentially the same as its determinisation (according to the standard powerset construction). For system automata, this is justified because their labels are enriched so that

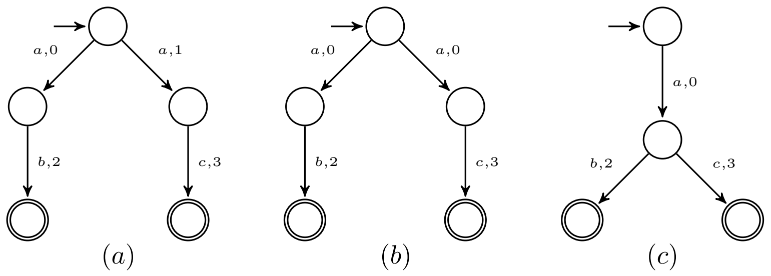


Figure 2.11: Different types of non-determinism

they do not only contain rule names but also application identifiers. Only the rule names are typically observable; for instance, verification methods such as model checking only take note of the rule name part of the labels.

If we would project all Rule × Id-labels of a system automaton onto their first components, the resulting automaton would in general be non-deterministic; but this type of non-determinism can *not* be resolved without changing the meaning of the automaton; typically some form of bisimilarity is imposed instead. This implies that this projection does not preserve the intended semantics of the automata. For instance, in Fig. 2.11, automata (b) and (c) are language equivalent, whereas (a) is different; however, after projection onto the rule names, (a) and (b) appear essentially the same (viz., isomorphic) whereas (c) appears to be different.

In other words, after projection onto the rule names, equally named transitions may either be from rule applications with equal or with different identifiers. In a free automaton, non-determinism after projection is only caused by having multiple applications of the same rule in the same state, and never by having the same rule application twice. As we will see next, imposing control on a system automaton may cause spurious non-determinism (equal transitions represented more than once).

To avoid this type of confusion, we prefer to work only with deterministic system automata. Obviously, this can be achieved by determinising automata whenever necessary. However, this might not be the best technique, as determinisation can be exponential in the size of the automaton, and system automata are likely to be very large. Part of the contribution of this work is therefore a technique to avoid generating non-deterministic system automata altogether.

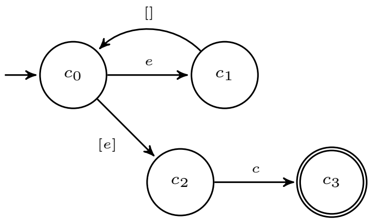


Figure 2.12: Example Control Automaton

2.3.3 Control Automata

Control automata are automata that can express, on the one hand, the application of a rule and on the other the observation that a given set of rules cannot be applied. The latter is called a *failure*. The set of all possible failures is given by $\mathsf{Fail} = 2^{\mathsf{Rule}}$.

Definition 2.16 (control automaton). *A* control automaton *is an automaton where $\Sigma = \mathsf{Rule} \cup \mathsf{Fail}$, such that:10*

- *For all $q \in Q$, $q \xrightarrow{F}$ with $F \in \mathsf{Fail}$ only if $\forall a \in F : q \xrightarrow{F_1 \dots F_n} \xrightarrow{a}$ with $a \notin F_1 \cup \dots \cup F_n$, $F_i \in \mathsf{Fail}$.*
- *For all $q \in S$, $a \in \mathsf{Rule}$ and $F \in \mathsf{Fail}$, there is no transition $q \xrightarrow{a}$ or $q \xrightarrow{F}$.*

A failure transition with failure F may exist from a state q only if from q each of the rules a in the failure can also be found on a normal transition via a path consisting of only failure transitions $F_1 \dots F_n$, and none of these failure transitions contains this a. This guarantees that observing the non-applicability has meaning, only if applying the rule does to. From a success state, no outgoing transitions are allowed. Having this constraint simplifies the definition of control automaton operations — such as sequential composition — by merging states.

The class of control automata is denoted CAut.

The empty failure means that all rules in the empty set are inapplicable, which is vacuously true; hence, an empty failure transition is effectively a λ-transition.

Example 2.17. *Figure 2.12 shows a control automaton in which rule e is scheduled as long as it is possible, after which c is scheduled. In the setting of Example 2.15, the door is closed only when no more passengers want to enter the train. Here, $c_0 \xrightarrow{[e]} c_2$ denotes a failure transition with label $\{e\}$. [] represents the empty failure, which (as noted above) has the same behaviour as a λ-transition.*

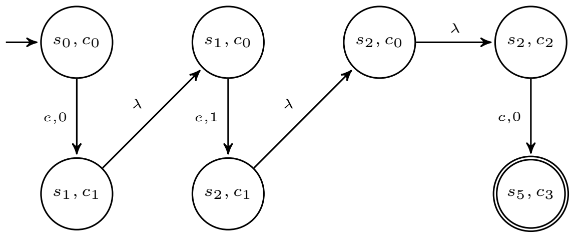


Figure 2.13: Example Product System Automaton

2.3.4 Combining System Behaviour and Control

The idea of a failure as the observation of a set of rules being inapplicable is given a meaning by defining the product of control and system automaton. This results in another system automaton, where states are tuples of a system state and a control state.

Definition 2.18 (product). *The* product *of a system automaton $\mathcal{A}$ and a control automaton $\mathcal{C}$ is defined as $\mathcal{A} \times \mathcal{C} = (Q_\mathcal{A} \times Q_\mathcal{C}, \Sigma_\mathcal{A}, \rightarrow, (q_{0,\mathcal{A}}, q_{0,\mathcal{C}}), S_\mathcal{A} \times S_\mathcal{C})$, where the transition relation is defined by the following rules:*

$$\frac{q_\mathcal{A} \xrightarrow{n,i}_\mathcal{A} q'_\mathcal{A} \quad q_\mathcal{C} \xrightarrow{n}_\mathcal{C} q'_\mathcal{C}}{(q_\mathcal{A}, q_\mathcal{C}) \xrightarrow{n,i} (q'_\mathcal{A}, q'_\mathcal{C})} \qquad \frac{q_\mathcal{C} \xrightarrow{F}_\mathcal{C} q'_\mathcal{C} \quad \forall n \in F, k \in \mathit{Id} : q_\mathcal{A} \not\xrightarrow{n,k}_\mathcal{A}}{(q_\mathcal{A}, q_\mathcal{C}) \xrightarrow{\lambda} (q_\mathcal{A}, q'_\mathcal{C})}$$

$$\frac{q_\mathcal{A} \xrightarrow{\lambda}_\mathcal{A} q'_\mathcal{A}}{(q_\mathcal{A}, q_\mathcal{C}) \xrightarrow{\lambda} (q'_\mathcal{A}, q_\mathcal{C})}$$

The control automaton constrains the system automaton on the name-component of its traces. In particular, in a combination of system state and a certain control state, a failure transition from that control state can be taken when for none of the names in the failure set an outgoing rule application exists in the system state. Because a failure transition in the control automaton results in a λ-transition in the product automaton, the alphabet of the product is the same as that of the input system automaton.

For example, Figure 2.13 shows the system automaton that is the product of the system and control automaton in Figures 2.8 and 2.12.

2.4 Control Language

For the convenient specification of control automata, we propose to use a control language along the lines of what has previously been proposed in, e.g., [HP01].

We will use the following grammar:

$$
\begin{array}{rcl}
P & := & \mathsf{rule} \mid \textbf{true} \mid P_1|P_2 \mid P_1;P_2 \mid P* \mid \textbf{alap}\ P \mid \textbf{try}\ P_1 \mid \textbf{try}\ P_1\ \textbf{else}\ P_2 \mid \\
 & & \textbf{if}\ (P_1)\ P_2 \mid \textbf{if}\ (P_1)\ P_2\ \textbf{else}\ P_3 \mid \textbf{while}\ (P_1)\ \textbf{do}\ P_2 \mid \textbf{until}\ (P_1)\ \textbf{do}\ P_2
\end{array}
$$

These constructs have the following intuitive meaning:

- rule schedules the execution of a single rule (named rule);
- **true** behaves like a rule that is always successful and does not change the underlying structure;
- $P_1|P_2$ is the *non-deterministic choice* of P_1 and P_2;
- $P_1;P_2$ is the *sequential composition* of P_1 and P_2;
- $P*$ (the *Kleene closure*) schedules P an arbitrary number of times;
- **alap** P (*as long as possible*) schedules P until it fails;
- **try** P_1 schedules P_1, and is skipped if P_1 fails;
- **try** P_1 **else** P_2 schedules P_1 first, and schedules P_2 in case P_1 fails;
- **if** (P_1) P_2 schedules P_1 first and schedules P_2 in case P_1 succeeds;
- **if** (P_1) P_2 **else** P_3 schedules P_1 first and schedules P_2 in case P_1 succeeds or P_3 in case P_1 fails;
- **while** (P_1) **do** P_2 schedules P_1 and then P_2 until P_1 fails;
- **until** (P_1) **do** P_2 schedules P_1, but P_2 and again P_1 in case P_1 fails;

Traditionally (e.g., in [HP01]), the effect of such control programs is defined in terms of the resulting input-output behaviour for the rule system at hand. Formally, this is captured by a function $[\![-]\!]_{\mathsf{io}} : \mathsf{Lang} \to \mathsf{Data} \times \mathsf{Data}$. For instance, for some of the operators, the defining clauses are as follows, where r is a rule and i is a application identifier:

$$
\begin{array}{rcl}
[\![\mathsf{r}]\!]_{\mathsf{io}} & = & \{(d,d') \mid \exists r,i : d \xrightarrow{r,i} d'\} \\
[\![P_1;P_2]\!]_{\mathsf{io}} & = & \{(d,d') \mid (d,d'') \in [\![P_1]\!]_{\mathsf{io}}, (d'',d') \in [\![P_2]\!]_{\mathsf{io}}\} \\
[\![P*]\!]_{\mathsf{io}} & = & \{(d,d) \mid d \in \mathsf{Data}\} \cup \{(d_0,d_n) \mid (d_0,d_1),\ldots,(d_{n-1},d_n) \in [\![P]\!]_{\mathsf{io}}\} \\
[\![\textbf{alap}\ P]\!]_{\mathsf{io}} & = & \{(d,d') \in [\![P*]\!]_{\mathsf{io}} \mid \nexists d'' : (d',d'') \in [\![P]\!]_{\mathsf{io}}\}
\end{array}
$$

Using control automata, we express the reactive rather than the input-output behaviour of Lang; or in other words, it is a small-step semantics rather than a big-step semantics. An advantage, furthermore, is that we capture the meaning of control expressions without reference to any particular rule system. However, our approach inevitably implies that the meaning of, for instance, **alap** changes with respect to the above definition: rather than considering a sub-expression "possible" if it can run to a successful completion, we consider it "possible" if it can do a *single* step. For example, in **alap**$\{a; b; \}$, the **alap** repeats its body while the first rule is possible — in this case when a is possible.

```
alap{ enter } ; close
```

Listing 2.1: Expression corresponding to Fig. 2.12

Given the language, we can formulate conditional scheduling of a rule, apply a rule as long as possible or any number of times. For example, the control automaton shown in Figure 2.12 corresponds to the control specification in Listing 2.1. The *enter* rule is scheduled repeatedly until it can no longer be applied and is then followed by the *close* rule.

To specify the example, we do not have to know the details of the rules. In principle, the control program can be applied to any rule system with rules named *enter* and *close*.

The "alap" and "try" constructs are the elementary constructs in the language, whereas other constructs are merely syntactic sugar of these. These are introduced to make the semantics of the language more intuitive:

- **if** (P_1) P_2 is the same as **try** $\{P_1; P_2\}$;
- **if** (P_1) P_2 **else** P_3 is the same as **try** $\{P_1; P_2\}$ **else** P_3;
- **while** (P_1) **do** P_2 is the same as **alap** $\{P_1; P_2\}$.

Next, we define the language semantics. Constructs that are syntactic sugar are omitted.

2.4.1 Semantics

The semantics of Lang is defined through a set of operators over CAut inductively. For this purpose, we first need to define what it means for an automaton to *fail*; this is an important concept in the definition of **alap** and **try-else**. The failure of an automaton is based on the non-applicability of its initial actions, being those rules that are scheduled as a first action. This set of initial actions is defined as follows:

Definition 2.19 (initial control actions). *The initial actions of a given control automaton $\mathcal{C}$ are defined by*

$$Init(\mathcal{C}) = \{a \mid q_0 \xrightarrow{F_1 \ldots F_n} \xrightarrow{a}\} \cup \{\delta \mid \exists q' : q_0 \xrightarrow{F_1 \ldots F_n} q' \in S\}$$

Here a represent a rule. The result is a set of rules that can be performed first, optionally after observing a number of failures. A δ in the result indicates that in the given automaton there is a path consisting of only failure transitions from the start state to a success state. Intuitively, this means that a success state can be reached without performing any actions (apart from observing what cannot be done); A δ in the set of initial control actions therefore represents a guaranteed success. Later we will see that this implies that the failure of the automaton is not defined.

Figure 2.14 shows the construction operations. In these definitions, we use control automata $\mathcal{C}_i = (Q_i, \Sigma, \rightarrow_i, q_{0_i}, S_i)$ for $i = 1, 2$, and we use distinct fresh states $q_N, q_M \notin Q_i$. The ? represents a “try (else)” operation, the $\downarrow$ represents the “alap” operations, and ! $\rightarrow$ represents an “until-do“. The following points are noteworthy:

- In the sequential composition $\mathcal{C}_1;\mathcal{C}_2$, every transition in $\mathcal{C}_1$ to a success state is redirected to the start state of $\mathcal{C}_2$.
- The **true** keyword is simply represented by a single success state.
- In the “non-determistic choice” $\mathcal{C}_1 \mid \mathcal{C}_2$, outgoing transitions of the original start states are replaced to use a fresh start state q_N.
- In the “alap closure” $\mathcal{C}_1\downarrow$, transitions to success states are redirected to the start state. Optionally, a failure transition of the *init* of the automaton is created to a fresh state q_N.
- In the “try-else” operation $\mathcal{C}_1?\mathcal{C}_2$, the start state of $\mathcal{C}_1$ is optionally connected to the start state of $\mathcal{C}_2$ by a failure transition.
- In the “until-do” operation $\mathcal{C}_1! \rightarrow \mathcal{C}_2$, an optional failure transition is created of the *init* of $\mathcal{C}_1$, sequentially composed with $\mathcal{C}_2$ and looped back to the start state.

We now state the following property:

Proposition 2.20. CAut *is closed under the constructions defined above.*

The proof is straightforward. We show that, given C_i for $i = 1..n$ in CAut, conditions 1 and 2 in definition 2.16 hold for all constructed automata.

$$
\begin{array}{rcl}
\mathcal{C}_{\mathsf{rule}} & = & (\{q_M, q_N\}, \Sigma, \{(q_N, \mathsf{rule}, q_M)\}, q_M, \{q_N\}) \\
\mathcal{C}_{\mathsf{true}} & = & (\{q_N\}, \Sigma, \emptyset, q_N, \{q_N\}) \\
\mathcal{C}_1 \mid \mathcal{C}_2 & = & (Q, \Sigma, \rightarrow, q_N, S_1 \cup S_2), \text{where} \\
 & & Q = Q_1 \setminus \{q_{0,1}\} \cup Q_2 \setminus \{q0, 2\} \cup \{q_N\} \\
 & & \rightarrow\, = \,\rightarrow_1 \setminus \{(q_{0,1}, x, q) \mid q_{0,1} \xrightarrow{x}_1 q\} \cup \{(q_N, x, q') \mid q_{0,1} \xrightarrow{x}_1 q'\} \\
 & & \qquad \cup \rightarrow_2 \setminus\{(q_{0,2}, x, q) \mid q_{0,2} \xrightarrow{x}_2 q\} \cup \{(q_N, x, q) \mid q_{0,2} \xrightarrow{x}_2 q\} \\
\mathcal{C}_1 ; \mathcal{C}_2 & = & (Q_1 \setminus S_1 \cup Q_2, \Sigma, \rightarrow, q_{0,1}, S_2), \text{where} \\
 & & \rightarrow\, = \,\rightarrow_1 \setminus \{(q, x, q') \mid q \xrightarrow{x}_1 q' \in S_1\} \\
 & & \qquad \cup \{(q, x, q_{0,2}) \mid q \xrightarrow{x}_1 q' \in S_1\} \cup \rightarrow_2 \\
\mathcal{C}_1 \star & = & (Q_1 \cup \{q_N\}, \Sigma, \rightarrow, q_{0,1}, \{q_N\}), \text{where} \\
 & & \rightarrow\, =\rightarrow_1 \cup \{(q, \lambda, q_N) \mid q \in S_1\} \\
\mathcal{C}_1 \downarrow & = & (Q_1 \setminus S_1 \cup \{q_N\}, \Sigma, \rightarrow, q_{0,1}, \{q_N\}), \text{where} \\
 & & \rightarrow\, = \,\rightarrow_1 \setminus\{(q, x, q') \mid q \xrightarrow{x}_1 q' \in S_1\} \cup \{(q, x, q_{0,1}) \mid q \xrightarrow{x}_1 q' \in S_1\} \\
 & & \qquad \cup \{(q_{0,1}, \mathit{Init}(C_1), q_N) \mid \delta \notin \mathit{Init}(C_1)\} \\
\mathcal{C}_1 ? & = & (Q_1 \cup \{q_N\}, \Sigma, \rightarrow, q_{0,1}, S_1 \cup \{q_N\}), \text{where} \\
 & & \rightarrow\, = \,\rightarrow_1 \cup \{(q_{0,1}, \mathit{Init}(C_1), q_N) \mid \delta \notin \mathit{Init}(C_1)\} \\
\mathcal{C}_1 \mathbin{?} \mathcal{C}_2 & = & (Q_1 \cup Q_2, \Sigma, \rightarrow, q_{0,1}, S_1 \cup S_2), \text{where} \\
 & & \rightarrow\, = \,\rightarrow_1 \cup \rightarrow_2 \cup \{(q_{0,1}, \mathit{Init}(C_1), q_{0,2}) \mid \delta \notin \mathit{Init}(C_1)\} \\
\mathcal{C}1! \rightarrow \mathcal{C}2 & = & (Q_1 \cup Q_2, \Sigma, \rightarrow, q_{0,1}, S_1), \text{where} \\
 & & \rightarrow\, = \,\rightarrow_1 \cup \{(q_{0,1}, \mathit{Init}(C_1), q_{0,2}) \mid \delta \notin \mathit{Init}(C_1)\} \\
 & & \qquad \cup \rightarrow_2 \cup \{(q, x, q_{0,2}) \mid q \xrightarrow{x}_2 q' \in S_2\}
\end{array}
$$

Figure 2.14: Construction operators over CAut.

Proof.

Condition 1. All states of C_i fulfil the condition. Any removed transitions are replaced by new transitions using the same source state. Therefore, the property cannot be invalidated for any already existing failure transitions. Addition of failure transitions is always done by using the start state of an automaton (C_i) as its source state; the failure on the transition becomes $Init(C_i)$. Therefore, by definition (Def. 2.19), the source state of newly created failure transitions satisfy property 1 of Def. 2.16.

Condition 2. In all constructions, S is formed by using S_i (already fulfilling the conditions) and/or a fresh state q_N; no outgoing transitions are created from any of these states. □

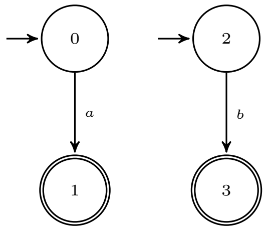
This gives rise to the following semantic function $[\![-]\!]_{\mathsf{aut}} : \mathsf{Lang} \to \mathsf{CAut}$:

$$
\begin{aligned}
[\![\mathsf{rule}]\!]_{\mathsf{aut}} &= \mathcal{C}_{\mathsf{rule}} \\
[\![\mathsf{true}]\!]_{\mathsf{aut}} &= \mathcal{C}_{\mathsf{true}} \\
[\![P_1|P_2]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}} \mid [\![P_2]\!]_{\mathsf{aut}} \\
[\![P_1;P_2]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}; [\![P_2]\!]_{\mathsf{aut}} \\
[\![P_1*]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}\star \\
[\![\textbf{alap } P_1]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}\downarrow \\
[\![\textbf{try } P_1]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}? \\
[\![\textbf{try } P_1 \textbf{ else } P_2]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}} \mathbin{?} [\![P_2]\!]_{\mathsf{aut}} \\
[\![\textbf{until } (P_1) \textbf{ do } P_2]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}! \to [\![P_2]\!]_{\mathsf{aut}}
\end{aligned}
$$

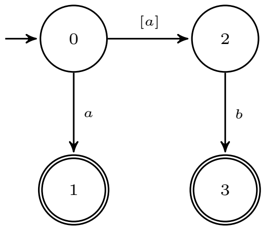
As we have pointed out above, this differs from the semantics studied in [HP01] (and elsewhere) in the treatment of failure, which for us is the failure of a small step but for them the failure of a big step. For instance, the control program $P =$ **alap** $(\mathsf{a};\mathsf{b});\mathsf{c}$ imposed on a rule system where $d \xrightarrow{\mathsf{a},i} d'$ and $d \xrightarrow{\mathsf{c},j} d''$ but $\nexists k : d' \xrightarrow{\mathsf{b},k}$ gives rise to $(d, d'') \in [\![P]\!]_{\mathsf{io}}$ ($\mathsf{a};\mathsf{b}$ fails and is completely skipped by backtracking), whereas $[\![P]\!]_{\mathsf{aut}} \times \mathcal{A}_d$ only contains the transition $(d, q_0) \xrightarrow{\mathsf{a},i} (d', q_1) \notin S$, without further outgoing transitions.

We illustrate the process of control automata construction for a control program for the program **alap** $\{$**try** a **else** $\mathsf{b}\};\mathsf{c}$. The states are numbered for clarity reasons.

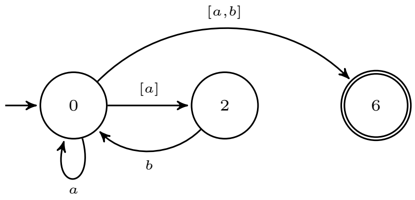
First we start with the base automata $[\![\mathsf{a}]\!]_{\mathsf{aut}}$ and $[\![\mathsf{b}]\!]_{\mathsf{aut}}$.



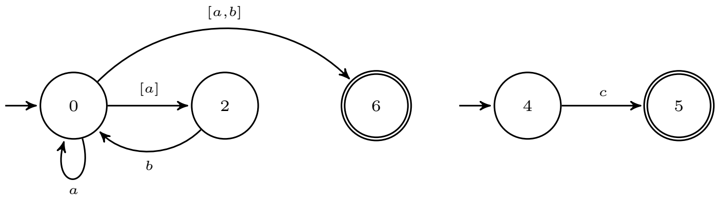
For the "try-else" construction, a single failure transition is added between states 0 and 2. The failure consists of $Init([\![\mathsf{a}]\!]_{\mathsf{aut}}) = \{a\}$. State 0 becomes the start state of the constructed automaton.



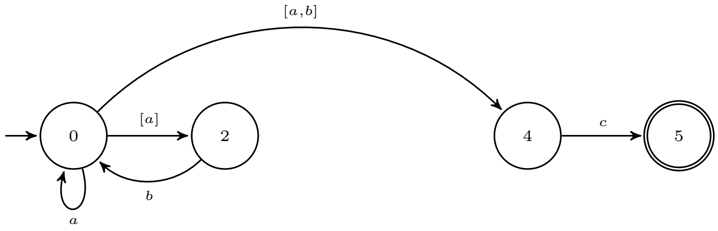
In the “alap” construction, transitions to final states are redirected to the initial state, and from this initial state, a failure transition is created to a fresh state 6. The failure consists of $Init([\![\textbf{try}\ \mathsf{a}\ \textbf{else}\ \mathsf{b}]\!]_{\mathsf{aut}}) = \{a, b\}$



Next, the automaton $[\![\mathsf{c}]\!]_{\mathsf{aut}}$ is created.



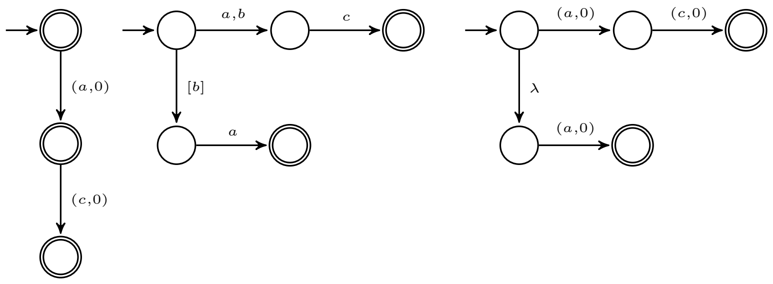
Finally, for the sequential composition, the incoming transitions of state 6 are substituted with transitions to state 4. Here it becomes clear why having no outgoing transitions from success states is convenient.



2.5 Guarded Control Automata

The product operation defined in Definition 2.18 can result in a system automaton that is non-deterministic in the sense of Def. 2.13, even when system and control automaton are both deterministic.

Example 2.21. *Consider the following automata:*



The system automaton on the left is clearly deterministic in the sense of Def. 2.13. In its product with the control automaton in the middle, shown on the right, there is a λ-transitions. Both before and after this transitions the rule application $(a, 0)$ *occurs. However, after the top-most* $(a, 0)$ *another transition* $(c, 0)$ *is possible. Thereby, the mechanism for combining a system automaton with a control automaton has created spurious non-determinism.*

The desired result of synchronising a control automaton and a deterministic system automaton is a deterministic controlled system automaton. Failure transitions give rise to λ-transitions in the product automaton. After such failures, the same rule applications may be scheduled. This can cause different traces to be generated in the product after λ-transitions.

The introduction of such non-determinism is undesirable. For that purpose, we introduce *guarded control automata*. Here, every transition consists of a rule name with a positive and negative guard, both of which are sets of rules, and n is the rule that is applied. For n to be enabled, all rules in the negative guard must fail to be applicable, and all rules in the positive guard must be applicable (i.e. each of the rule in the positive guard must have at least one match). We also introduce a *determinisation* operation for normal control automata that produces a guarded control automaton.

We use the notation $q \xrightarrow{[F|A]n} q'$ for transitions with guards, where F is the negative and A is the positive guard. When $A = \emptyset$, we use the notation $q \xrightarrow{[F]n} q'$; $q \xrightarrow{n} q'$ denotes that $F = A = \emptyset$.

Definition 2.22 (guarded control automaton). *A guarded control automaton is a deterministic automaton with $\Sigma = \mathsf{Fail} \times 2^{\mathsf{Rule}} \times \mathsf{Rule}$ and $S \subseteq Q \times \mathsf{Fail}$. Transitions should satisfy the following constraints for all $q \in Q$:*

1. *$q \xrightarrow{[F|A]n}$ implies $F \cap A = \emptyset$*
2. *$q \xrightarrow{[F_1|A_1]n}$ and $q \xrightarrow{[F_2|A_2]n}$ implies $F_1 \cup A_1 = F_2 \cup A_2$*
3. *$q \xrightarrow{[F|A]n}$ implies $q \xrightarrow{[F \cup A]n}$.*

The transitions in a guarded control automaton represent the scheduling of a rule n and the negative guard, consisting of the rules that may not be applicable for the rule to be scheduled. However, there may be multiple transitions n with different negative guards. If a negative guard F_1 is satisfied in a system state, then a negative guard $F_2 \subset F_1$ is automatically also satisfied. To have only deterministic synchronisation, we must therefore also specify which rules have to be applicable.

It is possible that there are more transitions with the same rule n but with different negative and possible guards F and A. However, in a specific system state, only one distinct combination of F and A can be satisfied: a rule can only be enabled or disabled, and both guarded transitions must have the same rules divided over F and A.

Since only one combination of A and F can be satisfied in a system state, there is at most one n transition in a guarded control state that can be synchronised with an application of n from a system state. Success states are now also conditional (or guarded). The class of guarded control automata is denoted GAut.

We define the failure dependency function fd from actions and sets of states to sets of actions. It gives for each action n and states qs the union of all possible failures that lead to a state where n is allowed.

Definition 2.23 (failure dependency). *The failure dependency in a set of states qs for a rule n is defined by:*

$$fd(qs, n) = \bigcup \{F_i \mid \exists q \in qs : q \xrightarrow{F_1 \ldots F_n} q' \xrightarrow{n}\}$$

Determinisation of a control automaton is given by a function det:

Definition 2.24 (control automaton determinisation). *Given a control automaton $\mathcal{C}$, $det(\mathcal{C})$ is an automaton with $Q = 2^{Q_\mathcal{C}} \setminus \emptyset$, $q_0 = \{q_{0,\mathcal{C}}\}$, where $\rightarrow$ is defined by:*

$$\frac{F \subseteq fd(q, n) \quad A = fd(q, n) \setminus F}{qs \xrightarrow{[A|F]n} \{q'_\mathcal{C} \mid q_\mathcal{C} \in qs : q_\mathcal{C} \xRightarrow{F_1 \ldots F_n} \xrightarrow{n} \xRightarrow{\varepsilon} q'_\mathcal{C}, F_1 \cup \ldots \cup F_n \subseteq F\}}$$

The set of success states is defined by:

$$S = \{(qs, \bigcup_i F_i) \mid \exists q_\mathcal{C} \in qs : q_\mathcal{C} \xrightarrow{F_1 \dots F_n} q'_\mathcal{C} \in S_\mathcal{C}\}$$

Intuitively, all failure-observations are collected that lead to a state where a certain action is possible; these observations performed in the system automaton influence what is reachable in the control automaton. All possible different results of the set of observations are represented by a transition, with each observation in either F or A.

The states in the guarded control automaton are sets of states of the original control automaton. The target state of a transition $q \xrightarrow{[A|F]n} q'$ in the guarded control automaton is defined as the set of all states in the original control automaton that can be reached with the failures in F followed by a rule name n. The positive guard A of a transition contains the names of those rules that must be enabled, which are those rules that are not in F but are in the failure dependency of n in the source state. Given n, F and A, the exact set of reachable states in $\mathcal{C}$ can be determined, which becomes the target state of the guarded transition. In the result, each transitions that applies rule n will have a different F and A leading to a different set of target states.

A success state is a set of tuples consisting of states and failures. Success is a conditional property; we show in the product definition that a product state is a success state if a failure is satisfied by the system component in the product state. A state with the empty failure is an unconditional success state.

We state the following property:

Proposition 2.25. *Given a control automaton $\mathcal{C}$, the automaton $det(\mathcal{C})$ is a guarded control automaton.*

Proof. The proof that the created automaton satisfies the requirements (1), (2), and (3) of Def. 2.22 follows directly from the construction of $\rightarrow$ in Def. 2.24. Let $\mathcal{G} = det(\mathcal{C})$. For all $q_\mathcal{G} \in Q_\mathcal{G}$ with $q_\mathcal{G} \xrightarrow{[F_1|A_1]n}$ and $q_\mathcal{G} \xrightarrow{[F_2|A_2]n}$ it follows that:

1. $FD = fd(q_\mathcal{G}, n), F_i \subseteq FD, A_i = FD \setminus F$, which implies that $A_i \cap F_i = \emptyset$.
2. $FD = fd(q_\mathcal{G}, n)$ and $F_1 \cup A_1 = F_2 \cup A_2 = FD$.
3. There must be a transition $q_\mathcal{G} \xrightarrow{[F]n} q'_\mathcal{G}$. Let $F = fd(q, n)$, then $q'_\mathcal{G}$. The required transition's target state contains all possible failure paths to a state where n is allowed (the union of q' in Def. 2.23). Since $F = fd(q, n)$, A is empty. The constructed automaton must also be deterministic. Given

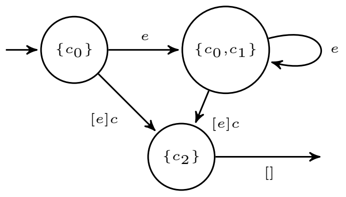


Figure 2.15: Example Extended Control Automaton

transitions $q_{\mathcal{G}} \xrightarrow{[F_1|A_1]n} q'_{\mathcal{G}}$ and $q_{\mathcal{G}} \xrightarrow{[F_2|A_2]n} q''_{\mathcal{G}}$, if $F_1 = F_2$, this implies that $A_1 = A_2$. The construction of the target state is deterministic given the source state qs (i.e. set of states from $\mathcal{C}$), n and F, since all reachable states are collected. The identity of constructed state is based on the collected set of states from $\mathcal{C}$. Thus, this implies that $q'_{\mathcal{G}} = q''_{\mathcal{G}}$.

□

Example 2.26. *Figure 2.15 shows the guarded control automaton for the control automaton of Figure 2.12. From the start state, an e transition is possible to $\{c_0, c_1\}$, which can be repeated as long as e is possible. From both $\{c_0\}$ and $\{c_0, c_1\}$ a c transition is possible to $\{c_2\}$ if e fails. The outgoing transition in $\{c_2\}$ represents the success condition. Since there is only the empty failure set, it is an unconditional success.*

For the definition of the product of system automata and guarded control automata, we introduce the function *enabled*, which returns a set of rules that are enabled in a given system state or a state reachable after an arbitrary sequence of λ's.

Definition 2.27 (enabled rules). *Given a system automaton $\mathcal{A}$, the function enabled : $Q_{\mathcal{A}} \to 2^{\mathit{Rule}}$ is defined as:*

$$\mathit{enabled}(q_{\mathcal{A}}) = \{n \in \mathit{Rule} \mid \exists i \in \mathit{Id} : q_{\mathcal{A}} \overset{(n,i)}{\Longrightarrow}_{\mathcal{A}}\}$$

The semantics of a guarded control automaton is given by the product with a system automaton. This results in another system automaton, where states are tuples of system states and guarded control states, defined as follows:

Definition 2.28 (guarded product). *Given a system automaton $\mathcal{A}$ and a guarded control automaton $\mathcal{G}$, the product $\mathcal{A}\times\mathcal{G}$ is a system automaton, with $Q \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{G}}$, $q_0 = (q_{0,\mathcal{A}}, q_{0,\mathcal{G}})$, and $\to$, S are defined by:*

$$\frac{q_{\mathcal{A}} \xrightarrow{(n,i)}_{\mathcal{A}} q'_{\mathcal{A}} \quad q_{\mathcal{G}} \xrightarrow{[F|A]n}_{\mathcal{G}} q'_{\mathcal{G}} \quad F \cap \mathit{enabled}(q_{\mathcal{A}}) = \emptyset \quad A \subseteq \mathit{enabled}(q_{\mathcal{A}})}{(q_{\mathcal{A}}, q_{\mathcal{G}}) \xrightarrow{(n,i)} (q'_{\mathcal{A}}, q'_{\mathcal{G}})}$$

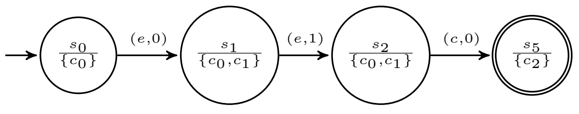


Figure 2.16: Example Guarded Product Automaton

$$\frac{q_{\mathcal{A}} \xrightarrow{\lambda}_{\mathcal{A}} q'_{\mathcal{A}}}{(q_{\mathcal{A}}, q_{\mathcal{G}}) \xrightarrow{\lambda} (q'_{\mathcal{A}}, q_{\mathcal{G}})} \qquad \frac{q_{\mathcal{A}} \in S_{\mathcal{A}} \quad (q_{\mathcal{G}}, F) \in S_{\mathcal{G}} \quad F \cap \mathit{enabled}(q_{\mathcal{A}}) = \emptyset}{(q_{\mathcal{A}}, q_{\mathcal{G}}) \in S}$$

A transition with rule n in $\mathcal{G}$ is paired with rule applications of n in the system automaton $\mathcal{A}$ when none of the rules in the negative guard F are applicable in $q_{\mathcal{A}}$, and all rules in the positive guard A are applicable in $q_{\mathcal{A}}$.

The success states are those states where $q_{\mathcal{A}}$ is a success state, and the failure it is combined with in S is satisfied in $q_{\mathcal{A}}$.

Example 2.29. *Figure 2.16 shows the product of the guarded control automaton of Figure 2.15 and the system automaton of Figure 2.8.*

As said before, the purpose of guarded control automata is to be able to produce deterministic controlled system automata. We state the following property:

Proposition 2.30. *Given a guarded control automaton $\mathcal{G}$ and a deterministic system automaton $\mathcal{A}$, $\mathcal{A} \times \mathcal{G}$ is deterministic.*

Proof. Given that $\mathcal{A}$ is deterministic, it contains no λ-transitions. Let $P = \mathcal{A} \times \mathcal{G}$. Then, by construction (Def. 2.28) P contains no λ-transitions either. For all $(q_{\mathcal{A}}, q_{\mathcal{G}}) \in Q_P$ with $(q_{\mathcal{A}}, q_{\mathcal{G}}) \xrightarrow{(n,i)} (q'_{\mathcal{A}}, q'_{\mathcal{G}})$ and $(q_{\mathcal{A}}, q_{\mathcal{G}}) \xrightarrow{(n,i} (q''_{\mathcal{A}}, q''_{\mathcal{G}})$ we need to show that $(q'_{\mathcal{A}}, q'_{\mathcal{G}}) = (q''_{\mathcal{A}}, q''_{\mathcal{G}})$. Since $\mathcal{A}$ is deterministic, we know that $q_{\mathcal{A}} \xrightarrow{(n,i)}_{\mathcal{A}} q'_{\mathcal{A}}$ and $q_{\mathcal{A}} \xrightarrow{(n,i)}_{\mathcal{A}} q''_{\mathcal{A}}$ implies that $q'_{\mathcal{A}} = q''_{\mathcal{A}}$. Also, for $q_{\mathcal{G}} \xrightarrow{[F_1|F_2]n}_{\mathcal{G}} q'_{\mathcal{G}}$ and $q_{\mathcal{G}} \xrightarrow{[F_1|F_2]n}_{\mathcal{G}} q''_{\mathcal{G}}$, Def. 2.22 implies that $q'_{\mathcal{G}} = q''_{\mathcal{G}}$ (it is deterministic). For $q_{\mathcal{G}} \xrightarrow{[F|A]n}_{\mathcal{G}}$, there is at most one combination of F, A where $F \cap \mathit{enabled}(q_{\mathcal{A}}) = \emptyset$ and $A \subseteq \mathit{enabled}(q_{\mathcal{A}})$, since a rule can only be enabled or disabled. Def. 2.28 implies that $(q'_{\mathcal{A}}, q'_{\mathcal{G}}) = (q''_{\mathcal{A}}, q''_{\mathcal{G}})$. □

2.5.1 Equivalence

We will now show that guarded control automata serve their intended purpose, namely that the product of a system automaton with a control automaton is

$$\begin{tikzcd}
\mathcal{C} \arrow[r] \arrow[d, "det"'] & \times \arrow[r] \arrow[d, leftrightarrow, "\mathcal{A}"] & \mathcal{A} \times \mathcal{C} \arrow[d, leftrightarrow, "\mathcal{L}\text{-equivalent}"] \\
det(\mathcal{C}) \arrow[r] & \times \arrow[r] & \mathcal{A} \times det(\mathcal{C})
\end{tikzcd}$$

Figure 2.17: Schematic Representation of the Equivalence in Theorem 2.31

essentially the same as its product with the determinised guarded control automaton. "Essentially the same" means that they have the same language in terms of (Rule $\times$ Id)-traces. This is depicted in Figure 2.17 and Theorem 2.31.

Theorem 2.31. *For all system automata $\mathcal{A}$ and control automata $\mathcal{C}$, $\mathcal{L}(\mathcal{A} \times \mathcal{C}) = \mathcal{L}(\mathcal{A} \times det(\mathcal{C}))$*

To prove this, we show inclusions in both directions, using two distinct notions of simulation.

Definition 2.32 (forward simulation)**.** *Given two system automata $\mathcal{A}_1, \mathcal{A}_2$, a relationship $\rho \subseteq Q_1 \times Q_2$ is called a* forward simulation *if:*

$$(q_{0_1}, q_{0_2}) \in \rho \tag{2.1}$$

and for all $(q_1, q_2) \in \rho$:

$$q_1 \xrightarrow{(n,i)}\xRightarrow{(n,i)} q_1' \Rightarrow \exists q_2 \xRightarrow{(n,i)} q_2' \wedge (q_1', q_2') \in \rho \tag{2.2}$$

$$q_1 \overset{\varepsilon}{\Rightarrow} q_1' \in S_1 \Rightarrow \exists q_2 \overset{\varepsilon}{\Rightarrow} q_2' \in S_2 \tag{2.3}$$

Proposition 2.33. *If there exists a forward simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{T}(\mathcal{A}_1) \subseteq \mathcal{T}(\mathcal{A}_2)$ and $\mathcal{T}^{\surd}(\mathcal{A}_1) \subseteq \mathcal{T}^{\surd}(\mathcal{A}_2)$.*

Proof. The proof is by induction over the length of the traces.

Hypothesis ($\mathcal{T}$) If there is a $q_{0_1} \overset{w}{\Longrightarrow} q_1$ then there is also a $q_{0_2} \overset{w}{\Longrightarrow} q_2$, and $(q_1, q_2) \in \rho$.

Basis. Let w be a trace in $\mathcal{T}(\mathcal{A}_1)$. If w is of length 0, it is also a trace of $\mathcal{A}_2$. It follows from 2.1 that $(q_1, q_2) \in \rho$.

Inductive Step. Let $w = w'(n,i)$ be a trace in $\mathcal{T}(\mathcal{A}_1)$. Then, for some q_1', $q_{0_1} \xRightarrow{w'} q_1' \xRightarrow{(n,i)} q_1''$. Since w' is shorter then w we can by induction assume that there is a $q_{0_2} \xRightarrow{w'} q_2'$ such that $(q_1', q_2') \in \rho$. From 2.2 it follows that there exists a $q_2' \xRightarrow{(n,i)} q_2''$. Thereby, w is a trace in $\mathcal{T}(\mathcal{A}_2)$, and $(q_1'', q_2'') \in \rho$.

We must do the same for the successful traces. Since $\mathcal{T}^{\surd}(\mathcal{A}_1) \subseteq \mathcal{T}(\mathcal{A}_2)$, we can take a shortcut here, by referring to the hypothesis above for all traces, which we already have proved to be true. For all traces $w \in \mathcal{T}^{\surd}(\mathcal{A}_1)$, it follows that $w \in \mathcal{T}(\mathcal{A}_1)$. Then there is a $q_{0,1} \xRightarrow{w} q_1$, $q_{0,2} \xRightarrow{w} q_2$, and $(q_1, q_2) \in \rho$. Also, because w is a successful trace, there exist a $q_1 \xRightarrow{\epsilon} q_1' \in S_1$. From Def. 2.3 it follows that there is a $q_2 \xRightarrow{\epsilon} q_2' \in S_2$, thus $w \in \mathcal{T}^{\surd}(\mathcal{A}_2)$. □

Proposition 2.34. *Let $\mathcal{A}$ be a system automaton and $\mathcal{C}$ a control automaton; then the relation defined by $\rho = \{((q_\mathcal{A}, q_\mathcal{C}), (q_\mathcal{A}, qs)) \mid q_\mathcal{C} \in qs\}$ is a forward simulation between $\mathcal{A} \times \mathcal{C}$ and $\mathcal{A} \times det(\mathcal{C})$.*

Proof. We now give the proofs that (2.1), (2.2) and (2.3) of Def. 2.32 hold for the proposed ρ. Let $\mathcal{A}_1 = \mathcal{A} \times \mathcal{C}, \mathcal{G} = det(\mathcal{C}), \mathcal{A}_2 = \mathcal{A} \times \mathcal{G}$.

(2.1) By construction (Def. 2.18 and Def. 2.28) $q_{0,1} = (q_{0,\mathcal{A}}, q_{0,\mathcal{C}})$ and $q_{0,2} = (q_{0,\mathcal{A}}, \{q_{0,\mathcal{C}}\})$; hence it follows that $(q_{0,1}, q_{0,2})$.

(2.2) Let $(q_\mathcal{A}, q_\mathcal{C})\rho(q_\mathcal{A}, q_\mathcal{G})$ and let there be a transition $(q_\mathcal{A}, q_\mathcal{C}) \xRightarrow{(n,i)}_1 (q_\mathcal{A}', q_\mathcal{C}')$. Def. 2.18 implies $q_\mathcal{C} \xRightarrow{F_1 \ldots F_n} \xrightarrow{n} q_\mathcal{C}'$, $F \cap enabled(q_\mathcal{A}) = \emptyset$, with $F = F_1 \cup \ldots \cup F_n$, and $q_\mathcal{A} \xRightarrow{\epsilon} \xrightarrow{(n,i)} q_\mathcal{A}'$. We choose a failure set $F' = fd(q_\mathcal{G}, n) \setminus enabled(q_\mathcal{A})$ so that F' can be synchronised with any visible action from $q_\mathcal{A}$. Since $q_\mathcal{C} \in q_\mathcal{G}$, Def. 2.22 implies $F \subseteq F'$, $q_\mathcal{G} \xrightarrow{[A|F]n} q_\mathcal{G}'$ with $A = F' \setminus F$, and $q_\mathcal{C}' \in q_\mathcal{G}'$. From Def. 2.28 it follows that $(q_\mathcal{A}, q_\mathcal{G}) \xRightarrow{\epsilon} \xrightarrow{(n,i)}_2 (q_\mathcal{A}', q_\mathcal{G}')$, and $(q_\mathcal{A}', q_\mathcal{C}')\rho(q_\mathcal{A}', q_\mathcal{G}')$ by construction of ρ.

(2.3) Let $(q_\mathcal{A}, q_\mathcal{C})\rho(q_\mathcal{A}, q_\mathcal{G})$, and $\exists (q_\mathcal{A}, q_\mathcal{C}) \xRightarrow{\epsilon}_1 (q_\mathcal{A}', q_\mathcal{C}') \in S_1$. Def. 2.18 implies $q_\mathcal{A} \in S_\mathcal{A}$, $\exists q_\mathcal{C} \xRightarrow{F_1 \ldots F_n} q_\mathcal{C}' \in S_\mathcal{C}, F = F_1 \cup \ldots \cup F_n$, and $F \cap enabled(s) = \emptyset$. Def. 2.24 implies $(q_\mathcal{G}, F) \in S_\mathcal{G}$. Finally, from Def. 2.28 follows $(q_\mathcal{A}, q_\mathcal{G}) \in S_2$.

□

The other direction is also covered by a simulation.

Definition 2.35 (reverse simulation)**.** *Given two system automata $\mathcal{A}_1$ and $\mathcal{A}_2$, $\rho \subseteq Q_2 \times 2^{Q_1}$, $R \neq \emptyset$ for all $(q, R) \in \rho$ is called a* reverse simulation *if:*

$$(q_{0,2}, \{q_{0,1}\}) \in \rho \tag{2.4}$$

and for all $(q_{\mathcal{A}}, R) \in \rho$*:*

$$q_2 \xRightarrow{(n,i)}_2 q_2' \Rightarrow \exists (q_2', R') \in \rho . \forall r' \in R' . \exists r \in R . r \xRightarrow{(n,i)}_1 r' \tag{2.5}$$
$$q_2 \overset{\epsilon}{\Rightarrow} q_2' \in S_2 \Rightarrow \exists q_1 \in R . q_1 \overset{\epsilon}{\Rightarrow} q_1' \in S_1 \tag{2.6}$$

Proposition 2.36. *If there exists a reverse simulation between* $\mathcal{A}_1$ *and* $\mathcal{A}_2$*, then* $\mathcal{T}(\mathcal{A}_2) \subseteq \mathcal{T}(\mathcal{A}_1)$ *and* $\mathcal{T}^{\surd}(\mathcal{A}_2) \subseteq \mathcal{T}^{\surd}(\mathcal{A}_1)$*.*

Proof. The proof is by induction over the length of the traces.

Hypothesis ($\mathcal{T}$**).** If there is $q_{0,2} \xRightarrow{w} q_2$ then there is a $(q_2, R) \in \rho$, and for all $q_1 \in R$ there is a $q_{0,1} \xRightarrow{w} q_1$.

Basis. Let w be a trace in $\mathcal{T}(\mathcal{A}_2)$. If w has length 0 it is in $\mathcal{T}(\mathcal{A}_1)$, and $(q_{0,2}, \{q_{0,1}\}) \in \rho$ by 2.4.

Inductive Step. Let $w = w'(n,i)$ be a trace in $\mathcal{T}(\mathcal{A}_2)$. Then, for some q_2', $q_{0_2} \xRightarrow{w'} q_2 \xRightarrow{(n,i)} q_2'$. Since w' is shorter than w we can by induction assume that there is R such that $(q_2, R) \in \rho$ and for all $q_1 \in R$ there is a $q_{0_1} \xRightarrow{w'} q_1'$. From Def. 2.5 follows that there exists an R' such that $(q_2', R') \in \rho$ and for all $q_1' \in R'$ there is a $q_1 \in R$ with $q_1 \xRightarrow{(n,i) q_1'}$. Thereby, w is a trace in $\mathcal{T}(\mathcal{A}_1)$.

For the successful traces we again take a shortcut by using the proof for all traces. Let $w \in \mathcal{T}^{\surd}(\mathcal{A}_2)$, then also $w \in \mathcal{T}(\mathcal{A}_2)$. Then there is a $(q_2, R) \in \rho$ and trace w to q_2 and to all $q_1 \in R$. Since $w \in \mathcal{T}^{\surd}(\mathcal{A}_2)$, there is a $q_2 \overset{\epsilon}{\Rightarrow} q_2' \in S_2$. From 2.6 it follows that there is a $q_1 \in R$ for which there is a $q_1 \overset{\epsilon}{\Rightarrow} \in S_1$. Thus, $w \in \mathcal{T}^{\surd}(\mathcal{A}_1)$. □

Proposition 2.37. *Let* $\mathcal{A}$ *be a system automaton and* $\mathcal{C}$ *a control automaton, and* $\mathcal{G} = det(\mathcal{C})$*, then the relation defined by* $\rho = \{((q_{\mathcal{A}}, q_{\mathcal{G}}), R) \mid \forall q_{\mathcal{C}} \in q_{\mathcal{G}} : (q_{\mathcal{A}}, q_{\mathcal{C}}) \in R\}$ *is a reverse simulation between* $\mathcal{A} \times \mathcal{C}$ *and* $\mathcal{A} \times det(\mathcal{C})$*.*

Proof. We now give the proofs that (2.4), (2.5) and (2.6) of Def. 2.35 hold for the proposed ρ. Let $\mathcal{A}_1 = \mathcal{A} \times \mathcal{C}, \mathcal{A}_2 = \mathcal{A} \times det(\mathcal{C}), \mathcal{G} = det(\mathcal{C})$.

(2.4) By construction (Def. 2.18 and 2.28) $q_{0,1} = (\mathcal{A}, \mathcal{C})$ and $q_{0,2} = (\mathcal{A}, \{\mathcal{C}\})$; hence it follows that $(q_{0,2}, \{q_{0,1}\}) \in \rho$.

(2.5) Let $(q_{\mathcal{A}}, q_{\mathcal{G}}) \rho R$ and let there be a transition $(q_{\mathcal{A}}, q_{\mathcal{G}} \xRightarrow{(n,i)}_2 (q_{\mathcal{A}}', q_{\mathcal{G}}')$. From Def. 2.28 it follows there is a $q_{\mathcal{A}} \overset{\epsilon}{\Rightarrow} \xRightarrow{(n,i)}_{\mathcal{A}} q_{\mathcal{A}}'$ (any λ-transitions are caused by synchronisation with λ-transitions in the system automaton) and

$q_{\mathcal{G}} \xrightarrow{[A|F]n} q'_{\mathcal{G}}$, such that $F \cap \mathit{enabled}(s) = \emptyset$, and $A \subseteq \mathit{enabled}(s)$. Def. 2.22 implies $q'_{\mathcal{G}} = \{q'_{\mathcal{C}} \mid \exists q_{\mathcal{C}} \in q_{\mathcal{G}} : q_{\mathcal{C}} \xRightarrow{F_1 \dots F_n} \xrightarrow{n} q'_{\mathcal{C}}$ with $F_1 \cup \dots \cup F_n \subseteq F\}$. Def. 2.18 implies $\forall q'_{\mathcal{C}}.\exists q_{\mathcal{C}} \in q_{\mathcal{G}}.(q_{\mathcal{A}}, q_{\mathcal{C}}) \xRightarrow{(n,i)}_1 (q'_{\mathcal{A}}, q'_{\mathcal{C}})$. This corresponds to the construction of R' in 2.5.

(2.6) Let $(q_{\mathcal{A}}, q_{\mathcal{G}})\rho R$ and $(q_{\mathcal{A}}, q_{\mathcal{G}}) \in S_2$. Def. 2.28 implies $q_{\mathcal{A}} \in S_{\mathcal{A}}, \exists F.(q_{\mathcal{G}}, F) \in S_{\mathcal{G}} : F \cap \mathit{enabled}(s) = \emptyset$. Def. 2.22 implies $\exists q_{\mathcal{C}} \in q_{\mathcal{G}} : q_{\mathcal{C}} \xRightarrow{F_1 \dots F_n}_{\mathcal{C}} q'_{\mathcal{C}} \in S_c, F_1 \cup \dots \cup F_N = F$. Finally, Def. 2.18 implies $(q_{\mathcal{A}}, q_{\mathcal{C}}) \in R \wedge (q_{\mathcal{A}}, q_{\mathcal{C}}) \overset{\epsilon}{\Rightarrow} (q'_{\mathcal{A}}, q'_{\mathcal{C}}) \in S_1$.

□

With the help of the above results, we can now prove the theorem.

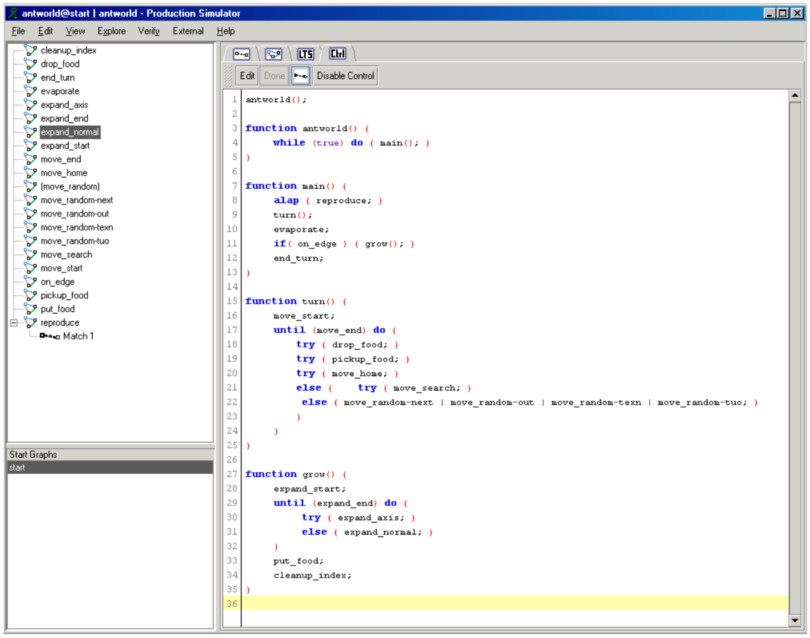
Theorem 2.31. From propositions 2.33 and 2.34 it follows that $\mathcal{T}(\mathcal{A} \times \mathcal{C}) \subseteq \mathcal{T}(\mathcal{A} \times \mathit{det}(\mathcal{C}))$ and $\mathcal{T}^{\surd}(\mathcal{A} \times \mathcal{C}) \subseteq \mathcal{T}^{\surd}(\mathcal{A} \times \mathit{det}(\mathcal{C}))$; from propositions 2.36 and 2.37 it follows that the inverse inclusions also hold. This implies the proof obligation. □

2.6 Implementation & Usage

The presented control language has been integrated into the GROOVE [Ren04] tool. A screenshot of the program editor view is shown in Figure 2.18. On top of the language constructs described in this chapter, it allows the specification of functions. During construction, a call to a function is replaced with a copy of the body of such a function.

By pressing a button, a dialog can be opened that displays the constructed (regular) control automaton; guarded control states are generated in a lazy manner (i.e., on demand) during simulation of a graph grammar. A screenshot of the automaton displaying view is shown Figure 2.19. It displays the constructed automaton for the program in Figure 2.18.

An unfortunate consequence of guarded control automata is the fact that there is an exponential blow-up in the possible number of states ($|Q_{\mathcal{G}}| = 2^{|Q_{\mathcal{C}}|}$) *and* transitions in applying the *det* function ($|{\rightarrow_{\mathcal{G}}}|$ is in the order of $2^{|\rightarrow_{\mathcal{C}}|}$) due to the adding of positive and negative guards. By using a lazy construction of $\mathcal{G}$ states during generation of the guarded system automaton, only those states are created that are (obviously) possible, but - more importantly - are in fact *used.* In a small test case that we have performed using a control program and a corresponding control automaton with 19 states and 9 failure transitions, only 17 distinct guarded control states were used during generation of the guarded system automaton (of



```
antworld();

function antworld() {
    while (true) do { main(); }
}

function main() {
    alap { reproduce; }
    turn();
    evaporate;
    if( on_edge ) { grow(); }
    end_turn;
}

function turn() {
    move_start;
    until (move_end) do {
        try { drop_food; }
        try { pickup_food; }
        try { move_home; }
        else {    try { move_search; }
         else { move_random-next | move_random-out | move_random-texn | move_random-tuo; }
        }
    }
}

function grow() {
    expand_start;
    until (expand_end) do {
         try { expand_axis; }
         else { expand_normal; }
    }
    put_food;
    cleanup_index;
}
```

Figure 2.18: A Screenshot of the Control Editor.

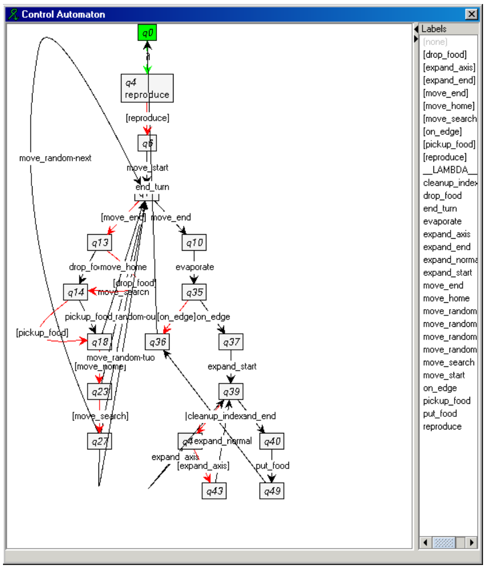


Figure 2.19: A Screenshot of the Automaton View.

a maximum possible count of 524288). In our experience, a single Rule is rarely used twice in a control program, limiting the number of guarded control states and the number of transitions a great deal.

By using the control language ourselves, we have established that it increases the ability to specify complex rule systems in a clear and intuitive manner. A good example is the "AntWorld" case study in the GraBaTs 2008 [RvG08] workshop, for which the behaviour of an ant colony has been specified consistent with a supplied description. Here, the use of a control program reduced the use of control information in the graph a great deal. Also a lot of negative application conditions could be left out. A summary and discussion of the specified solution can be found in [SR08]. The graph grammar can be downloaded from `http://alturl.com/896b`.

2.7 Conclusions

2.7.1 Related Work

There are two important areas of related work: on the one hand, other approaches to add control to rule-based systems, and on the other, results from process algebra.

Other approaches to control.

Although we have presented this work in the general context of rule-based systems, as far as we are aware most of the work on explicit control for such systems has been done for the special case of *graph transformation systems*; therefore, this is what we will focus on.

First let us remark that in this chapter we have concentrated on one particular method of controlling rule applicability. There are others, such as application conditions (e.g., [HHT96, HP05]), which have their own advantages and may very well be used in conjunction with control expressions. Where application conditions are typically suitable to specify constraints and conditions on the rule application based on the system state, the control expressions in the thesis are better suitable for simplifying rule dependencies and thereby reducing control information from the graphs. For example, there may be two or more rules to perform a specific action in different scenarios. Where the control program would schedule all of these rule in the right moment, application conditions can aid the rules in matching the right scenarios. Moreover, control can aid in matching these rules in a particular order to, which can increase the performance.

One of the first developments in the direction of using explicit control expressions for graph transformation was the PROGRES environment; see, e.g., [Sch90, SWZ99]. The aim here was to obtain a powerful and usable framework for programmed graph transformation, rather than study the theoretical properties of such a framework (although a translation into flow graphs was studied in [ZS92]). In the same vein, the control languages in tools like VIATRA2 [VB07] and VMTS [LLMC06] stress power and usability over theoretical properties. FUJaBA has the appealing graphical *storyboard* language for control [FNTZ00], but here the control is completely integrated with the rules and does not lend itself at all to an analysis like the one in this chapter.

The theory behind control conditions for rule-based systems has been studied intensively in the context of *graph transformation units* by Kreowski, Kuske and others; see, for instance, [KK96, KK99, Kus00]. However, they take a perspective that is quite different from ours, and indeed represents the input-output interpretation discussed in Section 2.4, insisting that "every description of a binary relation on graphs may be used as a control condition". Also in that interpretation, Schürr [Sch97] contains a systematic discussion of various operators and their meaning, and Habel and Plump in [HP01] show the minimality of the control language consisting only of choice, sequential composition and **alap**. In [PS04], a **while–do** is added to this language, to increase usability. We strongly believe that also the **try–else** operator of this chapter is useful in that regard.

In term rewriting, languages such as ELAN [BKKM02], Maude [CDE$^+$02], and Stratego [Vis01] allow defining transactions, strategies or programs over rules. These systems, typically used for program/model transformation, provide more control over the selection of rules and the order of normalization. Since the goal is the target model, these control directives have an input/output semantics, rather then a reactive semantics, such as presented in this chapter.

Finally, it is worth mentioning that the tools GrEAT [VNS$^+$06] and ATOM^3 [SV07] include facilities for rule scheduling based on *data flow* rather than control flow.

Failures in process algebra.

A weak link exists to the concept of failure semantics in process algebra, as developed in [BHR84], leading to the formalism of CSP [Hoa85]; and a slightly stronger one to the refusal testing semantics of [Phi86], where failures can be interspersed with ordinary actions. The failures in those papers, however, are solely used as *observations* of the execution capabilities of a process, never to *control* the process. Thus, despite the superficial similarities, the models and their purpose are quite different from the research reported in this chapter.

2.8 Future Work

By allowing named procedures in our control language, it is possible to simplify the product automaton to abstract from the details of such procedures, by replacing the application of the procedure with a single transition. Intuitively, control can be used to simplify rules that model complex processes, i.e. processes that can not be expressed by a single rule. By allowing this atomicity, we can visualise the process as a single action.

These same procedures can be used for specifying transactions. When a procedure can not be exited successfully, one could require to roll back the preceding rule application. Intuitively: the process spread over a group of rules, could not be finished completely. Would one consider the process being a single rule, it would not have been applied at all.

While we solved the problem of controlling the order of rules, we could extend this work with parameterized rule applications. A rule application would bind certain nodes to variables, where these bound variables can be used in the application other rules. This allows control of *where* the rule is applied (in case of more then one option). For example, one might like to do an operation on all elements of a linked list. Assuming the operation on a single element can be specified with a single rewrite rule, one could require the operation to be finished before applying any other rules by using the **alap** keyword. The structure of the linked list however, calls for coding the next element of the list to do the operation on; this avoids the complex task of finding the first element the operation has not yet been applied on (which might not even be possible). With parameters, the next element can be matched and bound to a variable, to be used as the current element in the next rule application. This avoid the use of control information in the states (and rules) even more. In fact, this extension is currently being formalised and implemented by a Master student.

2.8.1 Contributions

In this chapter, we have presented a control language for the specification of control expressions for arbitrary rule systems. It provides an intuitive way for the user to specify control expressions.

We have also defined an automaton formalism for controlling rule applications in rule-based systems. We have introduced the notion of failure as the non-applicability of a set of rules in a certain state, and used these failures as an element of control in our automata.

The semantics of the language is described as the control automata that represent programs written in the language. We have defined construction operations from language construct to control automata.

The resulting behaviour is defined as the product with a system automaton, an automaton representation of the uncontrolled rule system. The result is a reactive semantics for control expressions.

The control language and automata and product only require the existence of a rule system having rules with certain names, and the representation of a system as an automaton. Therefore, the control expressions are specified without reference to any particular rule system. The proposed approach can therefore be used by any rule base system that uses named rules.

We have explained how the defined product operation can introduce spurious non-determinism and how this can be harmful. To solve this issue, we have introduced a formalism for guarded control automata and the corresponding product operation, that — when applied to a deterministic system automaton — results in a deterministic controlled system automaton. We have proved that the languages of products using a normal and the corresponding guarded control automaton coincide.

We have implemented (a superset of) the control language and semantics presented here as an extension to graph transformations in the GROOVE tool. In future work, we foresee extensions the described control expressions with features such as atomic procedures, transactions and rule parameters.

Chapter 3

A Graph-Based Execution Semantics for Composition Filters

3.1 Introduction

In this chapter we present an execution semantics for the Composition Filters model. The Composition Filters model is an enhancement of the object-oriented model, that enables AOP by intercepting and manipulating messages between objects. The CF model has evolved from the first (published) version of the Sina language in the late 1980s [AT88, ABV92], to a version that supports language independent composition of crosscutting concerns [BA04, SEG09]. Despite the rich history of Composition Filters, the execution semantics has never been defined in a formal language. The specification language of our choice is graph transformations. In Section 1.3 we have explained the advantages of graph transformations over other formal specification techniques using textual notations.

This chapter is structured as follows. In the next section, we give an in-depth explanation of the Composition Filters Model. In Section 3.3, we elaborate our goal and illustrate the approach used. In Section 3.4 we discuss the graph-based representation of Composition Filters programs. In Section 3.5 we give these programs a control flow semantics, followed by the execution graphs in Section 3.6 and the run-time semantics in Section 3.7. In Section 3.8 we discuss certain properties of our approach, such as correctness, its usefulness, understandability, and extensibility. Finally, in Section 3.9, we compare our work to related work, discuss future work, and we elaborate on the contributions of the semantics presented in this chapter.

3.2 Introduction to Composition Filters

In this section, we discuss the aspect-oriented principles of the Composition Filters (CF) model. We start by giving a conceptual illustration of the Composition Filters model. Then, we explain the CF language and the model we use throughout this chapter to represent CF programs.

3.2.1 The Composition Filters Model

The Composition Filters model is a modular extension of the conventional object-based model. In object-oriented systems, objects send messages to each other, e.g. in the form of method calls or events. The Composition Filters model extends the concept of message sending with message interception and manipulation. This allows expressing many behavioural manipulations of object-based systems, since all visible behaviour of an object is triggered and manifested by the messages it sends and receives. Figure 3.1 illustrates the extension, which is realised by enhancing a traditional object — the so-called implementation object — with an outer layer of *filters*. The enhanced object is referred to as a *concern instance*. Traditional objects still exist as concern instances without filters. All incoming and outgoing messages must pass the filters of a concern instance. The filters define enhancements to the behaviour of objects. Each filter is intended to perform a specific manipulation to certain incoming and outgoing messages. Incoming messages pass through input filters, while outgoing messages pass through output filters. The filters are grouped into so-called filter modules, which are meant to group filters with a collaborative responsibility and allow reuse of this functionality. A filter module is in fact the unit of instantiation that is added to the traditional objects.

Filter modules also provide an execution context for the filters. This context consists of named variables referring to instances of other concerns. The CF model distinguishes two different kinds of variables in the execution context: *internals* and *externals*. Internals are objects that are local to the filter module instance. Externals are objects that represent shared state and can be shared between filter modules and the base program.

The filters express which messages are accepted and rejected by means of filter expression. A filter expression is a simple, declarative expression to match and modify messages. The filter expression may contain certain conditions on the state

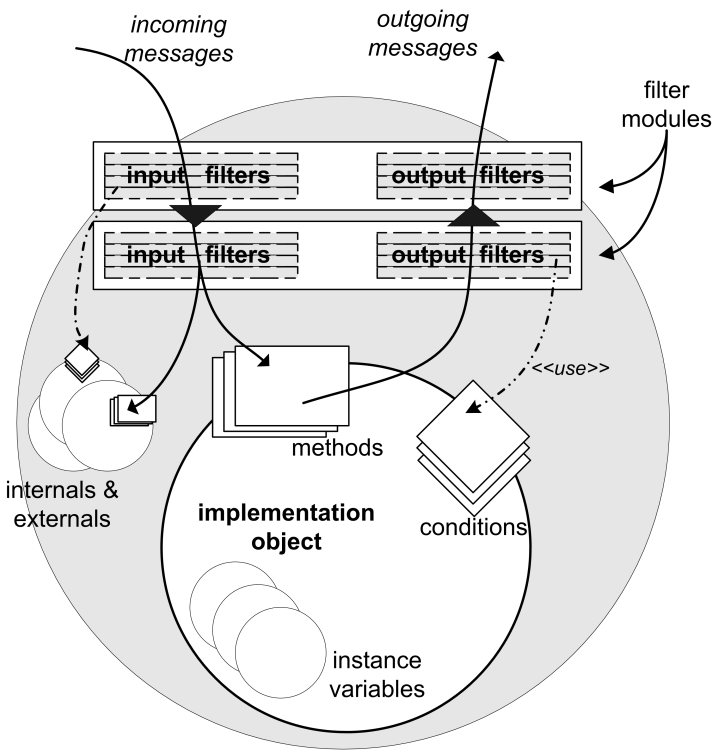


Figure 3.1: Simplified representation of a concern instance with filters.

and structure of the system; this state is reflected by the message, the implementation object and the execution context. Although the implementation object may be defined in any object-based language, the Composition Filters language is a language-independent extension to the underlying object-oriented language. To be able to specify expressions over this state, the following abstraction mechanisms are used:

- The run-time state is captured using conditions. These named variables refer to boolean methods (represented by the diamond shapes in Figure 3.1) in the implementation object or the execution context, which are expected to be side-effect free. Conditions are evaluated right before a message enters the filtering mechanism. A filter expression may refer to these conditions.

- Regular methods (represented by the rectangles in Figure 3.1) are identified by their name; they implement the functional behaviour of the object. A signature match — which can be specified in a filter expression — can test whether a message can be accepted by any of the methods in an objects interface.

- The name of the incoming message — reflecting the name of the called method — may be matched against a certain static value using a so-called name match. The name of a message is called a *selector*.

Each filter can either **accept** or **reject** a filtered message based on the filter expression. The semantics associated with acceptance or rejection depends on the type of the filter. These types define the execution of certain actions. Examples of predefined filter types and their semantics are:

- Dispatch: if the message is accepted, it is dispatched to a specified target object; if the message is rejected, it continues to the subsequent filter [ABV92].
- Substitute: filters with this type can substitute certain properties of messages explicitly on acceptance of the message, for instance the target object or the selector of the message. A rejected message continues to the subsequent filter [BA04].
- Error: if the filter rejects the message, it raises an exception; otherwise the message continues to the subsequent filter [ABV92].
- Meta: if the message is accepted, the message is reïfied and sent as a parameter of a new message to an internal or external; otherwise the message continues to the next filter. The object that receives the message can observe and manipulate the reïfied message and reactivate its execution [AWB+93].

Typically, messages travel sequentially along the filters until they are dispatched. Dispatching here means either to start the execution of a local method or to delegate the message to another object.

Composition Filters is among the AOP languages where aspects are added to the objects modularly (also called black-box AOP approach). Languages that introduce aspects as proxies/interceptors can also be put into this category, such as SpringAOP [JHA+]. Such approaches may have the following advantages:

1. The AOP extension *can* be designed in a base-language-independent way;
2. Advice respects the encapsulation of modules; aspects only affect incoming and outgoing calls;
3. Interface-based composition, due to encapsulation and well-defined method interfaces.

```
 1 concern JukeboxCredits {
 2    filtermodule TakeCredits {
 3       externals
 4          credits : jukebox.Credits =
 5                      jukebox.Credits.instance();
 6       conditions
 7          hasCredits : credits.hasCredits();
 8       inputfilters
 9          check : Error = { hasCredits => [*.*],
10                                  True ~> [*.play] };
11          withdraw : Meta = { True => [*.play]
12                                      credits.withdraw }
13    }
14
15    superimposition {
16       selectors
17          selection = { Class | isClassWithName(Class,
18                                  'jukebox.Jukebox') };
19       filtermodules
20          selection <- TakeCredits;
21    }
22 }
```

Listing 3.1: An example concern specification in Composition Filters

3.2.2 The Composition Filters Language

This section explains the syntax of Composition Filters. For illustration, we refer to an example of a Composition Filters specification, which is shown in Listing 3.1. It assumes there is an implementation of a jukebox.Jukebox class, which has a play method for playing songs. The specification in the example adds functionality to refuse playing of songs based on the number of credits, and withdraws one credit for each played song.

A **concern** is the main unit of modularization to specify crosscutting concerns in Composition Filters. Listing 3.1 starts with the concern specification of Jukebox-Credits. We first briefly discuss the example concern. Then, we discuss it in more detail.

Example 3.1. *Line 9 of Listing 3.1 shows the declaration of an input filter* check*, which has the filter type* Error *and on line 11 a filter* withdraw *is declared with filter type* Meta*. The ";" on the end of line 10 represents the sequential composition of the two filters.*

The check *filter consists of two filter elements:* hasCredits => [*.*] *and* True ~ >

[.play]. The "," in between represents the conditional OR.*

The first filter element consists of a condition expression hasCredits, a name match denoted by the square brackets [.*] and the default substitution part *.* (since no substitution part is given). The => is called a condition operator and means that no negation is used in the matching expression.*

The second filter element consists of a condition expression True, a name matching part [.play] — which matches all filtered messages with selector play — and again the default substitution part. Here, the $\sim >$ means that the matching expression is negated. Thus, the second filter element matches any message that does* not *have a selector play.*

The withdraw filter consists of a single filter element with a condition expression true, no negation, a name matching part [.play] and a substitution part credits.withdraw, which can be broken down to the target credits and a selector draw.*

The example does not have a signature match. An example signature match <credits> would test if the incoming message could be accepted by the declared external credits.

A concern consists of a set of **filter modules** (line 2-13), and a **superimposition** (line 15-21) specification. The filter module specifies *which* messages are intercepted and *how* they are manipulated, whereas superimposition specifies *where* the filter modules are applied.

Superimposition provides the mechanism to specify crosscutting concerns in a modular way; it allows to "export" behaviour to other concerns. A superimposition specification selects a set of relevant classes in the base program and superimposes filter modules on each class in this set. The superimposition clause consists of the following elements:

- A set of **selector declarations**, which refer to elements in the base program. These selectors use Prolog queries on the static structure of the base program. Line 16 in Listing 3.1 shows an example selector declaration. The selector selection uses a logic variable Class and a predicate isClassWithName in its declaration. This predicate will select the class jukebox.Jukebox (by using its fully qualified name). Note that the variable Class is a logic variable that may refer to not only one, but multiple program elements as well.

- **Filter module binding** specifications declare superimposition of a given filter module on the set of classes designated by a selector. This means that a new instance of the superimposed filter module is associated with each new instance of the designated classes, and each incoming and outgoing message is filtered by the superimposed filter module instances. Line 19 in

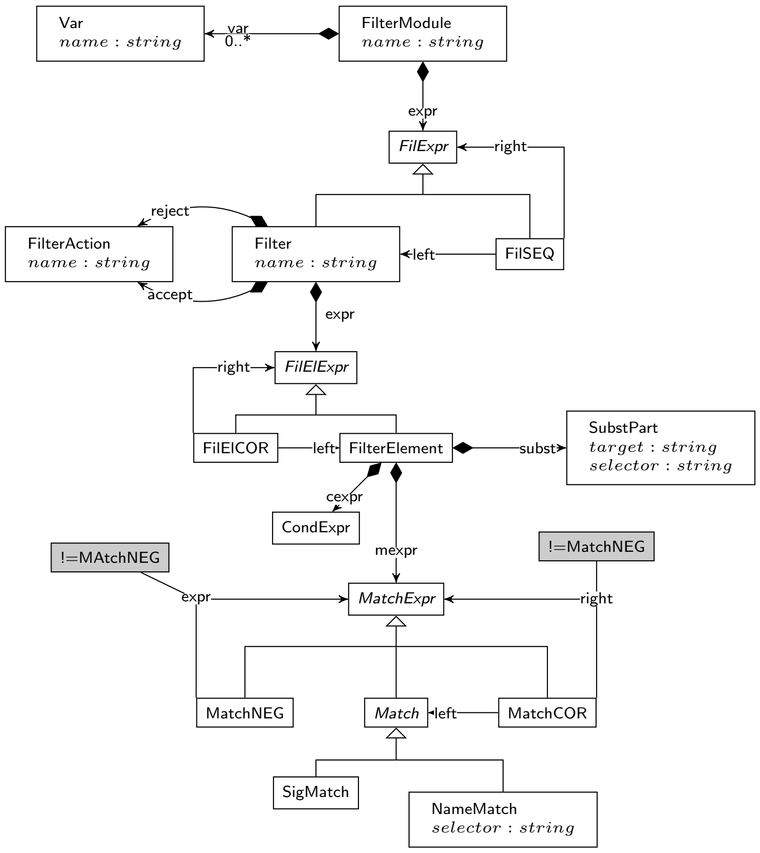


Figure 3.2: Abstract Syntax Tree of a Filter Module.

Listing 3.1 shows an example of the filter module binding specification: the filtermodule TakeCredits is superimposed on the selection selector, i.e. on class jukebox.Jukebox.

Filter modules are the units of reuse and instantiation of crosscutting behaviour. Lines 2 - 13 of Listing 3.1 present a filter module specification. As said, the filter module declares an execution context for the filters, consisting of internals, externals, and conditions:

- **Internal** and **external** declarations provide an execution context for the filters. Internals are objects pertaining to the filter module, while externals are references to instances created outside the filter module and concern, which are used for representing shared state. Line 4 of Listing 3.1 shows the declaration of an external credits using its fully qualified name — namely jukebox.Credits — following by the instantiation specification. Internals are declared in a similar way, and are preceeded by the "internals" keyword.

- **Condition** declarations declare a condition by its name and map it to a method. This method must implement a side-effect free Boolean expression and must be in the execution context of the filter module, i.e. in the implementation object or any of the internals or externals. Typically, conditions provide run-time information about the state of an object. Line 7 of Listing 3.1 shows the condition declaration hasCredits that is mapped to the result of the hasCredits() method of the external credits. Conditions are only evaluated once during the filtering of a message, namely when a message enters the filter module. This means that, even though the behaviour of the filter module can affect the result of a Boolean method referenced by a condition, this is not reflected by the value of that condition.

A **filter module** specification also consists of a set of **filter declarations**. Inputfilters manipulate incoming messages received by the implementation object, whereas outputfilters manipulate outgoing messages sent by the implementation object. Therefore, the Composition Filters model only deals with two kinds of join points: *message reception* and *method call.* For simplicity, we only handle inputfilters in this work. Outputfilters can however be dealt with in the same way.

We explain the structure of the filter declarations in more detail using the syntax model used in this work. This is shown in Figure 3.2. The italic labelled elements are abstract types. The top-level element of the model is the **filter module** (FilterModule), with a fully qualified name. The filter module consists of a number of named Var nodes, which represent the internals and externals. Conditions are not represented in the model; later in this chapter we explain how these are represented instead. A filter module has a filter expression (FilExpr), which can either be a filter (Filter) or a sequential composition operator (FilSEQ), with on the left side a Filter, and on the right side another FilExpr.

A **filter** has a name and a type. The type of the filters in the syntax is represented in Figure 3.2 by an *accept* and a *reject* FilterAction. A filter also contains a filter element expression (FilElExpr), which can be either a filter element (FilterElement), or a *conditional OR* (FilElCOR) with a FilterElement on the left and a FilElExpr on the right.

A **filter element** consists of a condition expression (CondExpr), a matching expression (MatchExpr) and a substitution part (SubstPart).

The **condition expression** is a boolean expression that may use the boolean values True and False, declared conditions in filter module, and condition operators

&& (*AND*), || (*OR*), and ! (*NOT*). The default boolean literal True is used when no condition expression is given.

A **substitution part** is an argument for the filter action and consists of a selector and a target part. The selector may for example be used to replace the selector of the filtered message. The target must be the name of one of the declared internals or externals, or one of the special keywords *inner* (the implementation object), or *sender*. The default value *.* is used when no substitution part is specified, and evaluates to the filtered message's current target and selector.

The **matching expression** can be a *negation* (MatchNEG) of a MatchExpr, a match (Match), or a *conditional OR* (MatchCOR) with a left-hand-side Match and a right-hand-side MatchExpr. A negation cannot be nested, i.e. only the entire top-level expression can be negated. A negation of a conditional OR yields true if and only if all nested matches yield false.

A **match** specification can be either a name match (NameMatch) or a a signature match (SigMatch). The name match compares the specified selector with the selector of the filtered message. A signature match, specified with a target object, tests whether the signature of the filtered message is accepted by a given object (i.e. if the called signature of the message is part of the interface of the class of the specified object). We do not represent the target argument in our model. Instead, we evaluate signature matches differently, which we will show in Section 3.7.

3.2.3 Meta Filters

The Meta-filter type has a special semantics among the predefined filter types: it reïfies the message that is matched by its filter specification, and sends it as an argument in a method call on an instance of an ACT (*AdviCe Type*) class. An ACT class is a regular object-oriented class with methods that take a single argument of the type *ReifiedMessage*. For instance, lines 11-12 in Listing 3.1 show an example of the Meta filter specification. If the incoming message is *play*, the filter *withdraw* will reïfy the message and pass it as an argument in the method call *withdraw* on the external credits. Execution continues at this method, which is shown in Listing 3.2. Calling *resume()* resumes the execution of the filters. Also note that the example also shows the implementation used for evaluating the hasCredits condition, which is a regular boolean method.

An ACT class can inspect and manipulate the message, and reactivate it in various ways [Sta05], namely:

- The **proceed()** method blocks the ACT and continues the evaluation of the filters. After dispatching the message to and returning from the message's

```
class Credits {
   private int credits = 0;
   ..

   public void withdraw(ReifiedMessage m) {
      this.credits--;
      m.resume();
   }

   public boolean hasCredits() {
      return (this.credits > 0);
   }

}
```

Listing 3.2: An example ACT class.

target the ACT resumes execution. The mechanics are similar to proceed as it is know from AspectJ [KHH+01].

- The **resume()** method continues the execution of the filters and continues the execution of the ACT in parallel.

- The **return(..)** will return to the caller — optionally with the given argument — immediately and will cause the evaluation of any other filters to be skipped.

- The **respond()** method will return a *null* value to the caller causing the caller to continue, and continues message evaluation in parallel. A return value resulting from dispatching the message will not reach the caller.

- The **send(Object target)** method will send a copy of the message to the given target. This will block the evaluation of the original message unless **respond()** has been called first.

3.3 The Goal and the Approach

In this section we elaborate on our goal and explain the approach used for achieving this goal.

3.3.1 Goal

Composition Filters allows for modular verification of crosscutting concerns. However, the combined behaviour of the base system and the aspect may not be trivial, as we explained in Section 1.2. Therefore, we want to be able to analyse and verify Composition Filters specifications. We choose to do that by defining a formal semantics of run-time behaviour of the language. The goal of this semantics is to use it for verification of Composition Filters specifications. This semantics must satisfy the following requirements:

- We want to be able to simulate Composition Filters specifications using the semantics. In other words, we need the semantics to be *executable*.
- To maximise the capability offered by the semantics for analysing properties of the Composition Filters specification, we want it to have a *reactive* nature, i.e. we want to be able to see all steps during the execution of the system.
- We want the semantics to offer an abstraction of the base system. This not only allows verification of larger systems, but may also allow proving certain properties for a Composition Filter specification composed with any base system that satisfies certain properties. Besides that, we do not want to specify the semantics of the base-language as well. In essence, it must be able to simulate Composition Filters specification in a *modular* way.
- We want the semantics to be *extensible*, to be able to incorporate future extensions of the language or user-defined filter-types. Ideally, it must therefore also be *understandable*.

3.3.2 Approach

We choose to specify a small-step execution semantics of Composition Filters. We use graph transformations as the language for specification of the semantics. We feel that graph transformations can model programs in a natural way (i.e. by representing classes and objects as nodes, and relationships as edges), thereby coming closer to our need for the semantics to be understandable as well as extensible.

The approach is visualised in more detail in Figure 3.3. Beginning from a program's source, specified in the Composition Filters language, the program is parsed and subjected to static analysis, resulting in an *abstract syntax tree* (AST). The AST — or a specific part of it — is transformed into a graph representation of the syntax, a so-called abstract syntax graph, corresponding to the model specified in Figure 3.2.

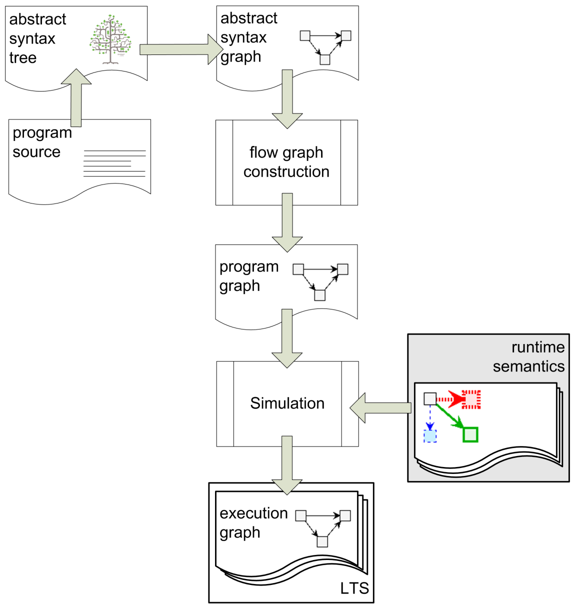


Figure 3.3: Overview of the approach.

The semantics of the language is specified using graph transformations. This is divided into two steps: a control flow semantics and a run-time semantics. Applying the control flow semantics to the abstract syntax graph creates explicit control flow information, resulting in the so-called program graph. The program graph and the run-time semantics combined can be used to simulate the execution of the program represented by the graph.

Depending on the part of the system that is represented (i.e., the classes and superimposed filter modules) the semantics can be used to simulate messages sent to instances of these classes. In Chapter 5 we show a use of the semantics where each generated graph represents one class and the superimposed filter modules.

The result of simulation is a state space represented as a labelled transition system (LTS), where states are represented as so-called execution graphs, and transitions are rule applications. Execution graphs are program graphs with additional information representing objects and processes (i.e., heap and stack). This reactive view on the execution of the system can be used for various kinds of analysis and verification.

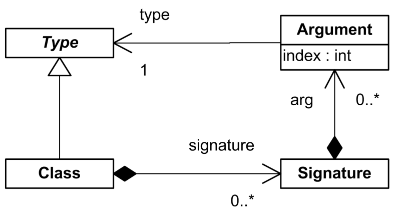


Figure 3.4: Type-Graph of the Base System Abstract Syntax Graph

3.4 Abstract Syntax Graphs

The generated abstract syntax graphs represent a number of classes and the filter modules superimposed on these classes. The type-graph of the filter module representation is shown in Figure 3.2.

The base syntax that is represented in the graphs consists of types and signatures; the type-graph is shown in Figure 3.4. Although concrete types are never labelled with Type, we use this label in the type-graphs in this section. Every type has a name, which we represent with string attribute values (see Section 2.2 for more information on attributed graphs). A class is a specialisation of a type. Classes are represented as Class-labelled nodes. Primitive types — although not supported natively — can also be represented using Class nodes. Each class is connected to a number of Signature nodes using signature edges; these represent the signatures of the methods that can be invoked on instances of the class. A Signature is connected to its arguments — Argument-labelled nodes with a type and an index — using argument edges. Identical signatures are represented by a single node; this allows for signature comparison by only looking at the node identity of the signature, without having to compare both sub-elements.

Superimposition is represented by filtermodule edges connecting FilterModule nodes to the classes that they have been superimposed upon. In other words, we represent superimposition after it has been resolved, which is done during compile-time by an existing compiler implementation. A signature, which is used to identify methods, is represented by a Signature-labelled node. A signature has a name — specified with a string attribute — that represents the name of the invoked method. A signature also has *arg* edges to Argument nodes. These nodes, which represent the formal arguments in the method signature, have a type edge to a Type node, and an index edge to an integer attribute value.

Figure 3.5 shows the abstract syntax graph that represents the example. It contains the syntax of the TakeCredits filter module, and the Jukebox class. The two are connected by a filtermodule edge. The class has a single Signature: the play method that has one Argument of type Song. To represent the argument type, the

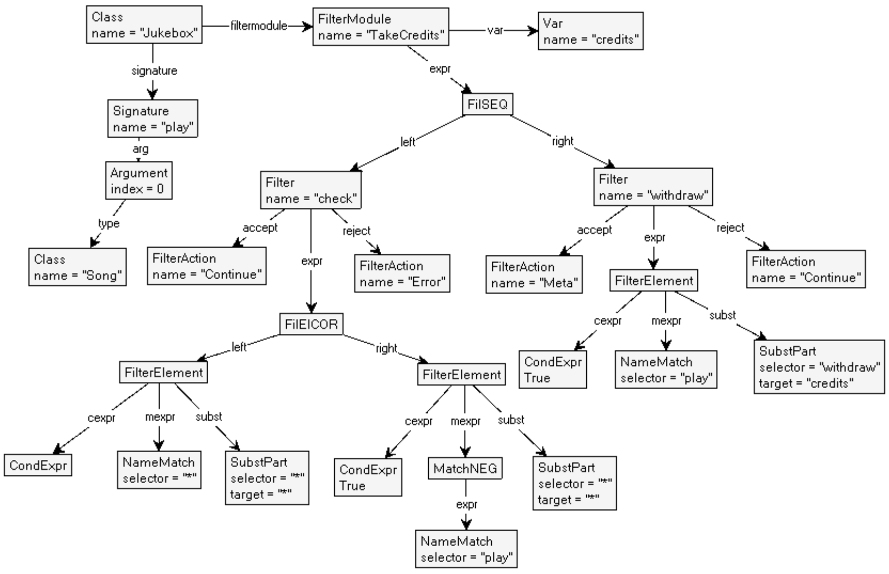


Figure 3.5: Abstract Syntax Graph of the Example.

Song class is also represented in the graph.

3.5 Control Flow Semantics

In this section, we present the control flow semantics of the Composition Filters language. With this semantics, we can add explicit control flow information to a syntax graph. This makes specification of the run-time semantics easier, because in each execution step, control can be passed forward in a uniform way. In this section, we first explain the details of the generated control flow graph, then we discuss the control flow specification language, and then we give the action control flow semantics.

3.5.1 Control Flow Construction

The control flow construction process is shown in Figure 3.6. The control flow semantics of syntax elements is expressed as graphs in a dedicated language for this purpose, defined in [SRK06]. Using another set of rules — the control flow meta rules — these graphs are transformed to rules themselves. The generated rules — control flow construction rules — together given to the simulator will add explicit control flow to an abstract syntax graph, resulting in a program graph.

Figure 3.7 shows the type-graph of the control flow information. Every program element the program counter may point to is recognised by a FlowElement node. These nodes are in fact syntax element (i.e. the elements of Figure 3.2) or auxiliary control flow nodes. A FlowElement node can have a `flow` edge to the next

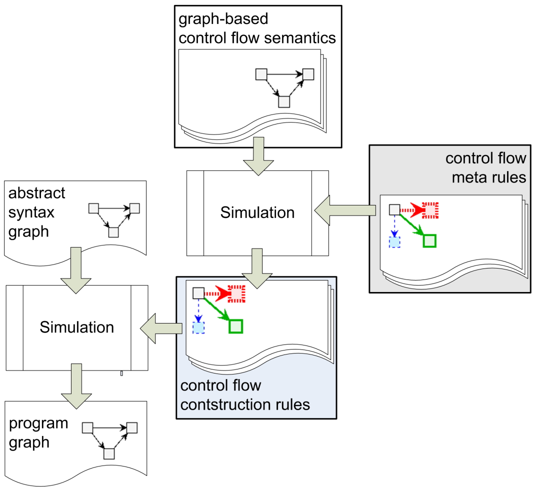


Figure 3.6: Overview of the control flow construction approach.

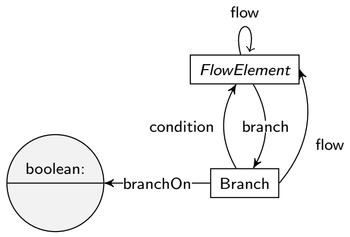


Figure 3.7: Type-graph of Explicit Control Flow

FlowElement, or specify conditional control flow. The latter is represented by connecting the FlowElement to two `Branch` nodes, connected by `branch` edges. The `branchOn` edge of the `Branch` node points to the required value of the branch. The `condition` edge points to the condition of the branch, which is another FlowElement, which must be a boolean expression. The outgoing `flow` edge of a `Branch` indicates the control flow when the value of the condition is equal to the value of the branch, as indicated by the `branchOn` edge.

3.5.2 The Control Flow Specification Language

The control flow specification language is a graph based language. A specification of the control flow of a syntax element starts with the graph representation of the static structure of that element, i.e. the element itself and its children. The element for which the graph specifies the control flow is tagged with a self-label `KeyElement`. An outgoing `exit` edge to an unlabelled node represents where flow goes when the element is finished. From the `KeyElement`, an `entry` edge specifies where the flow starts. The entry can be the key element itself — represented by a self-edge — or one of its children. Whenever — in the specification of another element — a flow edge is specified to the specified key element, in fact a flow edge is created to the entry of the key element. The actual control flow is specified either by `flow` edges between the syntax elements, or by `Branch` nodes, similar to the generated control flow graph explained above. For the clarity of the specifications in this thesis, we use an enhanced graphical denotation. An example is shown in Figure 3.8. Here, the black elements are all part of the abstract syntax graph. The *key element* (also labelled KeyElement) is denoted with a thick border. It has three child elements named Entry, Option1, and Option2.

The control flow is denoted using gray (blue) thick arrows. The flow entry of the key element (also labelled Entry) has an incoming flow edge that has no source node. It then goes to the KeyElement and to an auxiliary Flow node, also denoted using thick gray borders. These are used to take care of the flow behaviour of

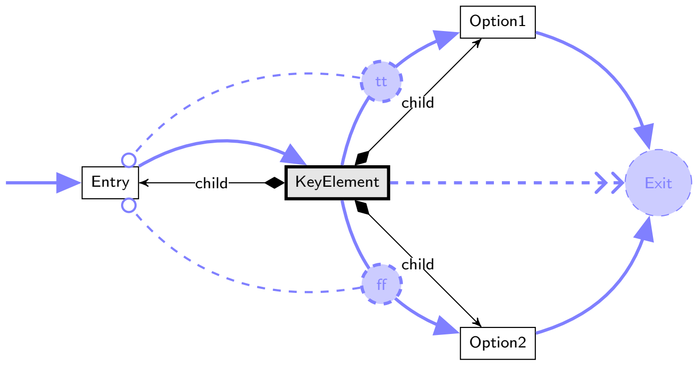


Figure 3.8: Control Flow Definition Example.

a syntax node that also has some other behaviour. Conditional flow is denoted with a dashed circle on the arrow. The circle is labelled with either tt or ff and is connected to its condition with a dashed arrow. These in fact correspond to *Branch* nodes with a *condition* and a branchOn edge. Flow ends at an Exit node connected to the KeyElement.

As mentioned, the control flow specifications are transformed into rules that add explicit control flow to an abstract syntax graph. For that purpose, first, each syntax element that is part of the control flow is labelled with a FlowElement self-edge. Then, the FilterModule node is given a KeyElement self-edge an exit edge to a new auxiliary Exitnode. Thereby, control flow generation starts at the FilterModule. Each control flow construction rule passes the Exit node and the KeyElement edge down to its children, until the control flow construction phase is finished.

3.5.3 Control Flow Semantics

We now enumerate the specifications of the language elements of Composition Filters. The following points are noteworthy:

- For the FilterModule (Figure 3.9), control enters at the FilExpr and from there flows to the exit;
- For the FilSEQ (Figure 3.10), control enters at the left Filter, then flows to the right FilExpr, and then to the exit;
- For a Filter (Figure 3.11), control enters at the FilElExpr, from there flows to the Filter itself, where it branches based on the evaluation result of the expr

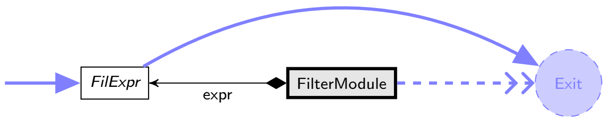


Figure 3.9: Control Flow Specification of a FilterModule.

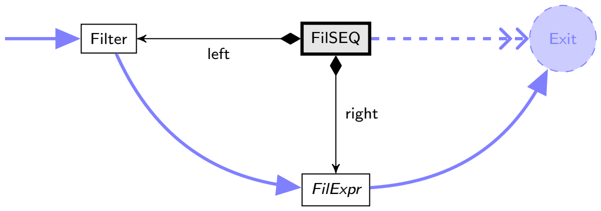


Figure 3.10: Control Flow Specification of a FilSEQ.

to either the accept (on *true*) or reject FilterAction (on *false*). From both these actions control flows to the exit;

- For a FilElCOR (Figure 3.12), flow enters at the left FilterElement, from there flows to an auxiliary Flow node, where flow branches based on the evaluation result of the filter element, to either the FilElCOR (on *true*), or via the right FilElExpr to the FilElCOR (on *false*). Finally control flows to the exit;

- For a FilterElement (Figure 3.13), control enters at the CondExpr, from there it flows via an auxiliary Flow and — on *true* — optionally via the MatchExpr to the FilterElement. From there, control flows to another auxiliary Flow node, from where optionally (on *true*) include the SubstPart is included, before it flows to the exit;

- For a MatchNEG (Figure 3.16), control enters at the MatchExpr, then flows to the MatchNEG, and from there to the exit;

- For a MatchCOR (Figure 3.17), control enters at the left Match, then flows to an auxiliary Flow node from where optionally the right MatchExpr is included, before it flows to the MatchCOR. From there, control flows to the exit;

- For the remaining elements, (the SubstPart in Figure 3.14, the CondExpr in (Figure 3.15, the NameMatch in Figure 3.18, and the SigMatch in Figure 3.19), control enters at the element itself and then flows to the exit;

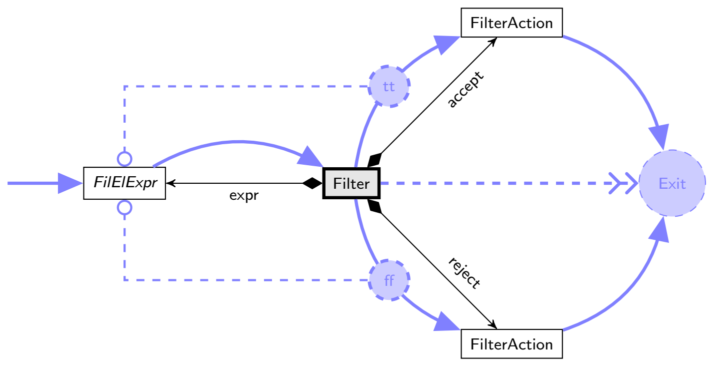


Figure 3.11: Control Flow Specification for a Filter.

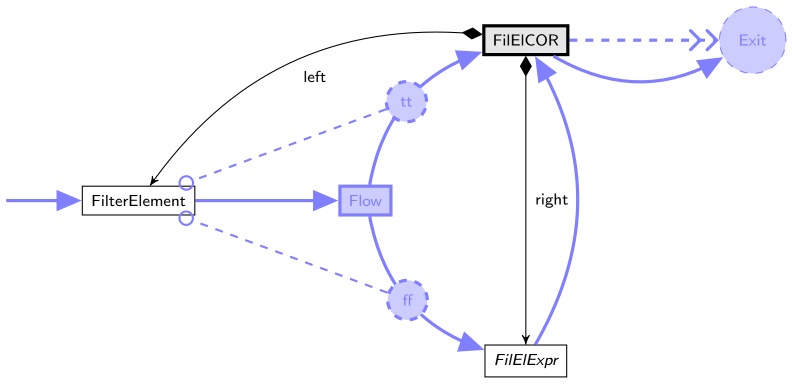


Figure 3.12: Control Flow Specification of a FilElCOR.

3.5.4 Example

Figure 3.20 shows a slightly edited syntax graph from the example, and the corresponding control flow graph. To reduce the size of this graph, only the first filter element of the first filter is included in the abstract syntax graph. Note that `flow` edges, Branch nodes, and auxiliary Flow nodes are added, as well as a single Exit for the FilterModule; this exit represents end-point of the control flow.

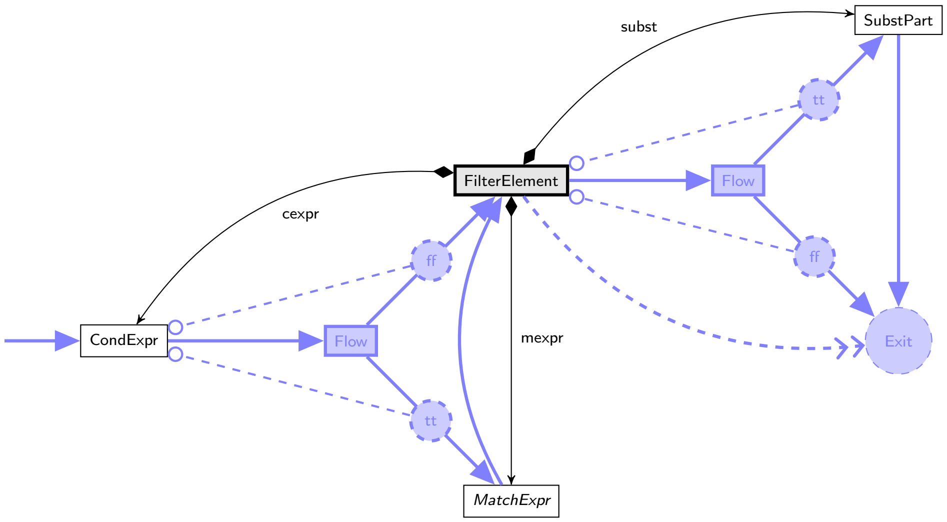


Figure 3.13: Control Flow Specification for a Filter Element

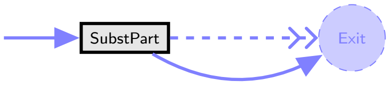


Figure 3.14: Control Flow Specification of a SubstPart.

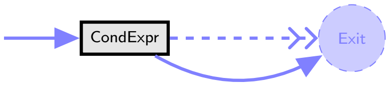


Figure 3.15: Control Flow Specification of a CondExpr.

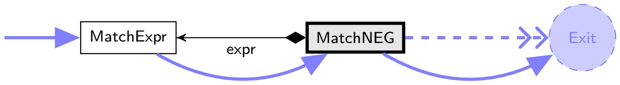


Figure 3.16: Control Flow Specification of a MatchNEG.

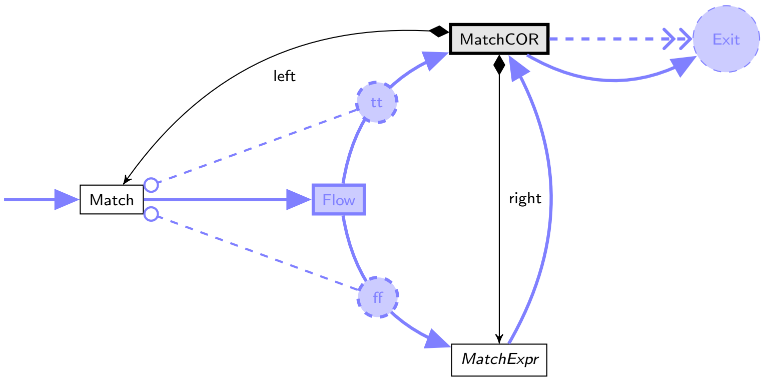


Figure 3.17: Control Flow Specification of a MatchCOR.

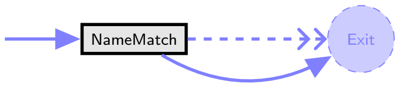


Figure 3.18: Control Flow Specification of a NameMatch.

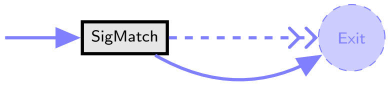


Figure 3.19: Control Flow Specification of a SigMatch.

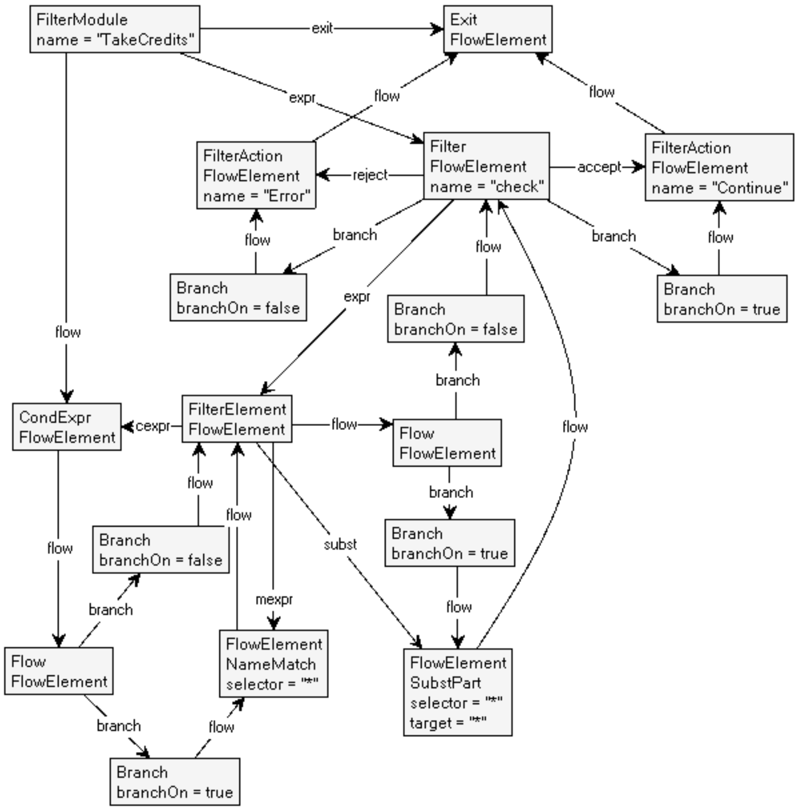


Figure 3.20: Example Control Flow Graph.

3.6 Execution Graphs

To specify the execution semantics of Composition Filters, we must also model the run-time state in the graphs. We extend the *abstract syntax graph* and *control flow graph* to an *execution graph*. These execution graphs essentially represent a part of a snapshot of the state of the system.

3.6.1 Value Graph

The value graph represents the values in a run-time state. To represent the execution of Composition Filters, at least the following values must be represented:

- The target of a method-call, which is an object (e.g. an instance of a class).
- The evaluation results of the actual parameters of a method-call; the parameters themselves — which are in fact expressions — are not required for the execution of CF; we assume that they have already been evaluated before the method-call is executed.
- The execution context of a filter module, consisting of instances for the internals and externals; although the filter module is seen as the unit of instantiation of a traditional object, we can simply extend the traditional object with values for each of the internals and externals, similar to class variables.
- Temporary results of evaluating filter expressions, which consist of boolean expressions.

The type-graph of the value graph is shown in Figure 3.21. To store a value we introduce Slot nodes, with a value edge to a Value node. Values of variables are stored in VarSlot-labelled nodes, which are related to a declared Var, for instance an internal or external. Temporary evaluation results are stored in AuxSlot-labelled nodes, which are related to an Expression. In fact, each syntax element used in a filter expression extends this abstract type, although we do not label them as such. For simplicity, we will also assume that an Argument extends the abstract type Expression; we can then represent actual arguments as AuxSlot nodes with an at edge to an Argument node. A Value node can currently only be an Object (i.e. an instance of a class) with an instanceof edge to its type. Primitive literals can however simply be represented by Object nodes with a instanceof edge to a Class node that represents a primitive type.

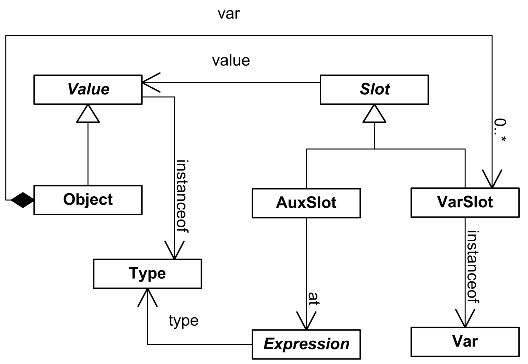


Figure 3.21: Type-Graph of the Value Graph.

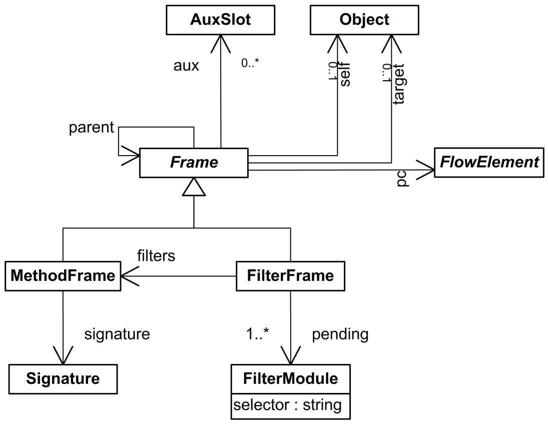


Figure 3.22: Type-graph of the Frame Graph.

3.6.2 Frame Graph

The *frame graph* represent the process part of the current state. In compiler terms, this corresponds to the *stack*. The type-graph of the frame graph is shown in Figure 3.22. The frame graph refers to elements of all other graphs, i.e. the abstract syntax graph, the control flow graph, and the value graph. Essentially, only one new type of node is introduced: the Frame node. In general, a Frame controls the execution of the code at a particular context. A Frame controls the execution of the corresponding code by maintaining a pc-edge (where pc stands for program counter) to the current FlowElement in the flow graph of a context (e.g. a method or a filtermodule). The pc-edge is moved to a successor in the flow graph at every execution step. When a method is called or a new object is constructed, a new, nested frame is created for it and the pc-edge is (temporarily) removed, indicating that the calling frame is passive while the nested frame is running. Obviously, each nested frame has a parent frame, referenced by a `parent`-edge. When a nested frame is finished, it is deleted and the program counter of the parent frame is restored; however, restoring parent frames is not part of this semantics, because the intercepted message still has to be dispatched (i.e., it has no program counter yet). Furthermore, frames can have local and auxiliary variables. In this semantics, we do not use local variables; filter modules do not have any. The auxiliary variables for storing evaluation results — which are part of the value graph — are referenced by aux-edges. A frame may have a self-edge to the object that is the context of the operation being executed. Furthermore, a frame may have a target edge, which is the reference used for meta-operations such as method-lookup and dispatch. The self-edge is in fact create at dispatching to the target of the target-edge. We distinguish between two kinds of frames.

- *Method Frames* represent the instantiation of a method context, and are responsible for everything dynamically related to the method execution.
- *Filter Frames* represent the instantiation of a filtering context and are responsible for everything dynamically related the filtering mechanism.

A MethodFrame refers to a Signature by means of a signature-edge. Besides that, a number of auxiliary edge can be used on the MethodFrame, that are used during the process of invocation:

- `init`. A new method frame starts in the *init* state, indicated by an `init`-self-edge.
- `filtering`. When the receiver of the method-call is enhanced with filter modules, the `init`-edge is replaced by `filtering`-edge. During the filtering process, the state is updated to either `dispatch` or `abort`.
- `dispatch`. This edge indicates that the frame can be dispatched. In fact, this edge triggers the method-lookup process, which is not part of this semantics. When the target of a method-call is not enhanced with filter modules, the `init`-edge is immediately replaced by a `dispatch`-edge edge.
- `abort`. This edge indicates that the message is aborted by an exception. When there is an abort edge, the message will not be dispatched.

A FilterFrame is connected to a method frame in the filtering state via a `filters` edge. This method frame represents the intercepted message in the CF model. The filter frame is connected via `pending` edges to the filter modules that are superimposed on the target's class. When a filter module is selected for execution, the pending edge is removed. When no more pending edges exist, the filtering process is finished. The outgoing `target` edge is the result of evaluting the target part of a substitution part; the `selector` edge is the result of evaluating the selector part of a substitution part. Since the target of the filtered message can be changed by a filter, the `self` edge of the FilterFrame points to the original target of the method-call. In Composition Filters, this object is referenced by the *inner* keyword.

3.6.3 Execution Graph

An execution graph used in the simulation process of a Composition Filters program consists of at least three frames:

- A MethodFrame that represents the message (i.e. the intercepted method-call); the `target` of this frame represents the *target* of the message in the Composition Filters model; the `name` of the Signature referenced by this frame represents the *selector* of the message;
- A MethodFrame that is the parent of the first frame. The `self` edge of this frame represents the *sender* of the message.
- A FilterFrame that is responsible for the filtering mechanism.

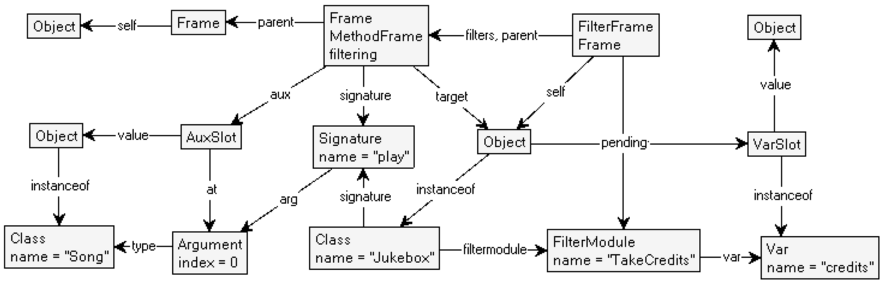


Figure 3.23: Example Execution Graph

An example of a minimal execution graph for the example is shown in Figure 3.23. It represents a snapshot of the run-time state *right before* a message (with selector play) sent to an instance of the JukeBox class is filtered. The graph can be used as a start graph for the semantics defined in the next section. The abstract syntax and control flow graph of the filter module has been omitted. The graph consists of three frames.

The left-most Frame has a `self` edge to the *sender* of the message. All other parts are not required for the execution .

The middle frame, a MethodFrame, has a `parent` edge to the first Frame, a `target` edge to the *target* of the message, an `init` self-edge to represents its execution phase, and an AuxSlot node for the method argument. The `name` of the Signature connected to the frame represents the *selector* of the message.

The FilterFrame is connected to the second MethodFrame with a `filters`-edge and a `parent`-edge. It has a `pending`-edge to the FilterModule that it will execute once the state of the MethodFrame is updated to `filtering`.

The target object has a `var` edge to a VarSlot node that holds the value of the external declared in the filter module. The FilterFrame has a `self` label to the target object. Although the `target` of the MethodFrame can be changed during the execution of the filter module, this `self` reference cannot, since it is used for locating internals and externals instances.

3.7 Runtime Semantics

In this section we will show the specification of the run-time semantics for Composition Filters as a graph-transformation production system. The run-time semantics has been defined by a single rule for each type of syntax element of the language. These rules are specified by hand but, once fixed, allow the method to be applied to any CF specification. The rules can be divided into two categories. The first set of rules is used to specify the semantics of the filtering language. When simulated, these rules evaluate whether a filter should execute either its accept or reject action. The other category consists of a rule for each type of filter action, and describes the actual behaviour of the filters: the effect of the action on the state of the system.

3.7.1 Filter Actions

The rules that specify the semantics of the different filter actions perform the actual behaviour of the filters. The rules involve a modification of the run-time state, represented by the execution graphs. We will illustrate this using the filter actions used in the running example. In this work we define the semantics of the actions actions Dispatch, Error, Substitute and Continue, which can be used to simulate filter with filter-types dispatch, substitute, and error. Filter type meta calls a method that is specified in the base-language. Specifying the precise semantics of a filter with this type involves a specification of the base-language as

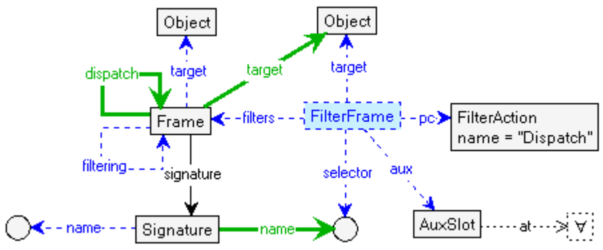


Figure 3.24: Dispatch

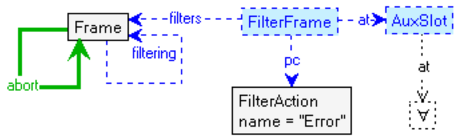


Figure 3.25: Error

well. Therefore we currently do not support this type in our semantics. A more detailed discussion about incorporating Meta filters is given in Section 3.8.

Dispatch

Figure 3.24 shows the rule for Dispatch action. It shows the FilterFrame with a `pc` edge to a FilterAction with name "dispatch". The `target` and `selector` of the MethodFrame being filtered are replaced by the values set at the FilterFrame. The FilterFrame is deleted — along with all AuxSlot nodes — and the state of the MethodFrame is updated to `dispatch`. In word, the action results in the (optionally altered) message being dispatched to the (optionally new) target.

Error

The rule for the Error action is shown in Figure 3.25. Again the FilterFrame and associated AuxSlot nodes are deleted. The state of the method frame is updated to `error`, indicating the occurrence of an exception.

Substitute

Figure 3.26 shows the rule for the Substitute action. The `target` and `selector` of the MethodFrame being filtered are replaced by the values set at the FilterFrame,

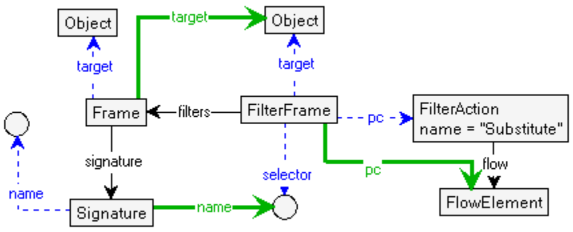


Figure 3.26: Substitute

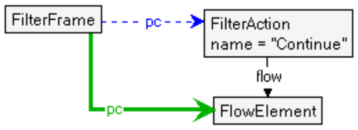


Figure 3.27: Continue

where they are deleted. The program counter is increased.

Continue

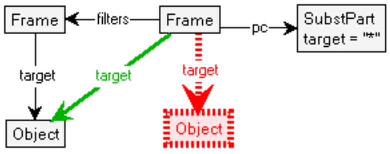
The rule specifying the Continue action is shown in Figure 3.27. This rule simply increases the program counter.

3.7.2 The Substitution Part

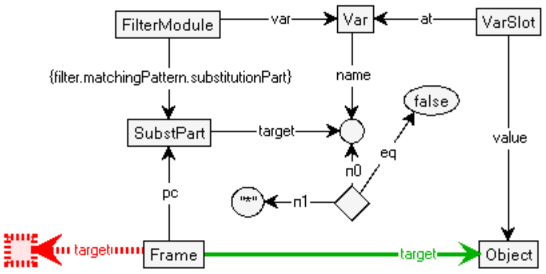
The SubstPart part is an argument for the accept action of a Filter. Its evaluation consists of two parts, namely evaluation of the target argument and evaluation of the selector argument.

The rules for evaluation of the target are shown in Figure 3.28. Figure 3.28a shows the rule that is used for evaluating the "*" argument and results in a `target` edge to the target of the filtered MethodFrame. Figure 3.28b shows the rule that is used for any target other then "*". In resolves to the object stored in the VarSlot of the Var with the selected name connected to the `self` object. Notice that the program counter is not increased. To make sure the rules for target evaluation are applied first, these are given a higher priority (as a property that can be given to the rule in GROOVE).

The rules for evaluation of the selector are shown in Figure 3.29. Again, two rules are needed. Figure 3.29a shows the rule that evaluates a "*" selector; the results



(a) SubstPartTargetStar



(b) SubstPartTargetOther

Figure 3.28: SubstPart Target Evaluation

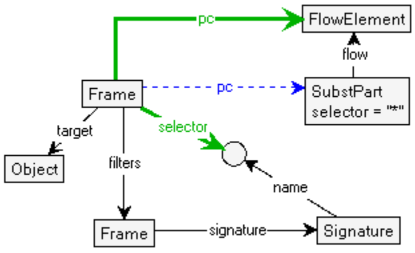
is a `selector` edge from the FilterFrame to the currect selector, i.e. the name of the Signature connected to the filtered MethodFrame. Figure 3.29b shows the rule for evaluating any selector other then "*", and results in a `selector` edge to the specified value. During program counter is increased during selector evaluation.

3.7.3 Filter Matching

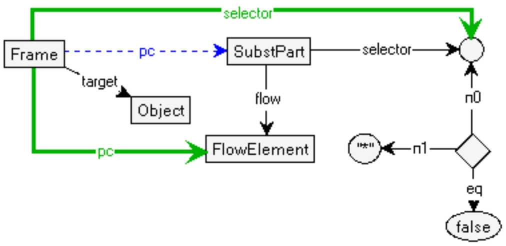
The only thing left to specify is the matching of a filter specification with a message. We enumerate the rules required for this process:

The following points are noteworthy:

- The CondExpr evaluation (Figure 3.30) is done by assuming and assigning either a *true* or a *false* value, non-deterministically; both rules are applicable in the same graphs.
- The NameMatch evaluation is shown in Figure 3.31. The specified selector is compared with the name of the signature of the filtered Frame. The value of the NameMatch is set to the result of the eq string operation.
- The SigMatch evaluation (Figure 3.32) is done by assuming and assigning either a *true* or a *false* value, non-deterministically; both rules are applicable in the same graphs.

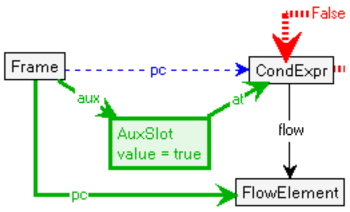


(a) SubstPartSelectorStar

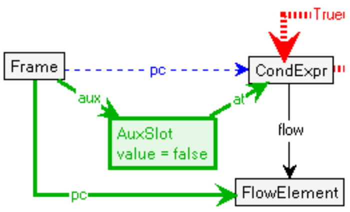


(b) SubstPartSelectorOther

Figure 3.29: SubstPart Selector Evaluation

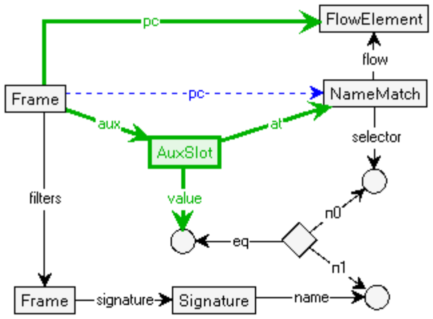


(a) CondExprTrue



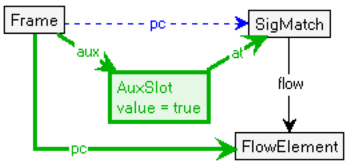
(b) CondExprFalse

Figure 3.30: CondExpr Evaluation

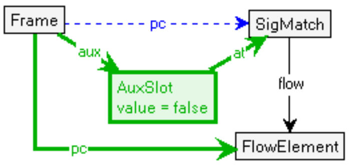


(a) NameMatch

Figure 3.31: NameMatch Evaluation

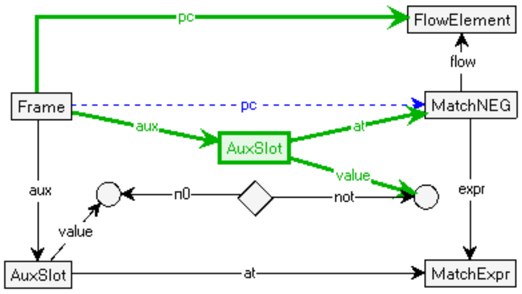


(a) SigMatchTrue



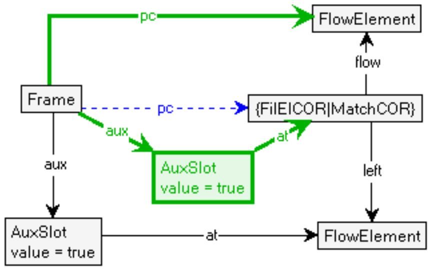
(b) SigMatchFalse

Figure 3.32: SigMatch Evaluation

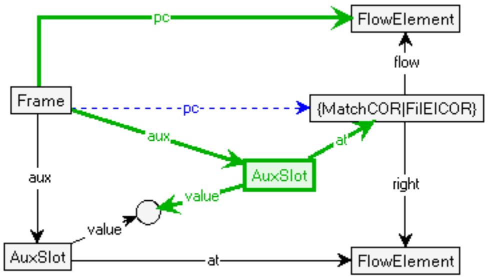


(a) MatchNEG

Figure 3.33: MatchNEG Evaluation

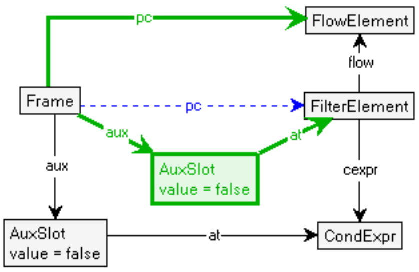


(a) CORLeft

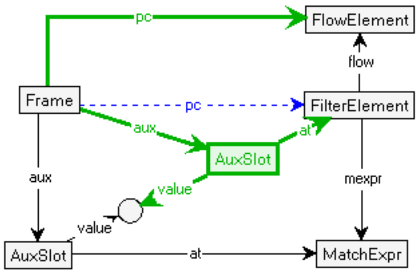


(b) CORRight

Figure 3.34: MatchCOR and FilElCOR Evaluation

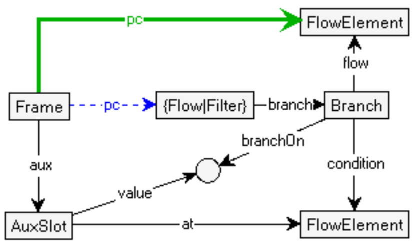


(a) FilterElementLeft



(b) FilterElementRight

Figure 3.35: FilterElement Evaluation



(a) Flow

Figure 3.36: FlowElement Evaluation

- The MatchNEG (Figure 3.33) simply negates the result of evaluating the expr node.

- The MatchCOR and FilElCOR are evaluated using the same rules. The rules in Figure 3.34 shows the two possible scenario's for evaluation. In Figure 3.34a the scenario is matched where the left argument has evaluated to *true* and the `right` argument has been skipped; the COR evaluates to *true* also. The other scenario, where the left argument evaluates to *false* is matched by the rule in Figure 3.34b, where the right argument is evaluated; the COR evaluates to the result of the right argument of the COR.

- The FilterElement evaluation is specified in Figure 3.35. The rules model an *and* operation; when the cexpr evaluates to *false*, the FilterElement is also evaluates to false; this is shown in Figure 3.35a. Otherwise, the mexpr is evaluated, and the FilterElement evaluates to the result of the mexpr; this is shown in Figure 3.35b.

- The branching that is required when the program counter is at an auxiliary Flow node or at the Filter node, is performed by the rule in Figure 3.36. The branch is selected that has a branchOn edge to the value of the expression that is selected as a condition; the program counter is updated to the flow target of this branch.

3.8 Evalution

In this section, we evaluate a number of important properties, such as the requirements presented in Section 3.3, but also some general properties of semantics, such as correctness and completeness.

Correctness

The semantics presented in this chapter is defined according to a description in natural language of the intended behaviour of the language. We therefore do not show any proof that our semantics is correct, as it is specified according to our interpretation of this description. However, as far as possible, the semantics has shown to be consistent with the executed behaviour of the current run-time implementation of Composition Filters.

Filter actions may be specified with a certain level of abstraction; the result of the actions is encoded in the scope of the message. This means that only the effect of the actions on the message is specified. For the default filter types, this is the correct behaviour. However, custom filter types may cause side-effects that require abstract modelling of the effect. Examples of such abstractions can be found in Chapter 5.

Completeness

The semantics is complete w.r.t. the filtering mechanism of Composition Filters. However, the base system may be represented in an abstract way, so that the filters are evaluated for any context and any message. We explain this in more detail in Chapter 5.

Certain parts of the Composition Filters have been omitted. First, only inputfilters are handled. However, most CF specifications only use inputfilters. Also, extending the semantics for outputfilters is a trivial task, since such filters are evaluated similar to inputfilters. Second, only the default filter types are included, and the meta filter type has been omitted. Extending the semantics for custom filter types is discussed in the next paragraph; extending the semantics for the meta filter type is discussed in our discussion of future work in Section 3.9.

Understandability & Extensibility

We believe that graphs are a natural way to model software. Also, once familiar with the notation of graph transformation rules, the results of applying such a rule are understood from these specifications easily.

In extending the semantics for custom filter types, each new filter action only requires a single rule to be specified, that specifies the existence and incrementing of a program counter, and the effect of the action on the message. Taking the rules for filter actions in this work as example should make extending the semantics a fairly simple task.

Verification and Analysis

As part of our requirements, we have said that the semantics must be a solution to being able to analyse and verify Composition Filters specification. We think this goal is satisfied. The semantics allows us to generate a state space representing the filtering mechanism of any message sent to an instance of a certain class in any context. The nature of the semantics — a small-step semantics — exposes all the steps of the execution. This can for instance be useful for model checking the system.

The graph-based nature of the semantics has another advantage. In representing run-time states as graphs, we are able to compare states by using an isomorphism comparison. This test is integrated into the simulator tool and takes care of representing isomorphic graph as identical states in the state space. In Chapter 5 we show how this can be used to detect interference between filter modules.

Base-System Abstraction

In defining an execution semantics for Composition Filters, the design of Composition Filters makes it possible to abstract from base-system details. In fact, part of a Composition Filters specification provides the interface (i.e. an abstraction) with the base-system. We have encountered these in the specifications of condition expressions and signature matching.

We deal with the base-system by evaluating any predicates on the base-system in a non-deterministic way, i.e. by assuming that a condition expression or a signature match can be evaluated to true or false. This is handled by two rules for each situation that can be applied in a state where the value of the predicate is yet to be determined, and applying these rules results in two states where the predicate is set to either true or false. In fact, the state space then represents the execution of the filter module in any base-system, where each trace in the state space represents the execution of the filter module in a set of concrete base-systems.

3.9 Conclusions

3.9.1 Related Work on AOP Semantics

The basic idea of using graph transformations for operational semantics is far from new: it ranges from a term graph-based semantics for functional languages (see Plump [Plu99]) to graph-based semantics for actor languages (see Janssens [Jan99]) and visual languages (e.g., [Hau06]). For object-oriented languages the first approach of this kind is by Corradini et al. [CDFR04]; the approach of this paper is inspired by [KKR06].

There are quite a few works describing a specification of an aspect-oriented operational semantics. Most of these approaches focus on — like this work — a simplified base language and aspect extension. In general, they focus on a certain feature of aspect-oriented languages. The works all use a textual, mathematical notation, whereas our notation is of a more intuitive visual kind. The works mentioned also do not provide a means for execution, whereas our work can be directly be used to visually represent the execution of a specific Composition Filters specifications. We now enumerate the most relevant work.

Djoko Djoko et al. [DDF08] presents the Common Aspect Semantics Base (CASB), which consists of an SOS semantics of two base languages — a simple functional language and AFJ — and a number of aspect-oriented programming language features. The work in Chapter 4 is based on the CASB.

Lämmel [Läm02] presents an operational semantics for an imperative object-oriented language with join-points for method calls. The goal of this work is to formally define the semantics of a single AOP feature and proof certain (safety and liveness) properties of the modelled language feature, whereas — in our work — we focus on verifying properties of programs specified in the language. Kiczales et al. [KD02] give a denotational semantics for a first-order procedural language with join-points for procedure-call, procedure execution, and advice execution. The work is intended to serve as a reference semantics for against which correctness results may be measured. Jagadeesan et al. [JJR03] give a calculus of untyped aspect-oriented programs. Their specification is class-based, and models multi-threaded programs. They also define formally define a weaving algorithm. The purpose of the work is to verify properties (correctness) of the weaving algorithm. Opposite to our work, no properties of the language or instances of the language are verified. Walker et al. [WZL03] present a core aspect-oriented calculus for a simple, idealised aspect language based on the simple-types lambda calculus. The goal of this work is to provide means for (largely) language independent studying of aspects. No claims are made of the properties that may be verified. Instead, the focus is on defining a semantics for a concrete language using the given core calculus. Bruns et al. [BJJR04] present a formal model for aspects in a functional language. The purpose of this work is to provide a different view on aspects, where aspects are the only computational entities. The semantics can serve as a "meta-language", but is mainly used to study the theory of aspects in general. Clifton et al. [CL05] provides a calculus for an imperative object-oriented language with advice bindings and proceed. The contribution of the work is to give a semantics of proceed that allows changing the target object, which in fact is also possible using Composition Filters. In general, the semantics is used to study properties of proceed.

3.9.2 Future Work: Possible Extensions

Currently we have not included the Meta filter type in our semantics. To be able to include simulation of the ACT method, we also require an execution semantics of the base-language. However, a simplified version of the Meta filter behaviour may already be useful. We could for example extract the changes that are made to the message and the reactivation kind that is used on the message by the ACT. Other side-effects, such as outgoing method-calls and changes to variables in the ACT itself are ignored. With this approach, we could still analyse the control properties of the message filtering process, including Meta filters.

Another possible extension is to make simulation more concrete for a specific base system. Regarding signature matches, this is done by simply adding the signatures of all classes that are used in the filter modules to the abstract syntax graph; signature matches can then be matched against the signatures of the specified object. This gives a more concrete simulation of the base-system at hand.

It is also possible to have less impossible run-time state represented in the state-space. For example, let us assume a filter module with two conditions **a** and **b**. For the condition expressions $a\&\&b$, and a, not all possible combinations of $true$ and $false$ are possible. For example, if a is $false$, $a\&\&b$ cannot be $true$. This can be done by representing declared conditions in the abstract syntax graph and evaluate these to true and false; the condition expressions using these conditions can then be evaluated accordingly. In doing this, less traces that cannot exist are created.

As we have mentioned, the semantics currently only supports inputfilters. Without much effort, the semantics can be extended to also support outputfilters. This may give rise to the simulation of longer filter module sequences, whereas a message being sent from an object with output filters may be sent to an object with input filters.

3.9.3 Contributions

This chapter describes an execution semantics for the Composition Filters model. We employ a graph-transformation-based formalism to specify the semantics. This involves giving a control-flow semantics and a run-time semantics of the aspect language, and a run-time semantics for the actions performed by filter modules to manipulate the base language by means of method-call interceptions

The formal semantics can help in understanding the language, as well as providing a means for investigating new language features. Also, by using graph-transformations as our specification language, we can directly execute this semantics given a program represented a a graph. Specifically for Composition Filters this means that we can simulate the result of a Composition Filters specification on the base-system state. The result of this — a state space represented as a labelled transition system — can be used for existing verification techniques for program specifications, such as model checking.

Defining a control flow semantics has learned us that the control flow of Composition Filters is not very straight-forward. In fact, the abstract syntax model for filter modules that we use in this chapter is an enhancement of the current language model; it can therefore be used as an improvement for the language.

In Chapter 5 we illustrate simulation of a program using the semantics and using the semantics for verification.

Chapter 4

Specification and Simulation of Featherweight AspectJ

4.1 Introduction

In this thesis we want to investigate formal techniques for aspect-oriented languages, i.e. to verify whether all possible system executions are correct with respect to its specification. We do this by first defining the semantics of the language at hand as an operational semantics. With such semantics, the maning of a program is represented as a sequence of computation steps that result from the program's execution.

In this chapter we present an operational semantics for Assignment Featherweight Java (AFJ), which is an extension to Featherweight Java (FJ), a minimal subset of Java. This simple language - although not suitable for industrial implementations - is often used for studying the consequences of language extensions. We study part of this language, namely *around advice*, which can — combined with a *proceed* statement – be used to represent before and after advice.

The semantics of the language and the extension are taken from a work called the Common Aspect Semantics Base (CASB) [DDF08]. The CASB is a reference model for the runtime semantics of aspect-oriented programming languages. This work presents a structural operational semantics (SOS) for the language at hand (AFJ with an aspectual extension).

The specification method in this thesis is graph transformation. In this chapter, we demonstrate the advantages of a graph-transformation based operational semantics over traditional, mostly textual notations:

- Specifying of a semantics can be a complicated task; mistakes are easily made. The directly executable nature of graph transformation increases ease and confidence of specification of a semantics by giving the user a way to test it without having to write a interpreter first(which may contain errors either copied from the semantics or made during implementation).
- By giving the semantics in this way, the road is opened towards applying existing verification methods, such as the work we present in Chapter 5. Also, the labelled transition system (LTS) (resulting from executing the semantics) lends itself directly for model checking (see [KR06]).
- We believe that the visual nature of the graph transformation rules will appeal to many readers that are not experts in mathematics. This benefits the understanding of a language based on the given semantics.

To increase confidence in the correctness of our definitions, we show that they coincide with the formal specification of the AFJ language in the CASB.

In the next section we will explain details of Assignment Featherweight Java with around advice. In Section 4.3, the graphs are described that represent AFJ programs with advice. In Section 4.4 we provide a detailed description of the graph transformation based semantics of chosen language. In Section 4.5 we discuss the reference semantics and show that our semantics is correct with respect to the reference semantics. Section 4.6 shows some example programs and illustrates the result of simulation. Finally, in Section 4.8, we discuss related work, future work, and the contributions of the work presented in this chapter.

4.2 Assignment Featherweight Java with Around Advice

In this section, we describe the syntax and run-time of the Featherweight AspectJ language.

4.2.1 The Featherweight AspectJ Language

Featherweight Java (FJ) [IPW99] is a subset of Java that contains only five forms of expression: object creation, method invocation, field access, casting, and variables. The language has no branching (i.e. conditional statements).

$$\begin{aligned}
\mathit{Prog} &::= \overline{L}; e; \overline{A} \\
L &::= \textbf{class}\, T\, \textbf{extends}\, T\, \{\overline{T\, f};\, \overline{M}\} \\
M &::= T\, m(\overline{T\, x})\{e;\} \\
e &::= x \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\, T(\overline{e}) \\
&\quad\mid e.f = e \mid (T)e \\
A &::= T\, \textbf{around}(\overline{Tx}) \;:\; P\, \{e'\} \\
P &::= \textbf{call}(T^*.m^*(\overline{T^*})) \\
e' &::= e \mid e.\textbf{proceed}(\overline{e}) \\
T^* &::= T \mid \textbf{*} \mid T+ \\
m^* &::= m \mid \textbf{*}
\end{aligned}$$

Figure 4.1: Grammar of Featherweight AspectJ.

Assignment Featherweight Java (AFJ) [MP05] has extended this language with mutable field variables to bring it closer to the way Java programs are usually written. The minimal syntax and operational semantics make it a handy language for conceptual studies on the implications of language extensions. This makes the language useful for trying AOP language features. We actually study an extension of the AFJ language with around advice. In this thesis, we will refer to the extended language as Featherweight AspectJ (FAJ). The grammar of FAJ is shown in Figure 4.1. Throughout the chapter we use the overbar notation for vectors.

- A program *Prog* consists of a vector of classes $\overline{L}$, a main expression e, and a set of advice declarations $\overline{A}$.
- A class L consists of a list of field names and types $\overline{Tf}$, and a list of methods $\overline{M}$.
- A method consists of a return type, an identifier, a list of arguments $\overline{T\overline{x}}$, and a method body, which is an expression.
- Expressions can be (from left to right) variables (e.g. a method parameters), fields accesses, method invocations with a sequence of expressions as arguments, object creations with a sequence of expressions as parameters, castings, assignments, and proceed expressions (see below). Object creation is not handled by an explicit constructor. Instead, the ordered list of arguments is assigned to the ordered list of fields.

- Aspects (A) are represented as (global) declarations of an around advice and a point-cut. Advices are methods with a method body e', that can optionally contain a *proceed* expression. As usual, we can use this also to mimic *before* and *after* advice, by adding a PROCEED instruction after or before the instructions of the advice, respectively.
- An advice declaration is combined with a point-cut declaration P that selects certain expressions. In this language we have limited the point-cut language to selection of method calls.
- A point-cut is specified as a call to a certain receiver type T^*, which can be a concrete type, a wild-card, or $T+$, selecting a type and all its sub-types. The same is used for the parameters of the call.
- The method identifier can be either a concrete method identifier or a wild-card selecting any identifier.

An example FAJ program — used throughout this chapter — is shown in Figure 4.2. It defines a class Exam, with a field Nat grade and a method Nat setGrade(Nat) to update the grade. The class NamedExam represents an exam where the student has filled in his name. The classes Nat, Zero and Succ represent the natural numbers; every natural number can be represented as a sequence of successors of zero. Nat defines the method Nat add(Nat) and Nat dec(), which return the sum of itself and the argument, and its value decreased by one, respectively. Since FAJ does not allow interfaces, abstract classes, or empty method bodies, the methods in Nat define the correct implementation for Zero, which is a sub-class of Nat. Note that the decrement of Zero is Zero, since -1 is not a natural number. The implementation for successors is defined in the Succ class. Two *main expressions* are specified (one commented) that represent the different cases we demonstrate in this paper. In the first, the exam is awarded a five; in the second a zero. After the main expression, two around advices are declared. The first specifies that, when a person filled in his name on the exam, he or she is awarded one extra point. Notice that the signature of the advice method contains the parameters of the intercepted method *plus* the type of the receiver. Also, proceed is syntactically designed as a method-call on the receiver object, with the signature of the intercepted method-call. The second advice describes a situation where the average score is too high and has caused a norm change: every exam is awarded one point less. When setGrade is called on an instance of NamedExam, both advices are triggered. This enables two interesting scenarios, caused by the initial grade (the argument of setGrade), and the order in which the advice are executed. When a person is awarded a grade greater than zero, the grade can be increased and decreased or decreased and increased, resulting in the same final grade. However, when a person is awarded a zero, if his grade is increased first and then decreased, the result is still zero; if the grade is decreased first and then increased, the final grade will

```
class Exam extends Object {
   Nat grade;
   Nat setGrade(Nat n) { this.grade = n; }
}

class NamedExam extends Object {
   String name;
}

class Nat extends Object {
  // default implementation
  Nat add(Nat n) { n }
  Nat dec() { this  }
}

class Zero extends Nat {
  // zero can use default implementation
}

class Succ extends Nat {
   Nat pred
   Nat add(Nat n) { new Succ(pred.add(n)); }
   Nat dec() { this.pred;  }
}
// scenario 1:
new NamedExam(new Zero()).setGrade(new Succ(new Succ(new Succ(
    new Succ(new Succ(new Zero())))));)
// scenario 2:
// new NamedExam(new Zero()).setGrade(new Zero())

// advice 1: extra point for filling in your name
Nat around(Nat n, NamedExam receiver) :
                call(NamedExam.setGrade(Nat) ) {
    receiver.proceed(new Succ(n));
}

// advice 2: norm change
Nat around(Nat n, Exam receiver) :
                call(Exam+.setGrade(Nat)) {
    receiver.proceed(n.dec());
}
```

Figure 4.2: Source code of the example FAJ program.

$$mcode(id, r) = S(mbody(id, r)); \textsc{return}$$

$$\begin{aligned}
S(x) &= \textsc{var}_x \\
S(e.f) &= S(e); \textsc{get}_f \\
S(e_0.f = e) &= S(e); S(e_0); \textsc{set}_f \\
S(\textsc{new}\, T(e_0, \ldots, e_n)) &= S(e_0); \ldots ; S(e_n); \textsc{new}_T \\
S(e.m(e_0, \ldots, e_n)) &= S(e_0); \ldots ; S(e_n); S(e); \textsc{call}_m \\
S(e.\textsc{proceed}(e_0, \ldots, e_n)) &= S(e_0); \ldots ; S(e_n); S(e); \textsc{proceed}
\end{aligned}$$

Figure 4.3: Sequentialisation

be a one. In Section 4.6 we will show that we can automatically visualise such differences.

4.2.2 Run-time Semantics

Next, we explain the run-time semantics defined in this chapter, which is a slight adaptation of the semantics defined in the CASB. Since in this work we concentrate on run-time semantics, we have decided to ignore issues of static semantics, such as typing. Therefore, type casting and method overloading are not treated in our semantics.

The run-time semantics of this language is specified in terms of sequences of instructions. That is, every expression in the grammar can be sequentialised into a sequence of stack-based instructions of the types: CALL, RETURN, NEW, VAR, GET, SET, and PROCEED. In our semantics, we assume that expressions are pre-evaluated into such sequences; whenever we represent an FAJ program, method-bodies consist of a sequence of instructions instead of an expression. We define the sequentialisation process as a function $mcode : Id \times T \to \overline{Instr}$ (Fig. 4.3) that turns a method identifier into a sequence of instructions. It uses the function $mbody : Id \times T \to Expr$ (as defined in the reference semantics), that finds the body expression for a method identifier and a receiver type. This is defined as follows: Thus, the *mcode* function will use the given identifier and type to the *mbody* function, which looks up the method and returns the body expression. This expression is broken down into a sequence of instructions by function S. Finally, a RETURN instruction is added.

Run-time information is stored in both a heap and a number of global stacks:

- A so-called *continuation stack* contains the currently scheduled instructions, the top instruction being the first to be executed. Execution terminates when the continuation stack is empty.
- The results of evaluating an expression are placed on a so-called *value stack*.

For executing around advice, the following concepts are required:

- a *proceed stack* is used for postponing an action that triggers an advice; PROCEED instructions pop the top of the proceed stack onto the continuation stack;

Furthermore, a number of auxiliary instructions will be used:

- a DO instruction is used for invoking advice;
- a PUSHP instruction pushes the top of the continuation stack on top of the proceed stack;
- a POPP instruction pops the top of the proceed stack;

When a CALL instruction is matched by any aspects, these aspects are first scheduled (in a certain order) by placing a DO instruction on either the continuation stack (for the first advice), or the proceed stack (for any other advice). The PUSHP and POPP instructions are added to achieve a uniform handling of multiple around advice, all of which may contain a PROCEED instruction.

4.3 Graph-Based States

In this section we explain the graph representation of FAJ states. For this, we use so-called meta-graphs (e.g. Fig. 4.4), which show the nodes used in the actual graphs (with a label), and all the edges that may be used to connect these nodes. The meta-graph does not give a multiplicity restriction. An ordinary graph conforms to this meta-graph when:

- each node in the ordinary graph can be mapped on a node in the meta-graph, which is possible when both nodes have the same label;
- for each edge in the ordinary graph, there must exist an equally labelled edge between the images in the meta-graph of the source and target node of that edge.

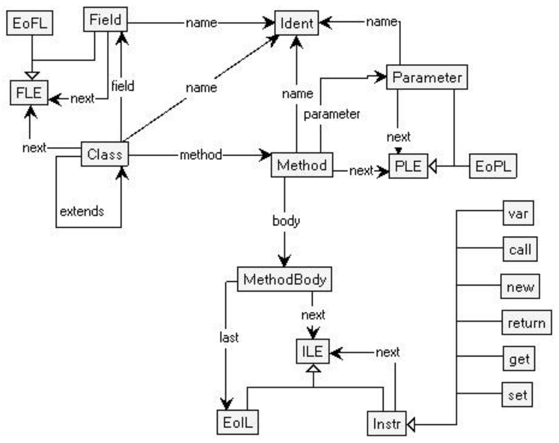


Figure 4.4: Meta-Graph of the Base Language Static Structure

We assume that some translator generates a graph for a given program given by *Prog* in the grammar described above. This graph contains the static structure of the base-program $\overline{L}$, static structure of point-cuts and advice $\overline{A}$, and some representation of run-time information, which is initialised by the main expression e on the continuation stack. We describe these parts one by one.

4.3.1 AFJ Program Graph

The static structure of the AFJ-part of the program (the base) is represented by a graph conforming to the type-graph shown in Figure 4.4. Most of it should be self-explanatory. Classes have fields and methods. Methods have parameters and a method body. A method body has a list of instructions. Classes, fields, methods, and parameters have names. Fields, parameters, and instructions are ordered in a list by `next`-edges. The last element of a list is an EoL node. List elements and EoL nodes can be either for fields (FLE,EoFL), parameters (PLE,EoPL) and instructions (ILE, EoIL).

The main expression — not in this figure — is on the continuation stack, which is described shortly. Since type-checking is neglected, return types, field types, and parameter types are not used and therefore not represented in the graph.

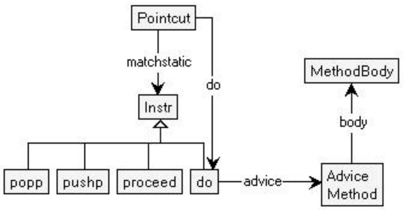


Figure 4.5: Meta-Graph of the Aspectual Static Structure

4.3.2 Aspect Program Graph

Advice is represented conforming to the meta-graph shown in Figure 4.5. Advice is modelled as a method, inheriting all options of regular methods. However, the body of an advice can contain a PROCEED instructions, where regular methods cannot. Point-cuts are connected to all instructions they match. A point-cut is associated with an advice through a DO instruction, which is used to invoke the advice.

4.3.3 Run-time Graph

Run-time information is represented conforming to the heap and stack meta-graph, which is shown in Figure 4.6. The heap is represented as Object-labelled nodes in the graph. Objects are connected to Class-labelled nodes representing their types. The values of the fields of an Object are stored in Var nodes. The values themselves are again Objects.

For method execution, a Scope node represents the scope of the method execution. Each variable in this scope (e.g. the value of parameters, and *this*) are connected to this scope. Instructions of the method body are also connected to the scope when they are pushed on the continuation stack, such that a VAR instruction can find the correct value.

Execution is modelled as interpretation of instructions on the continuation stack, and using values from a value stack. Therefore, most rules of the semantics described in this paper perform one or more pop or push operations. Stacks consist of a sequence of cells, the top one directly linked to the stack itself. Cells can contain either instructions or objects. A cell can be tagged, meaning that if this cell contains an instruction, this instruction cannot trigger an advice. Stacks are named as follows: C represents the continuation stack, S the value stack, P the proceed stack. (The names of the stacks are not in the meta-graph, they are instantiations of the node-type Stack).

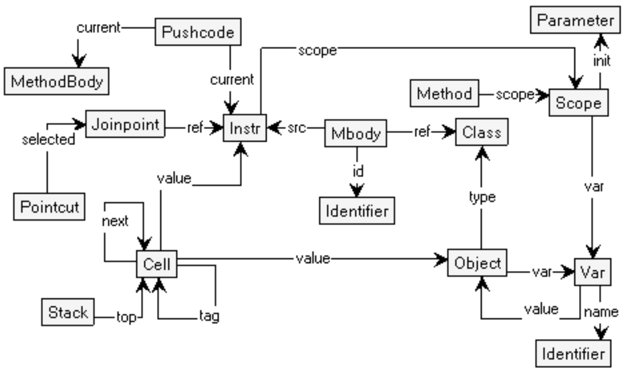


Figure 4.6: Meta-graph of run-time information.

Some auxiliary nodes can be created during execution that trigger and store information for certain sub-routines. For example, the `Mbody` is used for method lookup, and the `Pushcode` node is used for pushing the instructions of a method body on the continuation stack. Auxiliary instructions PUSHP and POPP can be created during advice execution.

4.4 Language Semantics

Given the graph-based representation of the system state as presented in the previous section, we can now give an operational semantics of the language, by specifying one or more rules for each instruction. Most rules match a certain instruction on top of the continuation stack. Other rules are used for executing sub-routines, e.g. method-lookup.
For the specification of such *internal* sub-routines, we choose priorities to help keep the specifications simple. A rule with a certain priority can only be applied when no rule with a higher priority matches. We will use three different priorities. The rules that match base-language instructions have the lowest priority (priority 0). To allow these rules to remove the instruction from the stack, but have sub-routines continue before the next instruction is matched, all base language sub-routine rules have priority 2, i.e. the next instruction continuation stack is matched no sooner than when all sub-routines for the previous instruction have finished. Rules for handling advice have priority 1: they have precedence over base program instructions, but do not interfere with sub-routines.

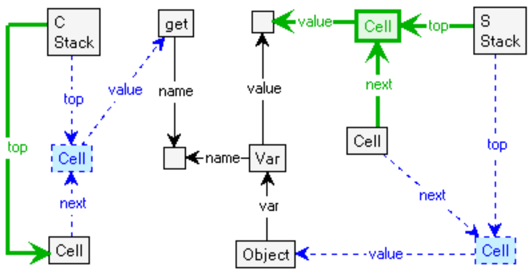


Figure 4.7: Rule for the GET instruction

4.4.1 AFJ semantics

There are rules for each of the instructions discussed above. We describe these rules one by one.

The get Instruction

Figure 4.7 shows the graph production rule for the GET instruction. In Fig. 4.3 we have shown that this instruction originates from an expression of the kind $e.f$. The rule applies when the GET instruction is on the continuation stack and the result of receiver object e is on the value stack; it is processed before the GET instruction. A `Var` node connected to the receiver object is selected, that has the same name as the name argument of the GET instruction. The receiver object is popped from the value stack, and the value of the selected `Var` node is pushed on the value stack.

The set Instruction

Figure 4.8 shows the rule for the SET instruction, which originates from an expression of the kind $e.f = e_0$. The rule applies when a SET instruction is popped from the continuation stack. The receiver object e and the new value e_0 are on the value stack. The variable to be updated is selected, that has the same name as the name argument of the SET instruction. The value of the variable is replaced, and the receiver is popped from the value stack. The new value remains on the value stack, since it is also the result of $e.f = e_0$.

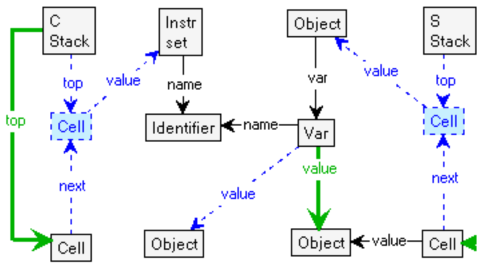


Figure 4.8: Rule for the SET instruction

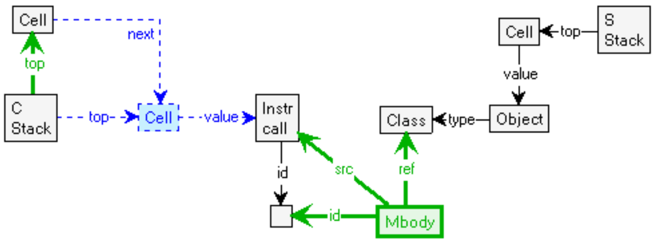
```
// phase 1: call
call;
// phase 2: method body lookup
until ( call_mbody_match ) do {
  call_mbody_up
}
// phase 3: parameter initialisation
alap { call_initparam }
// phase 4: pushing body on the continuation stack
call_pushcode_start;
until ( call_pushcode_end ) do {
  call_pushcode
}
```

Figure 4.9: The method-call sub-routine.

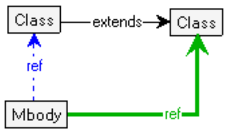
The call Instruction

This CALL instruction requires some more work to be done. First, a method-body lookup has to be performed. Second, arguments (that are already on the value stack) have to be transferred to local variables. Third, the instructions of the method-body have to be pushed on top of the continuation stack. To be able to model this, we need to apply a sequence of rules. The entire process of the CALL instruction is described in Figure 4.9 using the control language introduced in Chapter 2. (The control program is not actually part of the semantics; as mentioned above, we use priorities to control rule applications).

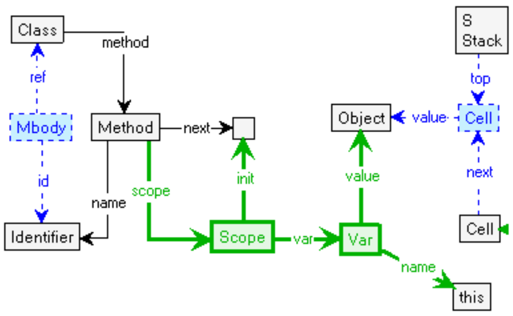
The process is divided into four phases. We describe these phases one at a time. The rules for this process are shown in Figure 4.10 and Figure 4.11.



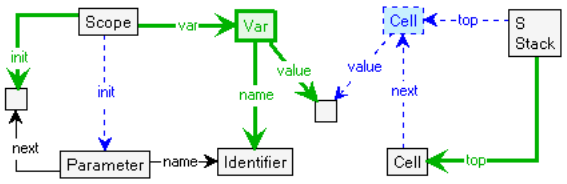
(a) *call*



(b) *mbody_up*

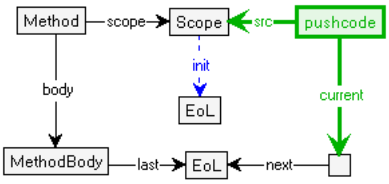


(c) *mbody_match*

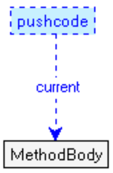


(d) *call_initparam*

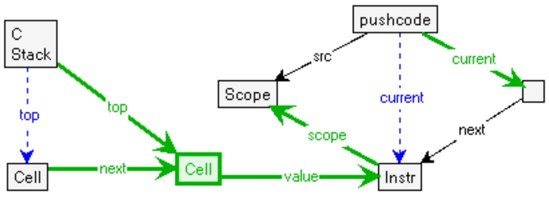
Figure 4.10: The rules for the CALL instruction phases 1-3.



(a) *call_pushcode_start*



(b) *call_pushcode_end*



(c) *call_pushcode*

Figure 4.11: Rules for the CALL instruction phase 4.

- *Phase 1:* A CALL instruction is popped from the continuation stack. The method lookup process is started by creating an `Mbody` node that refers to the type of the receiver object on top of the value stack, and the identifier of the method that is called.

- *Phase 2:* The `Mbody` node is used to search upwards in the class hierarchy for a method with an identifier that matches the associated identifier. When the method is found, a scope is created to contain the variables for the method parameters, and the *this* variable is created with the receiver as value, which is popped from the value stack. An `init` edge is created connecting the scope to the first parameter that is to be initialised.

- *Phase 3:* As long as the scope has an `init` edge to a Parameter, a value is popped from the value stack and a variable is created storing the value for the parameter. The parameters are initialised in backwards order, consistent with the order of the arguments on the value stack.

- *Phase 4:* (Figure 4.11) When all parameters have been initialised, a push-code process is created for pushing the method body on the continuation stack. Each time an instruction is pushed, the `current` edge of the `pushcode` node is updated to the next instruction. The instructions are pushed in reverse order (i.e. starting with the RETURN instruction) so that — when finished — the first instruction will be on top of the continuation stack.

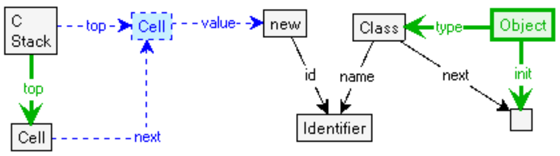


Figure 4.12: The *new* rule.

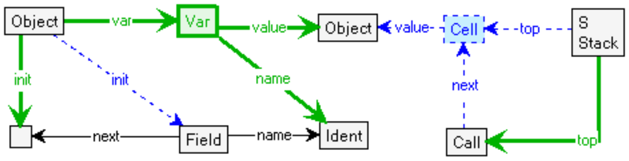


Figure 4.13: The *new_initfield* rule.

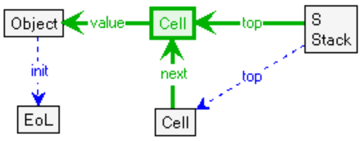


Figure 4.14: The *new_end* rule.

The new Instruction

As mentioned before, a class has no explicit constructor. Instead, when a NEW instruction is executed, the initial values of the class' fields are on the evaluation stack and will be assigned to the corresponding variables of the newly created object. When the execution of the NEW instruction is finished, the created object is placed on top of the continuation stack. The rules for the NEW instruction are executed consistent with the following control expression:

```
new; alap{new_initfield}; new_end
```

Figure 4.12 shows the rule that matches the NEW instruction on top of the continuation stack. It creates a new `Object` with a `type` edge to the class and an `init` edge to the first `Field` of the class, or `EoL` (the end of the list). The `init` edge is a trigger for the rule for field initialisation.

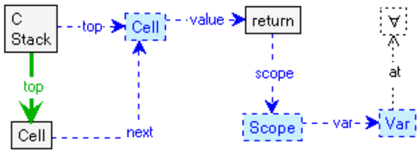


Figure 4.15: The *return* rule.

Figure 4.13 shows the rule for the initialisation of a field. It matches when an `Object` node has an `init` edge to a `Field` node. A `Var` node is created and a value is taken from the evaluation stack. The `Var` node is connected to its parent object and the name of the field. The `init` edge is redirected to the next Field (or `EoL`).

Figure 4.14 shows the end of the field initialisation *init* process. This rule is matched when the `init` edge is targeting `EoL` and removes that edge. The new object is pushed on the continuation stack.

The return Instruction

As defined by the sequentialisation function in Figure 4.3, every method body ends with a RETURN instruction. It cleans up the scope after method-execution. The corresponding graph production rule is shown in Figure 4.15. The `Scope` that was created during execution of the CALL instruction is deleted.

The var Instruction

Figure 4.16 shows the rule for the VAR instruction, which is the usage of a method argument. When the VAR instruction referring to a variable name is on top of the continuation stack, the rule will match the `Var` node in its scope that has that same name. The resulting value is pushed on the evaluation stack.

4.4.2 FAJ Semantics

Triggering advice

When an instruction is on top of the continuation stack and is matched by one or more point-cuts, the associated advices are scheduled by pushing the DO instructions (linking the point-cuts with the advices) on (i) the continuation stack, if the advice is scheduled first; (ii) the proceed stack, if the advice is scheduled after the

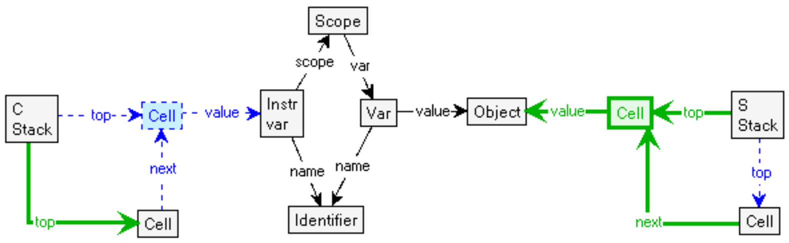


Figure 4.16: The *var* rule.

first advice. The original instruction is popped from the continuation stack and pushed on the proceed stack.

The *around* rule in Fig. 4.17a takes care of scheduling the first advice that is scheduled and creates a join point instance. This instance is used to trigger matching any subsequent advices that, instead of matching an instruction on the continuation stack, match an instruction on the proceed stack referred to by this join point. Matching these subsequent advices is done by a different rule.

The do instruction

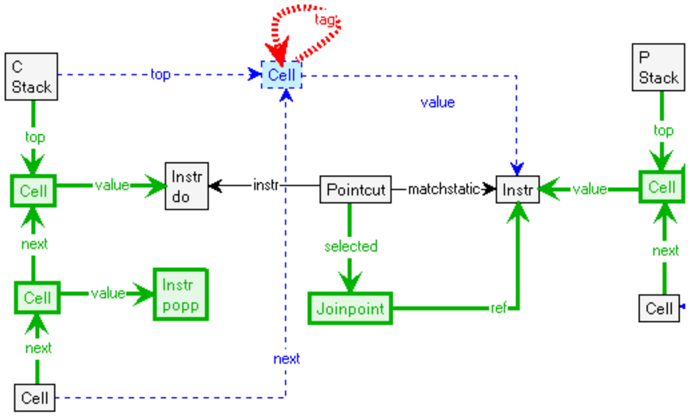
The DO instruction in Fig. 4.17b invokes the advice, in the same manner as is done by a CALL instruction; a scope instance is created, that will trigger rules for parameter initialisation and pushing the body on top of the continuation stack.

The proceed instruction

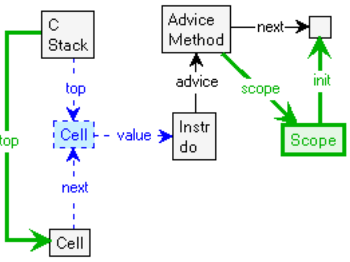
The PROCEED instruction in Fig. 4.17c pushes the top of the proceed stack back on the continuation stack. To not have this instruction trigger advice again, the cell containing the instruction was *tagged* already when the advice was scheduled (Fig. 4.17a); the tagged instruction cannot trigger the around rule.

The popp and pushp instruction

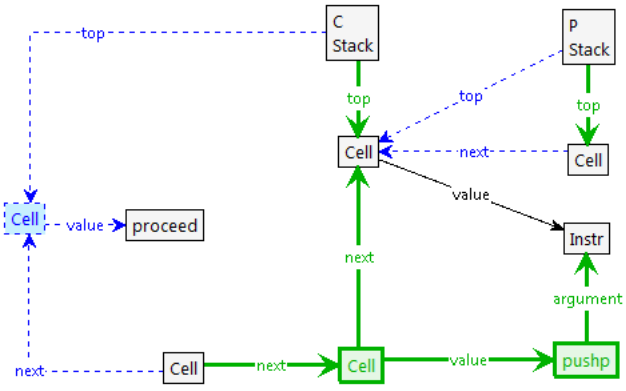
To handle advice without a PROCEED instruction, the around rule also puts a POPP instruction on the continuation stack first. After executing the advices, this instruction pops any remaining instructions for the join point from the proceed stack. For the POPP instruction to be able to function (it must pop at least one instruction), the proceed rule pushes a PUSHP instruction on the continuation



(a) *around*

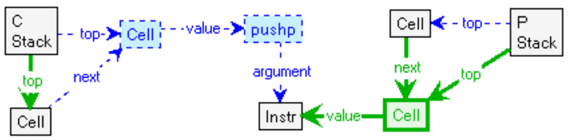


(b) *do*

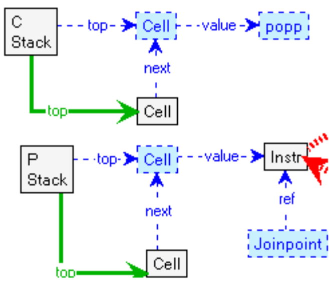


(c) *proceed*

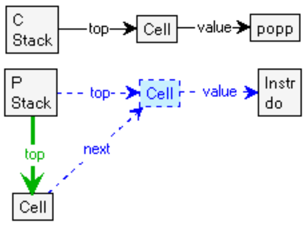
Figure 4.17: The main rules for advice execution.



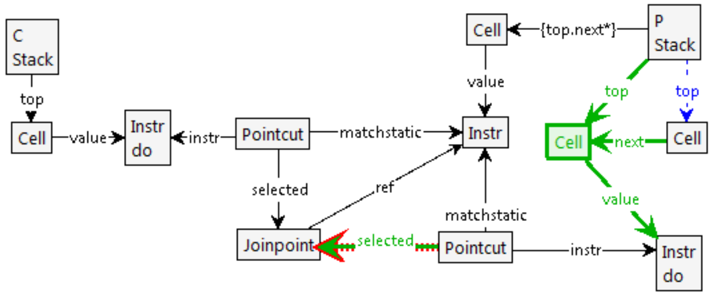
(a) *pushp*



(b) *popp*



(c) *popp_do*



(d) *around_more*

Figure 4.18: More rules for advice execution.

stack. This instruction, which is executed after returning from the PROCEED instruction, which pushes the instruction that was triggered by PROCEED back on the proceed stack, from which it will later be popped from by the POPP instruction. The rules for the PUSHP and POPP instruction are shown in Figure 4.18.

Triggering more than one advice

Figure 4.18d shows the rule that is required to handle more than one advice. Because the advice-triggering instruction was pushed on the proceed stack by the first advice, instead of matching an instruction on the continuation stack, the rule matches an instruction on the proceed stack that is referred to by a `Joinpoint`. This rule does not require the instruction to be untagged; it was tagged by the rule matching the first advice. The rule only matches point-cuts that have no `ref` edge to the `Joinpoint` node yet and, when applied, the rule adds this edge to the selected point-cut.

To illustrate the mechanism of advice execution, we explain the content of the stacks in a number of stages in the execution of the second example scenario, where the *setGrade* method of *NamedExam* is called with argument *Zero*.

The figures are shown in Figure 4.19:

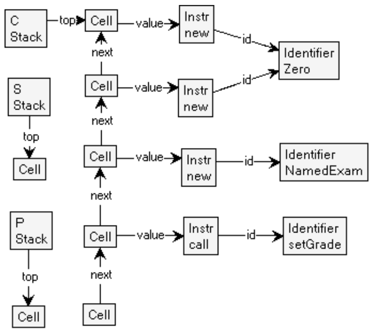
- *(a). Initial state*: The continuation stack C contains the main expression (see Fig. 4.2 line 28), which is sequentialised into:

  $$\text{NEW}_{Zero};\text{NEW}_{Zero};\text{NEW}_{NamedExam};\text{CALL}_{setGrade}.$$

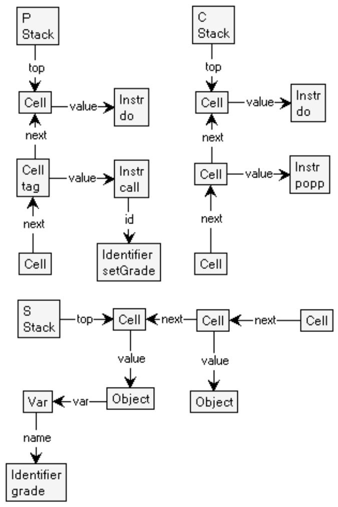
  Stack P and S are empty.

- *(b). After the two advices are scheduled*: The S stack contains the Zero object and the NamedExam object, with the `Var` representing grade zero. The continuation stack contains, from top to bottom: a DO instruction that will invoke the first advice; a POPP that pops any remaining instructions for the join point from the proceed stack. The proceed stack contains, from top to bottom: a DO instruction for the second advice, such that proceed in the first advice will execute the second advice; the tagged CALL instruction for setGrade, that will be executed by proceed in the second advice.

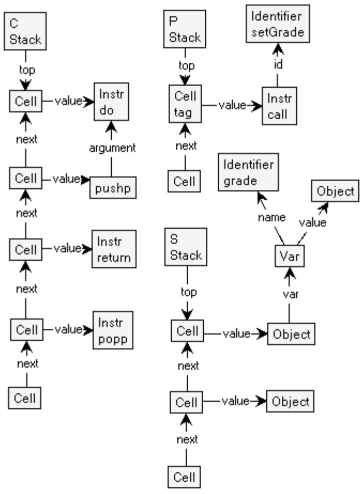
- *(c). After the first* PROCEED *instruction*: Since proceed is called on the initial receiver of the intercepted CALL, and has the arguments of the intercepted call as arguments, the S stack has not changed w.r.t. the previously described state. The proceed stack now contains only the CALL instruction that triggered the advices. The continuation stack contains, from top to bottom: the DO instruction that will invoke the second advice; a PUSHP instruction that will push the DO instruction back on the proceed stack (after



(a) Start state



(b) After scheduling advices



(c) After first proceed

Figure 4.19: Run-time part of Three States

the advice has returned); the RETURN instruction of the first advice; and the POPP instruction that will clean up the proceed stack.

4.5 Correctness of the Semantics

To increase confidence in the correctness of our semantics, we show that it coincides with the formal specification of the AFJ language in SOS (Structural Operational Semantics) style in [DDF08]. We will explain the proof strategy and the desired result of the proof. The proof itself, however, is only given partly to provide an intuition.

4.5.1 Notion of Correctness

We claim that the following correspondence holds between the applications of the SOS rule and the GT rules:

$$\begin{tikzcd}
(C, S, \Sigma, P) \arrow[r, "\text{SOS derivation}"] \arrow[d, "\mathit{Tra}"'] & (C', S', \Sigma', P') \arrow[d, "\mathit{Tra}"] \\
G \arrow[r, "\text{graph derivation sequence}"] & G'
\end{tikzcd}$$

The picture can be read top-down or bottom-up: for all single SOS derivations, there is a corresponding sequence of graph derivations, and for all sequences of graph derivations *between non-intermediate graphs* there is a corresponding SOS derivation — where a graph is intermediate if it is in between a number of graph derivations that together correspond to the SOS derivation, i.e. when a SOS rule is specified using more than one graph transformation rule.

In fact, we claim that the translation function is a *weak bisimulation* between SOS configurations and graphs. This means that intermediate graphs have a correspondence to the same SOS configuration as the resulting graph of a sequence of GT rule application.

For example, the SOS rule for the CALL instruction corresponds to a visible GT rule application of the CALL rule, followed by a number of internal rule applications for parameter transfer and method body pushing. These "internal" rules are always applied first, before the next instruction is executed. Thereby, the intermediate graphs implicitly correspond to the same SOS configuration as the final graph of the sequence.

To prove that Tra is a weak bisimulation we need to show that one SOS rule application corresponds to one or a sequence of GT rule applications. This involves showing:

- (1) The SOS rule *applies* if and only if the GT rule applies to the corresponding graph.
- (2) The *effect* of the SOS rule is "the same" as the effect of the GT rule (sequence), meaning that the resulting configuration and state again correspond w.r.t. the translation function.

We define a mapping from the configurations in the SOS semantics to graphs. (It should be noted that our language is a slight adaptation of that in [DDF08]; the most important difference is that we only allow call point-cuts, whereas they can define point-cuts for arbitrary instructions; on the other hand, we include parameters into the advice, which they do not. To establish correctness, we use an accordingly modified version of the SOS semantics.)

Then we give the actual proof, by illustrating that (1) and (2) hold for all SOS rules. For those derivations that correspond to a single graph transformation rule, we will see that the translated graphs for the configurations before and after $\rightarrow_b$ correspond to the LHS and RHS (see Chapter 2) of the graph transformation rule, respectively. Thereby, correspondence is automatically proved. We show this for the GET and SET instructions. For the sake of keeping this thesis interesting, we omit the proof for the RETURN instruction. Proof of correspondence for NEW and CALL is a little more complicated, as these derivations correspond to a sequence of rule applications.

4.5.2 The SOS Semantics

The static structure of a given FAJ program is captured by three partial functions:

- $FDecl : T \rightarrow (Ident \times T)^*$, yielding for each class type the sequence of field declarations (where $Ident$ is the universe of identifiers);
- $MDecl : (T \times Ident) \rightharpoonup (Ident \times T)^* \times T \times Expr$, yielding for each class type the corresponding method declarations. A method declaration consists of a list of parameters (pairs of identifiers and types), a return type and a method body.
- $ADecl : (T \times Ident) \rightarrow \mathcal{P}((Ident \times T)^* \times Expr)$, yielding for each pair of class and method identifier the set of aspects that statically match calls of that method.

For every program, this triple of functions together with the initial expression plays exactly the same role as the initial graph discussed in Sect. 4.3, except that there all expressions (i.e., the initial expression and the bodies of all methods and advices) have been sequentialised as discussed in Section 4.2.2. In fact, there is a straightforward translation from each valid combination $(\mathit{FDecl}, \mathit{MDecl}, \mathit{ADecl}, \mathit{Expr})$ to an aspect program graph; an intuition can be gained from the type graph of Figure 4.4. We define the translation in the next subsection. Below we use `Instr`, `Ident` and `Class` to denote the sets of nodes corresponding to *Instr*, *Ident* and T in the program.

The *dynamic* structure, i.e., the states of the program, are encoded in the SOS semantics as *configurations* (S, C, Σ, P) consisting of the same stacks and store as in our graph-based semantics:

- $C \in (\mathit{Instr} \times \mathit{Bool})^*$ is the continuation stack, containing the instructions to be executed, combined with booleans indicating whether the instruction has already been advised (corresponding to the `tag`-edge used for this purpose in Section 4.4.2).;
- $S \in \mathsf{Object}^*$ is the value stack, containing intermediate results;
- $\Sigma : \mathsf{Object} \to T \times (\mathit{Ident} \rightharpoonup \mathsf{Object})$ is the heap, containing the run-time type and field values of all objects;
- $P \subseteq (\mathit{Instr} \times \mathit{Bool})^*$ is the proceed stack, containing the (tagged or untagged) advices scheduled to be executed.

On the basis of these configurations, the SOS semantics consists of two types of rules: first, rules to sequentialise expressions to their corresponding instructions; and second, rules modelling the execution of the instructions. In our semantics we have chosen to do sequentialisation as part of the pre-processing in Section 4.2.2; for the purpose of showing correctness in this section, we assume the same has happened on the SOS semantics side; that is, we assume that all expressions are already transformed into sequences of instructions.

The execution rule of instruction INSTR always has the form

$$\frac{\mathit{side\ conditions}}{(\textsc{instr} : C, S, \Sigma, P) \to (C', S', \Sigma', P')}$$

meaning that, if the side conditions are fulfilled, a configuration in which the first instruction on the continuation stack is INSTR can perform a step, changing into the configuration on the right hand side. For instance, the rules for GET, SET, CALL, NEW and RETURN are:

$$\textsc{get}\quad \frac{\Sigma(v) = (T, F) \quad F(f) = v_2}{(\textsc{get}f : C, v : S, \Sigma, P) \rightarrow_b (C, v_2 : S, \Sigma, P)}$$

$$\textsc{set}\quad \frac{\Sigma(v_0) = (T, F)}{\begin{array}{l}(\textsc{set}_f : C, v_0 : v : S, \Sigma, P) \\ \quad \rightarrow_b (C, v : S, \Sigma[v_0 \mapsto (T, F[f \mapsto v])], P)\end{array}}$$

$$\textsc{call}\quad \frac{\Sigma(v_0) = (T, F) \quad \mathit{MDecl}(m, T) = ((x_1, \ldots, x_n), e)}{\begin{array}{l}(\textsc{call}^T_m : C, v_0 : v_1 : \ldots : v_n : S, \Sigma, P) \\ \quad \rightarrow_b (e[x_1/v_1, \ldots, x_n/v_n], \mathit{this}/v_0 : C, S, \Sigma, P)\end{array}}$$

$$\textsc{new}\quad \frac{\begin{array}{l}v \notin \mathit{dom}(\Sigma) \quad \mathit{FDecl}(T) = (f_1, T_1), \ldots, (f_n, T_n) \\ F = [f_1, \ldots, f_n \mapsto v_1, \ldots, v_n]\end{array}}{\begin{array}{l}(\textsc{new}_X : C, v_1 : \ldots, v_n : S, \Sigma, P) \\ \quad \rightarrow_b (C, v : S, \Sigma[v \mapsto (T, F)], P)\end{array}}$$

$$\textsc{return}\quad \frac{}{(\textsc{return} : C, S, \Sigma, P) \rightarrow_b (C, S, \Sigma, P)}$$

Note that in these rules, the continuation stack elements are given as plain instructions rather than pairs of instructions and booleans; this is to indicate that we do not care about the instruction tags.

The P-stack is only used for advice execution. Two example rules are given below: the AROUND-rule to schedule advice execution, and the rule for executing PROCEED.

$$\textsc{around}\quad \frac{\mathit{ADecl}(T, m) = \{a_1 \ldots a_n\}}{\begin{array}{l}((\textsc{call}^T_m, \mathbf{ff}) : C, \Sigma, P) \rightarrow \\ \quad (\textsc{do}_{a_1} : \textsc{popp} : C, \Sigma, \\ \qquad \textsc{do}_{a_2} : \ldots : \textsc{do}_{a_n} : (\textsc{call}_m, \mathbf{tt}) : P)\end{array}}$$

$$\textsc{proceed}\quad \frac{}{\begin{array}{l}(\textsc{proceed} : C, \Sigma, i : P) \\ \quad \rightarrow (i : \textsc{pushpp}_i : C, \Sigma, P)\end{array}}$$

$$\textsc{popp}\quad \frac{}{\begin{array}{l}(\textsc{popp}_n : C, \Sigma, i_i, \ldots, i_n : P) \\ \quad \rightarrow (C, \Sigma, P)\end{array}}$$

A derivation for PUSHP$_n$ has not been specified in the SOS semantics. Instead, it simply states that the instruction i is pushed back on the proceed stack. We can therefore only assume that the GT rule is correct.

4.5.3 From Programs to Graphs

The translation from each valid FAJ program to a graph is defined by

$$\mathit{Tra} : (\mathit{FDecl}, \mathit{MDecl}, \mathit{ADecl}, \mathit{Expr}) \mapsto [\![F]\!] \cup [\![M]\!] \cup [\![A]\!] \cup [\![E]\!]$$

where $[\![F]\!]$, $[\![P]\!]$, $[\![A]\!]$ and $[\![E]\!]$ are the graphs corresponding to the individual data structures; the combined graph is the union of these. The individual graphs in turn are defined as follows:

- For *FDecl*, we assume a set of nodes `Class` corresponding to the types in *dom*(*FDecl*), a set of nodes `Ident` and a set of nodes `Field` corresponding to the fields. These will be encoded as a combination of $t \in$ `Class` and a field declared in t:

  $$\mathtt{Field} = \{(t, id, t') \mid \mathit{FDecl}(t) = F \quad (id, t') \in F\}$$

  Using this set of nodes, the graph for *FDecl* is defined by:

  $$\begin{aligned}[\![F]\!] &= (\mathtt{Class} \cup \mathtt{Field} \cup \mathtt{Ident}, E_F) \quad \text{where}\\ E_F &= \{(t, \mathtt{field}, f) \mid f = (t, id, t')\} \;\cup\\ &\quad\; \{(f, \mathtt{name}, id) \mid f = (t, id, t')\} \;\cup\\ &\quad\; \{(t, \mathtt{next}, f) \mid f = (t, id, t'), \mathit{FDecl}(t)[0] = f\} \;\cup\\ &\quad\; \{(f, \mathtt{next}, f') \mid f = (t, id, t'), \mathit{FDecl}(t)[i] = f, \mathit{FDecl}(t)[i+i] = f'\}\end{aligned}$$

  Notice that we do not translate the type of fields; these are not required for the transformations. (For simplicity reasons, we ignore *EoL* nodes).

- For *MDecl*, we represent methods as a set of nodes `Method` ; these will be encoded (T, m), where $T \in$ `Class` and m is a method declared in T. Due to the way *MDecl* is defined, we need to extend our program graphs; for all methods declared in classes with subclasses, we create `Method` nodes for each subclass of the type the method is declared in. Thereby, each method that can be called on an object is directly available in the type of the object. We can then in fact use the rule *call_mbody_match* immediately when doing a method-lookup, without ever having to use *call_mbody_up*.

  We assume a set of nodes `MethodBody` corresponding to the expressions e of methods m, where $\mathit{MDecl}(m) = (P, t', e)$. We also need nodes to represent parameters; these will be encoded as pairs (m, p, T), where m is a method and p is a parameter of type T declared for m:

  $$\begin{aligned}\mathtt{Method} &= \{(t, id) \mid (t, id) \in \mathit{dom}(\mathit{MDecl})\}\\ \mathtt{Parameter} &= \{(m, id, t) \mid \mathit{MDecl}(m) = (P, t', e), (id, t) \in P\}\end{aligned}$$

Using this set of nodes, the graph for *MDecl* is defined by:

$$[\![M]\!] = (\texttt{Class} \cup \texttt{Method} \cup \texttt{Parameter} \cup \texttt{MethodBody} \\ \cup \texttt{Instr} \cup \texttt{Ident}, E_M) \quad \text{where}$$

$$\begin{aligned} E_M = \; & \{(t, \texttt{method}, m) \mid m = (t, id)\} \cup \\ & \{(m, \texttt{field}, id) \mid m = (t, id)\} \cup \\ & \{(m, \texttt{parameter}, p) \mid m = (t, id), p = (m, id, t)\} \cup \\ & \{(p, \texttt{name}, id) \mid p = (m, id, t)\} \cup \\ & \{(m, \texttt{next}, p) \mid \mathit{MDecl}(m) = (P, t', e), P[0] = p\} \cup \\ & \{(p, \texttt{next}, p') \mid p = (m, id, t), p' = (m, id', t'), \\ & \quad \mathit{MDecl}(m) = (P, t', e), P[i] = (id, t), P[i+1] = (id', t')\} \cup \\ & \{(m, \texttt{body}, e) \mid m = (t, id), \mathit{MDecl}(m) = (P, t', e)\} \cup \\ & \{(e, \texttt{next}, ins) \mid S(e)[0] = ins\} \cup \\ & \{(ins, \texttt{next}, ins') \mid S(e)[i] = ins, S(e)[i'] = ins'\} \end{aligned}$$

Here, S is the sequentialisation function of Figure 4.3. We require that instructions are never used twice; they are distinct and consist of the instruction type and a unique index. Notice that we do not translate the return type of a method.

- For *ADecl* , we need a set of nodes `Pointcut` that are encoded by their signature (t, id) representing a call to a method with identifier id in type t. Advices are methods and are translated to graphs the same as for *MDecl*, except that the sequentialised expression may contain a PROCEED instruction. Also, advices do not have a signature like methods, because they are not declared in a class. For simplicity, we encode instructions as pairs of a type and a list of parameters; CALL instructions are encoded as $(\text{CALL}, (t, id))$, with id a method identifier and t the type of the target object; DO instructions are encoded (DO, a), where a is an advice (i.e. a method).

  The graph for *ADecl* is defined by:

$$[\![A]\!] = (\texttt{Class} \cup \texttt{Pointcut} \cup \texttt{Method} \cup \texttt{Instr} \cup \texttt{Ident}, E_A) \quad \text{where}$$

$$\begin{aligned} E_A = \; & \{(pc, \texttt{matchstatic}, ins) \mid pc = (t, id), ins = (\text{CALL}, (t, id))\} \cup \\ & \{(pc, \texttt{field}, ins) \mid m = (t, id), ins = (\text{DO}, a), a \in \mathit{ADecl}(t, id)\} \cup \\ & \{(ins, \texttt{name}, a) \mid ins = (\text{DO}, a)\} \cup \end{aligned}$$

4.5.4 From Configurations to Graphs

The translation of SOS configurations to graphs is defined by

$$\mathit{Tra} : (C, S, \Sigma, P) \mapsto [\![C]\!] \cup [\![S]\!] \cup [\![\Sigma]\!] \cup [\![P]\!]$$

where $[\![C]\!]$, $[\![P]\!]$ etc. are the graphs corresponding to the individual data structures; the combined graph is the union of these. The individual graphs in turn are defined as follows:

- For each of the stacks, we introduce a single special `Stack`-node with the name (C,P, or S) of the stack, which stands for the stack as a whole, and `Cell`-nodes that stand for the stack positions. As representatives we can use integer numbers:

$$\begin{aligned}
&\texttt{Stack} = \{\text{"}C\text{"}, \text{"}S\text{"}, \text{"}P\text{"}\}\\
&\texttt{Cell} \;\; = \{0, \ldots, n\} \quad \text{where } n \text{ is the stack size}
\end{aligned}$$

  The nodes are linked with `top`-, `next`- and `value`-edges in accordance with the type graph of Figure 4.6. Using $|C|$ to denote the size of C and C^i to denote the value at position i (where the bottom position is numbered 0 and the top $|C| - 1$), the formal definition is:

$$\begin{aligned}
\llbracket C \rrbracket &= (\texttt{Stack} \cup \texttt{Cell} \cup \texttt{Instr}, E_C) \quad \text{where}\\
E_C &= \{(\text{"}C\text{"}, \texttt{top}, |C| - 1)\} \;\cup\\
&\quad\; \{(i, \texttt{next}, i - 1) \mid 0 < i < |C|\} \;\cup\\
&\quad\; \{(i, \texttt{value}, x) \mid 0 < i < |C|, C^i = (x, b)\} \;\cup\\
&\quad\; \{(i, \texttt{tag}, i) \mid 0 < i < |C|, C^i = (x, \texttt{tt})\}
\end{aligned}$$

  Note that stacks always contain a spurious `Cell`-node for the sake of uniformity, so that even the empty stack has a `top`-edge.

  The P-stack is encoded in the same way; so is the S-stack, except that the `value`-edges point to `Objects`, and no `tag`-edges are required.

- For the store, we assume a set of nodes `Object` corresponding to the objects in $dom(\Sigma)$, that is, those objects that are actually allocated on the heap. We also need auxiliary nodes to represent the object fields; these will be encoded as pairs (o, f) where $o \in$ `Object` and f is a field declared for o's type:

$$\texttt{Var} = \{(o, id) \mid \Sigma(o) = (t, Fd), id \in dom(Fd)\}$$

  Using this set of nodes, the graph for Σ is defined by

$$\begin{aligned}
\llbracket \Sigma \rrbracket &= (\texttt{Class} \cup \texttt{Object} \cup \texttt{Ident} \cup \texttt{Var}, E_\Sigma) \quad \text{where}\\
E_\Sigma &= \{(v, \texttt{name}, id) \mid v = (o, id)\} \;\cup\\
&\quad\; \{(o, \texttt{var}, v) \mid v = (o, id)\} \;\cup\\
&\quad\; \{(v, \texttt{value}, o) \mid v = (o', id), Fd(o') = o\} \;\cup\\
&\quad\; \{(o, \texttt{type}, t) \mid \Sigma(o) = (t, Fd)\}
\end{aligned}$$

4.5.5 Proof of Correctness

We now give the actual proof, which involves showing for each SOS rule that:

- The SOS rule applies iff the GT rule sequence applies to the corresponding graph;
- The effect of the SOS rule is "the same".

get correspondence

First, we show correspondence between a GET derivation and the *get* rule in Figure 4.7. First, we give the graph translation of the requirements of the inference rule, i.e. the part on the left of the $\rightarrow_b$ arrow. We denote this graph $[\![\textsc{get}]\!]$. The graph after $\rightarrow_b$ is denoted $[\![\textsc{get}]\!]'$. The graphs consist of a set of nodes and edges, where the set of nodes are defined by the *Tra* functions for programs and configurations. We only give the set of edges. We omit E_P for base language instructions. We use $k = |E_C|, m = |E_S|, o = |E_P|$.

$$\begin{aligned}
[\![\textsc{get}]\!] &= (N, E_C \cup E_S \cup E_\Sigma) \quad \text{where} \\
E_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \;\cup \\
&\quad\; \{(|C|-1, \texttt{value}, (\textsc{get}, f))\} \;\cup \\
&\quad\; \{(|C|-1, \texttt{next}, k-1) \\
E_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup \\
&\quad\; \{(m, \texttt{value}, v)\} \;\cup \\
&\quad\; \{(m, \texttt{next}, m-1)\} \\
E_\Sigma &= \{(v, \texttt{type}, T)\} \;\cup \\
&\quad\; \{(v, \texttt{var}, (v, f))\} \;\cup \\
&\quad\; \{((v, f), \texttt{name}, f)\} \;\cup \\
&\quad\; \{((v, f), \texttt{value}, v_2)\}
\end{aligned}$$

This exactly corresponds to the left-hand-side of the rule. After application of $\rightarrow_b$, the configuration is translated to the following graph:

$$\begin{aligned}
[\![\textsc{get}]\!]' &= (N', E'_C \cup E'_S \cup E'_\Sigma) \quad \text{where} \\
E'_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \\
E'_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup \\
&\quad\; \{(m, \texttt{value}, v_2) \;\cup \\
&\quad\; \{(m, \texttt{next}, m-1)\} \\
E'_\Sigma &= \{(v, \texttt{type}, T)\} \;\cup \\
&\quad\; \{(v, \texttt{var}, (v, f))\} \;\cup \\
&\quad\; \{((v, f), \texttt{name}, f)\} \;\cup \\
&\quad\; \{((v, f), \texttt{value}, v_2)\}
\end{aligned}$$

This corresponds to the right-hand-side of the rule; therefore we have proved that the correspondence holds.

set correspondence

Now, we show correspondence of a SET derivation and the *set* rule. Again, the node set N is defined by the edgeset E and we omit P.

$$
\begin{aligned}
&[\![\textsc{set}]\!] = (N, E_C \cup E_S \cup E_\Sigma) \quad \text{where} \\
&\quad E_C = \{(\text{"}C\text{"}, \mathtt{top}, k)\} \cup \\
&\qquad\quad \{(k, \mathtt{value}, (\textsc{set}, f))\} \cup \\
&\qquad\quad \{(k, \mathtt{next}, k-1) \\
&\quad E_S = \{(\text{"}S\text{"}, \mathtt{top}, m)\} \cup \\
&\qquad\quad \{(m, \mathtt{value}, v_0)\} \cup \\
&\qquad\quad \{(m, \mathtt{next}, m-1)\} \cup \\
&\qquad\quad \{(m-1, \mathtt{value}, v)\} \cup \\
&\qquad\quad \{(m-1, \mathtt{next}, m-2)\} \\
&\quad E_\Sigma = \{(v_0, \mathtt{type}, T)\} \cup \\
&\qquad\quad \{(v_0, \mathtt{var}, (v_0, f))\} \cup \\
&\qquad\quad \{((v_0, f), \mathtt{name}, f)\} \cup \\
&\qquad\quad \{((v_0, f), \mathtt{value}, o)\}
\end{aligned}
$$

Since all variables (field instances) in FAJ programs must have a value, we represent this unmentioned value by an object o. This exactly corresponds to the left-hand-side of the rule. After application of $\to_b$, the configuration is translated to the following graph:

$$
\begin{aligned}
&[\![\textsc{set}]\!]' = (N', E'_C \cup E'_S \cup E'_\Sigma) \quad \text{where} \\
&\quad E'_C = \{(\text{"}C\text{"}, \mathtt{top}, k)\} \\
&\quad E'_S = \{(\text{"}S\text{"}, \mathtt{top}, m)\} \cup \\
&\qquad\quad \{(m, \mathtt{value}, v)\} \cup \\
&\qquad\quad \{(m, \mathtt{next}, m-1)\} \\
&\quad E'_\Sigma = \{(v_0, \mathtt{type}, T)\} \cup \\
&\qquad\quad \{(v_0, \mathtt{var}, (v_0, f))\} \cup \\
&\qquad\quad \{((v_0, f), \mathtt{name}, f)\} \cup \\
&\qquad\quad \{((v_0, f), \mathtt{value}, v)\}
\end{aligned}
$$

This graph corresponds to C, S and Σ in the right-hand-side of the rule; therefore we have proved that the correspondence holds. Indeed, the graph deletes the current value of field f in v_0, and sets the new value v.

call correspondence

We now show that the SOS derivation for the CALL instruction corresponds to a sequence of rule applications consistent with the following control expression:

```
call;
call_pushcode_match;
alap { call_initparam };
call_pushcode_start;
until ( call_pushcode_end ) do { call_pushcode }
```

We start with the graph translated from the SOS derivation before $\rightarrow_b$:

$$
\begin{aligned}
[\![\textsc{call}]\!] &= (N, E_C \cup E_S \cup E_\Sigma) \quad \text{where} \\
E_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \cup \\
&\quad\ \{(k, \texttt{value}, (\textsc{call}, (T, m)))\} \cup \\
&\quad\ \{(k, \texttt{next}, k-1) \\
E_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \cup \\
&\quad\ \{(m-i, \texttt{value}, v_i) \mid 0 \le i \le n\} \cup \\
&\quad\ \{(m-i, \texttt{next}, m-i-1) \mid 0 \le i \le n+1\} \\
E_\Sigma &= \{(v_0, \texttt{type}, T)\}
\end{aligned}
$$

Besides the configuration, we know that there is a $(m, T) \in$ `Method` connected to $T \in$ `Class`. This method has `Parameter` nodes for $x_1, \ldots, x_n$, and a `MethodBody` connected to a sequence of instructions that corresponds to $S(e)$.

The derivation yields an unsequentialised expression on the continuation stack. In the SOS semantics, this expression is sequentialised on-the-fly. Also, usage of method arguments is represented as a substitution of elements of the expression with arguments of the method-call. For the sake of simplicity, we assume that $e[x_1/v_1, \ldots, x_n/v_n]$ corresponds to $S(e)$, with e containing $\textsc{var}_x$ instructions such that these variables yield the substituted values. Now, the configuration after $\rightarrow_b$ is translated to the following graph, ignoring parameters:

$$
\begin{aligned}
[\![\textsc{call}]\!]' &= (N, E_C \cup E_S \cup E_\Sigma) \quad \text{where} \\
E'_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \cup \\
&\quad\ \{(k-i, \texttt{value}, S(e)[i]) \mid 0 \le i < |S(e)|\} \cup \\
&\quad\ \{(k-i, \texttt{next}, k-i-1) \mid 0 \le i < |S(e)| \\
E'_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \\
E'_\Sigma &= \{(v_0, \texttt{type}, T)\}
\end{aligned}
$$

Now we show that the corresponding graph transformation yield a corresponding graph. From (N, E) we first apply the *call* rule. The top of C is removed; the rest of the rule adds control information that triggers rule *call_mbody_match* for the method with identifier m in class T. This yields the following configuration:

$$\begin{aligned}
&[\![\textsc{call}]\!]^1 = (N^1, E^1_C \cup E^1_S \cup E^1_\Sigma) \quad \text{where} \\
&\quad E^1_C = \{(\text{"}C\text{"}, \texttt{top}, k)\} \\
&\quad E^1_S = \{(\text{"}S\text{"}, \texttt{top}, m)\} \ \cup \\
&\qquad\qquad \{(m - i, \texttt{value}, v_i) \mid 0 \leq i \leq n\} \ \cup \\
&\qquad\qquad \{(m - ii, \texttt{next}, m - i - 1) \mid 0 \leq i \leq n + 1\} \\
&\quad E^1_\Sigma = \{(v_0, \texttt{type}, T)\}
\end{aligned}$$

Next, the *call_mbody_match* rule binds the value for the *this* variable and triggers parameter transfer at the correct `Method` node. This yields a `Scope` node with connected `Var` nodes for all parameters and *this*, later to be used by VARinstructions. We leave the correspondence of parameter transfer in our semantics and parameter substitution in the SOS semantics at that. After parameter transfer, the graph of the configuration is:

$$\begin{aligned}
&[\![\textsc{call}]\!]^2 = (N^2, E^2_C \cup E^2_S \cup E^2_\Sigma) \quad \text{where} \\
&\quad E^2_C = \{(\text{"}C\text{"}, \texttt{top}, k)\} \\
&\quad E^2_S = \{(\text{"}S\text{"}, \texttt{top}, 0)\} \\
&\quad E^2_\Sigma = \{(v_0, \texttt{type}, T)\}
\end{aligned}$$

Stripped down to an almost empty configuration, we want to stress that the graph also encodes values for the method parameters. Also, the final application of *call_initparam* triggers the *call_pushcode_start* rule. This rule creates control information to trigger the *call_pushcode* rule for all instructions in $S(e)$ in a backwards fashion until *call_pushcode_end* can be applied. We show the resulting configurations after the first application of *call_pushcode* and after the application of *call_pushcode_end*. After pushing the last instruction of $S(e)$:

$$\begin{aligned}
&[\![\textsc{call}]\!]^3 = (N^3, E^3_C \cup E^3_S \cup E^3_\Sigma) \quad \text{where} \\
&\quad E^3_C = \{(\text{"}C\text{"}, \texttt{top}, k)\} \ \cup \\
&\qquad\qquad \{(k, \texttt{value}, ins) \mid S(e)[k] = ins, k = |S(e)| - 1\} \ \cup \\
&\qquad\qquad \{(k, \texttt{next}, k - 1)\} \\
&\quad E^3_S = \{(\text{"}S\text{"}, \texttt{top}, m)\} \\
&\quad E^3_\Sigma = \{(v_0, \texttt{type}, T)\}
\end{aligned}$$

After all instruction have been pushed onto C, the final configuration is:

$$\begin{aligned}
&[\![\textsc{call}]\!]^4 = (N^4, E^4_C \cup E^4_S \cup E^4_\Sigma) \quad \text{where} \\
&\quad E^4_C = \{(\text{"}C\text{"}, \texttt{top}, k)\} \ \cup \\
&\qquad\qquad \{(k - i, \texttt{value}, S(e)[i]) \mid 0 \leq i \leq |S(e)| - 1\} \ \cup \\
&\qquad\qquad \{(k - i, \texttt{next}, k - i - 1) \mid 0 \leq i \leq |S(e)|\} \\
&\quad E^4_S = \{(\text{"}S\text{"}, \texttt{top}, m)\} \\
&\quad E^4_\Sigma = \{(v_0, \texttt{type}, T)\}
\end{aligned}$$

Thus, $[\![\textsc{call}]\!]' = [\![\textsc{call}]\!]^4$ and we have proved the correspondence (assuming the shortcuts in our proof for some of the invisible steps are correct).

new correspondence

Next, we show that the derivation for the NEW instruction corresponds to a sequence of graph transformations. The configuration translated to a graph before the derivation is:

$$\begin{aligned}
[\![\textsc{new}]\!] &= (N, E_C \cup E_S \cup E_\Sigma) \quad \text{where}\\
E_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \;\cup\\
&\quad\; \{(k, \texttt{value}, (\textsc{new}, X))\} \;\cup\\
&\quad\; \{(k, \texttt{next}, k-1)\\
E_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup\\
&\quad\; \{(m-i, \texttt{value}, v_{i+1}) \mid 0 \le i \le n-1\} \;\cup\\
&\quad\; \{(m-i, \texttt{next}, m-i-1) \mid 0 \le i \le n\}\\
E_\Sigma &= \emptyset
\end{aligned}$$

Also, the graph contains a translation of $FDecl(X) = (f_1, T_1), \ldots, (f_n, T_n)$. The translated configuration after $\rightarrow_b$ is:

$$\begin{aligned}
[\![\textsc{new}]\!]' &= (N', E'_C \cup E'_S \cup E'_\Sigma) \quad \text{where}\\
E'_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\}\\
E'_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup\\
&\quad\; \{(m, \texttt{value}, v)\} \;\cup\\
&\quad\; \{(m, \texttt{next}, m-1)\}\\
E'_\Sigma &= \{(v, \texttt{type}, X)\} \;\cup\\
&\quad\; \{(v, \texttt{var}, (v, f_i)) \mid 1 \le i \le n\} \;\cup\\
&\quad\; \{((v, f_i), \texttt{value}, v_i) \mid 1 \le i \le n\}
\end{aligned}$$

Now, we show that the derivation corresponds to rule applications consisting with the following control expression:

```
new; alap{new_initfield}; new_end
```

First, we show the configuration graph after application of the *new* rule:

$$\begin{aligned}
[\![\textsc{new}]\!]^1 &= (N^1, E^1_C \cup E^1_S \cup E^1_\Sigma) \quad \text{where}\\
E^1_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\}\\
E^1_S &= \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup\\
&\quad\; \{(m-i, \texttt{value}, v_{i+1}) \mid 0 \le i \le n-1\} \;\cup\\
&\quad\; \{(m-i, \texttt{next}, m-i-1) \mid 0 \le i \le n\}\\
E^1_\Sigma &= \{(v, \texttt{type}, X)\}
\end{aligned}$$

Also, the graphs contains control information to trigger the *new_initfield* rule for the first parameter and the newly created object v. After the first application of *new_initfield*, the configuration graphs is:

$$\begin{aligned}
&[\![\textsc{new}]\!]^2 = (N^2, E^2_C \cup E^2_S \cup E^2_\Sigma) \quad \text{where}\\
&\quad E^2_C = \{(\text{"}C\text{"}, \texttt{top}, k)\}\\
&\quad E^2_S = \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup\\
&\qquad\qquad \{(m-i, \texttt{value}, v_{i+2}) \mid 0 \le i \le n-2\} \;\cup\\
&\qquad\qquad \{(m-i, \texttt{next}, m-i-1) \mid 0 \le i \le n-1\}\\
&\quad E^2_\Sigma = \{(v, \texttt{type}, X)\} \;\cup\\
&\qquad\qquad \{(v, \texttt{var}, (v, f_1))\} \;\cup\\
&\qquad\qquad \{((v, f_1), \texttt{value}, v_1)\}
\end{aligned}$$

The rule is applied exactly n times. Then, all fields are initialised and *new_end* can be applied. The resulting configuration graph is:

$$\begin{aligned}
&[\![\textsc{new}]\!]^3 = (N^3, E^3_C \cup E^3_S \cup E^3_\Sigma) \quad \text{where}\\
&\quad E^3_C = \{(\text{"}C\text{"}, \texttt{top}, k)\}\\
&\quad E^3_S = \{(\text{"}S\text{"}, \texttt{top}, m)\} \;\cup\\
&\qquad\qquad \{(m, \texttt{value}, v)\} \;\cup\\
&\qquad\qquad \{(m, \texttt{next}, m-1)\}\\
&\quad E^3_\Sigma = \{(v, \texttt{type}, X)\} \;\cup\\
&\quad = \qquad \{(v, \texttt{var}, (v, f_i)) \mid 1 \le i \le n\} \;\cup\\
&\quad = \qquad \{((v, f_i), \texttt{value}, v_i) \mid 1 \le i \le n\}
\end{aligned}$$

Because $[\![\textsc{new}]\!]' = [\![\textsc{new}]\!]^3$, we have proved correspondence.

around correspondence

Next, we now show correspondence of the AROUND derivation. The configuration before the derivation is translated to the following graph:

$$\begin{aligned}
&[\![\textsc{around}]\!] = (N, E_C \cup E_S \cup E_\Sigma \cup E_P) \quad \text{where}\\
&\quad E_C = \{(\text{"}C\text{"}, \texttt{top}, k)\} \;\cup\\
&\qquad\qquad \{(k, \texttt{value}, ((\textsc{call}, (T, m)), \mathbf{ff}))\} \;\cup\\
&\qquad\qquad \{(k, \texttt{next}, k-1)\}\\
&\quad E_S = \{(\text{"}S\text{"}, \texttt{top}, m)\}\\
&\quad E_\Sigma = \emptyset\\
&\quad E_P = \{(\text{"}P\text{"}, \texttt{top}, o)\}
\end{aligned}$$

Although we know that Σ contains the target of the CALL and optionally values for the parameters of the CALL , we can assume they do not exist for this derivation. Also, besides the configuration, we know that the graph contains a translation of $ADecl(T,m) = \{a_1,\ldots,a_n\}$, with DO instructions for all advices. After the derivation, the configuration is translated to the following graph:

$$\begin{aligned}
&[\![\text{AROUND}]\!]' = (N', E'_C \cup E'_S \cup E'_\Sigma \cup E'_P) \quad \text{where}\\
&\quad E'_C = \{(\text{"}C\text{"}, \texttt{top}, l)\} \cup\\
&\qquad\quad \{(k, \texttt{value}, (\text{DO}, a_1))\} \cup\\
&\qquad\quad \{(k-1, \texttt{value}, (\text{POPP}))\} \cup\\
&\qquad\quad \{(k, \texttt{next}, k-1)\} \cup\\
&\qquad\quad \{(k-11, \texttt{next}, k-2)\}\\
&\quad E'_S = \{(\text{"}S\text{"}, \texttt{top}, m)\}\\
&\quad E'_\Sigma = \emptyset\\
&\quad E'_P = \{(\text{"}P\text{"}, \texttt{top}, o)\} \cup\\
&\qquad\quad \{(o-i, \texttt{value}, (\text{DO}, a_{i+2})) \mid 0 \le i \le n-2\} \cup\\
&\qquad\quad \{(o-n+1, \texttt{value}, ((\text{CALL}, (T,m)), \texttt{tt}))\} \cup\\
&\qquad\quad \{(o-i, \texttt{next}, o-i-1) \mid 0 \le i \le n-1\}
\end{aligned}$$

We show that this corresponds to one application of the *around* rule followed by $n-1$ applications of the *around_more* rule. After the *around* rule, the graph contains the following configuration:

$$\begin{aligned}
&[\![\text{AROUND}]\!]^1 = (N^1, E^1_C \cup E^1_S \cup E^1_\Sigma \cup E^1_P) \quad \text{where}\\
&\quad E^1_C = && \{(\text{"}C\text{"}, \texttt{top}, k)\} \cup\\
& && \{(k, \texttt{value}, (\text{DO}, a_1))\} \cup\\
& && \{(k-1, \texttt{value}, (\text{POPP}))\} \cup\\
& && \{(k, \texttt{next}, k-1)\} \cup\\
& && \{(k-1, \texttt{next}, k-2)\}\\
&\quad E^1_S = && \{(\text{"}S\text{"}, \texttt{top}, m)\}\\
&\quad E^1_\Sigma = && \emptyset\\
&\quad E^1_P = && \{(\text{"}P\text{"}, \texttt{top}, o)\} \cup\\
&\quad \{(o, \texttt{value}, ((\text{CALL}, (T,m)), \texttt{tt})\} \cup\\
&\quad \{(o, \texttt{next}, o-1)\}
\end{aligned}$$

The *around_more* rule is triggered by the CALL instruction somewhere in P and the DO instruction on C. Each application of *around_more* pushes the next advice on top of the proceed stack. However, the advices are not actually ordered in the graph. This has been done on purpose, to be able to simulate all possible advice orderings. To achieve correspondence, the *around_more* rule is assumed to be applied corresponding to the ordering of the advices in the derivation. The `selected` edge is created to ensure all advices are only scheduled once. The `Joinpoint` node is created to collect these edges each time a CALL instruction is adviced. After $n-1$ applications of *around_more*, the configuration graph corresponds to $[\![\text{AROUND}]\!]'$. Thereby, we have proved correspondence.

proceed correspondence

Correspondence for PROCEED is again a straight-forward translation corresponding to the LHS and RHS before and after the derivation, respectively. Since no derivation is given for PUSHP we cannot give correspondence there either. However, the derivation for POPP again requires a sequence of applications of graph transformation rules.

popp correspondence

Next, we prove correspondence for the POPP instruction. In the SOS derivation, when n advices $(a_1, \ldots, a_n)$ are scheduled for execution, the POPP instruction is annotated with a natural number n; it pops n instructions from the *proceed* stack. We explain that after advice execution is finished (leaving the POPP instruction on the continuation stack), the proceed stack always starts with $(\textsc{do}_2, \ldots, \textsc{do}_n, \bar{i})$. We distinguish two situations.

First, all advices $a_1, \ldots, a_n$ contain a PROCEED instruction. Before executing the first DOinstruction on the continuation stack, there are $n-1$ DO instructions and one CALL instruction on the proceed stack. For each PROCEED executed in an advice except the last advice, a DO instruction is popped from the proceed stack. A PUSHP instruction pushes the DO back on the proceed stack when the corresponding advice is finished. A proceed in the final advice pops the intercepted instruction and pushes this instruction together with a PUSHP for this instruction on the continuation stack.

Second, if advice k does not contain a PROCEED, it finishes leaving do_i with $i > k$ on the proceed stack and PUSHP instructions for $\textsc{do}_i$ with $2 \leq i \leq k$ on the continuation stack.

We show the correspondence between (1) the SOS derivation for $\textsc{popp}_n$ and (2) $n-1$ application of the *popp_do* rule followed by a single application of the *popp* rule. The configuration before $\rightarrow$ is translated to the following graph:

$$\begin{aligned}
[\![\textsc{popp}]\!] &= (N, E_C \cup E_P) \quad \text{where} \\
E_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \cup \\
&\quad\ \{(k, \texttt{value}, ((\textsc{popp}, n))\} \cup \\
&\quad\ \{(k, \texttt{next}, k-1)\} \\
E_P &= \{(\text{"}P\text{"}, \texttt{top}, o)\} \cup \\
&\quad\ \{(o-i, \texttt{value}, x_i) \mid 0 \leq i \leq n-1\} \cup \\
&\quad\ \{(o-i, \texttt{next}, o-i-1) \mid 0 \leq i \leq n-1\}
\end{aligned}$$

After $\rightarrow$, the configuration is translated to the following graph, where C and P

are empty:

$$\begin{aligned} [\![\textsc{popp}]\!]' &= (N', E'_C \cup E'_P) \quad \text{where} \\ E'_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \\ E'_P &= \{(\text{"}P\text{"}, \texttt{top}, o)\} \end{aligned}$$

Although we have shown the $[\![\textsc{popp}]\!]$ with instructions $x_1, \ldots, x_n$, we know in fact what these instructions are: $x_i = (\textsc{do}, a_{i+2})$ for $0 \leq i \leq n-2$ and $x_n = (\textsc{call}, \mathbf{tt})$. Also, in the graph production rules, the POPP instruction does not have an argument n. The *popp_do* rule removes a DO instruction from the proceed stack, but leaves the POPP instruction on the continuation stack. After $n-1$ applications of *popp_do*, only the intercepted instruction remains on the proceed stack. This leaves the following configuration graph:

$$\begin{aligned} [\![\textsc{popp}]\!]^1 &= (N^1, E^1_C \cup E^1_P) \quad \text{where} \\ E^1_C &= \{(\text{"}C\text{"}, \texttt{top}, k)\} \cup \\ &\quad\ \{(k, \texttt{value}, \textsc{popp})\} \cup \\ &\quad\ \{(k, \texttt{next}, k-1)\} \\ E^1_P &= \{(\text{"}P\text{"}, \texttt{top}, o)\} \cup \\ &\quad\ \{(o, \texttt{value}, (\textsc{call}, \mathbf{tt}))\} \cup \\ &\quad\ \{(o, \texttt{next}, o-1)\} \end{aligned}$$

Applying *popp* removes the POPP instruction from the continuation stack and the tagged instruction from the proceed stack, thereby leaving a configuration graph that is equal to $[\![\textsc{popp}]\!]'$. Thereby, we have proved correspondence.

4.6 Graph-Transformation-Based Simulation

We demonstrate the simulation capabilities of the operational semantics by means of the two scenarios of the example shown in Section 4.2. Simulation is done by using the GROOVE Simulator, which requires a graph production system and a start state. We will demonstrate this also in Chapter 5.

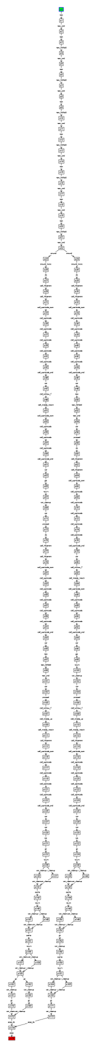
When using a "full" simulation strategy, the GROOVE simulator will try to match and apply all rules in all states (while respecting any priorities). This results in a state space, where states are graphs, and transitions are rule applications labelled with the name of the applied rule. This provides a simple and intuitive visual representation of the execution of the simulated system, the LTS.

The LTS can be used for analysis and verification, e.g. for model-checking. Also, the work described in Chapter 5, which is described using the semantics in Chapter 3, also can be applied to our FAJ semantics.

Figure 4.20 shows the generated LTS for the two scenarios. In Figure 4.20a, the setGrade method is called on an instance of NamedExam with argument *five*. The first branching (non-determinism) visualises the two choices in scheduling the two advices. When, in both cases, the advices and the setGrade method have returned, the two branches merge into the same final result, where the grade is set to *five*.

Figure 4.20b, the argument of the setGrade call is zero, causing the second advice in the example to behave differently: zero decreased by one remains zero. Again, after one-third the LTS branches into the two scheduling scenarios. In the left branch, the first advice of the example is executed before the second advice, causing the final result to be Zero (it is first increased and then decreased). In the right branch, the second advice of the example is executed first, causing the final result to be *one*. This is visualised by the two distinct final states: the branches are non-confluent.

In Chapter 5 we show how this non-confluence can be used as an indicator of aspect interference. Here we like to point out that the executable nature of the graph-transformation based semantics enables the use of such verification methods without any additional work needed.


(a) Scenario 1

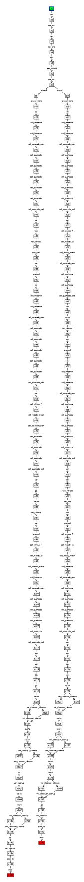

(b) Scenario 2

Figure 4.20: LTS for the Example Scenario's.

4.7 Evaluation of the Semantics

In this section we evaluate the contribution of our specified semantics. We discuss our observations while comparing the graph-based semantics to the corresponding SOS semantics. Then, we evaluate the predicted advantages of graph-transformation-based operational semantics over textual semantics, such as SOS. Finally, we compare the semantics to the graph-based semantics that was presented in Chapter 3.

4.7.1 Observations

First, we briefly discuss our observations while comparing the graph-based semantics with the corresponding SOS:

- The CALL- and NEW-rules in the reference SOS semantics include substitutions over lists. Graph transformation are not suitable for doing such at once (with the used graph-based state representation). Therefore, we required multiple rules that specify such behaviour, and more than one rule application to execute the behaviour;
- Instead of a VAR instruction, the SOS rules substitute variables with their values when pushing the method body on the continuation stack. This may give incorrect results when variables are first re-assigned (i.e. set) and then used.
- In the SOS rules, the result of the NEW instruction was originally pushed on the continuation stack. We found no need for this — it actually required an additional derivation that moves an object on the continuation stack to the value stack — and have modified the SOS rules accordingly.
- The graph-based semantics is in fact more precise, as it not only specifies a signature for the auxiliary functions *FDecl*,*MDecl* and *ADecl*, but also defines how input is mapped to output. Proof of correspondence is therefore under the assumption that these functions also correspond.
- To be able to compare graphs (using isomorphism, see Section 4.6), we found that a rule was required for *garbage collection*, which is not regarded in, nor required for the SOS semantics.
- As explained in Section 4.5, we interpret the POPP instruction in a different way, yet yielding the same result.

4.7.2 Evaluation

We have shown in Section 4.6 that we can simulate FAJ specifications using the specified operational semantics. We have not implemented a translator from FAJ programs to graphs; instead we have given a definition of the structure these graphs are required to adhere.

The executable nature of the semantics is a direct benefit for the rigour of the specification approach. Using a (number of) test programs, simulating these will quickly shows if the semantics is "working": any missing structure will fail the execution.

Some auxiliary rules can be used to simplify this testing-approach. For example, a rule *program_end* can be used that matches an empty continuation and proceed stack. A simple visual analysis of the graph can determine if each execution trace leads to a state where the *program_end* condition holds.

The graph transition system represents the execution state space of a single FAJ program. Such a state space can be used for verification approach such as model checking. Also, additional rules can be added that express certain conditions (similar to the *program_end* rule described earlier) that must hold during the execution of the program. For aspects that match the same instructions, the transition system represents all possible execution orders of such advices. We have already illustrated that this can lead to non-deterministic execution. In Chapter 5 we describe this in more detail as a verification approach, which is directly applicable to the simulation results of the semantics as it is described in this chapter.

In addition, we believe that the visual nature of the graph transformation rules will appeal to many readers that are not experts in mathematics. Programmers are used to think of software in terms visual artefacts, since the use of visual editors are a common way to design software.

Since we have used an existing (SOS) semantics as our notion of correctness, one may question the need for a new (graph) semantics for the same language. To motivate this once more, let us point out that our semantics is directly executable, in contrast to the SOS rules which (among other things) contain substitutions over lists, as well as many auxiliary functions. In fact, in setting up our rules we have had the opportunity to repair a number of smaller and larger oversights (described above) in the SOS semantics, precisely because these were shown up by the simulation of the rules. Had the SOS semantics been specified for MAUDE [CDE$^+$02], which allows execution of SOS semantics, then (1) none of the short-hand (non-formal) notations could have been used, (2) all functions had to be defined and (3) execution of the semantics would not yield a easy to use, visual result.

4.7.3 Comparison with CF

We take this opportunity to compare the semantics of FAJ defined in this chapter, with the operational semantics for Composition Filters of Chapter 3.

In the FAJ semantics, method-bodies are represented as sequences of instructions. After parsing the program, these sequences are generated by applying a sequentialisation function. This requires a stack-based model, where the instruction on top of the continuation stack is executed, optionally using arguments that are pushed onto the value stack earlier. Since the language does not support any jump operations, control flow is directly represented by the ordering of the instructions. In contrast, the CF semantics uses a graph-based representation of the abstract syntax tree (AST), which is a direct product of a parser. To be able to execute this syntax representation, control information must first be added explicitly. Then, the "stack" is represented as nodes representing frames and an edge representing the program counter. The low-level language model used by the FAJ semantics resembles an actual implementation of a (virtual) machine that executes FAJ programs. It can be used as a formal model of such a machine. In contrast, the model used in the CF semantics closely resembles the syntax, which increases the readability of the semantics. It is more likely to serve as a reference for the language semantics.

The aspectual extension of both semantics allows the interception of method calls. In the FAJ semantics, this is done by matching a CALL instruction on top of the continuation stack, resulting in the scheduling of advices by pushing DO instructions (that are similar to CALL instructions, but instead invoke an advice). In the CF semantics, the scheduling of aspects is triggered by the initialisation of a frame node that has a target object of which the type is enhanced with filter modules.

The CF semantics specifies the entire aspectual extension of Composition Filters. The FAJ semantics merely includes a small aspectual extension to Featherweight Java. The language model, however, limits the aspectual extension to the interception of instructions. The aspectual language then falls in the category of languages with fine-grained joinpoint models (such as AspectJ), whereas CF is more likely to be used as a composition mechanism (see Chapter 1.1). The specified advice language of CF consists of filter evaluation and the invocation of filter actions that does not use any base language constructs. The semantics can be used to simulate *join points*: simulation starts when filter module execution is triggered by a specific method-call and stops where base program execution would continue. This gives a certain level of abstraction: a single simulation may represent execution of a groups of joinpoints in different programs. In the FAJ semantics, advices are special methods (invoked by DO instructions) that contain PROCEED instructions. Being able to execute FAJ advice therefore requires the full specification of the base language. By representing the entire program as a graph, The entire program can be simulated in a concrete fashion. Both semantics can be used to simulate different advice orders non-deterministically. In the next chapter we will show how this feature can be used to detect the presence of *aspect interference*.

4.8 Conclusion

4.8.1 Related Work

There are quite a few works describing a specification of an aspect-oriented operational semantics. Most of these approaches focus — like this work — on a simplified base language and aspectual extension. In general, they focus on a certain feature of aspect-oriented language; such investigations typically aim to improve understandability of a feature, or to experiment with a new feature or a variation of the feature.

Clifton et al [CLW03] give a formal definition of the parameterised aspect calculus. This is a purely mathematical specification of a base language and a variety of point-cut description languages. The specification does not provide a direct means for execution. Kiczales et al. [KD02] give a denotational semantics for a first-order procedural language with join-points for procedure-call and procedure and advice execution. Denotational semantics do not expose the execution steps that we want to use for analysis and verification. Lämmel [Läm02] presents an operational semantics for an imperative object-oriented language with join-points for method calls. Walker et al. [WZL03] present a core aspect-oriented calculus based on the simple-types lambda calculus. Jagadeesan et al. [JJR03] give a calculus of untyped aspect-oriented programs. Their specification is class-based, and models multi-threaded programs. Clifton et al. [CL05] provides a calculus for an imperative object-oriented language with advice bindings and proceed.

The works mentioned all use a mathematical notation, whereas our notation is of a more intuitive visual kind. The works mentioned also do not provide a means for execution, whereas our work can be directly be used to visually represent the execution of a specific FAJ program.

4.8.2 Contributions

In this chapter we have defined a graph transformation-based semantics for a simple object-oriented language with around advice. The specified language, Assignment Featherweight Java, lends itself very well for studying the implications of language extensions. We have extended this language with around advice bound to point-cuts that select certain instructions.

We have explained that a graph transformations is a formal specification technique, and have illustration that a graph transformation based operational semantics can be complete with respect to a certain reference semantics.

A graph-tranformation based semantics is directly executable. This can help in finding bugs and testing the semantics. Due to its executable nature, the graph transformation-based specification has led to the discovery of errors (e.g. a missing rule) in the specification that is used as a correctness criterion; we see these errors as an unfortunate consequence of the (more traditional) textual formal notation used.

We have demonstrated that a graph transformation based specification allows simulation of a program written in the language, if this program is represented as a graph as described in this paper. This gives a simple and intuitive view on the execution of the program, and opens the road towards applying existing verification methods such as analysis based on model checking. We have shown that our simulation tool allows non-deterministic execution, which opens possibilities for the verification of ordering conflicts between aspects.

Chapter 5

Analysing Aspect Interference on Shared Join Points

5.1 Introduction

Aspect-oriented programming languages allow the modular specification of crosscutting concerns, thereby improving separation of concerns at the implementation level. We have illustrated in Section 1.2 that when multiple aspects are applied to a system unexpected results can emerge: two or more aspects behaving correctly when applied in isolation, may interact in an undesired matter when applied together. This phenomenon is called *aspect interference*.

In this chapter we take a look at a specific kind of aspect interference, namely when it happens at a join point that is selected by more than one pointcut, a so-called *shared join point*. At such points, two or more advices are scheduled for execution. When no fixed order of advice execution is determined by the program directives, but the order of advice execution affects the result, a shared join point can lead to unpredictable and undesired behaviour of the woven system, or even an ambiguous system. Other studies such as [NBA05] have already indicated that special attention must be paid to shared join points.

In this chapter we propose a detection mechanism for aspect interference at shared join points. The approach is based on simulation using a semantics specified with graph production rules. Such a simulation results in a *graph transition system*, a state space where states are in fact graphs, and transitions are graph transformation rule applications and are labelled by the name of the rule. We will show that, by simulation of different advice orderings in a specific way, aspect interference will result as non-confluence in the state space, making detection an almost trivial task. We demonstrate the approach for the Composition Filters language, for which we have defined such a semantics in Chapter 3. We will argue that CF is typically but not solely suitable for our approach.

This chapter is organised as follows. In Section 5.2 the problem of aspect interference at shared join points is discussed in more detail and an example is introduced. In Section 5.3 we will explain our approach in detail. In Section 5.4 we present an adaptation of the CF semantics presented Chapter 3 to implement our approach and to use it for the example used in this chapter. In Section 5.5 we show the illustrate simulation and present the experimentation results of the approach applied to the example. In Section 5.6 we evaluate the approach for a number of properties. Finally, in Section 5.7 we discuss related work, give a discussion possible future extensions and present the contributions of our work.

5.2 Problem Definition

In this section we will elaborate on the problem of interference among aspects at shared join points. Then we illustrate the problem with an example specified in the Composition Filters language.

5.2.1 Aspect Interference at Shared Join Points

Interference between aspects at shared join points will only occur when aspects depend on the same system state during the execution of the advice. Three different kinds of interaction between aspects can cause interference to occur at shared join points.

1. The first problem is what we refer to as *scheduling interference.* Advices can be woven conditionally, meaning that whether the advice is invoked depends on the result of evaluation of a dynamic predicate. At run-time, such predicates typically involve the evaluation of a boolean expression. When more aspects are scheduled, an earlier executed aspect may influence the evaluation of this boolean expression. Thereby, it can change the scheduling of an advice. Scheduling interference is caused by a common feature of aspect-oriented languages; they allow pointcuts to consist of predicates over both the static structure of the program and the run-time state. The latter predicates allow the aspect to be applied only at a certain run-time state.

Where the static predicates are typically resolved during weaving - resulting in a set of so-called *shadow join points* - the dynamic predicates (so-called *pointcut residues*) are typically woven in with the advice as an *if*-statement.

2. The second problem — *control interference* — occurs when one aspect changes or aborts the control flow of the system, for example by aborting the join point action and all succeeding aspects to be executed at that join point. A typical *AspectJ* example of this behaviour is an *around* advice without a *proceed()* statement.

3. Aspect interference can also occur if one aspect writes to variables or fields that are later used by other aspects, such that the behaviour of a succeeding advice is affected. As this *data interference* is a general cause for aspect interference, it can also occur on shared join points. It can happen when one aspect writes to a variable that is read by another aspect, or when both aspects write to the same variable. In the latter case, the operations may be orthogonal causing no problem at all; but they can also disable or undo each other.

Scheduling interference cannot occur in Composition Filters. Pointcuts consisting of dynamic predicates can be expressed in CF as *conditions*. We have explained in Chapter 3 that these conditions are evaluated once for every joinpoint, namely when a message enters a set of filters. Any change of the run-time state caused by the filters is not reflected by the value of the conditions.

5.2.2 Example Aspects

We will illustrate the problem by examples of aspect interference. Consider a system where String objects are sent between objects and assume we add the following four aspects to this system:

- **Logging**: The "logging" aspect logs method-calls to a certain location (e.g. to a file or to the console).
- **Authorisation**: The "authorisation" aspect will disallow unprivileged users from using certain methods in the system. These methods will be blocked by aborting the method dispatch process and throwing an exception.
- **Parent**: The "parent" aspect removes inappropriate words in a String that is sent.
- **Encryption**: The "encryption" aspect encrypts the String that is sent for secure transportation in the system.

The combination of the logging and authorisation aspects illustrates an example of interference type 2 as presented above: interference caused by a control-flow modification. When the logging advice is executed before the authorisation advice, a method can be logged without being executed. In reverse order, the method may be aborted before the method-call is logged.

The parent and encryption aspects combined illustrate an example of interference type 3: both aspects change the same field. Encrypting the string after filtering inappropriate words is the obviously desired behaviour. In reverse order, the profanity filter will be applied to an encrypted string and will not be able find any profane language.

The logging and encryption combined are another example of interference type 3. When Logging is executed first, the original string is logged; in reverse order the encrypted string is logged.

5.2.3 Example Code

We now give an implementation of the aspects described earlier in this section in the Composition Filters language. To be able to specify a small-sized and to-the-point example, we have defined a number of so-called *user-defined filter types*. Such filter types are implemented by the user, but are intended to have a reusable nature. We merely describe the run-time behaviour of these filter types (i.e. no actual run-time implementation exists of the actions associated with the filter types used in the example).

Listing 5.1 shows the complete specification of the logging aspect. Worth mentioning is the use of filter-type Log. The result of this specification is that each incoming message with name send to an instance of class Server is logged.

Similarly, Listing 5.2 shows the filter module specification of the other three aspects: authorisation, parent and encryption, that make use of filter-types Abort, Parent and Encrypt, respectively.

At run-time, the used filter types are associated with filter actions "LogAction", "AbortAction", "ParentAction" and "EncryptAction"; these are executed when the filter accepts a message. All used filter types perform the default "ContinueAction" when the filter rejects the message. This action is executed when the filter rejects a message, and will continue the message to the next filter. If the filter is last in line, the message will be dispatched afterwards.

```
concern Logging {

  filtermodule Logging {
    inputfilters:
      logging: Log = { [*.send] *.* }
  }

  superimposition {
    filtermodules
      classes = { x | ClassByName{ x, 'Server' } };
    superimposition
      classes <- Logging;
  }
}
```

Listing 5.1: Composition Filters source code of the Tracing aspect

```
  filtermodule Authorisation {
    externals
      user: User = User.instance();
    condition
      isAllowed = user.isAllowed();
    inputfilters:
      auth: Abort = { !isAllowed => [*.send] *.* }
  }

   filtermodule Parent {
      inputfilters
        profanity: Parent = { [*.send] *.* }
   }

   filtermodule Encrypt {
     inputfilters
       encrypt: EncryptString = { [*.send] *.* }

   }
```

Listing 5.2: Composition Filters source code of the Authorisation aspect

5.3 Approach to Aspect Interference Detection

In the previous section we have explained the problem of advice interference at shared join points. In this section, we present our approach to verify that, given an aspect-oriented specification, no interference occurs. The absence of interference means that, at all join points, each execution order for the scheduled aspects results in the same resulting state.

We propose to use simulation of all possible advice orderings at all shared join points. We represent such a joinpoint (i.e. a run-time state where advices are scheduled) as a graph (a so-called *joinpoint graph*), and use a graph-transformation based operational semantics to simulate the execution of the aspect composition.

5.3.1 Analysis

We now explain the shape of the generated state space of simulating a join point graph and explain the analysis of this state space for the detection of aspect interference. When simulating the full execution space of a joinpoint graph and the run-time semantics, we see two kinds of branching occur in this state space.

1. First, dynamic predicates — condition expressions and signature matches in Composition Filters — are evaluated to true or false non-deterministically; the transition system branches into a path where the predicate is true and a path where the predicate is false. The number of different evaluations for all signature matches and condition expressions is 2^p, where p represents the number of predicates.

2. Second, once all predicates are evaluated, the transition system branches each time an advice (a filter module in CF) is picked from the pool of pending advices. Each choice is represented by a distinct state (i.e. a graph). The worst-case number of different orders (and paths) is $a!$, where a represents the number of advices at the shared join point.

A generalised shape of the generated LTS is shown in figure 5.1. In AOP terminology, the initial state represents a shadow joinpoint; it reflects a statically matched join point. Branching occurs when a rule has multiple matchings in one graph or when different rules match in the same graph. The branching above the dotted horizontal line represents assigning values to the predicates that do not have a value. The states on the dotted line represent all *actual joinpoints* for the shadow joinpoint. As stated above, the number of paths to these states is 2^c, where c represents the number of conditions.

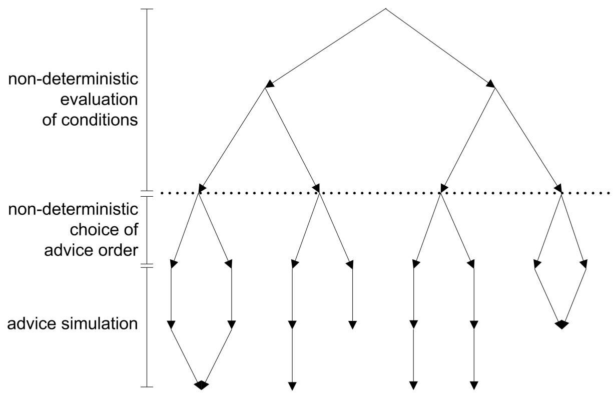

non-deterministic evaluation of conditions
non-deterministic choice of advice order
advice simulation

Figure 5.1: General shape of the LTS, with two predicates and two advices.

Below the dotted line, branching occurs when the first filter module is selected, after which all filter modules are executed. The figure illustrates that for different values of conditions, different shapes in the LTS can occur.

When all advice orders on a shared join point have been simulated, we can analyse the composition of the advices by looking at the shape of the LTS. We base our analysis on the shape of the actual joinpoint.

When advice actions are commutative — the order of execution does not affect the resulting state — the execution traces of the different orders are confluent, because the same state is automatically represented by the same "box" in the LTS. The equivalence of states is based on an isomorphism check that is based on labels, not on node identities. Confluence is visualised in the left-most and right-most diamond shape in Figure 5.1.

When the order of execution of a number of advices does not affect the result, we can not immediately conclude that this is also the desired result. Imagine a logging advice that logs an immutable copy of the (original) argument of a method, and another advice that modifies the value of a mutable variable containing the argument. Both orders of advice execution would log the original argument and make the same change to the argument. In other words, the LTS would be confluent. However, if the behaviour of the logging advice is documented as "logging the value of the argument passed to the method body", the combined behaviour is not correct. The expectation is only satisfied when the argument is not changed. Although we cannot be certain that a result from confluent orderings is the "expected" result, at least we are certain that the all orders of the advices yield the same result. The problem in the scenario we just illustrated is caused by an assumption that holds when the aspect is applied in isolation, but may not hold when composed with other aspects.

When advice units are not commutative, the order of execution affects the resulting state and the execution of different advice orders will result in different final states. This is illustrated in the two middle cases in Figure 5.1. One of these states might be the desired result, or both might be undesirable. In any case, we can conclude that the advices interfere: the changes made to the state by one advice affects the applicability of the other advice or the state change made by the other advice. This definition of interference helps us understand when confluence can also occur even though the advices interfere. The state change made by the first advice and the effect of the first advice on the state change made by a second advice might "accidentally" add up to an identical resulting state. However, by intuition we believe this to occur only very rarely.

5.3.2 Implementation Requirements

For the simulation of advices expressed in Composition Filters (i.e. filter modules) we use the operational semantics presented in Chapter 3. To be able to use this semantics for our approach, and to illustrate this for the given example, we must take the following preparations:

1. We generate graphs that represent one class, signatures of the methods defined in this class, and all filtermodules that are superimposed on the class. To represent a shadow joinpoint, we must add a representation of a method call to an instance of this class. To represent all distinct method-calls, we use the different *selectors* used in the filter modules for the name of the called method. Also, we need to represent the instances of internals and externals.

2. After the previous step, the graph is an abstraction of the possible states of the system when a shared joinpoint is reached. In particular, the abstraction pertains to the value of condition expressions and signature matchings. For interference analysis, we need to make this more concrete. In fact, to realise the proposed LTS shape, the predicates are required to be evaluated *before* executing the advices. Additional rules need to be specified for this pre-evaluation.

3. To simulate the execution of the filtermodules in the proposed manner, we need to specify rules for the non-deterministic scheduling of these filtermodules.

4. The presented example uses user-defined filter types. To simulate the run-time behaviour, we need to specify rules for the filter actions associated with these filter-types. Because the graph is an abstraction of a run-time state, we also specify the behaviour of the filters in an abstract way.

In the next Section we show the additional rules needed to apply our verification to the presented example. Then, in Section 5.5, we show our experimentation results.

5.4 Extended Composition Filters Semantics

We now give the additional rules needed as discussed in the previous section. The rules for the user-defined filter-types are specifically needed for the example; the other rules merely extend the semantics resulting in an implementation for our verification approach.

5.4.1 Message Creation

We use a Composition Filter specificiation to generate a graph for each class that is enhanced with filtermodules. This graph then represents the class, the signatures of methods defined in the class, and the syntax of the filter modules. The latter is consistent with the type-graph shown in Figure 3.2.

To simulate all distinctly recognised messages to an instance of such a class, we add specific frames to this graph. Each frame corresponds to a level in a call-stack. When a new frame is created, the creating frame has to wait for the created frame to finish execution. For simulation of the filtering mechanism at a single joinpoint we need three frames, namely (1) a frame that is executing the code where the message originates (i.e., with a method-call statement), (2) a frame for executing the called method, and (3) a frame for filtering the message. The name of the called method is initialised to any selector used in the filter modules. This selector can also be "*"; the graph then represents all message that are not explicitly referenced by the filter modules. If the chosen selector matches a signature in the class, this signature is used to initialise the arguments of the method-call.

The resulting graphs are the start graphs of our simulation. The graphs coincide with the type-graphs in Figure 3.22 and 3.21. We briefly discuss the rules that we have specified to generate these graphs:

- The rule in Figure 5.2 creates the three frames that are required to represent the message. The filtered MethodFrame is created with a name-less Signature, i.e. no selector is specified yet. Also, the target object is created.

- The rule in Figure 5.3 initialises the name of the method-call as any selector that is used in a filter module.

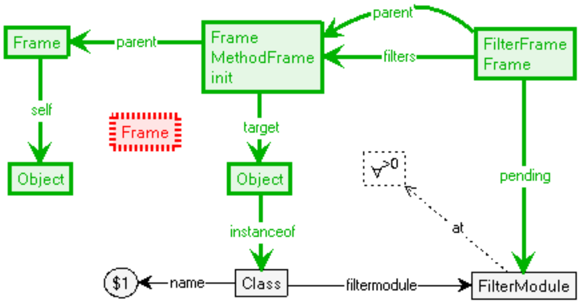


Figure 5.2: Rule for Frame Creation

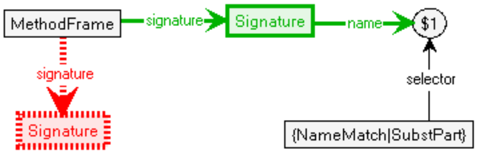


Figure 5.3: Rule for Selector Creation

- The rule in Figure 5.4 replaces the created signature with an equally named signature in the target class. If no such signature is found (i.e. for the "*" selector), nothing happens.
- The rule in Figure 5.5 creates values for the parameters of the signature. The default signature has no arguments.
- The rule in Figure 5.6 creates values for each of the internals and externals that are referenced in the filter modules.

The graphs resulting from applying the rules above represent *shadow join points*, and are the start graph of the simulation of filter module execution.

5.4.2 Predicate Evaluation

The start graph of the simulation represents an abstraction of the actual run-time states where filter modules are triggered. This abstraction lies in absence of the part of the state that is used by expressions in the filter modules. In the original CF semantics (see Chapter 3), condition expressions and signature matches are evaluation on the fly (i.e. when the program reaches the expression). The abstraction then causes non-determinism during the execution of the filter

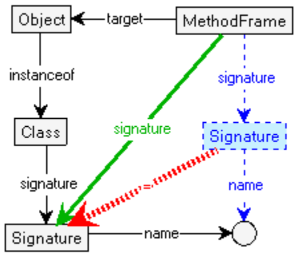


Figure 5.4: Rule for Signature Resolving

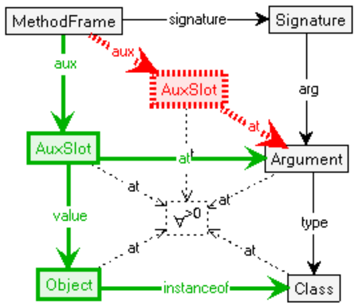


Figure 5.5: Rule for Argument Initialisation

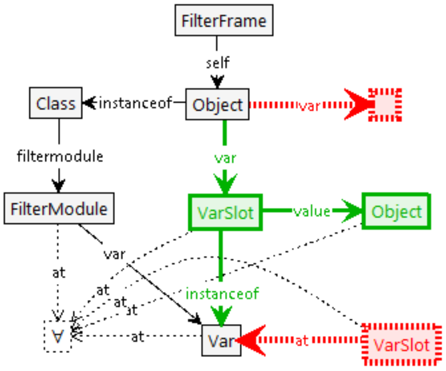
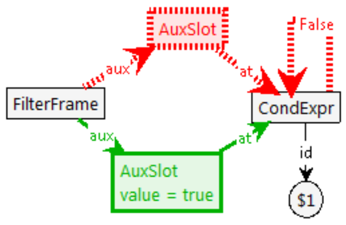
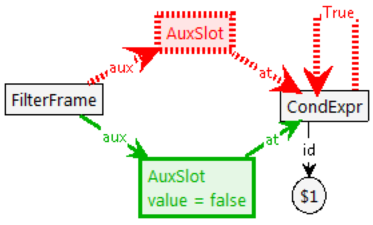


Figure 5.6: Rule for Context Initialisation

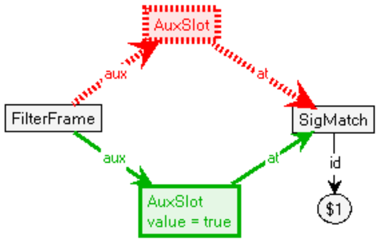


(a) PreCondExprTrue

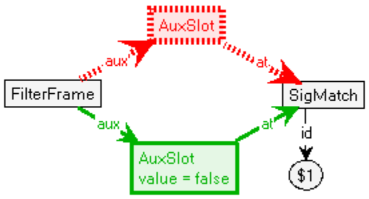


(b) PreCondExprFalse

Figure 5.7: Early CondExpr Evaluation



(a) PreSigMatchTrue



(b) PreSigMatchFalse

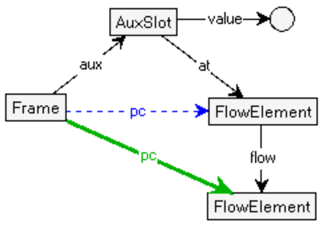
Figure 5.8: Early SigMatch Evaluation

modules, due to the fact that the rules for evaluating the expression to *true* and *false* are both applied to the same graph.

To compare the different executions of filter modules at a specific (non-abstract) run-time state, we must *first* evaluate condition expressions and signature matches, so-that the execution of a filter-module becomes deterministic.

To achieve this, rules are added that resolve condition expressions and signature matches before the filter modules are executed. The rules for the CondExpr expressions are shown in Figure 5.7; one rule for an evaluation to *true* and one rule for an evaluation to *false*. Similarly, the rules for the SigMatch are shown in Figure 5.8. Both rules can be applied to the same graph, causing non-determinism and creating different concrete possibilities for an abstraction.

The difference with the rules in Chapter 3 is that there is no program counter required for the evaluation of the expressions. However, now that these expressions are evaluated before the filter modules are executed, we need an additional rule to skip expressions that have been evaluated earlier. This rule is shown in Figure 5.9.



(a) Skip

Figure 5.9: The Skip Rule

5.4.3 Non-Deterministic Scheduling of Filtermodules

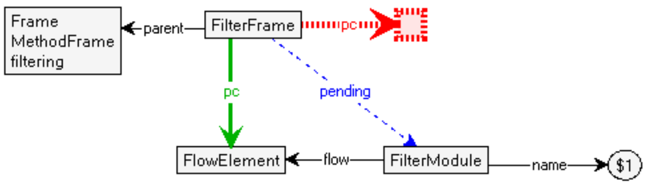
Figure 5.10 shows the rules that are involved in the scheduling of filter modules. The following points are noteworthy:

- The rule in Figure 5.10a selects any of the filter modules that must be executed at the join point.
- The rule in Figure 5.10b detects when the execution of a filter modules has ended, and adjusts the graph so that the previous rule can be applied again.
- The rule in Figure 5.10c dispatches the message when there are no more filter modules left to execute.

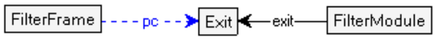
5.4.4 Filter Actions of the Example

For simulating the example presented in Section 5.2 we must extend the CF semantics with rules for the user-defined filter types, namely Abort, Log, Encrypt, and Parent. We now describe these rules.

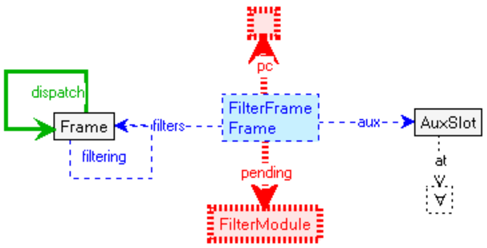
- The rule specifying the FilterAction Log is displayed in Figure 5.11. It creates a `logged` edge to selector and arguments of the message.
- Figure 5.12 shows the production rule of the FilterAction Abort. The filter frame is removed, ending the filter evaluation. The method frame — currently in the `filtering` state — is updated to the `abort` state, to indicate that the message was aborted.
- Figure 5.13 shows the specification of the Encrypt action. All arguments of type “String” are replaced with new objects. To indicate that these objects



(a) FilterModuleSelect



(b) FilterModuleEnd



(c) Dispatch

Figure 5.10: Filter Modules Scheduling Rules.

are encrypted versions of the original objects, it adds `encrypted` edges between the old and new objects. The rule for the Parent action (Figure 5.14) is similar, but adds a `parental` edge between the objects.

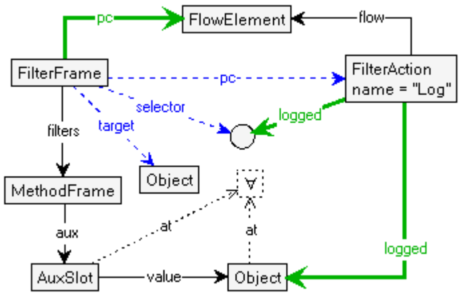


Figure 5.11: Rule specifying the Log action

The rules illustrate that we can specify the behaviour of the filter actions in an abstract way. We merely need an abstract representation of the effect of advice

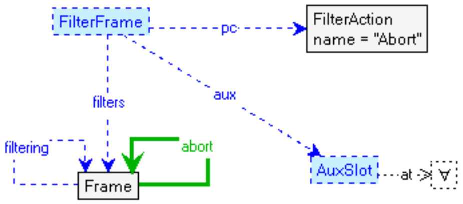


Figure 5.12: Rule specifying the Abort action

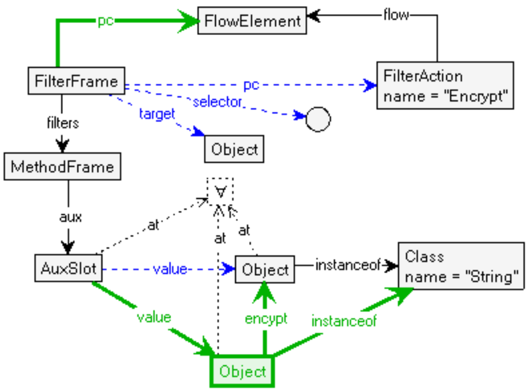


Figure 5.13: Rule specifying the Encrypt action

application on the represented part of the state, such that different states can be identified. For the logging action, for example, we only care about *what* is logged, and therefore only that information determines the resulting state. The encryption and parental actions do not actually represent modified strings, but rather encode a relationship between the new and the old values.

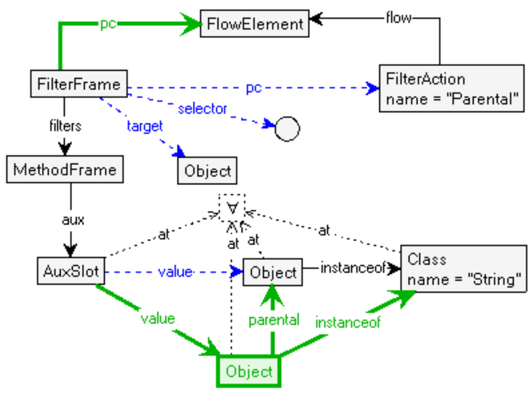


Figure 5.14: Rule specifying the Parental action

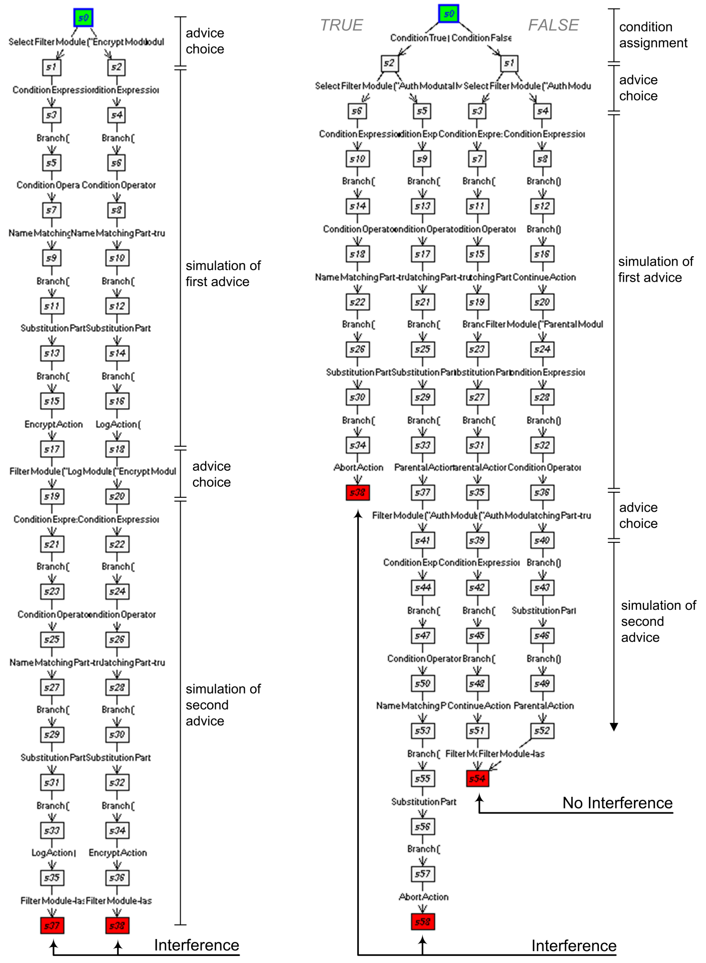
5.5 Experimentation

We now present the results of our interference detection experiments with the example presented in Section 5.2. First we show some of the generated transition systems and give an intuition of the analysis. Then we give a detailed report of data gathered from an implementation that automates the approach.

5.5.1 Generated State Spaces

Figure 5.15a shows the generated state space for a shared join point with the *logging* and *encryption* aspects. The two paths are not confluent, since the Log action tags a different string; the advices interfere. Figure 5.15b shows the generated state space for a shared join point with the Authorisation and ProfanityFilter aspects. In the first branching, different values are assigned to the condition expression of the authorisation advice, resulting in two actual join points. Then, on the left side (where the condition is true), the branches do not merge because the authorisation advice aborts the flow either before or after the profanity filter has executed, resulting in different final states. On the right side (where the condition is false), the condition of the authorisation advice fails, so that it does not abort, in which case the advices do not interfere.

Figure 5.16 shows the generated state space for all our four aspects, which is a bit harder to analyse visually. The first branching again represents the evaluation of the condition expression in the Authorisation aspect. We can see that more than one final state can be reached via the 24 ($a!$, where $a = 4$) different execution orders for each of the two actual join points. The figure can be used to analyse the possible executions of the join point.



(a) Logging and Encrytion (b) Authorisation and ProfanityFilter

Figure 5.15: Generated transition systems

5.5.2 Analysis Report

We now give a detailed report of our experiment. We have implemented a tool that performs the analysis automatically. An example output of the tool is shown in Listing 5.3. For each run-time simulation it performs, it shows the target and

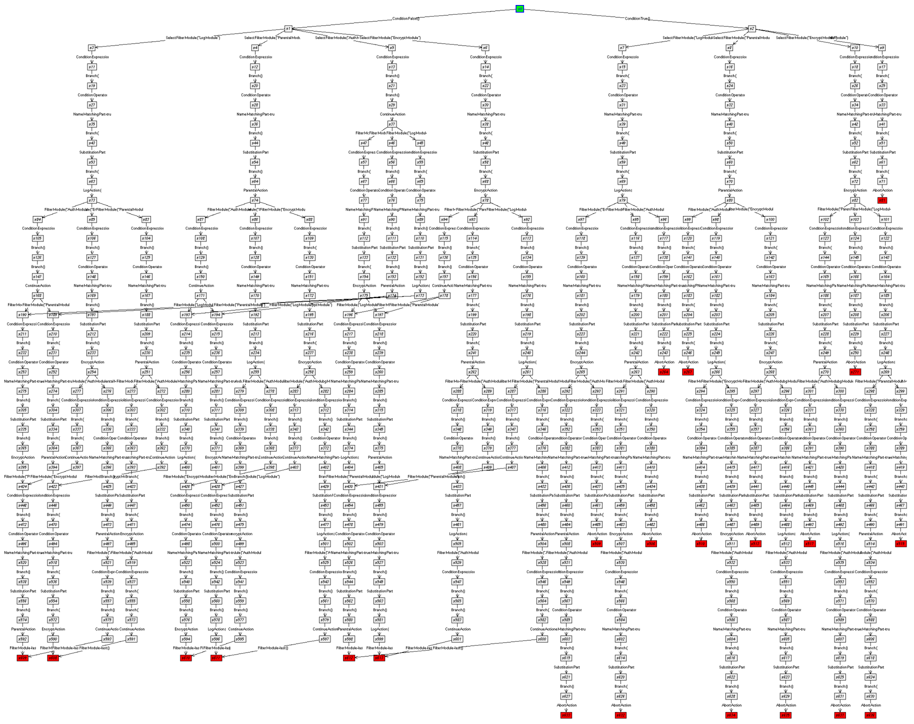


Figure 5.16: Generated transition system of the example program with four aspects

selector of the encoded message (i.e. the type of the target and the name of the method-call). Then it reports details of the generated LTS, such as the number of states and transitions, and the number of final states. For each conflict it detects, it gives a summary of the predicate assignments at the actual run-time state, and then shows the counter example: the orders of the execution filter modules and the (different) final states. Finally, it reports the total time elapsed for the analysis. The example output reports one conflict that corresponds to the interference we have shown in Figure 5.15b. It indicates that the conflict appears when the condition expression in the first filter element (0) of the auth filter in the Authorisation() filter module is true. This is followed by the counter example: the filter module sequences that are executed from this state, and the final states of these sequences.

Table 5.1 shows a summary of the output of running the analysis tool with different compositions of aspects. We denote the aspects with L,P,E,A for Log, Parent, Encrypt and Authorisation, respectively. We give the number of nodes and edges in

```
+++ SIMULATION RESULT +++
message target: "grasstest.Server"
message selector: "send"
nodes: 63
edges: 198
++++++++++++++++++++++
+ grammar: runtime
+ states:75
+ transitions:80
+ final states:3
+ open states:0
++++++++++++++++++++++
*** CONFLICT
*** STATE SUMMARY:
 - "AuthorisationConcern.Authorisation.auth.0.CondExpr" = true
*** TRACES:
 + s6 >> "Authorisation" >> s46
 + s6 >> "Parent" >> "Authorisation" >> s75
****************
Fin. (0:3.234s)
```

Listing 5.3: Analysis report for Authorisation and Parent

the start graph, the number of states, final states and transitions in the generated LTS, the elapsed time, and the number of conflicts (counter examples). We only give the results for the analysis of send messages to an instance of class Server. Other messages (such as *) do not result in interference.

The number of conflicts is limited to the number of actual joinpoints in the graph, namely 2^p, where p is the number of predicates (condition expressions and signature matches). In the example aspects, only the authorisation aspect has a condition. The rows with aspect A therefore can have 2 conflicts, and have significantly more states compared to the compositions with an equal number of aspects that have no predicates and thus can have a maximum of one conflict.

For illustrational purposes, we take a closer look at the analysis report of the composition of all four aspects. The found conflicts are shown in Listing 5.4.

The thing we want to point out is that the report does not show all traces that lead to different states. For example, the first conflict only shows the execution orders that start with the Authorisation aspect. Since this aspect does not do anything when the condition is false, any other position of the Authorisation aspect gives the same final state. The analyser reports each final state only once, and only one execution order to reach this final state.

```
*** CONFLICT
*** STATE SUMMARY:
 - "AuthorizationConcern.Authorisation!classes.log.0.CondExpr"
     = false
*** TRACES:
 + s23 >> "Authorisation" >> "Logging" >> "Encrypt" >> "Parent"
      >> s785
 + s23 >> "Authorisation" >> "Logging" >> "Parent" >> "Encrypt"
      >> s786
 + s23 >> "Authorisation" >> "Parent" >> "Encrypt" >> "Logging"
      >> s783
 + s23 >> "Authorisation" >> "Parent" >> "Logging" >> "Encrypt"
      >> s784
 + s23 >> "Authorisation" >> "Encrypt" >> "Parent" >> "Logging"
      >> s781
 + s23 >> "Authorisation" >> "Encrypt" >> "Logging" >> "Parent"
      >> s782

*** CONFLICT
*** STATE SUMMARY:
 - "AuthorizationConcern.Authorisation!classes.log.0.CondExpr"
     = true
*** TRACES:
 + s24 >> "Parent" >> "Logging" >> "Authorisation" >> s619
 + s24 >> "Logging" >> "Parent" >> "Authorisation" >> s623
 + s24 >> "Encrypt" >> "Logging" >> "Parent" >> "Authorisation"
      >> s794
 + s24 >> "Parent" >> "Encrypt" >> "Authorisation" >> s617
 + s24 >> "Logging" >> "Encrypt" >> "Authorisation" >> s621
 + s24 >> "Parent" >> "Logging" >> "Encrypt" >> "Authorisation"
      >> s796
 + s24 >> "Encrypt" >> "Parent" >> "Logging" >> "Authorisation"
      >> s793
 + s24 >> "Encrypt" >> "Logging" >> "Authorisation" >> s615
 + s24 >> "Parent" >> "Authorisation" >> s319
 + s24 >> "Authorisation" >> s105
 + s24 >> "Encrypt" >> "Parent" >> "Authorisation" >> s613
 + s24 >> "Parent" >> "Encrypt" >> "Logging" >> "Authorisation"
      >> s795
 + s24 >> "Logging" >> "Encrypt" >> "Parent" >> "Authorisation"
      >> s797
 + s24 >> "Logging" >> "Parent" >> "Encrypt" >> "Authorisation"
      >> s798
 + s24 >> "Logging" >> "Authorisation" >> s322
 + s24 >> "Encrypt" >> "Authorisation" >> s316
```

Listing 5.4: Conflicts for all four aspects.

	nodes	edges	states (final)	transitions	time (s)	# conflicts
LA	62	198	75 (3)	80	3.78	1
LP	60	196	50 (2)	50	2.95	1
LE	60	196	50 (2)	50	2.97	1
AP	63	198	75 (3)	80	3.23	1
AE	63	198	75 (3)	80	3.30	1
PE	59	196	50 (2)	50	2.92	1
LAP	82	280	232 (7)	251	3.95	2
LEP	79	279	179 (6)	183	4.78	1
LAE	82	280	232 (7)	251	4.15	2
APE	82	280	232 (7)	251	4.62	1
LAPE	101	362	798 (22)	863	10.13	2

Table 5.1: Analysis results of all possible compositions.

5.6 Evaluation of the Approach

In this section we evaluate the approach for a number of properties that we consider important.

5.6.1 Detection of Interference

The approach allows abstractly specifying the behaviour of advice actions, so that only relevant behaviour is incorporated. Of course this also means that the specification can be over-abstracted causing certain problems to be undetectable.

Although we cannot guarantee that a composition of aspects is free of interference, we can warn the user (with certainty) for interference in case of a non-confluent result. When the result is confluent the advices are either free of interference or coincidentally cause the same wrong result with all orders of execution (which we consider very rare). We believe that when advices are commutative for every combination of condition values, the shared join point is highly likely free of interference.

The visual nature of the result — the LTS — can help in understanding the composition of advice, even as simply as seeing different shapes under different condition values. This can help in understanding if a result is desirable or help debugging a problem.

The approach allows to distinguish results for different run-time states, by tailoring the conditional assignments. Some false positives (detection of interference that will never actually happen) can occur when certain combinations of condition values (i.e. run-time states) are never found when the program is executed. Again, the visual nature of the LTS can help in analysing the different scenarios.

5.6.2 Modularity

Modularity in the context of verification of aspect-oriented software development typically means that it is possible to analyse and verify aspects and aspect compositions independently of a base system. Our approach is modular in such a way that the source-code of the base system is merely used to extract shared join points. We have shown that our approach can detect aspect interference by simulation of advices alone. When the base system's source code is not available, it is possible to simulate every combination of two advices. This can, however, lead to false negatives and false positives for actual base programs, when the advices are never composed at shared join points.

5.6.3 Usability

Since the base system is not simulated, our approach requires only the advice language to be specified formally using graph-transformation rules. For the Composition Filters language this is a reasonable task, although we have not included *Meta* filters. Such filters call a method specified in the base-language. To incorporate analysis of meta-filters, we would not only have to specify base language semantics, but also have to incorporate a larger subset of the base system's run-time state. Determining this subset might be a difficult task.

CF comes with a base set of filter types. These filter types are argued to be useful for a large number of advice specifications, and are therefore elements of reuse. The specification of the language as described in this Chapter can already be used to analyse a large number of CF programs, namely all programs that only use default filter types.

However, custom filter types can be added to the Composition Filters language. In this case, either these custom types can be neglected during analysis, or the developer could specify the behaviour as a graph production rule to be able to include the custom filter type in the analysis.

In fact, the example presented in this chapter is based on the use of custom filter types (Log, Abort, Parent, and Encrypt). However, this was done just to create a simple and small yet interesting example.

5.6.4 Scalability

The complexity of our analysis can be broken into different phases.

The first phase consists of the generation of the graphs contain a class and number of advices. The complexity of this phase is of linear order to the number of classes (C) that are enhanced with more then one aspect.

The second phase is the phase that adds a message to the graphs, with a complexity that is linear to the number of selectors (s) that are referenced in the filtermodules.

The complexity of the runtime simulationa can be divided into two parts. In the first part, the different actual joinpoint graphs are generated, which is a function of the number of different dynamic prediates (p) used in the filter modules. In the second part the execution of all different filtermodule orders at each joinpoint are simulated. The complexity is $a!$, where a is the number of filter modules. The complexity of the simulation is of order $2^p \times a!$ per analysed join point, which expresses the number of paths. The number of conditions will most likely remain small compared to the number of advices, since conditions in pointcuts are commonly known to be a run-time performance bottle-neck. In future work (Section 5.7.2) we propose a different simulation strategy that reduces the size of the generated transition system.

Simulation of a single filter takes approximately ten rule applications. In a (bad-case) scenario where every filter module contains one filter and one condition expression, the size of the state space approaches $10 \times 2^a \times a!$. GROOVE is able to generate state spaces with size up to an order of 10^6 in a timescale of seconds. This allows us to simulate up to around six advices in the assumed scenario (one condition per filter module). Although the occurrence of shared join points is not rare, finding shared join points of six or more advices will not be a common case.

The complexity of the entire analysis of a CF program is of the order $C \times s \times 2^p \times a!$.

5.6.5 Tool support

The approach has been implemented as a *Compose** compiler module, which is a compile-time and run-time implementation of the Composition Filters language. Compose* is available for both the Java and .NET platform. The implementation consists of the generation of program graphs for all classes with more then one filtermodule. The GROOVE API is then called to invoke different graph production systems, namely to:

- add control flow information to the graph (see Chapter 3.5);
- generate the different shadow joinpoint graphs as described in Section 5.4.1;

- generate the LTS representing the executions of the shadow joinpoint, as illustrated in Section 5.5.

Although visually presented in this chapter, the analysis of the LTS is done automatically on the corresponding data structure. Output of this analysis is illustrated in Section 5.5. The tool can optionally show a viewer of the LTS when it detects interference.

The tool has also been integrated into the Common Aspect Proof Environment (CAPE) [DK06], a framework for aspect verification and analysis tools and modules over various aspect languages.

5.7 Conclusions

In this section, we discuss related work, future work, and the contributions of the work described in this chapter.

5.7.1 Related Work

A lot of work has been done in the area that investigates the problems that can occur when aspects are composed. We try to discuss the ones closely related to our approach.

Douence at al. [DFS02] describe a framework to identify overlapping join points and detect possible aspect interference. The abstract formalism can be a basis for future analysis tools, but these have yet to be implemented.

Dürr et al. [DSBA05] propose a semantic conflict detection model for Composition Filters. This model translates the semantics of filter action to operations on resources. The desired behaviour can be specified by means of patterns of operations on a resource, either being a conflict or a requirement. To be able to detect aspect interference, a pattern must exist that can identify the faulty execution. However, the abstraction level of the resource-operation model from the exact behaviour is quite high. Depending on the kind of the interference, it might not be possible to describe the behaviour of the filters on this level such that it is still possible to detect the problem.

Pawlak et al. [PDS05] present a language called *CompAr*, which allows the programmer to abstractly define an execution domain, the advice semantics and the execution constraints of around advices in order to check if the execution constraints are fulfilled when the aspects share a join point. The difference with our approach is that the aspects need to be specified in another language in order to be analysed whereas our approach uses the aspect specification directly (once the aspect language has been specified once).

Havinga et al. [HNBA07] present a graph-based approach to detect composition conflicts related to introductions. These introductions are translated to graph-transformation rules, which are applied to a graph representing the static structure of the program. After applying the transformations, the resulting structure is analysed to detect structural conflicts caused by aspects or aspect compositions.

Lagaisse et al. [LJD04] stress the need to express which modules may use and affect each other in the module composition process. Artifacts can be equipped with contracts that specify the provided functionality and dependencies on other components. For aspects, however, the accepted notion of a contract is no longer sufficient. They propose that aspects require to obey the contractual obligations of the components, such as not allowing breaking scope qualifiers (public/private/protected). They call breaking a contract *uncontrolled semantic interference.* So-called aspect integration contracts are introduced, which specify the permitted interference between an aspect and a base component. Essentially, the work focuses on aspects interfering with the composition of the base system, where our approach focusses on interference among aspects.

Zhang et al [ZCvdBG07] propose to reduce aspect interference problems at a higher level of abstraction – before proceeding to the implementation level – to enhance the reusability of the aspects. They describe how precedence can be declared at the modelling level. Based on precedence declarations, the underlying composition mechanism derives an appropriate weaving and execution order automatically. The work does not concentrate on reasoning about the correctness of a system after composing multiple aspects simultaneously, which is the focus of our approach.

Rinard et al. [RSB04] propose a classification for aspects interacting with methods. The work also mentions that the same classification can be used between aspects. This classification proposes aspects to have interfering scopes when both aspects write to the same field. The classification system is supported by analysis tools that identify classes of interactions and hence help developers to detect potentially undesired interactions. It is left to the user to decide what is problematic and what is not.

Katz [Kat06] extends the aspect classification from [SK03] and, for each category of aspects, classes of properties preserved by the aspects are defined. Katz and Katz [KK08] define semantic interference between aspect and propose an incremental way of checking whether aspect interfere. The technique is based on model checking and requires a formal "assume-guarantee" specification of the aspects. In [KK09] a modular verification technique is proposed for one of the above mentioned categories, namely *strongly-invasive.*

Störzer at al [SF06] also identify the problem of non-commutative advices at shared join points, the so-called advice precedence problem. A mechanism is proposed to detect relevant undefined advice precedence, by detecting common fields used in read and write operations for advice that share join points. In our approach, we also recognise the problem of interference based on read and write operations on fields. However, this does not imply that the advice are interfering. It is possible that two advices make an orthogonal change to the same field. Simulation will indicate whether the operations are orthogonal and commutative or not.

Goldman et al. [GK07] present a modular approach to verify correctness of an aspect relative to a formal specification. The approach is based on model checking using linear temporal logic. We only look at the actual state change of aspects to detect interference, and cannot reason about the intended behaviour. Kniesel [Kni09] presents a analysis method for weaving interaction and interference. It is based on a logical model of aspects, which specifies conditions and operations on program elements. There is some overlap between their approach and ours, but their approach can not detect run-time data interference between aspects.

5.7.2 Future Work

Applicability to other Aspect Languages

Composition Filters has proved itself to be a suitable language for our approach, due to the nature of the advice language. In CF, aspects are be specified in a base language-independent way. The advice language itself is not an extension of the base language, but instead consists of a fixed set of filter-types. Also, advice does not call base-language behaviour but is self-containing. Therefore, simulation of advice does not involve simulation of part of the base-system at all: only the semantics of the filtering mechanism and the fixed set of filter-types have to be specified.

Two issues have to be dealt with in applying the approach to other languages.

First, many other aspect-oriented languages have an advice language that is a variant of the base language. Specifying such an advice language is possible, but is a much larger task than specifying the small CF language. The entire base-language needs to be specified using graph transformation rules. However, applying the approach to a simplified base-language with *proceed* may already be interesting. As a matter of fact, the work presented in Chapter 4 embodies the application of the approach to the Common Aspect Semantic Base (CASB) [DDF08], a formal model of Featherweight Java with assignments and Featherweight AspectJ. The work resulted in a graph production system for simulation of entire programs written in this language, not just joinpoints.

Second, languages like AspectJ employ more comprehensive join point models that allow interception of assignments, constructor calls, static initialization, throwing exceptions, etc. In our approach, the interception mechanism is modelled as a change of the method dispatch mechanism, which is based on method frames. This is feasible because, in the Composition Filters model, the only kind of join points are messages (or method calls) between objects. For more fine-grained join point models, a different graph representation is needed, as not only a run-time process is changed, but in essence any instruction can be intercepted. Automatic join point graph generation becomes a lot more complicated. However, proxy/interceptor based languages such as Spring AOP [JHA$^+$] are becoming more and more popular. These languages typically only intercept messages between objects. Therefore, it is likely that such joinpoints can be represented as presented in this paper — using frames for those messages — since the joinpoint models of these languages are similar to that of CF.

Using Classifications for Optimisation

Classification of aspects helps in understanding the behaviour of aspects. In [SK03], a classification of aspects was proposed as spectative, regulative and invasive types. In [Kat06] these categories of aspects are extended and specified in a more precise way. Also, similar aspect classifications are in [RSB04] and [CL02]. The classification presented in [SK03] can be summarized as follows:

- Spectative aspects produce side-effects orthogonal to the base system. They do not change the base system.
- Regulatory aspects change or abort the control flow of the system.
- Invasive aspects change variables in the system.

These classifications can be easily mapped to the causes for interference at shared join points that were introduced in Section 5.2. Interference between aspects can occur with different combinations of aspect types; interaction of a regulatory or invasive advice and any other type can result in interference. Using this knowledge, we could optimise our approach by skipping join points that are shared by only spectative aspects.

Simulation Strategies

Currently, our approach employs a simulation strategy where unknown fields – needed by an advice – are evaluated before simulating the advice. The advantage of this is that it allows to visually detect interference by looking at confluence of traces that have branched after these fields are evaluated (i.e. shadow join points versus actual join points). The disadvantage is that the advice that uses a condition might not be reached due to a preceding advice aborting the message, such that unnecessary states are created. Traces which involve variables whose values are not going to be read or evaluated (because message abortion occurs before the respective aspect are reached) are essentially equivalent — independent of the values of those variables. An optimization w.r.t. the size of the state-space (and thus the simulation time) would be to lazily assign values to these variables (i.e. when the program counter has reached the actual use of a variable). This however, makes the visual detection of interference hardly feasible and requires detection of interference to be automated to provide understandable analysis results.

In future work, we would like to prove that commutativity is also compositional: if all possible compositions of two aspects from a set of aspects (i.e. for a_1, a_2, a_3 this means the composition of (1) a_1 and a_2, (2) a_2, and a_3, and (3) a_1 and a_3) are free of interference, then the total composition (i.e. a_1, a_2 and a_3) is also free of interference. Instead of simulating all possible advice orders on the shared join points that occur in a given program, this allows to verify just every couple of aspects and will greatly reduce the complexity, especially since this verification can be done *once and for all* for a given set of aspects (and any base system). One could verify a set of aspects to be free of interference by merely analysing every combination of two aspects.

5.7.3 Contribution

In this chapter we present a novel approach to detect aspect interference at shared join points. We use the graph transformation-based Composition Filters semantics presented in Chapter 3. By modelling the specification of aspects and a join point as a graph, we can simulate the execution of the aspects, resulting in a state space of the execution of the join point.

By evaluating the expressions in the advices that depend on the base program and run-time state in any possible way, we can create graph representations for every possible join point; thereby, we can represent the executions of the advices for all join points they apply to in the state space.

Simulating different advice orders allows us to detect aspect interference by analysing whether or not aspects are commutative – whether the order of advice execution at a shared join point affects the result – by analysing confluence of execution paths for different orders in the state space.

The analysis of aspects is done independently of the base system, making the approach more scalable. Once the language semantics supports random advice scheduling, no additional specification is required, making the approach very practical. The approach has been implemented for the Composition Filters language but the approach in general is applicable also to other aspect-oriented languages. We have explained the implications of implementing the approach for languages like AspectJ.

Chapter 6

Verification of Dynamic Constraints

6.1 Introduction

Aspect-oriented programming introduces new composition methods of modules that may complicate the ability of a developer to comprehend the composed behaviour. The obliviousness property of aspect may cause developers to even be un-aware of the existence of aspects. We feel that this increases the need for verification.

System verification aims at verifying whether a system satisfies a set of requirements. Formal verification techniques provide means to determine whether a system is correct with respect to a set of requirements, often called properties, based on a *model* of the system. One such technique is model checking, where the central idea is to verify all possible executions of a model of the system and check whether they satisfy the required properties.

Model checking is based on a modal extension of propositional logic. The properties are specified on the base level by propositions that are satisfied by a subset of the model being checked. The information in each of the states is abstracted to the subset of properties that is satisfied there. Only the information remains that is considered interesting for verification. On top of that we define a modal logic, in which the properties of the lower level are treated as propositions.

There are, however, system properties that are relevant to the correctness of a system and yet cannot be expressed in this two-layered setup. Typically, these are properties where the behaviour of individual entities over time is at issue. It therefore involves tracking individual objects between states (i.e. over time).

In this chapter, we illustrate a mechanism for the verification of such properties of systems by means of an example. This example, that we will be using throughout this chapter, involves the observer pattern [GHJV95] and involves the property "all and only the registered observers at the time of a state change receive a notification of the new state". The time between identifying the observers that require notification and the notification itself leaves existing model checking techniques unsuitable.

Concurrency

This property to be invalidated requires that an observer is added or removed during the execution of the notification. This typically only occurs in concurrent systems. In fact, in concurrent systems, the state of an object may change unexpectedly (by another thread), whereas in linear execution it is much easier to foresee the possible states of the execution. Therefore, we consider our verification mechanism to be typically suitable for concurrent systems.

Aspect-oriented programming

When considering the use of aspect-oriented programming, one may implement the observer pattern using either the object-oriented or aspect-oriented language features. In fact, it may even occur that the implementation "evolves" from OOP to AOP during the development life-cycle. We will illustrate our problem for both paradigms and consider it important that our verification approach can handle such evolution.

We have now identified the following requirements:

- The approach should work for implementations using either object-oriented programming and aspect-oriented programming (Java and AspectJ).
- It should be possible to express dynamic (modal) properties, that express requirements of how the system behaves.
- Property definitions should not be impacted by implementation choices; this means that when implementation changes — even from using Java to using AspectJ — it should take less or no effort to reuse the definitions of the properties that need to be verified.

We define a run-time semantics of Java and AspectJ, based on a common run-time state representation. This semantics is specified using graph transformation rules. Furthermore, predicates are specified as *additional rules* that query the graphs but do not change the model in any way. The use of graphs and graph transformation to represent states, allows us to *add information to the graphs* for the purpose of tracking individual objects; this information is then used by predicate-rules. We use a addition set of rules to initialise the system with a certain number of threads and a number of random actions to be executed per thread, thereby executing possible execution scenarios.

This chapter is organised as follows. In the next section, we illustrate the problem by an example: the observer pattern. In Section 6.3, we explain our approach. In Section 6.4 we give the semantics of Java and AspectJ required for the approach, and in Section 6.5 we explain how we verify the example problem. In section 6.7 we give an evaluation of our approach. In Section 6.8 we discuss related work, future work and the contributions of the work presented in this chapter.

6.2 Motivation by Example: the Observer Pattern

In this section, we motivate the work presented in this chapter by illustrating the problem using the well-known observer pattern as example. We start by explaining the intention of the observer pattern. Then we show problems that can arise when using the observer pattern; first using Java, then with AspectJ.

The intention of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" ([GHJV95]). The dependent are the so-called *observers*, whereas the single object is often referred to as the *subject*. In general, any implementation of the pattern must contain the following features:

- Identify the *subject* and *observer* roles;
- Maintain a mapping between subjects and observers;
- Capture state changes in the subject;
- Implement update logic: notify observers when a subject's state has changed

6.2.1 The Observer Pattern in Java

In Listing 6.1, a very simple observer pattern implementation in Java is shown. The StateHolder class — having the subject role of the pattern — has a field

```java
import java.util.ArrayList;
import java.util.Collection;

public class StateHolder {
   State state = new State();
   Collection<Observer> observers = new ArrayList<Observer>();

   void addObserver(Observer observer) {
      this.observers.add(observer);
   }

   void removeObserver(Observer observer) {
      this.observers.remove(observer);
   }

   void setState(State state) {
      this.state = state;
      for (Observer observer : this.observers) {
         observer.update(state);
      }
   }
}
```

Listing 6.1: Observer Pattern implementation in Java

state of type State; observers want to keep track of changes to this field. A Collection observers contains the registered observers. Registering is done using the addObserver(Observer) method, whereas unregistering is done using the removeObserver(Observer) method. The method setState(State) assigns a new value to the class variable state. After doing so, the registered observers are notified by means of an invocation of the update(State) method on each of them.

In a concurrent setting (i.e. the StateHolder is accessed by more then one thread), a number of undesirable scenarios may occur:

- After the state update, but before or during the notification of the observers, another observer may be registered. This results in the notification of an observer that was not actually registered at the moment the state change took place.

- After the state update, but before or during the notification of the observers, one of the observers may be unregistered. This result in an observer not being notified although it was registered at the actual moment the state change took place.

The observer pattern may also lead to infinite loops, which may happen even in single-threaded programs. This may be caused by cyclic subject-observer relationships, or by state updates of the subject in an update method of an observer (thereby triggering another update etc). In this example, we only focus on the concurrency-related problems.

An obvious solution to the scenario's explained above is to make the three observer-related methods in the StateHolder class synchronised. However, although not directly caused by the observer pattern, this may easily cause deadlock; when an observer requires a lock on a second object during the execution of the update(State) method and the thread holding the lock on the second object executes a call to addObserver on the StateHolder instance, both threads are blocked with no hope of being released. This sort of potential deadlock lurks in many programs that use monitors. To prevent this, we require the subject to have no outgoing method calls while it is locked.

The implementation that satisfies these constraints is shown in Listing 6.2. It uses a synchronised(this) statement in the setState(State) method of the subject. In this block, the new value is assigned and a copy is made of the registered observers. Then, when the lock is released, the observers are notified using this copy.

The above examples and discussions shows that — even for this simple example — a number of unforeseen complications can occur in a multi-threaded setting; this stresses the need for verification of the observer pattern implementation. In the remainder of this section we will show that these complications are even less obvious when aspect-oriented programming is used to implement the observer pattern functionality.

6.2.2 AspectJ

In AspectJ, crosscutting concerns are modularised into class-like modules called *aspects*. Besides elements that are allowed in regular classes, we explain two additional features that aspects can express: introductions, and pointcut/advice declarations. We illustrate these by means of an AspectJ version of Listing 6.1, which is shown in Listing 6.3. First, a stripped-down version of the StateHolder class is shown, with only the required functionality for changing the state. Then, an implementation of aspect ObserverPattern adds the observer pattern implementation to the StateHolder class.

Introductions

Introductions change the static structure of a program. Among others, inheritance relationships can be expressed, and class variables and methods can be added to

```java
class StateHolder {
   State state = new State();
   Collection<Observer> observers = new ArrayList<Observer>();

   synchronized void addObserver(Observer observer) {
      this.observers.add(observer);
   }

   synchronized void removeObserver(Observer observer) {
      this.observers.remove(observer);
   }

   void setState(State state) {
      Collection<Observer> copy = null;
      synchronized (this) {
         this.state = state;
         copy = observers.clone();
      }
      for (Observer observer : copy) {
         observer.update(state);
      }
   }
}
```

Listing 6.2: A synchronised observer pattern implementation in Java

classes (other then the aspect itself). This allows functionally that is required to execute in the context of another class to be specified in a modular way. Examples of introductions are shown in Listing 6.3 on lines 10-18. The observers variable and the addObserver(Observer) and removeObserver(Observer) methods are declared as members of the StateHolder class.

Pointcuts & Advice

Pointcuts are declarations that select events during the execution of a program. This can for example be a method call, execution of a method body, the creation of a new instance, or the usage of a class variable. Lines 16-18 of Listing 6.3 show the declaration of a pointcut setState, which selects the execution of the setState(State) method in the StateHolder class. The target and args expressions are used to bind variables.

Advice is a special kind of method that is executed when an event selected by a pointcut — or so-called *joinpoint* — occurs. Advice can be executed before,

```
 1 class StateHolder {
 2   State state = new State();
 3
 4   void setState(State state) {
 5     this.state = state;
 6   }
 7 }
 8
 9 public aspect ObserverPattern {
10    private Collection<Observer> StateHolder.observers = new
         ArrayList<Observer>();
11
12    void StateHolder.addObserver(Observer observer) {
13       this.observers.add(observer);
14    }
15
16    void StateHolder.removeObserver(Observer observer) {
17       this.observers.remove(observer);
18    }
19
20    pointcut setState(StateHolder subject, State state):
21       call(void StateHolder.setState(State)) && target(subject)
            && args(state);
22
23    after(StateHolder subject, State state): setState(subject,
         state) {
24       for( Observer observer: subject.observers) {
25          observer.update(state);
26       }
27    }
28 }
```

Listing 6.3: Example AspectJ Observer Implementation

after, or around the event. In the latter case, a special *proceed* statement is used to continue the execution of the event that triggered the advice. Advice may be given access to certain variables in the context of the joinpoint. On lines 23-27, an advice is specified that is executed *after* every event matched by the setState pointcut. It calls the update(State) method on each of the observers registered to the StateHolder instance, i.e. the subject.

```
class StateHolder {
   State state = new State();

   synchronized void setState(State state) {
      this.state = state;
   }
}

aspect ObserverPattern {
   Collection<Observer> StateHolder.observers;
   synchronized void StateHolder.addObserver(Observer observer)
        { .. }
   synchronized void StateHolder.addObserver(Observer observer)
        { .. }

   pointcut setState(StateHolder subject, State state) :
      execution(void StateHolder.setState(State)) && target(
          subject) && args(state);

   after(StateHolder subject, State state) : setState(subject,
       state) {
      Collection<Observer> copy;
      synchronized(subject) {
         copy = subject.observers.clone();
      }
      for(Observer observer : copy) {
         observer.update(state);
      }
   }
}
```

Listing 6.4: AspectJ observer implementation using *after* advice

6.2.3 The Observer Pattern using AspectJ

We have already seen that is quite hard to get the implementation of the observer pattern right using Java code. The example AspectJ code shown in Listing 6.3 corresponds to the code shown in Listing 6.1. Again, observers may be added or removed before all observers have received notification of a state change. Synchronisation is required to prevent such scenarios from occurring. We illustrate two variations, one using an after advice, and one using an around advice.

In Listing 6.4, an improved implementation of the observer pattern is given. The addObserver(..) and removeObserver(..) methods have been made synchronised, as well as the setState(State) method in class StateHolder. We have left out some of the method bodies; they remain unchanged. In the *after* advice, a copy is made of the registered observers within a synchronised block that requests a lock on the subject. The around advice is executed *after* execution of the synchronized setState(State) method. It requires in-depth knowledge of AspectJ run-time semantics to actually know whether the after advice is executed before or after the lock on the subject is released.

If the advice is executed during the lock, the synchronised block in the advice has no meaning (because the subject is already locked), and our previously scenario of deadlock is in place; invocations of update(State) on the observers are performed *during* the lock.

If the advice is executed after the lock on the subject is released, there is no actual cause for deadlock. No outgoing method-calls (observer notifications) are made while the subject is locked. A copy of the collection of registered observers is made while a lock is kept on the subject; only after the lock is released, the observers are notified. However, we are unsure if these locks are enough to prevent observers to be added or removed in between the state change and the notification. It may be possible that observers are registered or unregistered between the two locks.

Furthermore, while different developers may be responsible for different modules, the developer responsible for the StateHolder class may not see any reason for the setState method to be synchronised; he may be unaware of aspects that *advice* the execution of the method and require the method to be synchronised.

A possibly "better" implementation is shown in Listing 6.5. Here, the StateHolder implementation has no observer-pattern code at all, and the setState(State) method is not synchronised. The ObserverPattern aspect again introduces the observer collection, and synchronised methods for registering and un-registering observers to the StateHolder class. The pointcut remains unchanged. The advice is now specified as a so-called *around* advice. It replaces the execution of the selected event. In the advice, a synchronised block is used that requests a lock on the subject. In this block, a copy of the observers is made, and a proceed(..) is called, which continues the execution of the matched event (i.e. the setState(State) method in the subject). Because proceed is called while in the synchronised block, there is no need to declare the setState(State) method synchronised. After the lock is released the observers are notified in a deadlock-free manner.

One can see that it requires a lot of knowledge and understanding of AspectJ runtime semantics and concurrency to understand the behaviour of an implementation in a multi-threaded setting. Although aspects modularise the functionality of the observer pattern, this complicates the understandability of the system as a whole. This can become even worse for large-scale projects, where not all developers are aware of the existence of the aspects, or know where they apply. In the worst case,

```
class StateHolder {
  State state = new State();

  void setState(State state) {
    this.state = state;
  }
}

aspect ObserverPattern {
   Collection<Observer> StateHolder.observers;
   synchronized void StateHolder.addObserver(Observer observer)
        { .. };
   synchronized void StateHolder.addObserver(Observer observer)
        { .. };

   pointcut setState(StateHolder subject, State state) :
      execution(void StateHolder.setState(State))
         && target(subject) && args(state);

   void around(StateHolder subject, State state): setState(
       subject, state) {
      Collection<Observer> copy;
      synchronized(subject) {
         copy = subject.observers.clone();
         proceed(subject, state);
      }
      for(Observer observer : copy) {
         observer.update(state);
      }
   }
}
```

Listing 6.5: AspectJ observer implementation using *around* advice

changes to the base code may even break the specification of a pointcut, or add events that are matched by the pointcut.

A possible solution is to automatically verify the implementation of a system for certain — dynamic — properties the system must satisfy. However, the functionality that requires verification may be implemented in numerous variations, and may change over time. Refactorings may even change the implementation from Java to AspectJ. Therefore, we argue that software verification should be possible in a robust manner (i.e. regardless of implementation changes), and be independent of whether AOP is used.

Specifically for the observer pattern, we have established the following properties that we want to verify:

- All and only the registered observers at the time of a state change receive a notification of the new state before the method that implements the state update has finished executing;
- To prevent deadlock occurring caused by the implementation of the observer pattern, if the subject is locked, no outgoing method calls may occur to objects other than the object with a lock.

The first property contains the statement "all on only the registered observers". This requires us to keep track of the actual objects. Only these object may then receive a notification of the corresponding state update. In the next section we introduce our approach for tracking objects.

There are other constraints possible on the observer pattern. For example, we could have a requirement that observers must be notified of state changes in the same order the state changes occur. However, it is not the goal of this work to give a complete observer pattern specification.

6.3 Approach to Verification of Aspect Oriented Programs

In this previous section we have illustrated the difficulty of understanding the behaviour of an implementation, and how this is worsened by the use of aspect-oriented programming. We propose to solve this by automatic verification of system properties. We describe the approach step by step.

First, we define an execution semantics of a programming language run-time. The semantics can be used to simulate graphs representing programs in this language, so-called *program graphs*. This is illustrated in Figure 6.1. The specification language of our choice is graph transformation. By applying all rules in all states, a labelled transitions system (LTS) of the program's execution is generated, where the graphs are the states, and rule applications are the transitions. The *execution graphs* consists of the program graphs and a part representing the run-time state. A similar approach is used in Chapter 3 and Chapter 4.

We define an additional set of rules for verification of properties. The rules serve as state predicates and are applied at the same time as the execution semantics. This is illustrated in Figure 6.2. The rules do not change the execution semantics. Optionally, the rules may add information to the execution graphs to track individual objects; the information may then be used within these rules to specify

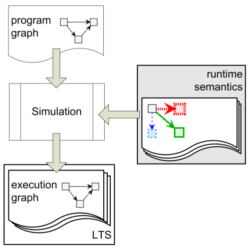


Figure 6.1: Using a runtime semantics.

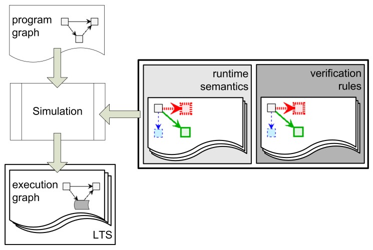


Figure 6.2: Adding rules for verification.

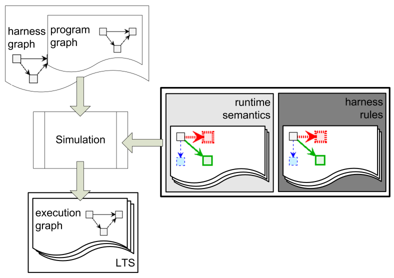


Figure 6.3: Adding a harness.

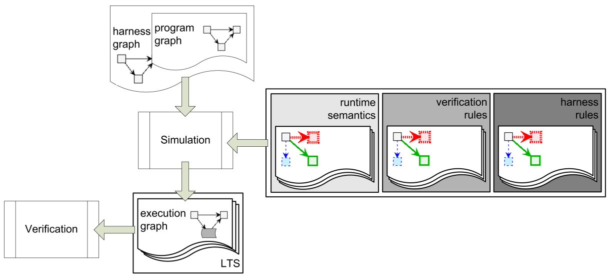


Figure 6.4: Complete overview of the Approach.

state predicates that involve specific objects. The resulting LTS corresponds to the original LTS and additionally reveals applications of verification rules.

Only the part of the program that is relevant for the properties that are checked is simulated in this way. Around this part, a so-called *harness* is added, which specifies instantiation of objects, creation of threads and method invocations. This is illustrated in Figure 6.3. The harness is responsible for simulating all distinct executions of the system that are relevant for the verification of the specified properties. A number of special *harness rules* are used to simulate harness specific actions. Method calls in the harness are handles by the execution semantics. The resulting LTS represents all the executions specified in the harness.

The complete approach is shown in Figure 6.4. The resulting LTS represents all executions that are interesting for verification, and reveals the propositions needed for this verification in the form of applications of verification rules. The LTS is then used for verification. For example, model checking can be applied by using a modal logic over the propositions. Since the verification rules can match specific objects, we can now track individual objects between propositions.

Execution Semantics

The graph based semantics consists of a set of rules for the simulation of Java programs. This semantics includes a representation of the run-time state of a system. Each graph contains such a structure. This is extended by additional rules for pointcut matching, advice execution, and aspect instantiation. We simulate concurrency by executing all threads simultaneously. This means that — in each graph — a rule matches that represents the next execution step of each thread. The result of a full simulation (i.e. exploring all rule applications in all states) represent all possible interleavings of the actions of all threads.

Verification

Although the specified aspectual behaviour requires unusual actions (e.g. triggering methods without the existence of an explicit method call), the run-time states created after application of the rules are very similar to regular Java states. We specify properties that need to be verified as verification rules that reason about this run-time state representation. This should provide the independence of implementation detail we require. As mentioned earlier, applications of these rules also appear in the LTS that is the result of simulation. Verification is done by analysing the occurrence of verification rules in the LTS. For example, some rules should never be applicable, whereas others are required. We illustrate the use of verification rules for the properties identified at the end of Section 6.2. To simulate all possible executions, we use a harness that creates a (configurable) number of threads. Each thread then performs a (configurable) number of actions. Each of these actions is non-deterministically selected from the set of possible actions: addObserver(Observer), removeObserver(Observer), and setState(State). Using multiple threads and multiple actions, simulation includes all possible interleavings of actions (i.e. one thread adds an observer while another thread is updating the state) while having different numbers of registered observers.

Simplifications

The purpose of this work is not to show that it is possible to specify a run-time semantics of an imperative language. We merely want to show that — given such a semantics — we can add rules to be able to verify properties as proposed. Therefore, the run-time semantics only incorporates those language constructs needed to illustrate that the approach is working for the presented example. We argue that this semantics can be extended to incorporate the complete language. For the base language (Java), our semantics includes method-calls, (synchronised) method execution, using variables, assignment, and synchronisation blocks. Other features

— such as inheritance and dynamic method lookup — are omitted. However, rules for these features can be found in [KKR06]; our semantics is based on the semantics defined in this work.

For the aspectual extension (AspectJ), our semantics includes matching *execution* pointcuts, after and around advice, and the proceed statement. We assume that all aspects use the *perTarget* initialisation scheme, which is part of AspectJ. This means that a new instance of the aspect is created for each distinct *target* of a matched event.

For the purpose of running the example, some ad hoc rules are used regarding the handling of collections. Instead of specifying the methods in a concrete Collection implementation, we assume the existence of AddElement, RemoveElement and Clone statements.

To reduce the size of the generated state space, we only simulate the part of the program involved in the properties that are verified. For the observer pattern, this means that we look at state changes (the assignment of a field marked as *state*), and notification of observers (method calls to elements of field of type *Collection* marked as *observers*).

Rule Categories

The approaches uses a lot of rules, which can be grouped into classes for each purpose. We explain these classes one by one:

- **Multi-Threaded Java Semantics**: A set of platform specific rules that models the execution semantics of Multi-threaded Java programs. The rules are specified once and can be used to simulate all programs that use the specified language features. We extended the semantics in [KKR06] with multi-threading and synchronisation, which we explain in Section 6.4.1. The rules for the Java semantics are explained in Section 6.4.2.
- **AspectJ Semantics**: this set of rules extends the Java semantics with a minimal set of AspectJ features. These rules are explained in Section 6.4.3.
- **Harness Semantics**: this set of rules is used to initialise a program with a specified number of threads, initialise objects, and to non-deteministically invoke a number of methods on these objects. The semantics can be reused for different harnesses. The most important rules are explained in Section 6.5.1. We focus on the graph representation of the (problem specific) harness. The rules however can be used to simulate any harness.
- **Verification Semantics**: a set of rules has been defined specifically for verification of properties of observer pattern implementations. We use these

rules to explain how to keep track of individual objects by adding information to the graphs and using this information in the rules. The rules are explained in Section 6.5.2.

6.4 Semantics

In this section we explain the execution semantics of the Java and AspectJ language. The semantics is based on the semantics defined in [KKR06], and similar to the semantics defined in Chapter 3. We have extended this semantics with synchronisation. We first explain the synchronisation model. Then we explain the rules for Java language. Readers already familiar with the semantics can safely skip this part. Then we explain the rules of the AspectJ semantics and the ad hoc rules used for dealing with collections.

6.4.1 Multi-Threaded Java

To be able to simulate multi-threaded Java programs, we introduce a number of concepts to our graphs. These concepts extend the model for normal Java programs, which we discuss in more detail in the next subsection. The type graph of the concepts is shown in Figure 6.5. The most import new elements are `Thread` nodes. Each frame — an element that controls the execution of a method (we discuss these in more detail later) — is executed in a thread, as indicated by a `thread` edge.

A thread can have a lock on an object, which is indicated by a `thread` edge. When a lock exists, synchronised events that refer to the locked object must wait until the object is released (i. e. the lock is removed). We illustrate this by showing the relevant parts of rules that specify such events. The rules are explained in more detail later.

Figure 6.6 shows part of the rule that matches a method call to a synchronised method; the signature referenced by the method call has a `synchronized` self-

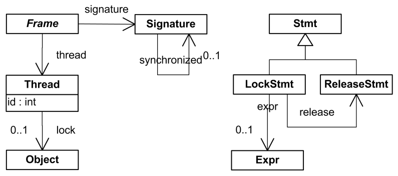


Figure 6.5: Type-Graph for representing multi-threading.

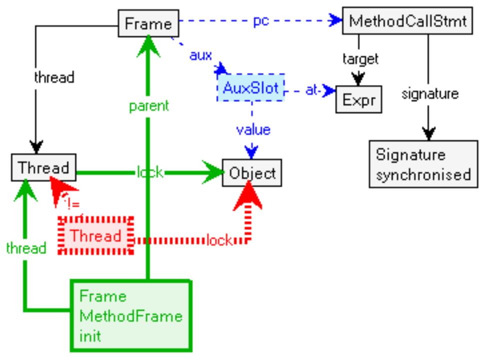
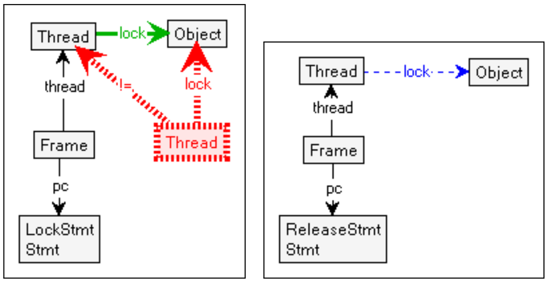


Figure 6.6: Locking mechanism using a synchronised method call.

edge. When the rule is applied, a new frame is created that has a lock on the object. However, this can only be done when *no other thread* already has a lock. The lock is released when the frame is deleted. A synchronized block is represented



(a) Lock (b) Release

Figure 6.7: Locking mechanism using a `LockStmt` and `ReleaseStmt`.

by a Lock and a Release statement, which are executed at the beginning and the end of the block, respectively. The `LockStmt` (Fig. 6.7a) has an argument which evaluates to the object on which a lock is required. Again, the rule can only be applied when no other thread already has a lock on the object. The lock is released when the `ReleaseStmt` (Fig. 6.7b) is executed.

The behaviour of the system having to wait on a lock being released is represented by the rules not being applicable because of an existing lock. The same is seen in simulation of the rule system. Where simulation of other threads may continue (by applying rules on other parts of the graph), the waiting thread can only continue until the rule becomes applicable.

6.4.2 Java Semantics

Graphs

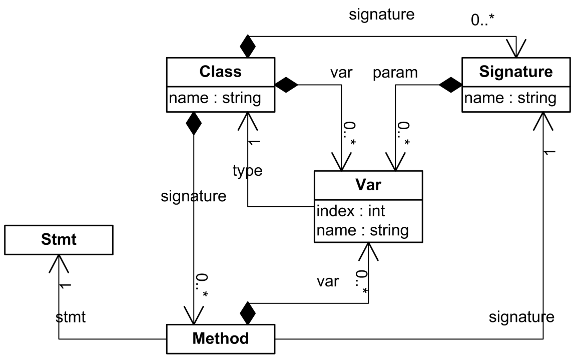


Figure 6.8: Abstract Syntax Graph for Java Programs.

To simulate a program using graph transformations, we must represent it as a graph. The type-graph of such a representation — the so-called *abstract syntax graph* — is shown in Figure 6.8. It consists of `Class` nodes, with `signature` edges to `Signature` nodes, and `method` edges to `Method` nodes. Methods again refer to one of the signatures supported by a class. A signature consists of a `name` and a number of parameters. These parameters are represented by `Var` nodes connected to the signature by `param` edges. Both classes and methods have `var` edges to the `Var` nodes representing class and local variables, respectively. The outgoing `var` edges of a method point to the `var` nodes used by its signature for the parameters. A method has a single statement — most likely a block statement — represented by the `Stmt` node connected to the method with a `stmt` edge.

We must also encode run-time state in the graphs. The type-graph of a heap representation — the so-called *value graph* — is shown in Figure 6.9. It consists of `Object` nodes, with an `instanceof` edge to the `Class` node representing their type. Objects can be stored in slots. `VarSlot` nodes represent containers for the value of a `Var` node (as indicated by the `instanceof` edge, whereas `AuxSlot` nodes are containers for temporary evaluation results (as indicated by the `at`).

Figure 6.10 shows the type graph of the stack frame representation. These frames are used to control the execution of methods, by means of a program counter to the current program element. This program counter is updated until execution is finished. Then, the program counter of the parent frame — which was halted by removing the program counter — is resumed by restoring the program counter. A `Frame` has:

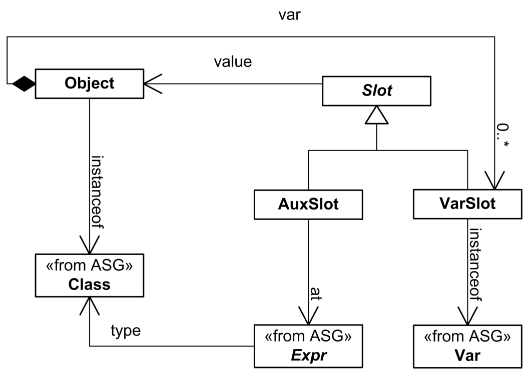


Figure 6.9: Value Graph for representing Heap.

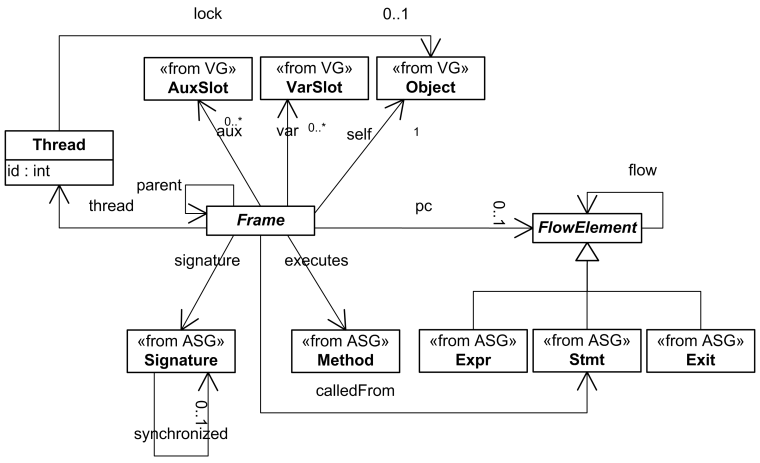


Figure 6.10: Frame graph and Flow graph for representing Stacks Frames.

- a `self` edge to the `Object` in which it executes;
- an `executes` edge to the `Method` it is executing;
- a `signature` edge to the signature of the method;
- `var` edges to `VarSlot` nodes for the values of local variables of the method;
- a `parent` edge to the `Frame` that executed the statement (i.e. a method call) for which the frame was created, and a `calledFrom` edge to that statement;
- optionally a number of `aux` edges to `AuxSlot` nodes that store temporary evaluation results;

- a `pc` edge to the current program element, conveniently always labelled with a `FlowElement` edge. This can be a statement, an expression, or the exit of a method. Before simulation, the internal control flow in method bodies is resolved and explicitly added in the form of `flow` edges.

An *abstract syntax graph* is extended with a so-called *flow graph* to easily update a program counter. The abstract syntax graph and the flow graph together form the so-called *program graph*. A program graph, value graph and frame graph together forms a so-called *execution graph*.

We now enumerate the rules for each of the statements and expressions supported by our semantics. The abstract syntax graphs of statements is omitted, but are also in the rules.

Creating new objects

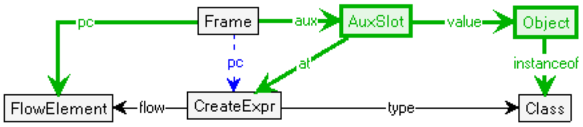


Figure 6.11: The CreateExpr rule

A constructor call is represented by a `CreateExpr` node, which has a `type` edge to the `Class` that is instantiated. The rule for this expression is shown in Figure 6.11. Notice that the rules matches a `Frame` with a `pc` edge to the expression. This `pc` edge is deleted and a new `pc` edge is created to the next `flow` element. As a result of the create expression, a new `Object` is create with an `instanceof` edge to the type referenced by the create expression. This new object is stored in the frame by an `AuxSlot`, with an `at` edge to the create expression.

We do not deal with constructor bodies in this semantics, nor do we initialise class variables; these features are not required for the simulation of the observer pattern implementation.

Variable Usage

So-called *getting* of variables is expression using two different expressions: a `SelfExpr` and a `VarCallExpr`.

The `SelfExpr` represents a *this* reference. The rule for this expression is shown in Figure 6.12. A new `AuxSlot` is created for the expressions; the value of the `AuxSlot` is set to the `Object` node referenced by the `self` edge of the `Frame`.

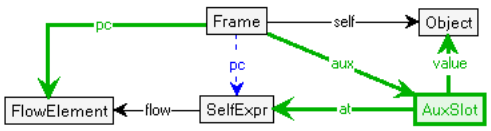


Figure 6.12: The *self* expression.

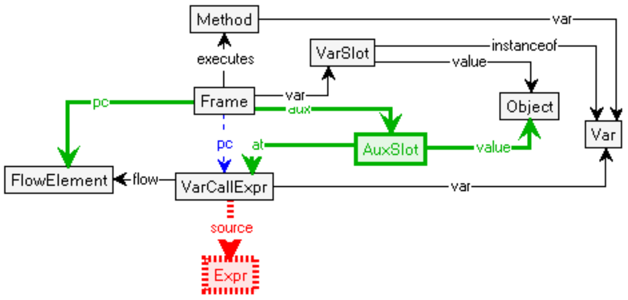


Figure 6.13: Local variable usage.

The rule shown in Figure 6.13 is used for evaluating a `VarCallExpr` without an outgoing `source` edge; it evaluates to the value of the referenced `Var` node, which is a local variable. The value of a new `AuxSlot` is set to the value of the `VarSlot` that is an instance of a local `Var` referenced by the `Frame`.

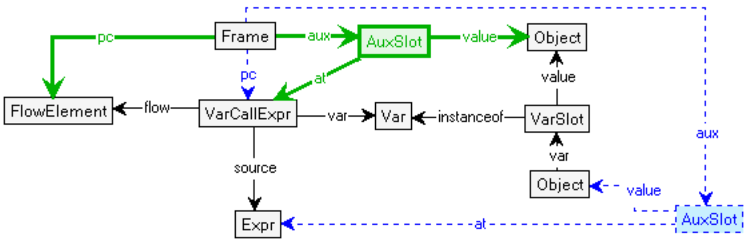


Figure 6.14: Variable use from a source.

The rule shown in Figure 6.14 is used for evaluating a `VarCallExpr` with an outgoing `source` edge. The expression evaluates to the value of the referenced `Var` node as a field of the `source` edge.

Assignment

Assignment of variables is represented by the `AssignStmt` and can be used for local variables and for variables of an object.

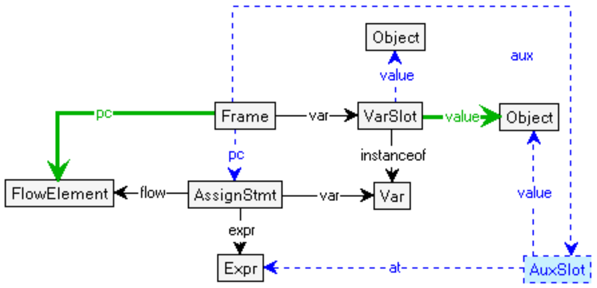


Figure 6.15: The Assign Statement for Local Variables.

Figure 6.15 shows the rule for assigning values to local variables. It matches a `VarCallExpr` without a `source` edge. The value to be assigned is the evaluation result of the expression referenced by the `expr` edge, and is stored in the corresponding `AuxSlot`. The `Var` node referenced by the `VarCallExpr` represents the variable that this value is assigned to. The value of the corresponding `VarSlot` in the `Frame` is deleted, and the new `value` edge is created.

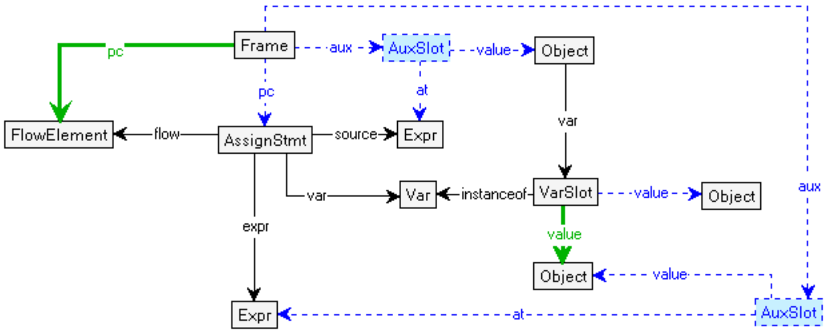


Figure 6.16: The Assign Statement for Class Variables.

Figure 6.16 shows the rule for assigning values to class variables. The difference with the previous rule is that this statement has a `source` edge to another `Expr`. The assignment is performed on a variable of the evaluation result of this expression.

Method Calls

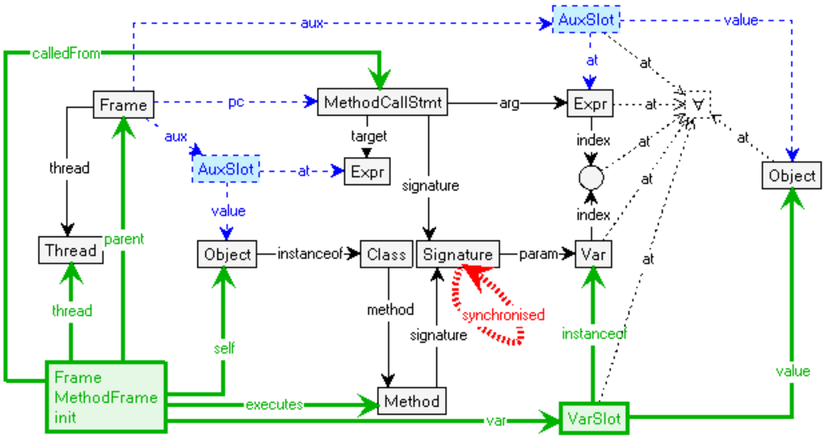


Figure 6.17: The Method Call Statement.

Method calls are represented by a `MethodCallStmt`. The corresponding rule is shown in Figure 6.17. The statement has a `target` edge to an expression; the result of this expression is the target of the method call. The `signature` referenced by the method call is matched with a method with this signature in the class of the target object. A new Frame is created with a parent edge to the current frame. The frame is initialised with an `init` edge that is used later. The program counter of the current frame is replaced by a `calledFrom` edge from the new frame, and the `self`, `executes`, and `thread` edges are created. The remaining nodes and edges represent a parameter transfer from values of arguments to variables of the new frame.

Notice that the `Signature` node may not have a `synchronized` labelled self-edge. We have a seperate rule for synchronised methods.

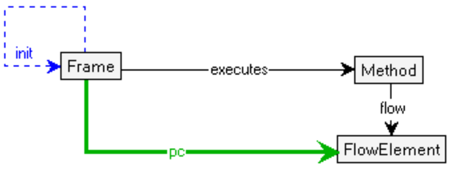


Figure 6.18: Invoke a Method.

Handling method calls and starting executing of the invoked method are separated into two steps. This is done to be able to support around execution advice correctly, as we will see later in this section. The rule in Figure 6.18 represents starting the execution of a method. It matches a `Frame` with an `init` edge. It creates a program counter to the first flow element of the method that it executes.

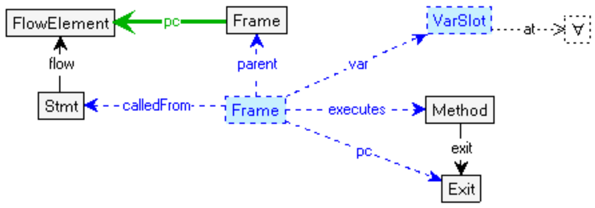


Figure 6.19: Exiting a Method Body.

Once the program counter of a frame reached the end of the control flow of a method body — represented by an `Exit` edge — the frame is removed and the parent frame is resumed. This is done by the rule shown in Figure 6.19. It deletes the current frame, and restored the program counter of the parent from at the next flow element from the statement referenced by the `calledFrom` edge. Notice that all `VarSlot` nodes connected to the `Frame` are deleted also.

Synchronisation

Synchronisation can be specified in two ways, either by declaring a method synchronised, or by using a synchronised block.

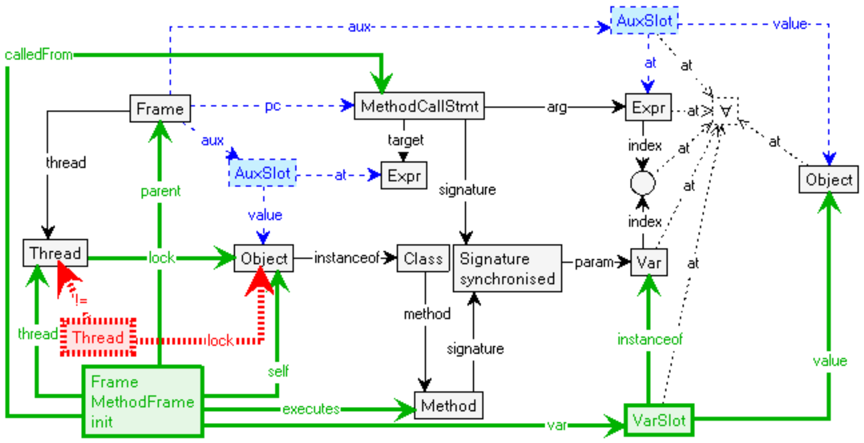


Figure 6.20: A Synchronized Method Call Statement.

Synchronised methods are represented by a special rule for a `MethodCallStmt`. This rule is shown in Figure 6.20. It matches a call with a `synchronized` signature. The rule is identical to the rule for a normal method call, with the exception that it also requires a lock on the target object.

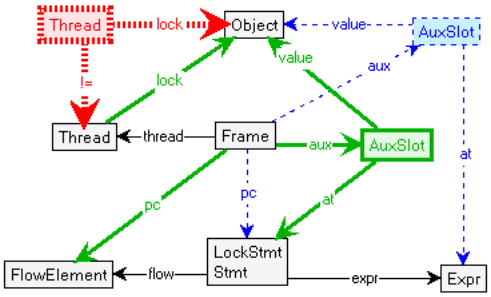


Figure 6.21: The Lock Statement.

As mentioned earlier, synchronised blocks are represented by a `LockStmt` and a `ReleaseStmt`. The rule for the `LockStmt` is shown in Figure 6.21. The statement has an `expr` edge to the expression that evaluates to the object on which a lock is required. For example, for the code synchronized(this){}, the expression is a `SelfExpr`. Again, no other thread may exist with a lock on the object, and when the rule is applied, the `lock` edge is created from the current thread to the object. An AuxSlot is created for the `LockStmt` that referenced the locked object.

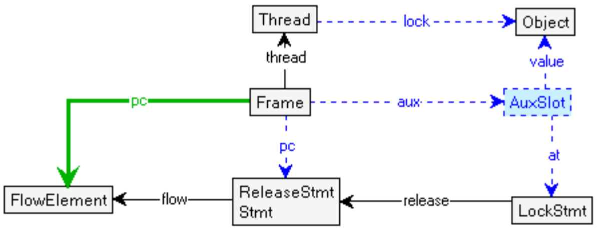


Figure 6.22: The Release Statement.

The rule for the `ReleaseStmt` is shown in Figure 6.22. It is targeted by a `release` edge from the corresponding lock statement; this is used to matched the locked object. When the rule is applied, the lock on this object is removed.

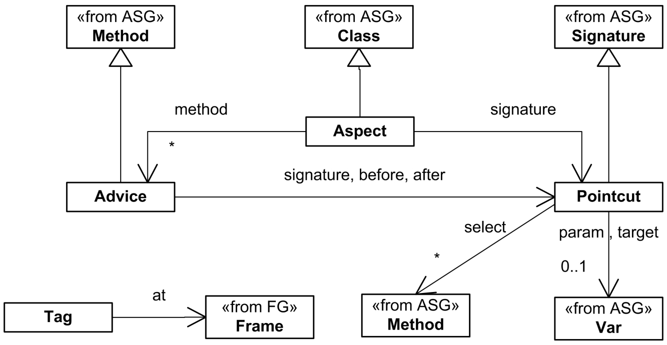


Figure 6.23: Aspect syntax graph of aspects.

6.4.3 Aspects

We specify only a small part of the AspectJ semantics, namely *execution* pointcuts, after and around advice, and the proceed statement. The type graph of the program graph corresponding to the syntax of the aspects is shown in Figure 6.23. Aspects extend regular classes and can thus contain variables and methods. Aspects can also contain advices and pointcuts, which extend regular methods and signatures, respectively. The signature edge of an `Advice` node points to a `Pointcut` node. An additional `after` or `around` edge is required to specify the kind of binding (*before* advice is not included in the semantics). The pointcut selects a number of syntax elements by means of a `select` edge; since the examples only use *execution* pointcuts, the `select` edge always points to a `Method`. Whenever, this method is executed, the execution of the advice is triggered in a way corresponding to the binding specification (i.e. `after` or `around`). The parameters of the advice are given by the parameters specified in its signature. Besides these parameters, a pointcut is specified to bind one more parameter, namely the *target*; for method execution this corresponds to the object in which the method executes. A `Tag` node is used to prevent frames from triggering advice more then once.

The rule shown in Figure 6.24 represents matching an execution pointcut; the `Pointcut` node has a `select` edge to a `Method`. An `Advice` is matched that has an `after` edge to this pointcut. The advice is executed after the selected method by matching a frame that has a `pc` edge to the `Exit` of this method. This frame may not have a connected `Tag` node, which prevents the frame from being matched twice. A frame is created for executing the advice, with the matched frame as its parent. The pointcut acts as a signature for the advice. Parameter transfer is done similar to method calls. An additional variable is initialised to `self` object of the matched frame.

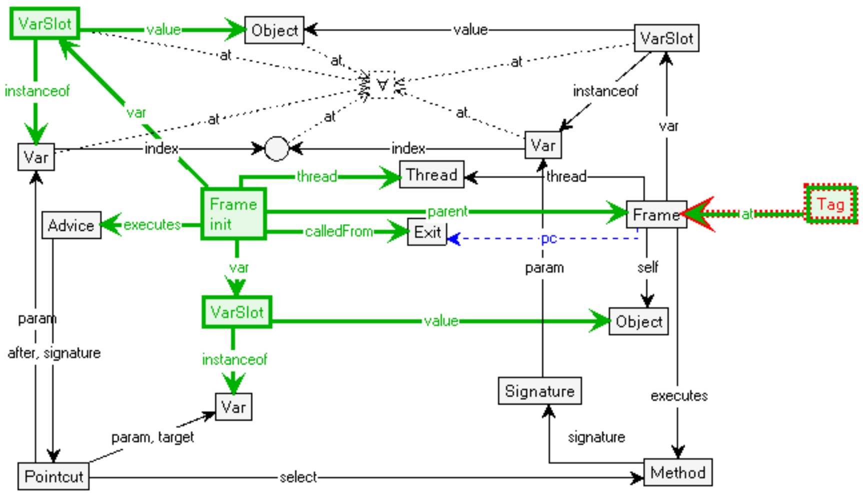


Figure 6.24: After-Execution Advice.

Figure 6.25 shows a similar rule, but for an around advice and an execution pointcut. The different is that it matches a `Frame` with an `init` edge, meaning that execution has not started yet. For around advice, a frame is inserted between the matched frame (corresponding to the execution pointcut) and its parent frame. With the resulting structure, the advice can activate the matched frame; when it is finished, the execution of the advice is resumed. Only when the advice is finishes, the parent frame is resumed.

The created frame gets a `parent` edge to the parent frame of the matched frame. The matched frame is referenced by the advice frame via a `proceedFrame` edge, which indicates that this frame is executed when a proceed statement is executed.

Since we model advice as a method execution, the frame that executes the advice must have a `self` edge. This self is the instance of the aspect. We limit the model to *perTarget* instantiation. This means that an instance of the aspect is created for each distinct target of an event intercepted by the aspect. This instance of the aspect is the context of advice execution. Instantiation only happens once; the second time a target is used, the instance that has been created is reused.

The instance is stored in a `VarSlot` of the object. No corresponding `Var` node exists in the type of the object.The rule in Figure 6.26 creates a new instance of the aspect for the target object if no such instance can be found, and creates a `self` edge from an advice frame to this instance. The rule in Figure 6.27 locates an already existing instance of the aspect, and creates the `self` to this instance.

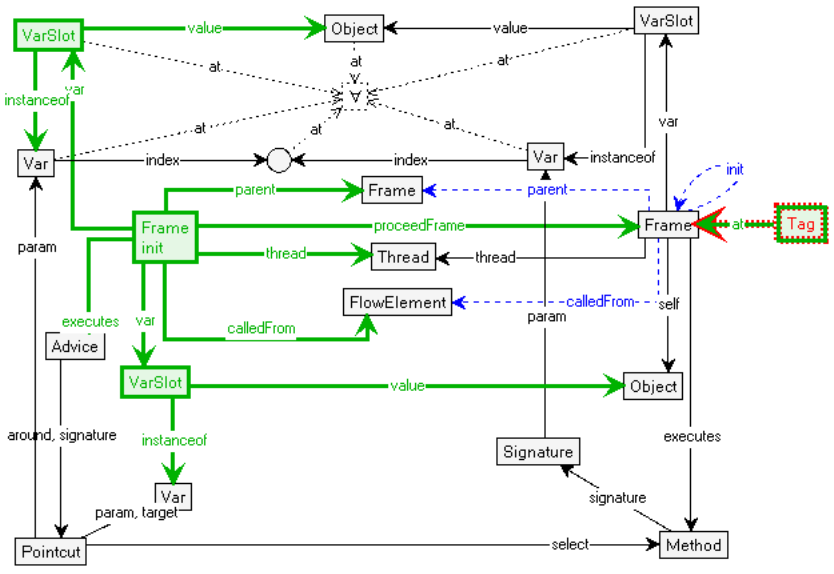


Figure 6.25: Around-Execution Advice.

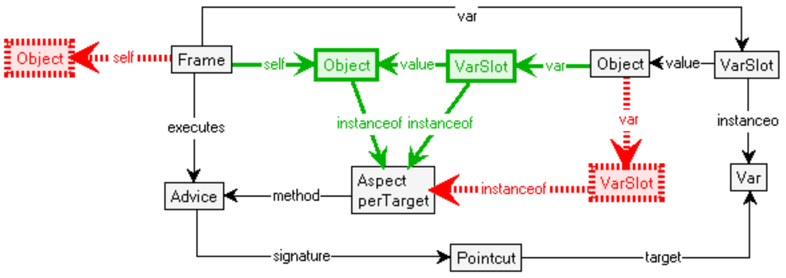


Figure 6.26: Creating an Aspect Instance.

Figure 6.28 shows the rule that matches a program counter at a `ProceedStmt`. The `init` edge of the `proceedFrame` is restored, with the advice frame as its target; when the proceeding frame has finished execution, the advice frame is resumed at the `ProceedStmt`.

A special rule is required for an exit statement in an advice body. When a program counter is at a regular exit (see Figure 6.19), the program counter of the parent frame is restored at target of the outgoing `flow` edge from the node targeted by the `calledFrom` edge. The rule in Figure 6.29 is applied when an exit in an advice occurs and the advice is triggered by the existence of a program counter on the exit

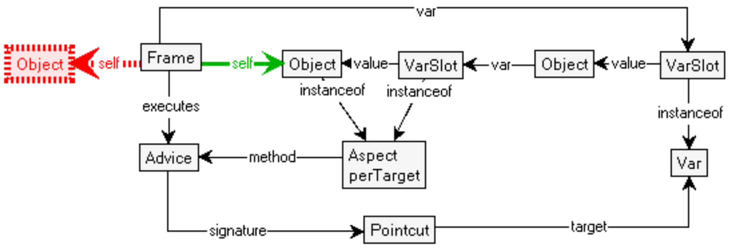


Figure 6.27: Resolving Existing Aspect Instances.

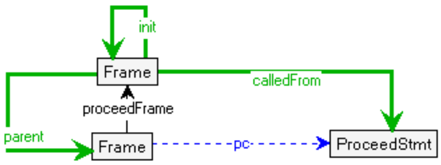


Figure 6.28: The Proceed Statement.

of a method (i.e. an *after execution* advice). Since this exit has no outgoing `flow` edge, the regular exit-rule cannot be applied. Therefore, the program counter is restored at the exit itself. The `Tag` node that was added to the frame prevents the parent frame from being adviced again. The tag is deleted when the corresponding frame no longer exists. This is done by the rule shown in Figure 6.30.

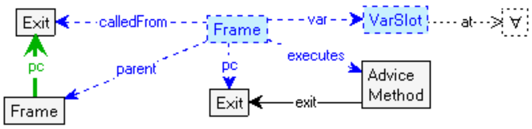


Figure 6.29: Exiting an Advice Body.

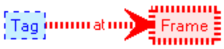
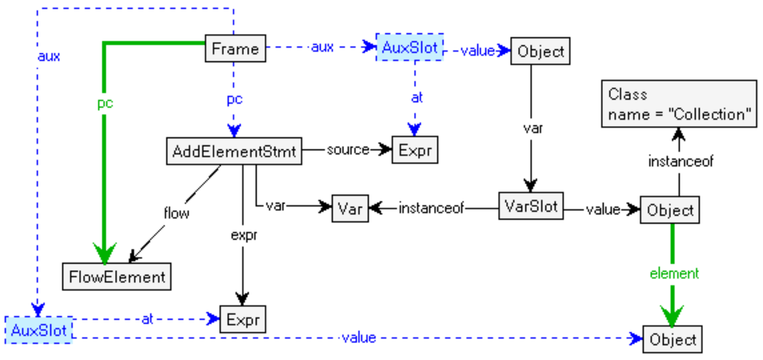
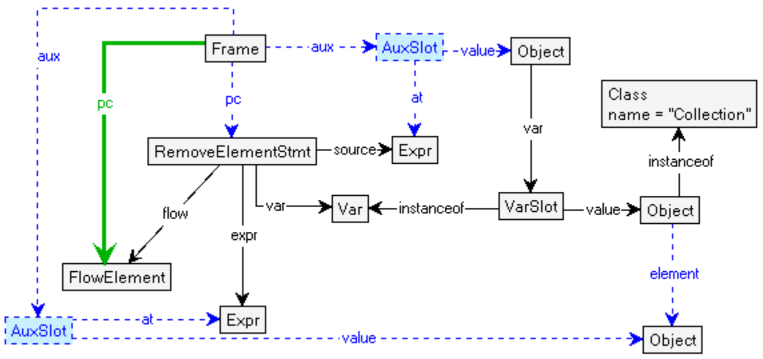


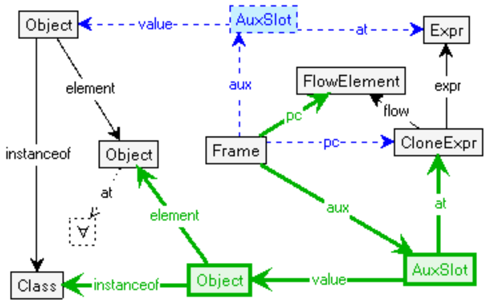
Figure 6.30: Tag Removal.



(a) Adding



(b) Removing



(c) Cloning

Figure 6.31: Rules for using collections.

.

6.4.4 Ad Hoc Rules: Collection Handling

A number of special rules is used to make the semantics described work for the observer pattern examples. These rules allow us to deal with collections without representing the implementation of this class in the graph. We introduce the two special statements AddElementStmt and RemoveElementStmt. The rules are shown in Figure 6.31. They are very similar to calling and assigning variables w.r.t. the usage of the source, var and expr edges. For making copies of a collection, we introduce a CloneExpr, which results in a new object of type Collection, with element edges to all elements of the original collection. The rule is shown in Figure 6.31c.

Two rules are required for iterating over the elements of a collection. We introduce a ForAllExpr, which has an expr edge to the expression that evaluates to an object of type Collection. Furthermore, the ForAllExpr has two outgoing flow edges, namely inflow and outflow. The inflow edge is used while not all elements have been used, whereas the outflow is used when iteration has finished.

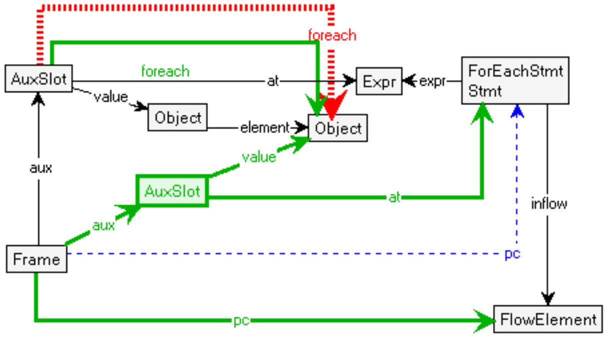


Figure 6.32: ForAllExpr with more elements.

Figure 6.32 shows the rule that matches when not all elements have been used. It matches an element in the collection that has no incoming forall from the AuxSlot, and adds this edge to the matched object. The object is set as the result of the ForAllExpr, can be used by successive flow elements, after which flow will return to the ForAllExpr.

Figure 6.33 shows the rule that matches when all element have been used. It matches when all elements of the collection object are connected to the AuxSlot, and creates a program counter to the target of the outflow edge. Although not specified in a very elegant manner, the rules provide a way to use collections without representing the implementation of a collection and iterator class in the graph.

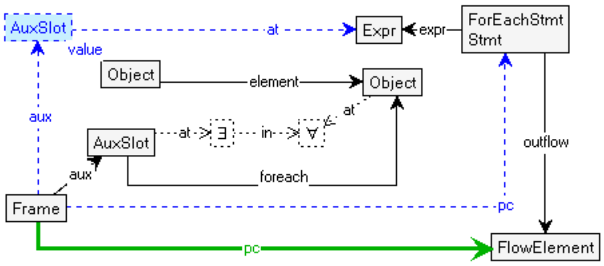
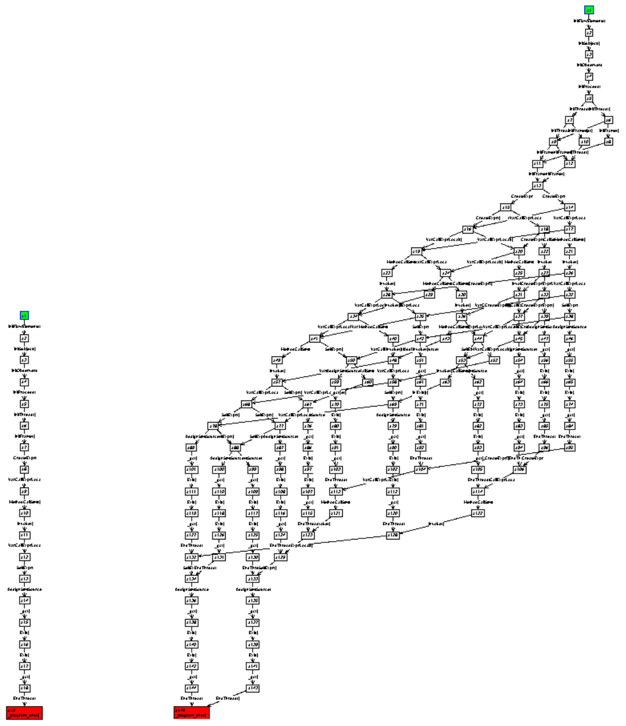


Figure 6.33: ForAllExpr with no more elements.



(a) One Thread (b) Two Threads

Figure 6.34: Resulting state spaces in single- and multi-threaded simulation.

6.4.5 Example

As an example of the difference between single- and multi-threaded simulation, we show the resulting labelled transition systems for a very small program. Figure 6.34a shows the resulting state space of a program that invokes a constructor. The program executes in a linear fashion: only one rule can be applied in each state. Figure 6.34b shows the state space for a program that creates two threads that both invoke the same constructor. Both threads execute in a linear fashion, however, the steps of the one thread can be interleaved by steps of the other thread. The states can be seen as an abstraction of the positions of the program counters of both threads. The state space converges into a single state when the program counters are deleted. The final state contains two objects.

Simulation of Java and AspectJ programs makes no essential difference, except that, for AspectJ, a larger set of rules is used.

```
class Client {
  public static void main(StateHolder sh) {
     int actions = 2;

     for( int i=0; i < actions; i++ ) {
        choose {
           sh.addObserver(new Observer());
        } or {
           sh.removeObserver(sh.observers.getRandom());
        } or {
           sh.setState(new State());
        }
  }
}

main() {
    int threads = 2;
    StateHolder sh = new StateHolder();
    for( int i = 0; i < threads; i++ ) {
       // virtual fork
       new Thread.run(Client.main(sh));
    }
}
```

Listing 6.6: Scenario Program

6.5 Verification of Observer Pattern Implementations

In Section 6.2 we have stated that the following properties must hold during any execution of a program using the observer pattern:

- All and only the registered observers receive a notification of a state change; this notification occurs before the method implementing the state update has finished execution;
- If the thread executing the notifications has a lock on an object, no outgoing method calls may be performed on objects other then the locked object.

We verify these properties by simulating the creation of a number of threads; each of these threads performs a predetermined number of method-calls on a single instance of the observable class (StateHolder). These actions are non-deterministically chosen and can be a call to addObserver(Observer), removeObserver(Observer), or setState(State). The described scenario is shown in pseude-

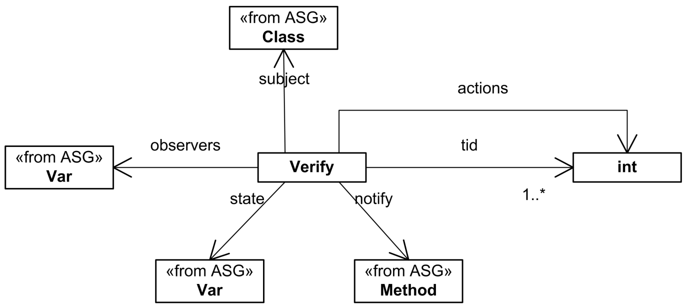


Figure 6.35: Verification and simulation parameters in the graph.

code in Listing 6.6, with two threads and two actions per thread. The main() method creates a single StateHolder instance, and performs a forked call main-(StateHolder) on a new instance of the Client class. There, a choice is made a number of times between the described actions. For adding an observer and setting the state, new instances are used as argument. The removeObserver method is called with a random registered observer as argument; when no observers are registered, it uses a dummy object.

In this section, we first discuss the harness semantics, which takes care of executing the pseudo-program shown in Listing 6.6. Then, we show the constraint semantics and explain how the generated LTS can be used for verification of an observer pattern implementation. Finally, we present our experimentation results.

6.5.1 Harness Semantics

A harness graph has a single `Program` node. This program, which can be executed by a so-called program frame (a special frame that has no parent frame), has a `main` edge to the initial method of the program. For the given pseudo-code, this edge points to the main(StateHolder) method of the Client class. The program node also has an outgoing `flow` edge to an `Exit`, such that its child frame can be finalised in a regular manner. The program is finished when the program counter of the program frame points to the exit of the program.

The harness uses a number of parameters that are represented in the graph as nodes and/or edges connected to a single `Verify` node. These include the roles of syntax elements in the observer pattern may have. The type-graph of the parameters is shown in Figure 6.35. The edges and their targets have the following meaning:

- an `actions` edge to an integer attribute determines the number of method-

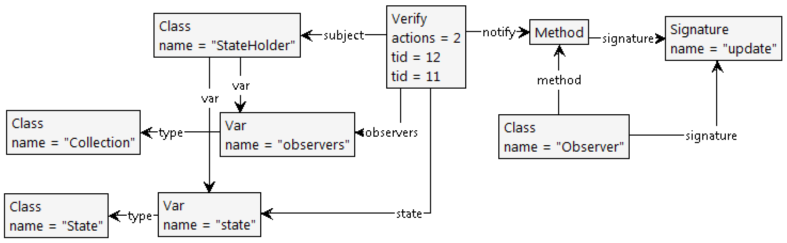


Figure 6.36: Harness parameter graph for the code in Listing 6.6.

calls performed by the Client;

- a number of `tid` edges to integer attributes correspond to thread identifiers; the number of such edges determines the number of threads that is created;
- a `subject` edge to the class that has the subject role; during simulation, this property is propagated to instances of the class (i.e. `Object` nodes);
- an `observers` edge to a `Var` node of type Collection, that contains the registered observers; during simulation, this property is propagated to the value (an `Object` node) of an instance (a `VarSlot`nodes) of the variable.
- a `state` edge to the `Var` node representing the variable of the subject; assignments to this variable correspond to the state changes of interest;
- a `notify` edge to the `Method` in the observers' class that corresponds to notification.

Figure 6.36 shows part of the graph that corresponds to the code in Listing 6.6. The `Verify` node has an `actions` attribute with value 2 and two `tid` attributes. The `subject` edge selects the StateHolder class. The state and observers variables of this class are selected by the `state` and `observers` parameters, respectively. The `notify` parameter is set to the update method in the Observer class.

Having these parameters in a central location in the graph, allows to easily configure the simulation scenario. Once the harness configuration is established for a certain implementation, the rules for verification of the implementation become implementation independent; they merely refer to syntax elements referred to by the harness. For the observer pattern, the above representation already imposes a number of constraints on the implementation. For example, the state of a subject must be represented by a class variable. We feel, however, that the chosen representation and the corresponding constraints implied on the implementation will be correct for most observer pattern implementations. As evidence, we found that all our examples satisfy such constraints.

Without showing the rules of the harness semantics, we describe the resulting behaviour in an informal way:

- A number of required objects is created using the `subject`, `state` and `observers` edge. For the example program, this results in a correctly initialised instance of the StateHolder class, with an empty collection of observers and an initial state.
- A `Thread` node is created for each of the specified thread identifiers.
- For each thread, a number of frames are created (one for each level in the call stack), such that the graph corresponds to a program that has reached line 3 of Listing 6.6. The harness uses the method that is pointed by the program as the `main` method for this step. The frames responsible for executing this method are given an integer attribute that indicates the number of actions left to be executed.
- From this point, the execution semantics takes over. The non-deterministic choice is modelled by having multiple method bodies connected to the "main" method. Each time the program counter is increased to the first `FlowElement` node, any of these method bodies can be selected.
- When the "main" method is at its exit, a special rule takes care of decreasing the number of actions it has left. While the number is greater then zero, the program counter is then reset to the start of the method.

6.5.2 Verification Semantics

Using the execution semantics and the harness we can simulate a program with multiple threads, each of which performs a number of non-deterministically selected actions on an instance of a *subject* class (i.e. the observer pattern implementation).

We now describe a number of rules that are applied during simulation of the program. Some of these rules — the ones that add information to the graph — are given a higher priority than the rules in the execution semantics. This is to make sure that they are applied before the program continues. However, they do not change the part of the graph that represents the program state; regular execution continues without change after the information has been added. In the generated LTS, the application of such a rule can be seen as a proposition that holds in the source state. Other rules — that merely query the graph for the existence of a certain structure — have the same priority as the rules in the language semantics. Since they do not change the graph, these rules can be seen as propositions that hold at a certain state.

First, we must express a rule that detects the occurrence of a state change. This is specified by the rule shown in Figure 6.37. It looks for the existence of an `AssignStmt` on a variable marked as *state* in an object marked as *subject*. Since we are interested in the notification of observers related to this state change, it requires that the number of registered observers is greater than zero.

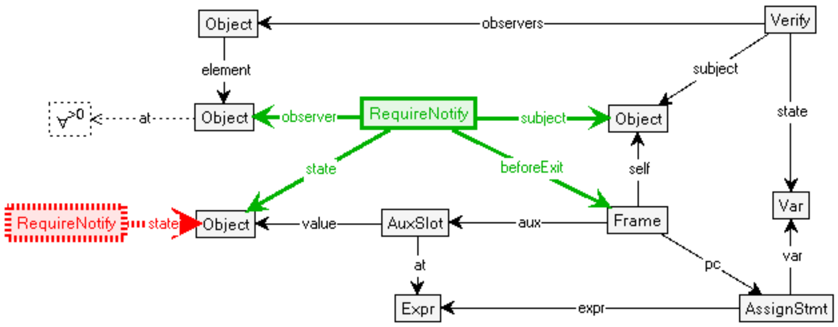


Figure 6.37: Detection of a state change.

A `RequireNotify` node is created that is used for analysis of notifications. It has the following outgoing edges:

- one or more `observer` edges to the registered observers; all and only these observers must receive notification of the state change;
- a `beforeExit` edge to a `Frame`; notification of the observers must occur before this frame finishes (i.e. is deleted);
- a `subject` edge to the object with the subject role;
- a `state` edge to the new value of the state.

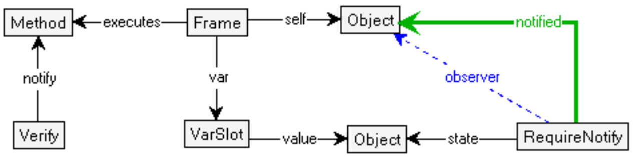


Figure 6.38: Detection of a (legal) notification.

After the detection of a state change, the graph contains a representation of the required notification. Now, we can start detecting these notifications and see if anything goes “wrong”. The rule in Figure 6.38 detects a scheduled notification.

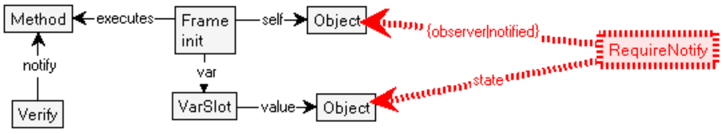


Figure 6.39: Detection of an illegal notification.

A `Frame` exists that executes the method that is marked as `notify` in an object that must be notified (as indicated by the `RequireNotify` node). The new state is passed as an argument of the method call. To "remember" the notification, the `observer` edge is removed and a `notified` edge is added.

Figure 6.39 shows the *_illegal_notify* rule. A notification is detected for a combination of state and observer that are not connected by a `RequireNotify` node. Notice that the rule does not change the graph; its applicability can be seen as a property that holds.

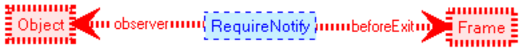


Figure 6.40: Detection of complete notification.

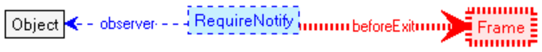


Figure 6.41: Detection of missing notifies.

Once the `Frame` that was marked as `beforeExit` is deleted, we can verify if all observers have received notification. Figure 6.40 shows the *_all_notified* rule that matches when notification for a state change is complete. The case where there are missing notifications is matched by the *_missing_notifies* shown in Figure 6.41. When applied, both rules delete the `RequireNotify` node and its incident edges.

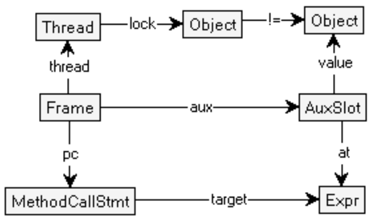


Figure 6.42: Detection of a deadlock causing method call.

Deadlock is detected by the *_deadlock* rule shown in Figure 6.42. It matches an outgoing `MethodCallStmt` from a locked object; the target of the method call must be another object. Obviously, this does not indicate actual deadlock; we merely want to make sure that — if deadlock occurs — it is not caused by the implementation of the observer pattern.

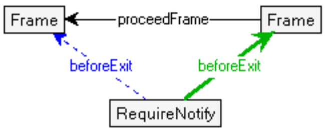


Figure 6.43: Propagation of the `beforeExit` label.

Currently, we have specified that the notifications must be completed before the `beforeExit` frame is deleted. In aspect-oriented implementations, we want to allow notification to take place also in an advice that is executed "around" the method that performs the state change. This is done by the rule shown in Figure 6.43. It propagates the `beforeExit` property of a frame to the frame that executes an advice around the frame. Thus, using the verification rules on aspect-oriented implementation requires only one additional rule.

6.6 Experimentation

Using the given semantics, we can now simulate observer pattern implementations according to the scenario described in the beginning of this section. This will give us a labelled transition system, with transitions for Java and AspectJ actions, as well as transitions that correspond to the rules for observer verification. Given these rules, we merely need to verify that a generated LTS does not contain any transitions labels *_illegal_notify*, *_missing_notifies*, or *_deadlock*.

We have run the described simulation for the following six graph representations of observer implementations:

- the **java** graph corresponding to the source code in Listing 6.1;
- the **java synchronised** graph corresponding to the same example, but with addObserver, removeObserver, and setState made *synchronised*;
- the **java synchblock** graph corresponding to the source code in Listing 6.2;
- the **after** graph corresponding to the source code shown in Listing 6.3;
- the **after synch** corresponding to the source code shown in Listing 6.4;

	threads	actions	states (final)	transitions	_missing_notifies	_illegal_notifies	_deadlock
java	1	2	98 (3)	107	N	N	N
java	1	3	197 (4)	215	N	N	N
java	2	2	15343 (5)	21690	Y	Y	N
java synchronised	2	2	1864 (5)	2519	N	N	Y
java synchblock	2	2	9525 (5)	12916	N	N	N
after	1	2	142 (3)	157	N	N	N
after	1	3	303 (4)	333	N	N	N
after	2	2	20435 (5)	27635	Y	Y	N
after synch	2	2	2848 (5)	3687	N	N	Y
around synch	2	2	12507 (5)	16512	N	N	N

Table 6.1: Verification results of all scenario's.

- the **around synch** corresponding to the source code shown in Listing 6.5;

The graphs that correspond to these implementations are specified by hand. For the usability of the approach an automatic graph generator needs to be implemented. We have experienced that this can be done in a straight-forward way; we have specified such a translator for the semantics in Chapter 3. The challenge lies in the identification of the relevant part of the implementation and the identification of the roles that are used during the simulation.

The simulation results are shown in Table 6.1. The different implementations are simulated for different numbers of threads and actions. To illustrate how the approach scales, we have shown the number of states and transitions in the generated LTS.

For verification, the table shows if the generated LTS contains a transition labelled by the mentioned rules. We merely need to check for the existence of transitions that correspond to rules that express a failure. When no problems occurs, each of these columns should contain an "N". The results correspond to the expectations that we discussed in Section 6.2. As expected, no problems occur in simulations with one thread. The described problems only occur when the implementation is used by more then one thread. The remaining results also satisfy our expectations. Without any form of synchronisation, observers can be added and removed during the notification process. When using only synchronised methods, the implementation can possibly cause deadlock. The Java and AspectJ implementations that use a synchronised block (in which the state is updated and a copy is made of the collection of registered observers) satisfy both our properties.

The number of final states actually corresponds to the different number of registered observers after execution is finished. This should correspond to the product of the number of threads and the number of actions (the maximum number of calls to the addObserver(Observer) method) plus one (0 observers).

The number of states explodes when we introduce concurrency by increasing the number of threads. Comparing the 2-action simulations of the java implementation with one thread and two threads, one could expect the number of states in the 2-thread simulation to have an upper bound of $98 \times 98 = 9604$. This would be true if both threads executed independently. However, data sharing amongst threads creates whole new states and executions, thereby increasing the number of states slightly more. When synchronisation is used, this number is again limited to a certain extend. We can, however, conclude that the simulation scales with the number of threads by s^t , with s the number of states in a single-threaded simulation and t the number of threads. With three threads and two actions, the minimal size of the state space becomes 941192 ($98 \times 98 \times 98$). This can be simulated, but it requires a good amount of memory and patience (or a very fast machine).

Each action causes a choice between the three different methods of the observer pattern implementation. Each time the simulation can choose, three branches are created. After each choice, each branch executions a method. When the method is finishes, another choice can be made. When simulating one action and with thread, the number of method bodies that are executed is 3^1. When simulating two actions with one thread, the number increases to $3^1 + 3^2$. In our experimentation results, the states spaces do not increase to that extend. In fact, the state space doubles in size. This is because a lot of branches merge during simulation because the resulting graphs are isomorphic. This reducing the size of the state space. Also, simulation involves a fixed amount of transitions taking care of initialisation.

In comparing the simulations of Java and AspectJ variations, we can see that AspectJ simulation takes slightly more transitions. The cause of this is an extra frame life-cycle for each joinpoint (i.e. frame creation, invocation, and exit). Also, some extra transitions are needed for creating an aspect instance the first time it is needed, and resolving it afterwards.

6.7 Evaluation of the Approach

In this section we evaluate our approach regarding the outstanding requirements formulated at the start of the chapter. We also give some remarks about the scalability issues of the approach. We base our evaluation on the experiments performed regarding the observer pattern.

6.7.1 Java and AspectJ support

We have shown the simulation of both Java and AspectJ specifications. The program and run-time models used for the approach can be used as a common model for most features of a lot of object-oriented languages. In fact, it is largely inspired by the work presented in [KKR06].

AspectJ programs can partially be represented as regular programs. Introductions for example do not have a special run-time semantics; they merely specify a transformation of the program specification. Once this transformation is resolved, the representation of the program as a graph is that of a typical Java program.

Pointcuts and advices do require some extensions to the Java program model. Pointcuts are represented as extensions of signatures, whereas advices are represented as special methods. During simulation, special rules are applied for matching pointcuts and invoking the corresponding advices. During execution, the run-time state representation of Java programs can be used to express AspectJ executions as well.

6.7.2 Ability to Verify Dynamic Constraints

We can verify properties of a design on existing implementations by specifying additional rules on the static structure and run-time state representation. Using this approach, we can add information to the states, to keep track of past events or required future events.

We have illustrated this for such events regarding the observer pattern. The occurrence of a state change is detected, and information is added to the state to keep track of the observers that need to receive a notification. Then, by detecting such notifications and the moment notification should be completed, we can reason if any notification problems occur.

We have equipped our execution semantics with support for multi-threading and synchronisation. Thereby, we are able to verify these properties in multi-threaded execution. The non-deterministic exploration character of graph transformations allows to easily simulate all possible interleavings of a concurrent system. Every program counter will be matched by a rule. This allows us to verify the specified properties in a concurrent setting.

6.7.3 Implementation Independence

Properties that need to be verified are specified as events related to certain objects in the heap and stack. The events can be recognised using the roles identified in the harness. Thus, once the syntax elements with these roles are identified, verification is done independent of the implementation.

We have illustrated this by our ability to verify some properties for different implementations using Java and AspectJ. The rules required for this verification are largely based on specific heap and stack structures that relate to program elements that are identified with role in the observer pattern.

6.7.4 Automation

To use the proposed approach to its full capacity, the verification must be (largely) automatic. When integrated into the software build process, this ensures that changes do not break properties of the system.

We have shown how a transition system can be generated by an abstract execution of part of the system. Once generated, this LTS can be analysed for the occurrence (as illustrated in the example) certain labels do not occur, or for model checking. Both these analysis methods are easily automated.

The properties need to be specified by hand; once specified, these rules are reusable for different implementations. One should keep that in mind while formulating the requirements as graph transformation rules, to not specify properties in an implementation-specific way.

The missing part to make this approach automatic is tool support for compiling source code to graph representation. The challenging part of implementing these tools is to resolve the relevant part of the source code, and needs to be represented in the graph. Annotations can be used for this purpose. For example, a certain field can be given an annotation that specifies it has the *state* role. A class can be annotated with the *subject* role. This information can be used by the compiler to extract the relevant source code. Warnings can be issued by the compiler about the occurrence of missing annotations.

6.7.5 Scalability

We have listed some scalability results in Section 6.6. As we have stated, the approach does not scale very well when simulating concurrent systems. It then requires to find a suitable pseudo-program — an abstract execution of the code of interest — that represents all scenarios where failure may occur. In our example, simulation of three threads is not possible for most scenarios due to the corresponding size of the state space.

6.8 Conclusions

6.8.1 Related Work

While there are many ways to verify software, (such as model checking, testing, or using assertions), checking run-time properties that require tracking of individual objects is even harder when dealing with aspects-oriented programming. To our knowledge, no verification technique exists that is able to work under the assumption that implementation may change from an object-oriented to an aspect-oriented representation. However, there are some approaches worth mentioning, that are commonly used to verify (concurrent) systems. We discuss model checkers and run-time verification approaches.

Model checking

In model checking of Java and AspectJ programs can can be done by using tools such as Java Pathfinder [RSEG08] (JPF). JPF is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. Two problems arise when using model checkers. First, it requires precise knowledge of the AspectJ weaver to understand the resulting byte-code given a certain AspectJ specification. This complicates the specification of properties in a language-independent manner. Second, it is not possible to track individual objects between states. The properties we have shown in this Chapter cannot be verified using regular model checking approaches.

Run-time verification

Run-time verification [KVK$^+$04] was proposed as a light-weight formal method applied during program execution. The program that is analysed is instrumented with verification logic, such as assertions or pre- and post-conditions. While running the program, the verification logic checks for violations of properties. Compared to our approach, these approaches have two main disadvantages. First, it is hard to verify all executions of the program. While some approaches use backtracking to execute different inputs, running the code in a standard execution environment does not give enough control over the execution to verify all possible inter-leavings of different threads. Second, assertions and pre- and post-conditions typically have a local view on the state of the system. For checking dynamic properties, these programs are required to maintain a global state representation. In our approach, the graph transformation rules have global access to the run-time state. Any additional information required can easily be added to the graph. Also, having to work with actual run-time states (i.e. not having a common abstract run-time representation) makes it very difficult to use run-time verification for both Java and AspectJ.

In general, graph transformation provide is a practical way to simulate all possible executions of a program combined with the ability to specify conditions and properties based on a global run-time state (i.e. without having to maintain a separate representation of this state). Also, because it is not based on a real virtual machine, a run-time state representation can be chosen that is suitable for both Java and AspectJ.

Observer Pattern

A general reference to the observer patterns can be found in [GHJV95]. A discussion on observer pattern implementation using AspectJ can be found in [HK02]. The problems concerning observer pattern implementations that were addressed in this chapter are largely inspired by [Lee06].

6.8.2 Future Work

The semantics given in this chapter is a partial execution semantics for Java and AspectJ. To use the proposed approach to its full potential, a complete semantics of Java and AspectJ is required; for Java, specification of a complete semantics is currently in progress.

Other instantiation strategies require that aspect instances are stored using the value graph model of the base language, i.e. by using `Var` an `VarSlot` nodes. For example, singleton aspect can be stored in a static instance variable of the aspect class.

6.8.3 Contributions

We have proposed to verify dynamic properties of systems using simulation. The proposed approach uses a graph-transformation-based execution semantics. This semantics can easily be used to simulate concurrent systems.

Verification of properties is done by specification of additional graph transformation rules that query the graph-based run-time state representation. If needed, extra information can be added to the graphs to track individual objects over time. Analysis can be done by looking at the existence of certain rule applications in the simulation result.

The approach is easily extended to support aspect-oriented programming. Because the run-time state of the base system and the aspects are represented using a common graph model, properties can be verified on different implementations. Using the approach, implementations can be verified after software evolution cycles, even when such cycles involve refactoring part of the implementation to the aspect oriented paradigm.

The approach has been illustrated for a number of observer pattern implementations and three properties of the execution that may fail when the implementation is used in a concurrent setting.

Chapter 7

Conclusions

7.1 Introduction

In this chapter, we highlight the contributions of the research presented in this thesis. In Chapter 1 we have introduced the basic concepts of the aspect-oriented programming paradigm. Its core features are *quantification* and *obliviousness*. With quantification we mean the ability to select elements of other modules and to add additional behaviour to these elements. Obliviousness means that you can't tell from looking at base code that aspect code will be executed. We have explained the benefits of AOP, that are directly related to its ability to improve *separation of concerns*.

In this thesis, we focus on a number of disadvantages of AOP. Where obliviousness has clear benefits towards the understandability of individual modules, it complicates the understandability of the system as a whole. This has a negative impact on other software engineering qualities of software, such as *maintainability* and *evolvability*. We have stressed the need for automatic verification approaches for aspect-oriented programming languages.

For verification, the general approach that is used in this thesis is the specification of execution semantics of languages. The method of choice for these specifications is graph transformation. They can be used for simulation of programs, and exposes the reactive behaviour of a program in the form of a *graph transition system* (GTS). This GTS is then used for formal verification of the simulated program.

7.2 Controlled Graph Transformation

The work presented in Chapter 2 is not directly related to AOP. In this chapter, we address the problem of controlling rule application when using rule-based specification techniques. Rules are stand-alone entities; they require to specify exactly the conditions required for the rule to be applied. Such conditions may become very complex and may use control information that is added to the state by other rules. Such information quickly complicates the comprehensibility of the entire rule system, and introduces hidden dependencies between rules.

The contribution of this chapter is the definition of control expressions for rule-based systems with a reactive semantics.

We have defined an intuitive language to express control programs over sets of rules. The semantics of the language is described as the control automata that represent programs written in the language. We have defined construction operations from language construct to control automata. The resulting behaviour is defined as the product with a system automaton (an automaton representation of the uncontrolled rule system). The result is a reactive semantics for control expressions. We have explained how the defined product operation can introduce spurious non-determinism and how this can be harmful. To solve this issue, we have introduced a formalism for *guarded control automata* and the corresponding product operation, that — when applied to a deterministic system automaton — results in a deterministic controlled system automaton. We have proved that the languages of products using a normal and the corresponding guarded control automaton coincide.

The work has been implemented in the GROOVE Tool Set for graph transformation-based specification and simulation.

Some valuable future extensions to the developed language are named procedures, atomic transactions, and parameterized rule applications.

7.3 Graph-Based Specification of AOP Execution Semantics

In this thesis, we address the *understandability* problem of aspect-oriented programming languages by specifying its execution semantics using graph transformations.

We have performed this task for three different languages, that all share the following main contributions:

- *Understandability*: We believe that the visual nature of the graph transformation rules will appeal to many readers that are not experts in mathematics. This benefits to understanding a language based on the given semantics. The fact that aspect-oriented programs as a whole can be hard to understand increases the need for means to assist in this matter.
- *Analysability*: By giving the semantics in this way, the road is opened towards applying existing verification methods. Also, the labelled transition systems that result from simulation can directly be used for model checking.

We now describe the main properties of each of the specified semantics, and the main contributions of the work involved:

7.3.1 Composition Filters

In Chapter 3 we have specified the execution semantics of Composition Filters (CF). The semantics described a control flow semantics and run-time semantics for filter matching, and a run-time semantics for the actions performed by the filters to manipulate the base program. No base language semantics is defined.

The main contributions are:

- The defined semantics can be used as a reference semantics for Composition Filters. Before, the run-time behaviour of the language was merely given informally, using natural language.
- The semantics can be used to simulate the execution of filter modules in a modular (i.e. without having a concrete base system). The nature of Composition Filters provides means to represent only an abstraction of the base program.
- Defining a control flow semantics has learned us that the control flow of Composition Filters is not very straight-forward. In fact, we have defined and used an improved abstract syntax model for filter modules to simplify the control flow semantics. We propose this model as an improvement for the language.

7.3.2 Featherweight AspectJ

In Chapter 4 we defined an execution semantics for a base language and an aspectual extension. The base language is Featherweight AspectJ with assignments; the aspectual extension consists of method call pointcuts and around advice with a proceed statement. The semantics can be used to simulate the entire execution of a program specified in the language. We have shown a correspondence relationship with a reference semantics, that formally describes the same

The main contributions are:

- We have demonstrated that a graph transformation based operational semantics is a formal speci

  cation technique and can be complete with respect to a certain reference semantics.

- We have shown that using graph-transformation for the specification of operational semantics benefits rigorness. The directly executable nature increases ease and confidence of specification of a semantics by giving the user a way to test the semantics without having to write a interpreter first, which may contain errors either copied from the semantics or made during implementation.

7.3.3 Java and AspectJ

In Chapter 6, we have defined an execution semantics for a subset of Java, largely based on an existing semantics. We have extended this semantics to support: (1) multi-threading and locking mechanisms; (2) a subset of AspectJ. The main contribution of the semantics is the definition of a multi-threading model using graph-transformations. We have demonstrated that this can be used to simulate all interleavings of a program.

7.3.4 Future Work

None of these semantics specifies all features of an aspect-oriented language, or lacks a full base language semantics. Future work in the developing formal verification techniques for aspect-oriented languages can greatly benefit from an execution semantics of a popular aspect-oriented programming language such as AspectJ (including Java), with no limitations whatsoever.

7.4 Analysis of Aspect Interference on Shared Joinpoints

In Chapter 5 we address the problem of aspect interference on shared joinpoints. Two or more aspects behaving correctly when applied in isolation, may interact in an undesired matter when applied together. We have identified *data interference*, *control interference* and *scheduling interference*, and have illustrated the first two by means of an example.

We have proposed an approach that involves simulation of all different advice orderings at such joinpoints. We use confluence analysis to detect if the order of execution affects the resulting state.

We have illustrated the approach for Composition Filters. Therefore, we have extended the semantics of Chapter 3 to support the approach. The approach in general is applicable also to other aspect-oriented languages. We have explained the implications of implementing the approach for languages like AspectJ.

The analysis of aspects is done independently of the base system, making the approach more scalable. If desired, however, the analysis can also be applied on a concrete program with aspects. For example, the approach is directly applicable using the semantics defined in Chapter 4.

The presented verification method can be applied to other aspect-languages. However, this requires that the semantics of these languages are specified using graph transformations, which — for many AO languages — requires the specification of the complete base language (i.e. Java or C#). Therefore, a highly recommended topic for future work is a full base language semantics.

Also, scalability can be improved by researching different possibilities for optimisation.

7.5 Analysis of System Properties under Concurrent Execution

In Chapter 6 we focused on verification of dynamic properties that require tracking of individual objects over time.

We have stressed the need for automatic verification of such properties, as its benefits correctness. This is even more the case when using aspect-oriented programming, as the obliviousness of AOP can cause unintended mistakes during refactoring.

The proposed approach involves augmenting an existing execution semantics with rules that describe the properties of state that are interesting for verification. If needed, information can be added to the states to “tag” objects. This information can then be read by verification rules that are applicable at a later moment.

The properties are described in a language-independent way, by referring to objects that are tagged with certain roles. Thereby, the approach is very tolerant w.r.t. implementation details, even when this involves refactoring from the OOP to the

AOP paradigm.

We have illustrated the approach for a number observer pattern requirements on different implementations.

Future work requires an in-depth study of the characteristics of these "modal object-constraints" and the development of a full verification approach, whereas in this thesis we merely described the problem and illustrated the approach by an example.

7.6 Reflection and Future Work

Graph transformations have proven to be an intuitive way to specify systems, and in particular object-oriented languages and programs. Using boxes for entities and edges for relationships between entities are not uncommon to many people; many have already used modelling languages like the UML. Personally, the author believes that the graphical notation is easier to learn to understand then textual formal specification languages. Using graphs for the purpose of specifying programs and semantics, however, requires a generator for the actual graphs, and proper support for debugging, because the graphs themselves easily grow too big to "read". Once familiarised, graphs and graph transformations are just a lot of fun.

The rule control language described in Chapter 2 has already proved itself even for different purposes then the intended specification improvements. In [Cir09], control expressions are used for checking CTL (like) expressions on-the-fly. For the authors, it has been a useful improvement for specifying readable graph-based operational semantics. A negative remark: we have not used the control language for any of the semantics in this thesis. The reason for this is that the control language was developed after having specified the semantics.

In Chapters 3, 4 and 6, we have used graph transformations for the specification of an operational semantics of different aspect-oriented languages. However, we have managed to specify only a portion of all features of a small number of aspect-oriented languages. There are many different aspectual extensions, some of which can even be used in conjunction with different object-oriented languages. The large variety of aspect-oriented composition features that exists, and the even larger number of ways to specify these compositions, is an indication that people in the field of language design are still struggling with a better way to achieve separation of concerns.

For aspect-oriented programming to grow into the next-generation programming paradigm, we feel a big step is needed into a paradigm where the composition mechanisms of object-oriented and aspect-oriented programming into an integrated whole of the best of both. In the introduction, we have already mentioned symmetric AOP; where this comes closer to a “grand vision” of AOP. Sadly, asymmetric AOP (i.e. extending an existing system with some features) has turned out to be more pragmatic. We like to add ”in the short term” to this sentence. The common practice of implementing aspect-oriented features on object-oriented virtual machines, has withheld us from taking that one large step further.

The verification approaches in this thesis (Chapter 5 and 6) are applicable to a number of these aspect languages. Each of these semantics has at least some limit w.r.t. its completeness or the usefulness of the specified language. Above we have mentioned that in future work, a more complete semantics of an accepted language like AspectJ is required. However, we feel that the most important benefit of such research lies in creating awareness of complications of the novel composition mechanisms that AOP offers. Future work may then lead to the checking of the described issues in aspect-oriented compilers.

Samenvatting

Aspect-geörienteerde software ontwikkeling is geïntroduceerd als middel om de modulariteit van software te verbeteren tijdens alle stadia van het software ontwikkelproces, van architectuur tot implementatie. Aspect-geörienteerd programmeren biedt bovenal de mogelijkheid om zogeheten "crosscutting concerns" op code niveau op modulaire wijze te specificeren.

In dit proefschrift trachten we verificatietechnieken te ontwikkelen voor aspect-geörienteerde programmeertalen in het bijzonder. Daarvoor modelleren we het gedrag van zulke talen als een operationele semantiek. We gebruiken graaf transformaties om deze semantieken te specificeren. Graaf transformaties hebben een wiskundige grondslag and bieden een intuïtieve manier om componentgebaseerde systemen — zoals software systemen — te beschrijven. Daarnaast hebben graaf transformaties een uitvoerbaar karakter en kunnen ze worden gebruikt voor het genereren van een toestandsdiagram van de uitvoering van programma's. Deze toestandsdiagrammen gebruiken we voor de verificatie van met behulp van aspecten geïmplementeerde programma's.

We beginnen met het definiëren van een uitbreiding op de specificatie van regelgebaseerde systemen. Pure regelgebaseerde systemen bestaan normaliter uit een ongestructureerde verzameling regels. Het gedrag van zulke systemen komt neer op het mogen toepassen van iedere regel in elke toestand. Regels kunnen zodoende alleen gedwongen worden in een bepaalde volgorde te worden toegepast door speciale elementen toe te voegen aan de toestanden, waarop vanuit de regels wordt getest. Met andere woorden, controle op regeltoepasbaarheid is niet expliciet, maar moet worden gecodeerd in de toestand; dit reduceert de begrijpelijkheid en onderhoudbaarheid van regelgebaseerde systemen in zijn geheel. We introduceren zogenaamde *controle automaten* die kunnen worden toegevoegd aan pure regelgebaseerde systemen. Het resulteren gedrag is gedefinieerd als het product van de oorspronkelijk toestandruimte en de controle automaat. Onze controle automaten bevatten zogenaamde mislukkingtransities, waarmee de ontoepasbaarheid van één of meer regels kan worden gerepresenteerd. Het resultaat is een reactieve semantiek voor controle expressies, dat zich onderscheidt van de gebruikelijke invoer-uitvoer semantiek. Controle automaten kunnen kunstmatig non-determinisme introduceren. Om dit ongewenste effect te vermijden introduceren we *bewaaktec ontrole automaten*, en we definiëren een semantiekbehoudende transformatie van normale naar bewaakte controle automaten.

In het volgende deel van het proefschrift beschrijven we de run-time semantiek van een aantal aspect-geörienteerde programmeertalen, te weten (1) Composition Filters, (2) Lightgewicht Java met assignment en een aspect-geörienteerde uitbreiding en (3) een deel van Java met meerdere threads en een deel van AspectJ. We laten zien hoe graafgebaseerde semantieken kunnen helpen bij het begrijpen van het run-time gedrag van een programmeertaal. Bovendien laten we zien dat zo'n semantiek can worden gebruikt om een (gedeeltelijk) programma te simuleren en dat we daarbij de afzonderlijke stappen van het programma zichtbaar maken. We illustreren dat de uitvoerbare natuur van graaf transformaties de strictheid van de specificatietechniek ten goede komt; fouten kunnen eenvoudig worden gedetecteerd door de simulatie uit te voeren. Als laatste laten we zien dat de uit de simulatie gelabelde transitiesysteem gebruikt kan worden voor bestaande verificatie technieken.

Daarna introduceren we twee nieuwe manieren voor het addresseren van complicaties veroorzaakt door het gebruik van aspect-geörienteerd programmeren.

De eerste aanpak richt zich op een probleem dat zich voordoet in verschillende aspect-geörienteerde talen. Aspecten die afzonderlijk correct functioneren, kunnen interactie vertonen wanneer te worden gecombineerd. Wanneer zulke interactie het gedrag van aspect beïnvloed of een aspect uitschakelt, noemen we dit interferentie. Een specifiek type interferentie vindt plaats wanneer twee aspecten van worden toegepast op een gedeeld verbindingspunt (*join point*), omdat dan de volgorde van uitvoeren van de aspecten het samengestelde gedrag kan beïnvloeden. We presenteren een aanpak voor het detecteren van zulke interferentie van aspecten op gedeelde verbindingspunten. De aanpak is gebaseerd op het simuleren van alle mogelijke volgordes van de door de verschillende aspecten uitgevoerde functies gekoppeld aan een gedeeld verbindingspunt. Een analyse van de confluentie van de resulterende toestandsruimte wordt uitgevoerd om te detecteren of de volgorde de resulterende toestand heeft beïnvloed.

De tweede aanpak richt zich op het verifiëren van dynamische eigenschappen van systemen. Zulke eigenschappen kunnen alleen worden geverifiëerd door het uitvoeren van het systeem te simuleren: een uitvoer semantiek is benodigd. Meer in het bijzonder richten we ons op eigenschappen waarvoor het benodigd is individuele objecten over de tijd te traceren. We benadrukken de noodzaak om zulke eigenschappen bij het gebruik van aspect-orientatie automatisch te kunnen verifiëren vanwege de onbewustheid-eigenschap van aspecten: een ontwikkelaar kan aan het basisprogramma niet zien dat er aspecten op van toepassing zijn. Wanneer de software zich wordt gewijzigd kan bestaande functionaliteit onbedoeld stuk raken. We stellen voor om bestaande uitvoersemantiek uit te breiden met speciale verificatieregels. Deze regels kunnen indien nodig informatie toevoegen aan de grafen om bepaalde objecten te kunnen volgen. Eigenschappen zijn gemodelleerd als interacties tussen objecten met een bepaalde rol. Zodra de objecten met deze rollen zijn geïdentificeerd kan een programma worden geverifiëerd, ongeacht hoe deze is geïmplementeerd. We laten zien dat de aanpak toepasbaar is op zowel object-geörienteerde als aspect-geörienteerde programma's.

Bibliography

[ABV92] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In O. Lehrmann Madsen, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP)*, pages 372–395. Springer-Verlag Lecture Notes in Computer Science, 1992.

[ARS09] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-Oriented Software Development*, pages 39–50, New York, NY, USA, 2009. ACM.

[AT88] M. Akşit and A. Tripathi. Data Abstraction Mechanisms in SINA/ST. In *Proceedings of the conference Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 23, pages 267–275. ACM Sigplan Notices, 1988.

[AWB$^+$93] Mehmet Akşit, K. Wakita, J. Bosch, L. M. J. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer Verlag, London, 1993.

[BA04] L. Bergmans and M. Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect Oriented Software Development*, pages 63–95. Addison Wesley, Boston, 2004.

[BHR84] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BJJR04] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In *In Concur*, pages 209–224. Springer, 2004.

[BKKM02] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a Rewriting Logic Point of View. *Theor. Comput. Sci.*, 285(2):155–185, 2002.

[CDE$^+$02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[CDFR04] Andrea Corradini, Fernando L. Dotti, Luciana Foss, and Leila Ribeiro. Translating java into graph transformation systems. In Hartmut Ehrig, Gregor Engels, Fransesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Second International Conference on Graph Transformation (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 383–389. Springer-Verlag, 2004.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, 1999.

[Cir09] S. Ciraci. *Graph Based Verification of Software Evolution Requirements.* PhD thesis, Univ. of Twente, Enschede, November 2009.

[CL02] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Ron Cytron and Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, March 2002.

[CL05] Curtis Clifton and Gary T. Leavens. MiniMAO: Investigating the semantics of proceed. In Curtis Clifton, Ralf Lämmel, , and Gary T. Leavens, editors, *FOAL 2005 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2005, Chicago, IL*, number 05-05 in TR, pages 57–67, Ames, IA, 50011, March 2005. Dept. of Computer Science, Iowa State University.

[CLW03] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Formal definition of the parameterized aspect calculus. Technical Report 03-12b, Iowa State University, Department of Computer Science, November 2003.

[DDF08] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Specialized aspect languages preserving classes of properties. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 227–236, Washington, DC, USA, 2008. IEEE Computer Society.

[Der05] Nachum Dershowitz. Term rewriting systems by "terese" (marc bezem, jan willem klop, and roel de vrijer, eds.), cambridge university press, cambridge tracts in theoretical computer science 55, 2003, hard cover: Isbn 0-521-39115-6, xxii+884 pages. *Theory Pract. Log. Program.*, 5(3):395–399, 2005.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *1st Conf. Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Berlin, 2002. Springer-Verlag.

[Dij74] Edsger W. Dijkstra. On the role of scientific thought. published as [Dij82], August 1974.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[DK06] Eyal Dror and Shmuel Katz. The revised architecture of the cape. In *Deliverable 42, AOSD-Europe, EU Network of Excellence in AOSD*, August 2006.

[DSBA05] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume II: Applications, Languages and Tools. World Scientific, Singapore, 1999.

[EEKR00] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Theory and Application of Graph Transformations (TAGT)*, volume 1764 of *Lecture Notes in Computer Science.* Springer, 2000.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation. part ii: single pushout approach and comparison with double pushout approach. In *Handbook of graph grammars and computing by graph transformation: volume I. foundations*, pages 247–312, River Edge, NJ, USA, 1997. World Scientific Publishing Co., Inc.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:167–180, 1973.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In Ehrig et al. [EEKR00], pages 296–309.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Boston, MA, January 1995.

[GK07] Max Goldman and Shmuel Katz. Maven: Modular aspect verification. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2007.

[GR82] Claude Girault and Wolfgang Reisig, editors. *Application and Theory of Petri Nets, Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets, Stasbourg 23.-26. September 1980, Bad Honnef 28.-30. September 1981*, volume 52 of *Informatik-Fachberichte.* Springer, 1982.

[Hau06] Jan Hendrik Hausmann. *Dynamic Meta Modelling: A Semantics Description Technique for Visual Modeling Languages.* PhD thesis, University of Paderborn, Germany, 2006.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.

[HNBA07] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings* *of the 6th international conference on Aspect-oriented software development, Vancouver, Canada*, pages 85–95, New York, NY, USA, 2007. ACM Press.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.

[HP05] Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In Hans-Jörg Kreowski and et al., editors, *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[Jan99] D. Janssens. Actor grammars and local actions. In Grzegorz Rozenberg, Hartmut Ehrig, et al., editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume III: Parallelism, Concurrency and Distribution, chapter 2, pages 57–106. World Scientific, Signapore, 1999.

[JHA$^+$] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, and Rick Evans. Aspect oriented programming with spring.

[JJR03] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *In European Conference on Object-Oriented Programming*, pages 54–73. Springer-Verlag, 2003.

[Kas08] H. Kastenberg. *Graph-based software specification and verification.* PhD thesis, Univ. of Twente, Enschede, October 2008.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. In Awais Rashid and Mehmet Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2006.

[KD02] Gregor Kiczales and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *ACM Transactions on Programming Languages and Systems*, pages 1–8, 2002.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[KK96] Hans-Jörg Kreowski and Sabine Kuske. On the interleaving sematics of transformation units - a step into GRACE. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Graph Gramars and Their Application to Computer Science (TAGT)*, volume 1073 of *Lecture Notes in Computer Science*, pages 89–106. Springer, 1996.

[KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig et al. [EEKR99], pages 607–638.

[KK08] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 29–38, New York, NY, USA, 2008. ACM.

[KK09] Emilia Katz and Shmuel Katz. Modular verification of strongly invasive aspects: summary. In *FOAL '09: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 7–12, New York, NY, USA, 2009. ACM.

[KKR06] H. Kastenberg, A. G. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. In R. Gorrieri and H. Wehrheim, editors, *Proceedings of the 8th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems, Bologna, Italy*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201, London, June 2006. Springer Verlag.

[Kni09] Günter Kniesel. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Programming V*, 2009. Special issue on Aspect Dependencies and Interactions, edited by Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, Arend Rensink.

[KR06] H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In A. Valmari, editor, *Model Checking Software (SPIN), Vienna, Austria*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305, Berlin, 2006. Springer-Verlag.

[Kus00] Sabine Kuske. More about control conditions for transformation units. In Ehrig et al. [EEKR00], pages 323–337.

[KVK$^+$04] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24(2):129–155, 2004.

[Läm02] Ralf Lämmel. A semantical approach to method-call interception. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55, New York, NY, USA, 2002. ACM.

[LCC$^+$96] M. Loewe, A. Corradini, A. Corradini, U. Montanari, U. Montanari, F. Rossi, F. Rossi, H. Ehrig, H. Ehrig, R. Heckel, R. Heckel, and M. L Owe. Algebraic approaches to graph transformation, part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1996.

[Lee06] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[LJD04] Bert Lagaisse, Wouter Joosen, and Bart De Win. Managing semantic interference with aspect integration contracts. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect*, March 2004.

[LLMC06] László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Model Transformation with a Visual Control Flow Language. *International Journal of Computer Science (IJCS)*, 1(1):45–53, 2006.

[MP05] Thomas Mølhave and Lars H. Peterseny. Assignment Featherweight Java: Bringing mutable state to Featherweight Java. Master's thesis, University of Aarhus, 2005.

[NBA05] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In Andreas Polze Robert Hirschfeld, Ryszard Kowalczyk and Mathias Weske, editors, *Proceedings of International Conference NetObjectDays (NODe)*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[PDS05] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier. CompAr: Ensuring safe around advice composition. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed*

Systems (FMOODS), volume 3535 of *Lecture Notes in Computer Science*, pages 163–178, 2005.

[Phi86] Iain Phillips. Refusal testing. In Laurent Kott, editor, *Automata, Languages and Programming (ICALP)*, volume 226 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 1986.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Plu99] D. Plump. Term graph rewriting. In Ehrig et al. [EEKR99].

[PS04] Detlef Plump and Ra Steinert. Towards Graph Programs for Graph Algorithms. In *In Proc. International Conference on Graph Transformation (ICGT 2004*, pages 128–143. Springer, 2004.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[Ren09] Arend Rensink. Repotting the geraniums: On nested graph transformation rules. In *GT-VMT: Graph Transformation and Visual Modeling Techniques*, 2009.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, Singapore, 1997.

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FOSE)*. ACM, October 2004.

[RSEG08] NASA Ames Research Center Robust Software Engineering Group. Java pathfinder, 2008.

[RvG08] Arend Rensink and Pieter van Gorp, editors. *GraBaTs: Graph-Based Tools: The Contest*, 2008.

[Sch90] Andy Schürr. Introduction to PROGRES, an attribute graph grammar based specification language. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1990.

[Sch97] Andy Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook on Graph Grammars: Foundations*, volume 1, pages 479–546. World Scientific, 1997.

[SEG09] University of Twente Software Engineering Group. Compose*, 2009.

[SF06] Maximilian Storzer and Florian Forster. Detecting precedence-related advice interference. In *ASE*, pages 317–322. IEEE Computer Society, 2006.

[SK03] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[SR08] Tom Staijen and Arend Rensink. A groove solution for the grabats'08 antworld case. In Rensink and van Gorp [RvG08].

[SR09] T. Staijen and A. Rensink. Graph-based specification and simulation of featherweight java with around advice. In *FOAL '09: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages, Charlottesville, Virginia, USA*, pages 25–30, New York, March 2009. ACM.

[SRK06] R. Smelik, A. Rensink, and H. Kastenberg. Specification and construction of control flow semantics. In J. Grundy and J. Howse, editors, *Visual Languages and Human-Centric Computing (VL/HCC), Brighton, U.K.*, pages 65–72, Los Alamitos, September 2006. IEEE Computer Society Press.

[Sta05] T. Staijen. Towards safe advice: Semantic analysis of advice types in compose*. Master's thesis, University of Twente, April 2005.

[SV07] Eugène Syriani and Hans Vangheluwe. Programmed graph rewriting with DEVS. In Manfred Nagl and Andy Schürr, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science. Springer, 2007.

[SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. [EEKR99], pages 487–550.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.

[vdBCC05] Klaas van den Berg, Jose Maria Conejero, and Ruzanna Chitchyan. AOSD ontology 1.0 - public ontology of aspect-orientation. AOSD-Europe-UT-01 D9, AOSD-Europe, May 2005.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 357–362, London, UK, 2001. Springer-Verlag.

[VNS+06] Attila Vizhanyo, Sandeep Neema, Feng Shi, Daniel Balasubramanian, and Gabor Karsai. Improving the usability of a graph transformation language. *ENTCS*, 152:207–222, 2006.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *In ACM International Conference on Functional Programming*, pages 127–139. ACM Press, 2003.

[ZCvdBG07] Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray. Aspect composition in the motorola aspect-oriented modeling weaver. *Journal of Object Technology*, 6(7), August 2007.

[ZS92] Albert Zündorf and Andy Schürr. Nondeterministic control structures for graph rewriting systems. In Gunther Schmidt and Rudolf Berghammer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 570 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 1992.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of

Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data

ence, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical