Graph-Based Software Specification and Verification

Harmen Kastenberg

Dissertation committee:

Prof. dr. ir. J. J. W. van der Vegt

Prof. dr. H. Brinksma (promotor, University of Twente)
Prof. dr. ir. M. Aksit (co-promotor, Universit of Twente)
Dr. ir. A. Rensink (assistent-promotor, University of Twente)

Prof. dr. J.C. van de Pol (University of Twente)
Prof. dr. G. Engels (Universität Paderborn)
Prof. dr. ir. J.-P. Katoen (RWTH Aachen University)
Dr. A. Corradini (University of Pisa)

Harmen Kastenberg

Graph-Based Software Specification and Verification
PhD Thesis, University of Twente, 2008
ISBN 978-90-365-2734-7

IPA Dissertation Series 2008-27
CTIT PhD Thesis Series (ISSN 1381-3617) 08-128

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and within the context of the Centre for Telematics and Information Technology (CTIT).

Printed by MailBalance, Veenendaal, The Netherlands

GRAPH-BASED SOFTWARE SPECIFICATION AND VERIFICATION

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 3 oktober 2008 om 13.15 uur

door

Hermannus Kastenberg

geboren op 7 maart 1981

te Rijssen

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. H. Brinksma
Prof. dr. ir. M. Aksit

Contents

Acknowledgements **xi**

Abstract **xiii**

1 Introduction **1**

1.1 Defining Formal Semantics 2

1.2 System Verification 4

1.2.1 Model Checking in Practice 6

1.2.2 Model Checking and Graph Transformations 7

1.3 Contributions 8

1.4 Outline of the Thesis 9

2 Background in Graph Transformation **11**

2.1 Graphs and Graph Morphisms 11

2.2 Graph Transformations 19

2.2.1 Graph Transformation Approaches 21

2.2.2 Example: Circular Buffer 26
2.3 Tools in the Field 28
2.3.1 The GROOVE Tool Set 28
2.3.2 Graph Transformation Tools 31
2.4 Conclusion 34

3 Uniform Attributed Graphs **35**
3.1 Introduction 35
3.2 Preliminaries 39
3.2.1 Signatures and Algebras 39
3.2.2 Categories of Signatures, Algebras, and Algebra Graphs . 42
3.3 Uniform Signatures 43
3.3.1 Flattening Arbitrary Signatures 46
3.3.2 Unflattening Uniform Signatures 48
3.3.3 Composing Signature Flattening and Unflattening 49
3.4 Uniform Algebras 51
3.4.1 Flattening Arbitrary Algebras 53
3.4.2 Unflattening Uniform Algebras 55
3.4.3 Composing Algebra Flattening and Unflattening 56
3.5 Uniform Attributed Graphs 58
3.5.1 Example: Vectors 59
3.5.2 Equivalent Categories of Attributed Graphs 60
3.5.3 Uniform Attributed Graph Transformations 69
3.6 Abstraction on Uniform Attributed Graphs 72

3.6.1 Abstraction Morphisms 72
3.6.2 Concrete versus Abstract Transformations 75
3.6.3 Non-Determinism and Case Merging 80
3.6.4 Abstraction to Final Algebras 84
3.6.5 Negative Application Conditions 84
3.7 Implementation Issues 85
3.7.1 Pre-defined and User-defined Algebras 85
3.7.2 Fixing Attribute Values 86
3.7.3 Dealing with Infinite Algebra Graphs 86
3.7.4 Visualizing Attributed Graphs 87
3.8 Conclusion 88
3.8.1 Summary 88
3.8.2 Discussion 89

4 Semantics Through Graph Transformations 91

4.1 Introduction 91
4.2 Approach 93
4.3 Abstract and Concrete Syntax 95
4.3.1 Types 96
4.3.2 Statements 98
4.3.3 Expressions 100
4.3.4 TAAL EBNF Grammar 102
4.3.5 Remarks 104
4.4 Two Example Programs 105

4.5 Flat Abstract Syntax Graphs 107
4.5.1 Inheritance 107
4.5.2 Ordered Associations 109
4.5.3 The Flower and Vase Example 110
4.6 Program Graphs 110
4.6.1 Flow Graphs 113
4.6.2 Flow Graph Construction 117
4.6.3 The Flower and Vase Example 122
4.7 Execution Graphs 123
4.7.1 Value Graphs 124
4.7.2 Frame Graphs 126
4.7.3 Program Simulation 130
4.7.4 Statement Execution 131
4.7.5 Expression Evaluation 133
4.7.6 Flow Graph Termination 134
4.7.7 Object Creation 135
4.7.8 Method Invocation 138
4.7.9 Examples 142
4.8 Analysis on TAAL Programs 145
4.9 Extensions 147
4.9.1 .NET Intermediate Language 147
4.9.2 Control Flow Specification Language 150
4.9.3 Remarks 155

4.10 Conclusion 156
4.10.1 Summary 156
4.10.2 Related Work 156
4.10.3 Discussion 158

5 Model Checking Graph Production Systems 161

5.1 Introduction 161
5.2 Model Checking 164
5.3 LTL Model Checking: Ingredients 167
5.3.1 Kripke Structures 168
5.3.2 Linear Temporal Logic 169
5.3.3 From Graph Production Systems to Kripke Structures . . 170
5.3.4 Example: Circular Buffer 172
5.4 On-The-Fly LTL Model Checking 173
5.4.1 Büchi Automata 174
5.4.2 The Automata Theoretic Approach 177
5.4.3 The Schwoon and Esparza NDFS Algorithm 178
5.5 Model Checking Graph Production Systems 179
5.5.1 Boundary Conditions 181
5.5.2 Approximation Sequences 182
5.5.3 From Graph Production Systems to Büchi Automata . . . 184
5.5.4 Example: Circular Extensible Buffer 185
5.6 On-the-Fly Bounded Model Checking Algorithm 186
5.6.1 Correctness and Relative Completeness 189

5.6.2 Complexity Analysis 190
5.7 Implementation and Experiments 196
5.7.1 From LTL Formulae to Büchi Automata 196
5.7.2 State Colouring 197
5.7.3 Boundary Conditions for GPSs 197
5.7.4 Experiments 198
5.7.5 Observations 203
5.8 Conclusion 205
5.8.1 Summary 205
5.8.2 Related Work 206
5.8.3 Discussion 207

6 Dynamic Partial Order Reduction Using Probe Sets 211
6.1 Introduction 211
6.2 Partial Order Reduction 213
6.2.1 Persistent Sets 214
6.2.2 Static versus Dynamic Partial Order Reduction 215
6.3 Stimulation, Disabling, and Reduction 217
6.3.1 Transition systems 220
6.3.2 Entity-based System Specifications 223
6.4 Missed Actions and Probe Sets 225
6.4.1 Missed Actions 226
6.4.2 Probe Sets 228
6.4.3 Correctness 231

6.5 The Probe Set Algorithm 235
6.5.1 Identifying Missed Actions 236
6.5.2 Ensuring Fairness 237
6.5.3 Constructing Probe Sets 238
6.5.4 Putting All Together 239
6.6 Probe Sets for Graph Production Systems 240
6.6.1 Graph Elements as Entities 241
6.6.2 Example: Concurrent Append 244
6.7 Conclusion 249
6.7.1 Summary 249
6.7.2 Related Work 250
6.7.3 Discussion 250

7 Conclusions **253**

7.1 Summary 253
7.2 Discussion and Evaluation 255
7.3 Limitations of the Graph Transformation Framework 257
7.3.1 Graph Matching Complexity 258
7.3.2 Scalability 258
7.3.3 Visualizing Graphs 259
7.4 Future Work 259
7.4.1 Implementing the Probe Set Algorithm 260
7.4.2 Graph-Based Language Engineering 260
7.4.3 Extensions to Modal and Quantified Logics 261

7.4.4 Graph-Based State Space Reduction Techniques 262
7.4.5 Directed Model Checking 264

Bibliography **265**

A Basic Category Theory **285**

A.1 Categories, Functors, and Natural Transformations 285
A.2 Pushouts and Pullbacks 287
A.3 Adhesive Categories 289

B Proofs of Chapter 3 **291**

B.1 Proofs of Section 3.2 291
B.2 Proofs of Section 3.3.1 293
B.3 Proofs of Section 3.3.2 298
B.4 Proofs of Section 3.3.3 301
B.5 Proofs of Section 3.4.1 303
B.6 Proofs of Section 3.4.2 306
B.7 Proofs of Section 3.4.3 308
B.8 Proofs of Section 3.5 310

C TAAL Artifacts **315**

C.1 TAAL EBNF Grammar 315
C.1.1 Non-terminals 315
C.1.2 Terminals 316

D Graph Production System of the Leader Election Protocol 319

D.1 The Leader Election Protocol 319

D.2 A Graph Transformation Implementation 320

D.2.1 Start Graph 320

D.2.2 Rules 322

D.2.3 Priorities 327

D.3 Artificial Error 327

E Proofs of Chapter 6 329

E.1 Proofs of Section 6.3 329

E.2 Proofs of Section 6.4.3 332

E.3 Proofs of Section 6.5.1 337

Samenvatting 339

Acknowledgements

It was a privilige having been a member of both the Software Engineering group and the Formal Methods and Tools group at the University of Twente. Therefore, I would like to thank my promotors Mehmet Aksit and Ed Brinksma, who gave me the opportunity to do research in their inspiring environment. Although you both have not actively been involved in the actual work, your vision as embodied by the other members of both groups has had great influence on my personal and professional development.

Most of the thanks goes to Arend Rensink, my daily supervisor. You were my compass while sailing on the ocean of research. During my master's you introduced me to the very interesting research field of Graph Transformations. Through you, I met a lot of interesting people all over the world. You also learned my how to fly by accompanying me on my first flight ever, to Rome.

Other people that have influenced to content of this dissertation, either through joint work, illuminating discussions, or comments on draft versions are Maarten Fokkinga, Frank Hermann, Anneke Kleppe, Wouter Kuyper, Rom Langerak, and Theo Ruys. Thanks to all of you for your valuable time.

Within the context of the SegraVis Research Training Network, I have been visiting the Technical University of Berlin, on invitation of Gabriele Taentzer for a period of about three months. I would very much like to thank you for letting me taste your way of doing research and giving me the opportunity to follow some of your courses. This visit enriched my knowledge of several computer science topics as well as the german language. In addition, I would like to thank Leen Lambers for experiencing the city of Berlin from her appartment in Kreuzberg.

Next, I would like to thank the people that have used (and thereby tested)

the results of my research activities. Among them are Christian Soltenborn, Christian Hofmann, and Selim Ciraci. Their feature requests often lead to interesting discussions; their bug reports made me experience that even very small bugs can cause a lot of confusion.

During the last four years I have been sharing offices with Iovka Boneva and Patrick Sathyanathan. I would like to thank both of you for the nice, humurous, and inspiring discussions we had. Iovka, a special thanks to you. You gave me some insight in how to deal with pregnant women. Hopefully, I can put this into further practice in the future. I wish you all the best for the three of you.

Furhtermore, I would like to thank Joost 'Monthy Python' Noppen. During our many fitness sessions you taught me, among many other things, the art of translating songs written in foreign languages to the Twents dialect.

I would also like to thank my waterpolo teammates. You all know and experienced my enthousiasm for sports in general, and waterpolo in specific. Thank you all for your understanding with respect to the many times I feld the need to prioritize my career over my sports.

From a more personal point of view I would like to thank my family and friends. Although none of you could really understand the topics I was working on, you have nevertheless giving me great support and motivation to make the last four years a great succes.

Wilma, my love. The last year has been very tough for both of us. My thanks to you cannot be expressed in words. I love you with all my hart.

Abstract

The (in)correct functioning of many software systems heavily influences the way we qualify our daily lives. Software companies as well as academic computer science research groups spend much effort on applying and developing techniques for improving the correctness of software systems. In this dissertation we focus on using and developing graph-based techniques to specify and verify the behaviour of software systems in general, and object-oriented systems more specifically. We elaborate on two ways to improve the correctness (and thereby the quality) of such systems. On the one hand, we investigate the potential of using the graph transformation technique to formally specify the dynamic semantics of (object-oriented) programming languages. On the other hand, we develop techniques to verify systems of which the behaviour is specified as graph production systems. Most of the techniques developed in this work have been implemented in the GROOVE Tool Set.

Typically, a system's state is identified by the values assigned to the system's state variables (often of primitive types such as integer and Boolean). In the object-oriented paradigm, objects (i.e., instances of classes) can be seen as state variables of which the internal state depends on the values of its attributes (or fields). We start with introducing a uniform framework for the specification and transformation of attributed graphs. In this framework, attributed graphs and their transformations are specified in terms of graph structures and graph morphisms only. One of the main advantages of such an approach is that it reduces the effort to implement a graph transformation engine for attributed graphs when compared to existing approaches. Additionally, we show that our uniform framework provides a natural way to deal with abstraction over the supported data domains, without the need to restrict to algebra homomorphisms.

Once a system has been designed it must be implemented in some programming language that best fits to the type of system to be produced. Examples of popular programming languages are Java, C, and C♯. The semantics of such programming languages are typically specified in natural language. Unfortunately, such specifications are often hard to understand. More importantly, they often leave room for multiple interpretations. That is to say, they are ambiguous. In this work we show how the graph transformation framework provides formal and intuitive means to specify operational semantics of programming languages. For that, we introduce an artificial, object-oriented programming language called Taal, and define its control flow and execution semantics in terms of graph transformation rules. We also provide the necessary tool support to actually simulate the behaviour of any Taal-program.

Once a system has been specified as a graph production system, its behaviour must be verified for correctness. Therefore, we introduce an algorithm that combines a well-known on-the-fly model checking algorithm with ideas from bounded model checking. We aim at verifying the temporal behaviour of such systems. This means that the properties to be verified are expressed as formulae in some (linear) temporal logic, e.g., *LTL*; the names of the graph transformation rules form the set of atomic propositions.

We have extended the Groove Tool Set with verification functionality instead of performing the model checking procedure using an existing model checking tool. The main motivation for this is the ability to investigate how to optimally benefit from the potential of the graph transformation framework, especially in the context of combating the state-explosion problem using partial order reduction techniques. Unfortunately, many such existing techniques are based on assumptions that do not hold in our setting, e.g., regarding the number of actions (or operations) that can be performed. Therefore, we have developed a new dynamic partial order reduction algorithm based on selecting so-called probe sets. The algorithm is based on asymmetric stimulation and disabling relations. We have developed the algorithm in the context of entity-based systems in which states are uniquely characterized as sets of entities and actions can read, delete, create entities and forbid the existence of entities. This setting has been chosen because of its nice match with the graph transformation framework. In fact, we describe how graphs can be encoded as set of entities and how rule applications can be translated to corresponding entity-manipulating actions. We show that our algorithm produces a correctly reduced state space in the sense that all possible system executions are preserved. As yet, our algorithm is not accompanied with an implementation and therefore we have no figures on the reduction that can be obtained for different types of systems.

1 Introduction

Nowadays, our daily lives heavily depend on the correct functioning of many different types of software systems. Such systems range from innocent smart applications on devices such as cell phones and tooth brushes, via systems that prevent small-scale accidents from happening such as software systems to control traffic lights or assist the brake process in recent cars, to software systems which can improve the quality of human life or help saving the environment, for instance, software systems embedded in medical-care systems or nuclear power plants. Malfunctioning of systems in the first category can be annoying or create inconvenient situations. If systems of the last category fail to work correctly, this may cause life-threatening situations. We all know about the software failure that made the Ariane-5 missile explode just forty seconds after lift-off. More recently, on March 27th 2008, at London Heathrow Airport a computer software failure, which affected more than 1,000 passengers, led to the cancellation of 12 flights in and 12 out; others were delayed by up to three hours. Up to the time of writing, around 1,000 bags get lost every day [178]. Both examples caused huge financial losses.

Before software systems are taken into production and sold, or embedded in their physical environment, they are usually thoroughly checked to be of sufficient *quality*. Software quality is a very broad concept. Here, with software quality we refer to the level to which the actual system satisfies its original requirements, and focus on requirements that specify how a system should *behave*. More specifically, we determine whether all possible *temporal orderings* of actions performed by the system are conform its specification. Stated differently, we *verify* whether all possible system executions are correct with respect to its

specification.

Traditionally, the behaviour of systems is specified in textual languages (e.g., programming languages or process algebras). The semantics of popular programming languages such as C or Java, are often described in natural language such as English or Dutch, thereby introducing ambiguity, whereas the proper use of process algebras requires a solid understanding of mathematics and logic.

This is the point where *graph transformations* come into play. Graph transformations have been introduced in the early 1970s [72] to generalize Chomsky's string grammars. They provide an intuitive and formal way of specifying local graph changes, i.e., creation or deletion of graph elements, in a rule-based manner. In this dissertation we investigate how graph transformations can help to alleviate the problems mentioned above. In particular, we focus on how we can benefit from this technique when formally defining the semantics of object-oriented (programming) languages. In addition, we introduce techniques to verify the correctness of systems of which the behaviour is entirely specified in terms of graph transformations.

1.1 Defining Formal Semantics

Research on the formal semantics of programming languages started in the 1960s, resulting in different methods and formalism. Most of these methods are based on one of the following two main approaches:

denotational semantics: meanings are denoted as mathematical objects. As originally developed by Scott and Strachey [172], denotational semantics are basically defined through semantic functions mapping inputs to outputs, without prescribing or revealing the way this mapping should or could be achieved in an implementation of the semantics.

operational semantics: the meaning of a program is represented as a sequence of computation steps that, for example, result from the program's execution. These sequences then actually *are* the meaning of the program. The level of granularity of those computation steps usually reflects the intended level of abstraction.

A good introduction to both approaches and related theory can be found in [198]. In the sequel, we focus on the second category of methods.

A traditional approach to defining language semantics operationally is called *structural operational semantics* (SOS, for short), originally proposed by Plotkin [151], and usually applied to give semantics to process algebras such as, e.g.,

Milner's CCS [141]. The idea is to express the evaluation of expressions and the execution of actions or statements by rules (often called *inference rules*) in a *syntax-directed* way. That is to say, for every (syntactic) way of composing programs from smaller (sub-)programs, an individual rule expresses how that type of composition influences the composed program's semantics. For researchers in the field, SOS is a fairly straightforward formalism with a concise syntax; for the average practitioner (e.g., software engineer or business analyst), who mainly thinks in terms of visual diagrams, SOS rules are much less compelling, due to their (mainly) textual notation and the underlying mathematics.

We believe that graph transformations provide a natural framework for formally defining the semantics of programming languages in the operational way. In particular, we aim at the operational semantics of *object-oriented* languages. The visual aspect of graph transformations is a first remedy for the gap between the (often) textual world of scientists and the diagrammatic world of practitioners. For example, when specifying the semantics of object-oriented programming languages like, e.g., Java, the process of *dynamic method lookup* can very naturally be defined through a number of graph transformation rules. In addition, each individual rule explicitly identifies the different phases of such processes and thus we obtain further insight in what is actually happening.

Another advantage of the graph transformation framework is the fact that, during the last decade, various tools have been developed, providing visual support for specifying and performing graph transformations. Some example graph transformation tools are Agg [182], Fujaba [142], GReAT [2], Progres [170], Augur [120], and the Groove Tool Set [157]. More recently, some of those tools are actively applied in practice for modelling and analyzing moderate-sized and complex systems. The basic functionality of graph transformation tools consists of visualizing graphs and graph transformation rules, and showing the effect of individual applications of such transformation rules. If, next to generating new graphs, all generated graphs and the actual rule applications are explicitly stored, a *graph transition system* can be constructed. The paths of such graph transition systems represent all possible *graph derivations* of the graph production system.

To close the cycle (from programming languages to using graph transformation for defining their semantics), assume we have a graph G representing the initial state of some program written in some language L, and $\mathcal{R}$ is a set of transformation rules defining the (execution) semantics of L. The paths in the generated graph transition system then represent the possible ways to sequence computation steps of the program, i.e., that graph transition system represents the program's semantics. Especially for languages in the object-oriented

paradigm, where systems are highly dynamic due to frequent (de)allocation of reference values, graph transformations provide a natural way of dealing with such dynamics.

1.2 System Verification

Determining whether a system satisfies a given set of requirements can be done in many different ways. Here, we distinguish between two conceptually different approaches, namely *testing* and *formal verification*.

Testing activities are performed on an actual *implementation* of the system [154]. The basic idea is to provide the implementation with some *input*, then observe its *output* (or *response*) and determine whether the output agrees with the desired or expected output. A single set of input values is often called a *testcase*. The main disadvantage of testing is that, typically, the number of testcases is very large and often even infinite. More importantly, subtle errors that only occur in highly exceptional cases will often not be identified with traditional testing techniques. Furthermore, testing activities must often be performed in short time periods and under high time-pressure (since project deadlines are then approaching quickly). Summarizing, with testing only, correctness of software systems can often not fully be guaranteed.

Formal verification techniques, on the contrary, provide means to assess whether a system is correct with respect to a set of requirements (then often called *properties*) based on a *model* of the system. Taking a model of the system as a starting point instead of the system itself, creates opportunities for applying additional techniques to combat the problem of determining correctness as will become clear later on. Different verification techniques have been developed, of which the most important ones are *theorem proving* (see, e.g., [58]) and *model checking* (see, e.g., [37, 4]). The basic idea of theorem proving is to determine whether a given statement (the *conjecture*) is a logical consequence of a set of statements (the *axioms* and *hypotheses*). The main disadvantage of the theorem proving technique is that the user must be proficient in logic theory and proof techniques, since theorem proving activities are often highly human-machine interactive.

Model checking [37, 4] is a fully automatic verification technique using *brute-force*. The central idea is to verify *all* possible executions of a *model* of the system and check whether they satisfy the required properties. The main advantage of the model checking approach is that for faulty models, a *counter-example* is provided (i.e., an execution representing the faulty behaviour). Counter-

examples can then serve as input for debugging the original system design or implementation. The down-side of model checking is that it suffers from the *state-explosion problem*: the size of a system's state space grows exponentially in

1. the number of *system components*;
2. the number of *state variables*.

Basically, every system component has its own local behaviour in terms of actions it can perform. Some actions might involve synchronization among components; some actions of one component might be causally dependent on actions of some other component. There are typically also many actions each component can perform independently of any other component (although different orderings of such actions may result in different behaviours of the whole system). The last class of actions may be performed in any possible order, thereby causing an explosion in the number of system executions.

In case the state-explosion problem with respect to the first cause is still manageable, the number of variables (of which the actual values uniquely identify the current system state) causes a further explosion on the number of system states. Typically, the domains of state variables can be huge, if not infinite. It is easy to see that state variables over infinite domains potentially give rise to infinite state spaces. Note that, in many cases, not all states are actually reachable from the system's initial state.

To tackle the state-explosion problem, the following techniques have been studied:

1. partial order reduction;
2. abstraction;
3. symmetry reduction.

Partial order reduction techniques aim at alleviating the state-explosion problem by tackling the first cause. In the literature, a number of partial order reduction techniques have been proposed; see, e.g., [185, 148, 91, 37, 85, 95]. The basic idea is that many actions of different system components (or *processes* in the context of process algebras) do not influence each other. For such actions, all different orderings have an equivalent overall effect. In many cases, it can be proved that picking one such ordering is sufficient. Typically, this significantly reduces the size of the state space that is effectively explored.

A popular approach to alleviate the pains of the second cause is to apply *abstraction* techniques (see, e.g., [36]). Instead of distinguishing between all

concrete values, sets of concrete values are mapped to single abstract values. In specific cases, infinite domains can be abstracted to finite ones.

A final technique that is often applied to further reduce the size of the state space and of interest in this work, is called *symmetry reduction* (see, e.g., [38, 106, 76]). Symmetry reduction techniques usually define a *simulation relation* between states based on specific requirements such as, among others, the local properties they satisfy. One special type of simulation relation is a *bisimulation* [141]. It is well-known that bisimilar states satisfy the same CTL^* formulae[1]. If there exists a non-trivial[2] bisimulation from a state space to itself, a reduced state space can be constructed by merging bisimilar states.

1.2.1 Model Checking in Practice

Originally, model checking techniques were applied in early phases of the software development process to verify the *design* of the system, or, stated differently, to verify a *pre-implementation* model of the system. The idea is that when errors are identified in early (e.g., design) stages, the cost of repairing the error are typically relatively low compared to when those errors would have been identified in later (e.g., testing) phases. Nevertheless, having a correct design does not guarantee the correctness of an actual implementation of that design. During the last decade, model checking techniques have been developed that work on *post-implementation* models, i.e., models that are derived from an actual implementation of the system under consideration. This approach is therefore called *software model checking*. Many software model checking techniques focus on system implementations in programming languages such as, e.g., C or Java (source or byte code). A few example software model checkers are Slam [11] and Blast [19] for C programs, and Java Pathfinder [194] for Java programs.

The model checker Spin [104] deserves some special attention. Over the past fifteen years, the Spin model checker and its related annual Spin workshops have been of primary guidance in the research field of model checking. Spin has been applied in various contexts, among which, the development of communication protocols. Nowadays, Spin is also often used as a *back-end* model checker: verification problems are then translated into specifications that can then be model checked by Spin. However, Spin does not provide natural means to

[1] CTL^* is a temporal logic often used in the context of model checking. Although CTL^* itself will not be discussed in this work, two sub-logics of CTL^* will, namely CTL [32] and LTL [133].

[2] The *trivial* bisimulation relation contains pairs of identical states only.

model, let alone verify, object-oriented programs, on which we focus. This is due to the underlying representation used by Spin to explicitly represent individual states, namely *bit-vectors*. The required length of such vectors is often determined statically and cannot change while the verification procedure is running. In the object-oriented paradigm, in contrast, system states are highly dynamic due to frequent (de)allocation of reference values. There are extensions of Spin that support some form of dynamics, for example dSpin [51]; they provide some practical work-around but do not change the fundamental limitations of the bit-vector formalism.

Graphs provide a natural way of representing the states of object-oriented systems. The main advantages of using graphs to model system states instead of bit vectors are twofold. Firstly, there is no need to *a priori* specify a bound on the size of the system states, since the size of the graph structure can freely vary over time. Secondly, the graph formalism provides a natural form of *symmetry reduction*, namely by merging states of which the internal graph structures are *isomorphic*, i.e., identical up to identities of individual graph elements.

1.2.2 Model Checking and Graph Transformations

When the semantics of some programming language has been defined in terms of graph transformations, and the state space of some program in that language can effectively be generated, the question arises whether we then can actually verify the behaviour of any such programs. In general, this question can be posed for arbitrary systems of which the behaviour is specified as graph transformations. Theoretically, this question can be answered positively to a certain degree, depending on what kind of properties we want to verify. In practice, however, some hurdles have to be taken, as explained in the following.

Needless to say, the state-explosion problem also occurs in the graph transformation framework. Unfortunately, many of the techniques proposed in the literature to alleviate those problems do not directly apply in the context of graph transformations. For example, traditional partial order reduction techniques assume that the system consists of multiple concurrent processes, whereas in the graph transformation framework there does not exist such a central notion of a process. Fortunately, as mentioned above the graph transformation framework provides natural ways to perform symmetry reduction by merging states with isomorphic internal graph structures. Although the problem of deciding whether two graph are isomorphic is, in general, computationally expensive, isomorphism checking potentially leads to significant state space reduction.

1.3 Contributions

From the above we can conclude that the graph transformation technique provides intuitive means to formally define (execution) semantics for object-oriented programming languages in an operational and syntax-directed way. In order for the graph transformation framework to be useful for more practical cases, the support for integrating (primitive) data values in the graphs is a prerequisite. Actually performing model checking activities on graph transformation systems requires either to *translate* the output of a graph transformation tool to an existing model checker or to *extend* existing graph transformation tools with model checking techniques; of these two possibilities, we have chosen the latter. One of the reasons for this is to investigate to what extent existing state space reduction techniques can be applied to the graph transformation framework.

The main contributions of this dissertation are the following:

1. *A uniform framework for specifying and transforming attributed graphs.*

Whereas in many other approaches to modelling and transforming attributed graphs the regular graph part and the data part are separated, in our approach these are combined in a uniform framework. This reduces the implementation effort when extending graph transformation tools to support graph attribution. In addition, our uniform framework creates new opportunities to perform data abstraction on attributed graphs, e.g., when used for modelling data-intensive (object-oriented) systems.

2. *A graph transformation based definition of the operational semantics of an object-oriented programming language.*

We show how graph transformation can be applied to define the dynamic semantics (i.e., control flow and execution semantics) of an (artificial) object-oriented programming language called Taal. The main result is that the execution of any Taal-program can actually be simulated with the Groove Tool Set.

3. *A model checking algorithm for graph transformation systems.*

In [113] we have provided a proof-of-concept that model checking techniques can effectively be applied to graph transformation systems that generate finite state spaces. To extend this to arbitrary graph transformation systems, we have implemented an *LTL* model checking algorithm that combines existing techniques and ideas from on-the-fly model checking and bounded model checking.

4. *A dynamic partial order reduction algorithm.*

Since traditional partial order reduction techniques do not directly apply to the graph transformation framework, we have developed a new algorithm using so-called *probe sets*. The new algorithm performs partial order reduction dynamically, i.e., it first underestimates the subset of enabled transitions to be explored from any state and repairs this choice if later on some behaviour of the system turns out to be omitted.

1.4 Outline of the Thesis

This thesis is structured as follows.

Chapter 2 introduces some definitions and concepts that will be used in the remainder of this thesis. It discusses the graph transformation technique by explaining different algebraic methods which are mostly applied in the field. This chapter ends with an overview of some tools that have been developed for performing graph transformations. It also includes an overview of the main features of the Groove Tool Set as it was before being extended with techniques resulting from work described in this thesis.

Chapter 3 discusses our uniform approach to graph attribution. Graphs are a very powerful formalism for modelling system structure and behaviour. In order to make the formalism more suitable for the object-oriented paradigm, we have extended the simple graph formalism with data types and provided an uniform algebraic framework allowing the specification of attributed graph and their transformation. This chapter also indicates the benefit of our approach in the context of abstraction. The core message is that since our attributed graphs are specified entirely in terms of graph structures, abstractions can be defined as (non-injective) graph morphisms.

This chapter extends the ideas originally presented in [110].

Chapter 4 defines an artificial object-oriented programming language called Taal. Although its concrete and abstract syntax are defined in terms of an EBNF-grammar and UML meta models, respectively, its control flow semantics and execution semantics are defined in terms of two separate graph transformation systems. After translating a textual Taal-program to a graph representation of its abstract syntax, applying both graph transformation systems results

in a simulation of the execution of the original program. We use two example Taal-programs to explain our approach.

This chapter is based on [111, 112] and the results of the master projects by Sombekke [176] and Smelik [175, 174].

Chapter 5 elaborates on how existing model checking techniques can effectively be applied to graph transformation systems. Since, in general, termination of graph transformation systems is undecidable, we have developed an on-the-fly bounded model checking algorithm. This algorithm verifies the system in an iterative fashion. By using proper boundary conditions, each iteration is guaranteed to generate a finite state space, which is then verified using well-known on-the-fly algorithms. Finally, this chapter reports on some experimental results.

This chapter is based on [115] and continues our work presented in [113].

Chapter 6 proposes a new dynamic partial order reduction algorithm. Traditional partial order reduction techniques, e.g., using persistent sets, are based on assumptions that do not hold for graph transformation systems. The new algorithm is based on an abstract framework of asymmetric enabling and disabling relations. This chapter explains the algorithm by instantiating the abstract framework in the context of entity-based systems. Whereas in the classical setting transitions can only perform read and/or write operations on shared or local variables, the actions of entity-based systems can also create new entities and delete superfluous entities. The algorithm produces a so-called *trace automaton* [91]. At the end, this chapter indicates how graph transformation systems can be encoded as entity-based systems, thereby proving its use in the graph transformation framework.

This chapter is based on [114].

Chapter 7 finishes this thesis by shortly summarizing the main results of our work. It discusses some of the limitations of our approach and looks ahead at some research topics that are to be investigated in the (near) future.

2 Background in Graph Transformation

In this chapter, we will introduce the basic concepts underlying the graph transformation technique. We start with introducing the central concepts of *graphs* and *morphisms*. Thereafter, we informally describe some basic categorical constructions on which the algebraic graph transformation approaches are based. Next, a short overview is presented of the actual algebraic graph transformation approaches. This chapter ends with a description of the starting position with respect to the GROOVE Tool Set. After that, we give an overview of the graph transformation tools closest related to the topic described in this work, listing their main features and relation to the GROOVE Tool Set.

Part of this chapter is based on some basic category theory. In Appendix A we have listed some of the basic category theoretical concepts.

2.1 Graphs and Graph Morphisms

When designing system architectures or specifying system behaviour, *graphs* provide a natural and intuitive way of showing what an architecture will look like, or how a system is supposed to evolve over time. Graphs are used in many contexts and there are almost as many different types of graphs as research fields in which they are applied. Therefore, it is necessary first to define what we mean with a graph.

The set of all graphs will be denoted $\mathcal{G}$, ranged over by G, H and defined over a (possibly infinite) set of labels Lab .

Definition 2.1 (graph, morphism)**.** *A* graph $G = (N, E)$ *consists of a (finite) set* N *of* nodes *and a (finite) set* E *of* edges*, together with functions* $\mathsf{src}\colon E \to N$*,* $\mathsf{lab}\colon E \to Lab$*, and* $\mathsf{tgt}\colon E \to N$*, which are called the* source*,* label*, and* target *function, respectively.*

Given two graphs G *and* H*, a* graph morphism $f = (f_N, f_E)$ *consists of two partial functions* $f_N\colon N_G \rightharpoonup N_H$ *and* $f_E\colon E_G \rightharpoonup E_H$ *such that* f *commutes with the source, label, and target functions, i.e.,*

$$\begin{aligned} f_N \circ \mathsf{src}_G &= \mathsf{src}_H \circ f_E \\ f_N \circ \mathsf{tgt}_G &= \mathsf{tgt}_H \circ f_E \\ \mathsf{lab}_G &= \mathsf{lab}_H \circ f_E \end{aligned}$$

We often represent edges as triples of which the components denote their source, label, and target. For example, representing an edge e as the triple (n_1, l, n_2) means that $\mathsf{src}(e) = n_1$, $\mathsf{lab}(e) = l$, and $\mathsf{tgt}(e) = n_2$.

Definition 2.2 (subgraph)**.** *Given two graphs* G *and* H*,* H *is a* subgraph *of* G*, denoted* $H \subseteq G$*, if* $N_H \subseteq N_G$ *and* $E_H \subseteq E_G$*.* H *is a* full *subgraph of* G *if the following condition is satisfied:*

$$\forall n_1, n_2 \in N_H\ \forall l \in Lab : (n_1, l, n_2) \in E_G \Rightarrow (n_1, l, n_2) \in E_H \ .$$

We call a graph G *discrete* if $E_G = \emptyset$, i.e., the graph G consists of nodes only. Specific properties of functions, such as *totality*, *injectivity*, and *surjectivity* can likewise be defined for graph morphisms through their constituent parts. That is, a graph morphism f is said to be *total* if both functions f_N and f_E are total. Similar definitions apply for injectivity, and surjectivity of graph morphisms. In this dissertation we extensively use *partial* graph morphisms.

Definition 2.3 (partial graph morphism)**.** *A graph morphism* $f\colon G \to H$ *is said to be* partial *if* f *is a total graph morphism for some subgraph* G' *of* G*.*

For a graph morphism $m\colon G \to H$, we usually define its *domain* and its *codomain* (or *range* in Set Theory), denote $dom(m)$ and $cod(m)$, respectively, such that:

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

For a graph morphism $m\colon G_1 \to G_2$, the *context of* m *in* G_2 are those edges

in G_2 that are not in the codomain of m, but of which the source or target node is in the codomain of m.

The above definition allows a graph to have *parallel edges*, i.e., multiple edges between the same nodes having equal labels. In this thesis we also explicitly work with graphs in which the set E of edges is a subset of $N \times Lab \times N$, which makes it impossible to have parallel edges. Such graphs are often referred to as *simple graphs*; in this context, the objects defined in Def. 2.1 are called *multigraphs*. In the sequel we will refer to different graph categories. We explicitly distinguish between the following ones:

- the category **Graph**, having multigraphs as objects and total graph morphisms as arrows;
- the category $\mathbf{Graph_P}$, having multigraphs as objects and partial graph morphisms as arrows;
- the category $\mathbf{SGraph_T}$, having simple graphs as objects and total graph morphisms as arrows;
- the category **SGraph**, having simple graphs as objects and partial graph morphisms as arrows.

There is a special class of graphs in which all nodes have at most one outgoing edge for any label. Such graphs are called *deterministic graphs*.

Definition 2.4. *A graph $G = (N, E)$ is said to be* deterministic *if for all nodes the number of outgoing edges of any label is at most one, i.e.,*

$$\forall n \in N, \forall l \in Lab : |\{e \in E \mid \mathsf{src}(e) = n \wedge \mathsf{lab}(e) = l\}| \leq 1$$

Restricting to deterministic graphs can have major impact on the analysis to be performed on them. For instance, Dodds and Plump have shown that on graphs satisfying this and similar conditions, graph transformation can be performed very efficiently [55]. Note that deterministic graphs are certainly simple.

Graphs can very naturally be depicted by representing nodes as bullets (or boxes) and edges as arrows. In Fig. 2.1, an example graph is depicted consisting of three nodes and three edges. Fig. 2.2 depicts two graphs (in the dashed rounded rectangles) and a partial graph morphism between them, specified by the dashed gray arrows. Edges are implicitly mapped through the mappings of their source and target nodes.

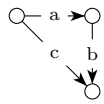


Figure 2.1: Example graph.

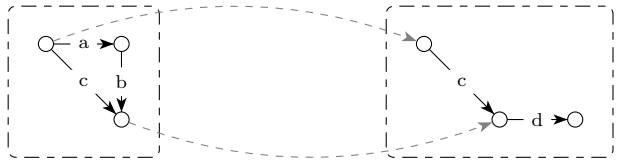


Figure 2.2: Example of a partial graph morphism.

Pushouts and Pullbacks, intuitively

The algebraic approaches to graph transformations heavily rely on special categorical limit and co-limit constructions, namely *pullbacks* and *pushouts*. In Appendix A, we have given a formal definition of pullbacks and pushout. Here, we give a very informal description of how pushouts and pullbacks are constructed in the category **SGraph**.

In the category **SGraph**, the pushout of a graph G and two partial morphisms $f: G \to G_1$ and $g: G \to G_2$ is constructed as the *component-wise union* of the two graphs G_1 and G_2 gluing together the common parts in G. Moreover, the elements of G that are not mapped by f are removed from G_2 and the elements in G_1 not having a pre-image in G are added to G_2. Stated differently, for elements in G the intersection is taken of G_1 and G_2, and for elements outside G the union is taken of G_1 and G_2. In this construction, the roles of G_1 and G_2 can be interchanged, when also f is replaced by g. For a formal description of the pushout construction in the category **SGraph**, the interested reader is referred to [162].

Example 2.5. *Let G, G_1, and G_2 be the graphs as shown in Fig. 2.3 and $f: G \to G_1$ and $g: G \to G_2$ be two partial graph morphisms (specified through the placing of the elements). The pushout of (G, f, g) is specified by the graph G_{12} and two partial morphisms $f': G_2 \to G_{12}$ and $g': G_1 \to G_{12}$.*

In a pushout diagram as shown in Fig. 2.3, the object G_2 is called the *pushout complement* of the span $G \xrightarrow{f} G_1 \xrightarrow{g'} G_{12}$ (dually, G_1 is the pushout complement of the span $G \xrightarrow{g} G_2 \xrightarrow{f'} G_{12}$).

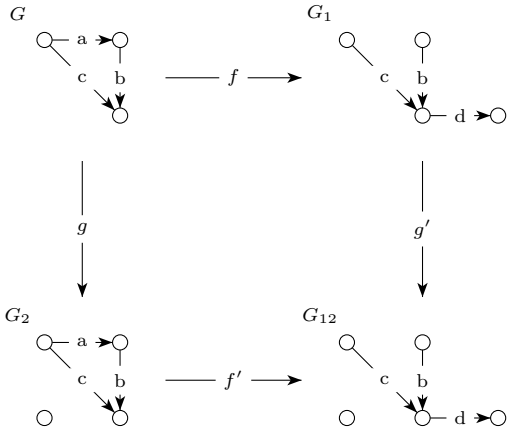


Figure 2.3: Example pushout in the category $\mathbf{SGraph_T}$.

Here, we address some interesting examples of pushouts and pushout complements in the different graph categories. In Example 2.6, Example 2.7, and Example 2.9 we consider the category **SGraph**; Example 2.8 involves the categories **Graph** and **SGraph**.

Example 2.6. *Suppose we have the graphs G, G_1, and G_2 as depicted in Fig. 2.4, with partial graph morphisms $f\colon G \to G_1$ and $g\colon G \to G_2$. The partial morphism f specifies the creation of a* c*-labelled edge. However, the graph G_2 already contains a* c*-labelled edge between the corresponding nodes. Since in the category $\mathbf{SGraph_T}$, there can at most be one edge between two nodes with a certain label, in the pushout graph G_{12} the created* c*-labelled edge will be identified with the already existing edge.*

Example 2.7. *In this example we construct the pushout of the morphisms $f\colon G \to G_1$ and $g\colon G \to G_2$ as depicted in Fig. 2.5. Here, the morphism specifies the deletion of the* b*-labelled edge and its target node. In the graph G_2, however, there exists an* c*-labelled edge pointing to this node. Although the deletion of this edge is not explicitly specified, it cannot be part of G_{12}, since that would cause G_{12} not being well-defined, or, stated differently, G_{12} would contain a so-called* dangling edge. *Therefore, this edge for which the target node is deleted by f will also be deleted.*

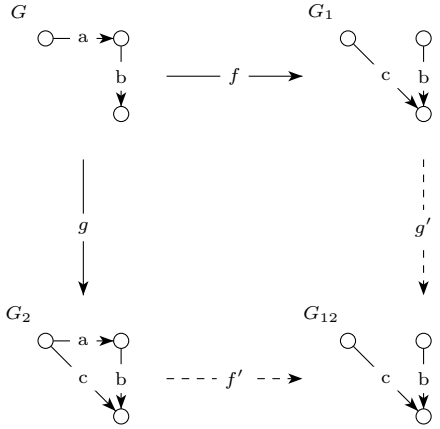


Figure 2.4: Diagram for Example 2.6.

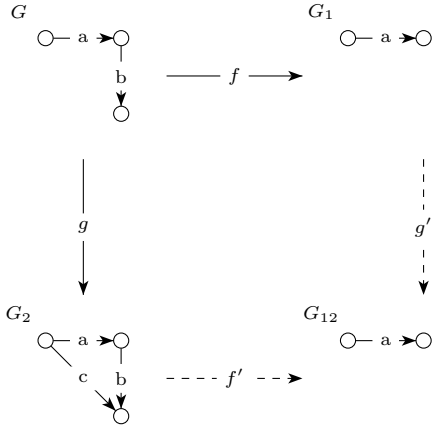


Figure 2.5: Diagram for Example 2.7.

Example 2.8. *In this example we start with the morphisms $f: G \to G_1$ and $g': G_1 \to G_{12}$ and construct the pushout complement. Fig. 2.6 depicts the unique pushout complement (up to isomorphism) in the category* **Graph**. *In the category* **SGraph**, *the situation occurs that the pushout complement is not unique. Fig. 2.7 depicts the two candidate pushout complements. Note the relation with Example 2.6, where the creation of an edge did not actually result in a fresh edge.*

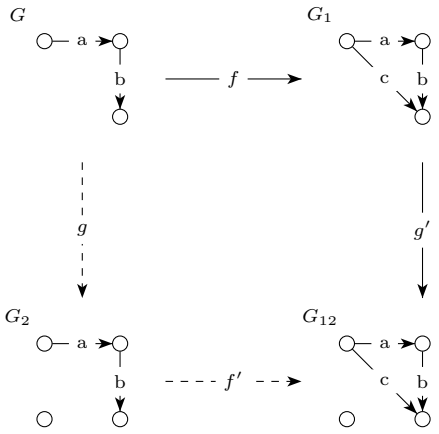


Figure 2.6: Diagram for Example 2.8 in the category **Graph**.

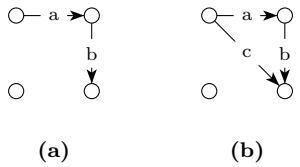


Figure 2.7: Candidates for the pushout complement of the morphism f and g' from Fig. 2.6 in the category $\mathbf{SGraph_T}$.

Example 2.9. *In this example we discuss a pushout construction involving a non-injective, partial graph morphism, i.e., a morphism that maps distinct nodes to the same image node. In Fig. 2.8, we have depicted such an example.*

The morphism f specifies the deletion of the a*-labelled edge and its target node. However, the morphism g maps the target node of this* a*-labelled edge and the isolated node to the same node in G_2, as indicated by the dashed gray lines. For the isolated node we have a preserve-delete conflict: on the one hand it should be preserved, whereas on the other hand it must be deleted. In such situations, deletion always wins.*

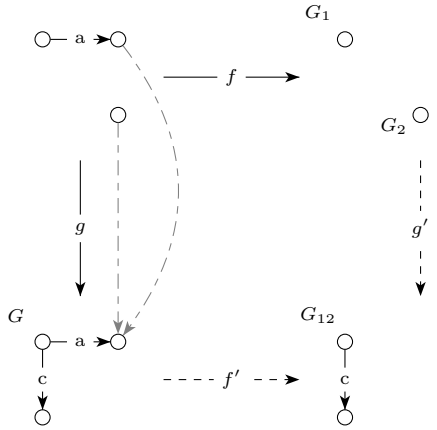


Figure 2.8: Diagram for Example 2.9.

Based on the above examples we introduce the notion of a *side-effect* of a pushout construction. A pushout construction has a side-effect if in the pushout object, graph elements are deleted for which this is not specified explicitly. For example, in Example 2.7, the c-labelled edge is removed since it would otherwise be a dangling edge; in Example 2.9, the isolated node is deleted due to the fact that there was a preserve-delete conflict in which cases deletion always wins.

In the category $\mathbf{SGraph_T}$, the pullback of two partial graph morphisms $f\colon G_1 \to G$ and $g\colon G_2 \to G$ is constructed as the *component-wise intersection* of the two graphs G_1 and G_2 identifying their common parts in G.

Example 2.10. *Let G_1, G_2, and G be the graphs as depicted in Fig. 2.9 and $f\colon G_1 \to G$ and $g\colon G_2 \to G$ be two partial graph morphisms (again specified through the placing of the individual elements). The pullback of (G, f, g) is specified by the graph G_{12} and the two morphisms $f'\colon G_{12} \to G_2$ and $g'\colon G_{12} \to G_1$.*

It is well-known that in the category **Graph** pushouts and pullbacks always

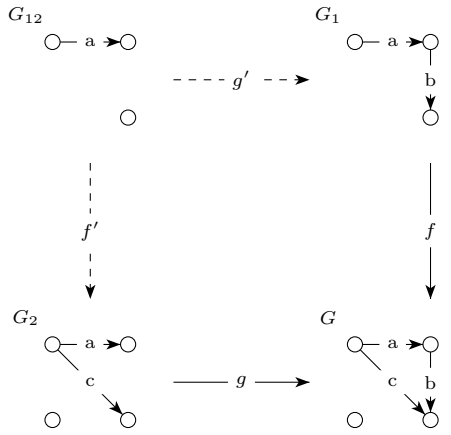


Figure 2.9: Example pullback in the category $\mathbf{SGraph_T}$.

exist. The same holds for the category **SGraph**. In addition, those pushouts and pullbacks are unique up to isomorphism.

2.2 Graph Transformations

The central idea of graph transformations is to generalize Chomsky's string grammars to arbitrary graph structures. During the last four decades, the theory of graph transformations has become quite mature [167, 67]. Recently, more energy is devoted to the practical use of this technique and tools have been implemented visualizing all aspects of graph grammars. A *graph grammar* (also called a *graph production system* or *graph transformation system*) consists of a set $\mathcal{R}$ of *graph rewrite rules* (also called *graph productions* or *graph transformation rules*) and a start graph G. With graph transformation rules one can specify how to generate graphs from other graphs. This has created opportunities for applying graph transformation in modern fields of research such as specifying *model transformations* (see e.g., [98], [181]) in the context of Model Driven Architecture (MDA) [143].

In the following we formally introduce further concepts in the graph transformation framework. We start with defining the basic building blocks, after which we briefly give an overview of the graph transformation approach used in this dissertation, namely the *algebraic approach* to graph transformation.

Definition 2.11 (graph production system)**.** *A* graph production system $P = \langle \mathcal{R}, G_0 \rangle$ *consists of a set* $\mathcal{R}$ *of graph transformation rules and a* start graph *(or* initial graph*)* G_0*.*

In practice, graph transformation rules are often named; we will denote the name of a graph transformation rule p with N_p. For a graph production system $P = \langle \mathcal{R}, G_0 \rangle$, we are interested in the set $\mathcal{G}_P$ of graphs that can be derived from G_0 by repeatedly applying the rules in $\mathcal{R}$. For a rule $p \in \mathcal{R}$ to be applicable to some graph G we need to identify a *matching* of p to G. For now it suffices to know that having a matching makes the rule applicable, without requiring to formally define the notion of a matching. Applying a rule p to a graph G via a match m gives rise to a graph H, often called the target graph; this is denoted $G \overset{p,m}{\Longrightarrow} H$. We now introduce the following notation: a graph G can be transformed to a graph H in zero or more steps, denoted $G \Rightarrow^* H$, if and only if there exist pairs (p_i, m_i) and graphs G_i, for $1 \leq i < n$, consisting of a rule $p_i \in \mathcal{R}$ and a *matching* m_i such that $G_{i-1} \overset{p_i,m_i}{\Longrightarrow} G_i$, $G_0 = G$, and $G_n = H$, as shown below.

$$G = G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots G_{n-1} \overset{p_n,m_n}{\Longrightarrow} G_n = H \ .$$

The set $\mathcal{G}_P$ can then be specified as follows:

$$\mathcal{G}_P \quad = \quad \{G \in \mathcal{G} \mid G_0 \Rightarrow^* G\} \ .$$

We can now introduce the notion of a *graph transition system*, in which states represent graphs and transitions represent rule applications.

Definition 2.12 (graph transition system)**.** *Given a set* $\mathcal{R}$ *of graph transformation rules, a* graph transition system *is a triple* $T = (S, \rightarrow, G_0)$ *where*

- $S \subseteq \mathcal{G}$ *is a set of* states*;*
- $\rightarrow \subseteq \mathcal{G} \times \mathcal{R} \times \mathcal{M} \times \mathcal{G}$ *is a set of* state *or* graph transitions*;*
- G_0 *is the* initial state*.*

We say that a graph production $(\mathcal{R}, G_0)$ *generates* a graph transition T if the transitions in T correspond to applications of rules in $\mathcal{R}$ to all reachable states. Since graphs are generated up to isomorphism, for every rule application $G \overset{p,m}{\Longrightarrow} H$ we require the existence of a graph transition in T with some target state H' isomorphic to H.

Definition 2.13. *Let* $P = (\mathcal{R}, G_0)$ *be a graph production system. Then,* P *is said to* generate *the graph transition* $T_P = (S, \rightarrow, G_0)$ *such that:*

- $G_0 \in S$;
- $G \in S$ *and* $G \overset{p,m}{\Longrightarrow} H$ *implies* $\exists (G, p, m, H') \in \rightarrowtail$ *with* H' *isomorphic to* H.

In traditional labelled transition systems, transitions are identified by means of their source state, target state and label. For graph production systems, rule applications are identified by means of their source graph, rule, and matching, together uniquely (up to isomorphism) determining the target graph. If the generated graph transition system would only label the transitions with rule names, different applications of the same rule leading to isomorphic states would be merged to one single transition. By including the matching in the transition, such rule applications have distinct counterparts in the generated graph transition system. In this work we will not make use of this fact. An example case in which this information is useful is when we are interested in the history of individual graph elements in a sequence of graph transitions.

2.2.1 Graph Transformation Approaches

Next to the specific graph formalism in which program structures are modelled, the specific graph transformation approach being applied determines what kind of results can be achieved. Two main aspects are (i) the conditions under which *direct derivations* are allowed and (ii) the fact whether they can be performed *unidirectional* or *bidirectional*. These two aspect are the main driving forces for either applying the Double Pushout approach [72, 45] or the Single Pushout approach [134, 69]. A more recently developed approach is the Sesqui Pushout approach[42], which combines the interesting features of the DPO and SPO. In the following paragraphs we will discuss each of them and discuss for what graph category they are defined.

Other graph transformation approaches that are no longer in the focus of interest (and will therefore not be discussed here) are the fibre approach by Kahl [109], the double-pullback approach by Heckel [99], and the pullback approach by Bauderon [15].

2.2.1.1 Single Pushout Approach

A liberal approach to graph transformation is the Single Pushout (SPO) approach [134, 69]. As the name indicates, the transformations are described by a single pushout construction. The SPO approach is based on graph categories

in which the arrows are partial graph morphisms, e.g., the category $\mathbf{Graph_P}$ or **SGraph**.

In the SPO approach, transformation rules are represented by two graphs L and R with a partial graph morphism $p\colon L \to R$.

Definition 2.14 (SPO transformation rule)**.** *An* SPO graph transformation rule $p\colon L \to R$ *consists of a left-hand-side graph L, a right-hand-side graph R, and a graph morphism p mapping elements from L to elements of R.*

Applicability of an SPO transformation rule is based on the existence of a *total* graph morphism $m\colon L \to G$, also called a *matching*. This means that we can only apply a rule, if all the elements of L are present in the graph G, often called the *host graph*. The application of a transformation rule is depicted in Fig. 2.10. The resulting graph H is constructed by removing those elements from G that are not in the domain of p and creating the elements in R that do not have a *pre-image* in L.

Definition 2.15 (direct SPO transformation)**.** *A* direct SPO transformation $G \overset{p,m}{\Longrightarrow} H$ *via a rule p and a matching m is given by the diagram in Fig. 2.10, where (1) is a pushout in the considered graph category.*

$$\begin{array}{ccc} L & \overset{p}{\rightarrow} & R \\ \downarrow m & (1) & \downarrow m^* \\ G & \overset{p'}{\rightarrow} & H \end{array}$$

Figure 2.10: Rule application in the SPO approach.

The interesting thing about the SPO approach is that the application of a rule p might have side-effects, i.e., the rule application removes more elements than specified by the rule explicitly. Previously, we have seen two side-effects that might both occur in the SPO approach, namely the deletion of dangling edges, and the deletion of nodes for which the preserve-delete conflict occurs.

2.2.1.2 Double Pushout Approach

Th second graph transformation approach is the Double Pushout (DPO) approach [72, 45]. The DPO approach is based on the category **Graph**. In the DPO approach, transformation rules are described by three graphs L, K, and R together with two total graph morphisms $l\colon K \to L$ and $r\colon K \to R$.

Definition 2.16 (DPO transformation rule)**.** *A* DPO graph transformation rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ *consists of a left-hand-side graph* L*, an interface graph* K*, and a right-hand-side graph* R*, together with two total graph morphisms* $l\colon K \rightarrow L$ *and* $r\colon K \rightarrow R$*, where* l *is required to be injective.*

For a rule p to be *applicable* to a graph G, we have to find a *matching* m of the left-hand-side (LHS) of p. The application of a transformation rule p via a matching m is depicted in Fig. 2.11. The target graph H is constructed in two phases. First, we construct the *pushout complement* D of the diagram $(K \overset{l}{\rightarrow} L, L \overset{m}{\rightarrow} G)$, if it exists and is unique. The pushout complement exists if and only if the following conditions are satisfied [45]:

dangling edge condition: no edge $e \in (E_G - m(E_L))$ is incident to any node in $m(N_L - l(N_K))$;

identification condition: there is no $x, y \in (N_L \cup E_L)$ such that $m(x) = m(y)$ and $y \notin l(N_K \cup E_K)$.

A matching that satisfies both the dangling edge and the identification condition is said to satisfy the *gluing condition*. The fact that l is injective, then guarantees that the pushout complement is unique (up to isomorphism). As mentioned above (and exemplified in Example 2.8), in the category **SGraph**, pushout complements are not always unique, regardless of the morphism l being injective or not. Therefore, the DPO approach cannot be applied to the category **SGraph**.

Intuitively, the first phase of the DPO construction removes the elements that are specified to be deleted by the rule; the second phase then adds those elements to D that have to be created by the rule.

$$\begin{array}{ccccccc} L & \leftarrow l - & K & - r \rightarrow & R \\ \downarrow & & \downarrow & & \downarrow \\ m & (1) & m' & (2) & m^* \\ \downarrow & & \downarrow & & \downarrow \\ G & \leftarrow l' - & D & - r' \rightarrow & H \end{array}$$

Figure 2.11: Rule application in the DPO approach.

Definition 2.17 (direct DPO transformation)**.** *A* direct DPO transformation $G \overset{p,m}{\Longrightarrow} H$ *via a rule* p *and a matching* m *is given by the diagram in Fig. 2.10, where (1) and (2) are pushouts in* **Graph**.

Whereas a rule application in the SPO approach might result in side-effects due to the removal of dangling edges and the identification conflict, DPO transformations will never have side-effects. Matchings that would lead to dangling edges or identification conflicts do not satisfy the gluing condition and are therefore not allowed in direct derivations.

2.2.1.3 Sesqui Pushout Approach

A more recently introduced approach to graph transformation is the so-called Sesqui Pushout (SqPO) approach [42] in the category **Graph**. 'Sesqui' is the Latin word for *one-and-a-half*, which indicates that this approach is a mixture of the DPO and SPO approach. In the SqPO approach, transformation rules are DPO-like spans of morphisms. A direct SqPO transformation step is defined comparable to a direct DPO transformation, where, however, pushout (1) in Fig. 2.11 is replaced by a pullback in which D is the *final pullback complement*. Properties of the l morphism determine whether the SqPO approach behaves like the DPO or the SPO approach. In the case l is an *monomorphism*, the final pullback complement coincides with the pushout complement D, if D exists, yielding a SqPO transformation which behaves equivalently to a DPO transformation. The non-existence of the pushout complement in combination with the existence of a final pullback complement results in SqPO transformations which behave equivalent to SPO transformations in which dangling edges are removed. For non-injective l morphisms, the SqPO approach models the effect of *cloning* graph elements. When the matching m does not satisfy the identification condition, the final pullback complement does not exist and transformation is not allowed.

2.2.1.4 Comparison

The main advantage of the SPO approach is that every match yields a direct derivation whereas in the DPO approach the gluing condition must be satisfied additionally. Stated differently, in the SPO approach one does not need to know the entire context of a match for a rule to be applicable; in the DPO approach a rule can only delete a node if it also deletes all incident edges.

The major advantage of the DPO approach is that rule applications never have side-effects. That is, a rule application only deletes elements for which deletion is specified explicitly. Suppose for a DPO rule $p : L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ we define the reverse rule of p, denoted $\bar{p}$ as the rule obtained by exchanging the roles of L and R, i.e., $\bar{p} : R \stackrel{r}{\leftarrow} K \stackrel{l}{\rightarrow} L$. Then, if r is also injective (like l),

it can be shown that every direct direction $G \overset{p,m}{\Longrightarrow} H$ there exists a direct derivation $H \overset{\bar{p},m^*}{\Longrightarrow} G'$, where m^* is the co-match of m, and G' is isomorphic to G. Basically, this means that DPO transformations can be "undone", in case r is injective.

As mentioned above, the SqPO approach is a mixture of the SPO and DPO approach. In fact, the SqPO approach combines the advantages of both the DPO and SPO approach depending on the injectivity of l and the existence of pushout complements and final pullback complements. Furthermore, SqPO transformations can specify graph elements to be cloned.

2.2.1.5 Application Conditions

So far, the applicability of graph transformation rules only depended on the existence of a matching m from the left-hand-side of the rule to the graph. In [96], Habel et al. introduced a way of equipping rules with so called *positive* and *negative application conditions*, independent of the underlying algebraic approach. In this thesis, we will equip the transformation rules with negative application conditions (*NAC*s, for short) in *conjunctive* form only. That is, *all* *NAC*s have to be satisfied for the rule to be applicable. Then, a transformation rule p is extended with a set $\mathcal{N}$ of negative application conditions with a total graph morphism $n\colon L \to N$ for each NAC $N \in \mathcal{N}$.

Definition 2.18 (*NAC* satisfaction)**.** *Given a transformation rule p with a left-hand-side L_p and a set $\mathcal{N}_p$ of negative application conditions, a graph G, and a matching $m\colon L \to G$. Let $N \in \mathcal{N}_p$ be one of the negative application conditions of p. Then, m is said to* satisfy *N if there does not exist a total graph morphism $m_N\colon N \to G$ such that $m_N \circ n = m$, where $n\colon L \to N$ is the graph morphism from L to N.*

In Fig. 2.12, *NAC* satisfaction is schematically depicted for a graph transformation rule in the SPO approach. A rule p for which the set $\mathcal{N}$ is non-empty is then said to be applicable to a graph G if there exists a matching $m\colon L \to P$ that satisfies *all* negative application conditions $N \in \mathcal{N}$.

2.2.1.6 The GROOVE Approach

In the Groove Tool Set (which will be discussed in more detail in Section 2.3) graph transformations are performed on simple graphs using the SPO approach. The main advantages of applying the SPO approach is that we do not have to check whether the gluing conditions are satisfied and single transformation steps

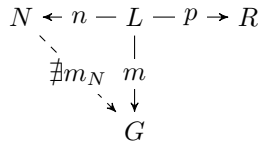


Figure 2.12: A rule applications with negative application conditions.

can be constructed fairly easily by removing and adding graph elements. The advantage of using simple graphs is that they can be used to specify binary relations over objects very naturally, e.g., in the context of first-order predicate logic (see, e.g., [157]). Binary relations contain each pair of objects at most once, which is intuitively reflected by the fact that in simple graphs for every distinct label there can exist at most one edge between two nodes.

When transforming simple graphs, the DPO approach cannot be applied. For a DPO transformation to be well-defined, the pushout complement should exist and be unique. For simple graphs, satisfying the gluing conditions does not imply the uniqueness of the pushout complement. A simple example is a rule that deletes an edge without deleting its source or target nodes. For every such an edge, the pushout complement might not or might still have this edge. When applying the SPO approach, every match is guaranteed to produce a unique pushout object (upto isomorphism).

2.2.2 Example: Circular Buffer

In this section, the concepts discussed in this chapter so far come together in a small example. We will model the behaviour of a circular buffer using graphs and graph transformations. The states of the buffer will be modelled as graphs and the operations that can be performed on the buffer as graph transformation rules. Fig. 2.13 depicts a graph representing a buffer containing two objects. The buffer-object is represented by a node labelled[1] Buffer; every cell of the buffer is represented by a node labelled Cell. The first Cell of the buffer is pointed to by an edge labelled first; the last cell is identified by a last-labelled edge. Objects are associated to Cells by means of val-labelled edges.

The operations being performed on the buffer determine whether the buffer behaves, for example, as a *first-in-first-out* or as a *first-in-last-out* buffer. For

[1]Formally (recall Def. 2.1) nodes are not labelled. However, we often depict labels of self-edges as node labels and accordingly speak about 'labelled nodes'.

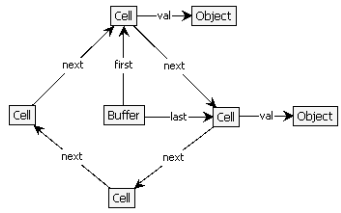


Figure 2.13: Graph represenbuffer-example-graphting a Buffer containing two Objects.

the sake of simplicity, we only consider the following two operations:

put: add an element to the buffer placing it in the first empty cell at the end of the buffer;

get: remove an element from the buffer from the first non-empty cell at the front of the buffer.

The transformation rules modelling these operations are shown in Fig. 2.14 (put) and Fig. 2.15 (get). The rules are shown as SPO rules, i.e., as two graphs L and R with a graph morphism $p\colon L \to R$, which is here specified implicitly through the placing of the graph elements. The put-rule includes a single NAC which requires that the next Cell (with respect to the current first Cell) does not yet have an Object associated to it.

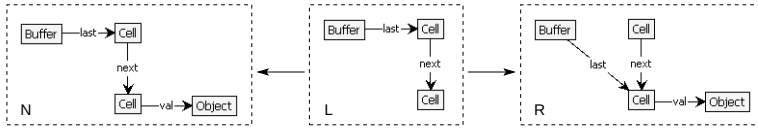


Figure 2.14: Graph transformation rule put.

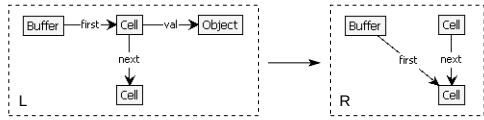


Figure 2.15: Graph transformation rule get.

Starting from the empty buffer and repeatedly applying both transformation

rules results in a graph transition system as shown in Fig. 2.16. State s_0, being the initial state as indicated by the single arrowhead, represents the state in which the buffer is empty; state s_4 represents the fully filled buffer. The internal graph structure of state s_2 has been depicted in Fig. 2.13. For readability we have omitted the matchings. Note that in this example, the put and get rule seem to be each other's *inverse*. In fact, this is not the case. Applying the get-rule immediately after a put-rule results in the same state due to the fact that both graphs are *isomorphic* and are therefore identified. We will come back to this in Section 2.3.1.1.

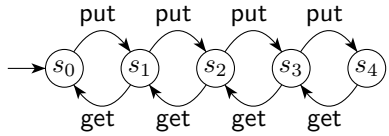


Figure 2.16: Graph transition system of the circular buffer example.

2.3 Tools in the Field

Part of the work described in this dissertation involved the development of a tool (set) that combines graph transformation functionality with a model checking engine that allows verifying the state spaces generated from graph production systems. In this section we will introduce the Groove Tool Set. Thereafter, we will give a (incomplete) overview of graph transformation tools closely related to the topic of this thesis, pointing out the main differences when compared to the Groove Tool Set.

2.3.1 The GROOVE Tool Set

As mentioned in Chapter 1, the Groove project is centered around the verification of object-oriented systems. The Groove Tool Set mainly focuses on the generation of transition systems from graph production systems that specify the behaviour of programs by means of graph transformation rules. In the next paragraphs, we will give a short description of the tools included in the Groove Tool Set, and discuss their main features.

The Groove Tool Set has been written entirely in Java and comprises approximately 400 classes and 150,000 lines of code. Basically, the Groove Tool Set consists of the following tools:

- the Groove Simulator,
- the Groove Generator,
- the Groove Editor, and
- the Groove Imager.

The Groove Simulator provides functionality for constructing state spaces from graph production systems manually (in a single step-wise fashion) or automatically (using different exploration strategies, discussed later on) through a graphical user interface. The Groove Generator is a command-line tool which can be used to generate partial or full graph transition systems without the graphical user-interface. For finite graph production systems the Groove Generator can be configured such as to store the set of all final states of the corresponding graph transition system (if there are any). The main advantage of the Groove Generator above the Simulator is that no time is spent on expensive graph rendering algorithms and no memory is used for constructing the corresponding data structures needed for that. The Groove Editor provides a graphical user interface for specifying graphs to be transformed and graph productions. Finally, the Groove Imager enables one to generate pictures from graphs or graph productions in various types of formats, among which, JPG, PNG, and EPS.

2.3.1.1 Single Graph Rule Presentation

In contrast to traditional approaches where graph transformation rules are depicted by separate graphs for the left-hand-side (LHS), the right-hand-side (RHS), and NACs, Groove represents rules in a single graph, using colours to distinguish the different roles graph elements can fulfill. We distinguish between the following roles:

- *reader*: reader elements are graph elements that are both in the LHS and the RHS. In the rule they are depicted as solid black boxes (nodes) and arrows (edges);
- *eraser*: eraser elements are graph elements that are in the LHS only. They are depicted as dashed blue (gray, in a black-and-white printout) boxes and arrows;

- *creator*: greater elements are those nodes and edges that are in the RHS only. In the rule they are visualized as solid fat green (light gray) boxes and arrows;
- *embargo*: embargo elements represents negative application conditions. They are depicted as dashed fat red (fat gray) boxes and arrows.

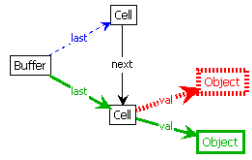


Figure 2.17: Single graph representation of the put-rule from Fig. 2.14.

Exploration Strategies. Both the Groove Simulator and the Groove Generator can be used to generate state spaces automatically using different *exploration strategies*. For arbitrary graphs, there can be a set of rule applications, all leading to different (or isomorphic) graphs. Which rule application to explore depends on the specific explore strategy chosen. In Groove, a variety of strategies have been implemented, among which are:

- *full*: for every reachable graph (starting from the start graph), all rule applications are explored;
- *linear*: for every graph, a single rule application is selected and explored. This strategy basically results in a linear execution path of the system;
- *barbed*: the resulting state space will look like barbed wire. This means that for every graph all rule applications are computed and added to the graph transition system as transitions. From the set of successor graphs, only one is selected for further exploration. This strategy can be useful when one is interested in local state properties along a linear path;
- *edge-bounded*: this strategy expects as input an upperbound on the number of the edges with a specific label that may occur in a graph for it to be further explored. For example, given 'Cell =50', this strategy explores all graphs containing less than 50 edges labelled Cell.

Priorities. The application of transformation rules can be further controlled by prioritizing the rules. Each rule can be assigned a single priority. The transformation engine will always first try to apply a rule from the set of rules with highest priority (0 being the lowest priority). Whenever a rule has been applied, new rule applications will again be computed for rules with the highest priority first.

Isomorphism Checking. One advantage of the graph formalism is that symmetry among states can be recognized through isomorphic graphs. In Groove, a mechanism has been implemented which is based on so called *graph certificates* for determining whether two graphs are isomorphic [158]. The basic idea is that all graph elements (nodes and edges) are assigned a number, the certificate, that is invariant under isomorphism. The certificate of the entire graph is then composed from the individual element certificates deterministically. Such certificates can then be used as a heuristic for identifying isomorphic graphs in a (previously computed) set of graphs. For state space exploration in Groove this means that whenever a rule application results in a graph G' isomorphic to a graph G that is already in the state space, G will be used as the target graph and G' does not need to be further explored. This is the reason why, in the circular buffer example from Section 2.2.2, the application of a get-rule immediately after the application of a put-rule (and vice versa) results in the same (isomorphic) graph (cf. also Def. 2.13 and Fig. 2.16).

Input/Output. In Groove, both graphs and transformation rules are stored externally in the GXL format [199], which is an XML-based format designed to be a standard exchange format for graphs. Graph grammars are stored in directories whose names end with "`.gps`". Graph grammars can be nested to an arbitrary depth.

2.3.2 Graph Transformation Tools

There are a number of other graph transformation tools around. Each of them focuses on special fields of research and therefore their features are quite diverse. In the following paragraphs, we will list a number of them which are applied in active fields of research. For each of them we will shortly discuss their main features. Other graph transformation tools that are less related to the topic of this work are, among others, Atom^3 [50], Viatra2 [12], VMTS [195].

2.3.2.1 AGG

The Attributed Graph Grammar System [182, 180], or AGG for short, is a graph transformation tool which mainly focuses on the development and implementation of theory on confluence and termination properties of attributed graph grammars. It includes algorithms for determining so-called *critical pairs* [100] among rule applications, algorithms for analyzing *termination of layered graph grammars* [63], and a mechanism for *consistency checking* [101].

With AGG, graph grammars can be specified including attributed graphs, i.e., graphs in which graph elements can be enriched with attributes of algebraic data types and, in the case of AGG, even JAVA expressions. From a graph grammar, one is able to manually or automatically (non-deterministically) derive graphs with AGG by repeatedly applying the transformation rules from the grammar. During transformation, intermediate graphs are not stored. This is one of the main differences compared to the GROOVE tool. Transformations in AGG are performed on multigraphs, in contrast to simple graphs as in GROOVE, using the SPO approach. By switching on the gluing conditions, it can also realize DPO transformations.

AGG also supports *typed* graph grammars. Graph grammars then include a specific graph called the *type graph* [44] to which every graph has a *typing morphism*. Whenever the start graph has a well-defined typing morphism and all transformation rules are guaranteed to preserve the typing constraints, the graphs that can be derived from the start graph through direct graph derivations of the rules can also be ensured to have well-defined typing morphisms.

Although AGG uses its own GGX-format for storing graph grammars, it also includes import and export functionality for GXL and export functionality for GTXL [130], which is an XML-based format to exchange entire graph transformation systems.

2.3.2.2 GreAT

The Graph Rewriting and Transformation (GREAT) language [2, 5] is a graphical language for specifying graph transformations among domain-specific modelling languages (DSMLs). The language is accompanied by a tool set which can be used to specify, debug, and execute such graph transformations. The GREAT language is split into three sub-languages: the *pattern specification language* (closely related to UML class diagrams), the *transformation rule language*, and the *sequencing of control flow language*. Together they allow the user to specify graph transformation rules based on a set of meta-models (for

the source and target language) and control the way they will be executed.

The effect of a transformation is specified by assigning different roles (one of *binding*, *delete*, or *new*) to pattern objects. Application constraints on the attributes of objects and associations are specified as OCL constraints. The transformations are then performed similar to SPO transformations.

Although GReAT provides intuitive means for specifying model transformations (which is not what we focus on) and guaranteeing termination of the transformation process, it does not include any means for verifying whether the end result is a (or *the*) correct model of the target language other than manually ensuring that the rules are correct and complete.

2.3.2.3 FUJABA

The primary topic of the Fujaba[2] Tool Suite [142] is to provide an easy to extend UML and graph transformation platform with the ability to add plugins. The Fujaba Tool Suite offers functionality to specify UML class diagrams and UML behaviour diagrams from which Java source code can be generated, resulting in an executable prototype. For the way back, Fujaba can parse Java source code and represent it within UML. The Fujaba Tool Suite RE Edition is especially configured with plugins for *reverse engineering* and design pattern recognition.

Transformations (or actually entire programs) are specified as so-called *Story Diagrams*. Story Diagrams are UML-based graphical specifications. They adopt UML activity diagrams to link different activities, being either program code or graph transformation rules (also called *Story Patterns*), using different control structures, thus providing a natural way of controlling the application of the different transformation rules or execution of actual program code.

The Fujaba environment supports the translation of such Story Diagrams to executable Java code, thus supporting rapid prototyping. Although Fujaba includes some mechanisms for checking certain consistency conditions on Story Diagrams, it does not provide any means of verifying the actual behaviour of the programs generated from them, other than including activities with attribute conditions and guards.

2.3.2.4 AUGUR

Augur [120] is a verification tool which analyzes graph transformation systems by approximating them with Petri nets [156]. Augur includes analysis algo-

[2] Fujaba is an acronym of "From UML to Java and back again".

rithms based for Petri nets based on so called *coverability graphs* and *backward reachability*. By modelling the graph grammar under consideration as a Petri graph [9] there exists a simulation relation between the reachable graphs of the graph grammar and the reachable markings of the net.

The properties to be verified need to be expressed either as regular expressions with the set of labels of the graph as the alphabet or as a monadic second order logic formula. The verification process is based on abstraction-refinement techniques. Whenever the tool finds a spurious counterexample, the abstraction is refined and the verification process is started again.

Augur supports GTXL as input format and produces GXL files as output representing counterexamples, if they exist.

2.4 Conclusion

In this chapter, a brief overview of concepts, and tools has been given in the area of graph transformation. We have started with introducing graph-related definitions, after which the most important algebraic graph transformation approaches have been discussed, being the Double Pushout Approach, the Single Pushout approach, and the Sesqui Pushout approach. The last has been introduced recently and is therefore not yet applied in large contexts. We have shown that graph production systems give rise to graph transition systems in which states have an internal graph structure and transitions represent applications of transformation rules. Next, we discussed the main features of the Groove Tool Set as it was before integrating the techniques explained and discussed in this thesis.

Finally, an (incomplete) overview has been given of related tools in the fields of graph transformation. The graph transformation tools included are Agg, GReAT, Fujaba, and Augur, of which only Augur focuses on verifying finite state graph production systems which is the central topic of this thesis.

3 Uniform Attributed Graphs

3.1 Introduction

When modelling object-oriented systems and verifying their behaviour using graphs and graph transformations, a key feature is the integration of data values in graph structures and algebraic operations in graph transformation specifications. Graphs in which nodes (and edges) can be assigned attributes of some data type are often called *attributed graphs*.

During the last fifteen years, a number of approaches to modelling attributed graphs and their transformations have been proposed (e.g., [135, 18, 100, 65]). The approaches mainly diverge in two dimensions: the first dimension being the way attributes are mathematically included in the graph, the second dimension concerns the way attribute values are changed. Löwe et al. [135] introduced *attribute carriers* being additional nodes that attach an attribute to a node or edge. Other approaches like the one proposed by Heckel et al. [100] and Ehrig et al. [65] use a graph structure in which attributes are referenced through special edges, directly connecting the node or edge to be attributed with the attribute itself. For changing attribute values, the two main alternatives are the *relabelling* approach and the *reconnecting* approach. In the relabelling approach, the nodes representing an attribute are preserved but the value represented by that specific node is changed by the transformation. This approach has been applied e.g., by Löwe et al. [135] and Plump et al. [152]. Changing attributed values in the reconnecting approach is basically established by replacing the attribute edge, i.e., the attribute edge pointing to the current attribute value

will be removed and an attribute edge referencing the new attribute value will be created. This approach is used by Ehrig et al. [65].

In traditional approaches, attributed graph transformations are described by including the *algebraic operations* as *terms* in the graph structure, but specifying the application constraints on the attributes externally (usually as *algebraic equations* over the terms). This makes the transformation specification less transparent and the implementation more involved. That is, next to graph matching algorithms, one has to implement mechanisms for variable evaluations and an equation solving engine which determines the solutions for the equations specified over the terms.

We propose a novel approach that has a close relation with the approach by Ehrig et al. [65]. Our approach introduces a *uniform* framework for the specification of attributed graphs and their transformation. In our approach, all these concepts are included in the graph formalism, thus providing a uniform and transparent attributed graph transformation framework. Such a uniform framework furthermore enables the tasks carried out when performing attributed graph transformations such as, e.g., the value assignment of variables and the evaluation of conditional expressions over variables, to be included in the graph matching algorithm in a natural way. We thus reduce the implementation efforts for extending the Groove Tool Set to support the use of attributes.

In this chapter we use an example graph transformation system to illustrate our approach. This example models the behaviour of a Vector in/from which we can insert/remove Objects. Figure 3.1 depicts how a typical rule in this example would be specified using the approach of Ehrig et al. [65].

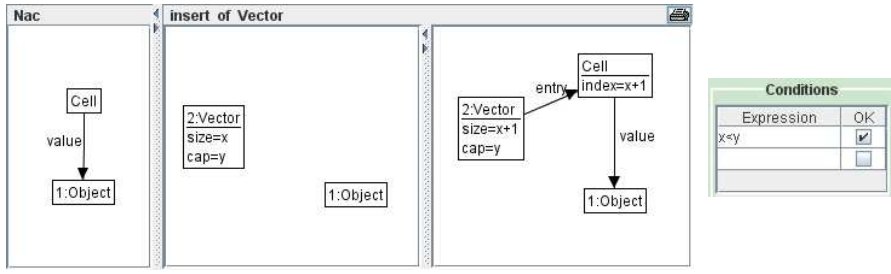


Figure 3.1: A typical graph transformation rule using the approach by Ehrig et al.

We aim at including both the conditional expressions on the variables and the expressions to update their actual values *in* the graph structure. This has

the additional advantage that the necessity of introducing variable names disappears. Figure 3.2 then depicts how the rule from Fig. 3.1 would be specified in Groove eventually.

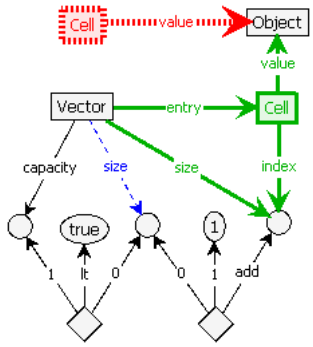


Figure 3.2: The rule of Fig. 3.1 specified in Groove.

Since Groove only supports the use of binary graphs, we have to close the gap between algebras that may have operations of arbitrary arities and the binary graph formalism. Modelling an arbitrary algebra as a graph, the so-called *algebra graph*, using binary edges can be done in several ways. The hyperedge-like structure of operations can be captured by introducing a separate node for every operation being applied on any combination of operands. Special edges then have to indicate the order of the operands and the result of the operation. This results in algebra graphs that are unnecessarily large (although they could be infinite anyway due to infinite data domains). Algebras that contain unary operations only (i.e., algebras of a so called *graph structure signature* [135]) can be modelled as graphs very naturally, since every operation instance can then be represented by a binary edge pointing from its operand to the corresponding result.

The main idea of our approach is to translate arbitrary algebras to equivalent algebras containing unary operations only. For this we first have to modify the underlying signature. We will show that from arbitrary signatures we can construct an equivalent graph structure signature by introducing product sorts and fresh functional and projection operation symbols. This construction will be called *flattening*. In order to prove the equivalence between the original and the modified signature we need to show that the original signature can be reconstructed from the flattened signature. This reconstruction is called

unflattening. The equivalence of arbitrary signatures and their corresponding flat signature can be lifted to the level of arbitrary algebras and to attributed graphs over arbitrary algebras. This allows us to prove one of the main results of this chapter, namely that our approach is categorically equivalent to the approach by Ehrig et al. [65], except from the fact that we do not support edge attributes and typing.

Eventually, we aim at model checking of object-oriented systems for which the state space has been generated by performing graph transformations. One way of alleviating the state-explosion problem is to apply abstraction techniques. Our uniform approach to modelling attributed graphs provides natural ways for abstraction on attribute values. Instead of interpreting attributed graphs and their transformation specifications on *concrete algebras*, the Groove Tool Set can be extended with *abstract algebras*. In this chapter we will elaborate on the consequences of performing what we call *abstract attributed graph transformations*.

Overview of the Chapter

This chapter is structured as follows. In Section 3.2 we start by recalling some basic definitions concerning algebraic specifications, both formally and intuitively, and discuss some additional categorical concepts that are used in this chapter. In order to guarantee that our approach produces correct results we show the relation between our approach and existing ones. This correspondence is based on the equivalence of the specific categories of attributed graphs. The equivalence is proven by first proving the equivalence of the signatures the attributed graphs in the different categories rely on. This is achieved by introducing *flattening* and *unflattening* functors at the level of signatures (Section 3.3) and algebras (Section 3.4). In Section 3.5 we introduce the notion of *uniform attributed graphs* and prove the equivalence with the approach introduced by Ehrig et al. [65]. Transformations of uniform attributed graphs can then be defined in the usual way, as discussed in Section 3.5.3. In Section 3.6 we elaborate on the way our approach provides a natural way of specifying abstractions on attribute values. In Section 3.7 we shortly discuss some implementation issues, after which we end this chapter with Section 3.8 containing some concluding remarks. The proofs of some major results are included in this chapter. The remaining proofs can be found in Appendix B.

3.2 Preliminaries

3.2.1 Signatures and Algebras

First we recall a number of basic definitions concerning algebraic specifications.

Definition 3.1 (signature)**.** *A signature* $SIG = (S, OP)$ *consists of:*

- *a set* $S = \{s_1, \ldots, s_n\}$ *of* sorts*;*
- *a set* $OP = \{o_1, \ldots, o_m\}$ *of* constant *and* operation symbols.

together with functions $\sigma\colon OP \to S^*$, $\tau\colon OP \to S$.

Given two signatures $SIG = (S, OP)$ *and* $SIG' = (S', OP')$*, a* signature morphism $h\colon SIG \to SIG'$ *is a pair of mappings* $h = (h_S, h_{OP})$ *such that* $\sigma(h_{OP}(o)) = h_S(s_1) \cdots h_S(s_n)$ *and* $\tau(h_{OP}(o)) = \tau(o)$ *for every* $o \in OP$ *with* $\sigma(o) = s_1 \cdots s_n$.

For every operation $o \in OP$, $\sigma(o)$ and $\tau(o)$ will be called the *parameter sort sequence* and the *target sort*, respectively. We often write $o\colon s_1 \cdots s_n \to s$ when $\sigma(o) = s_1 \cdots s_n$ and $\tau(o) = s$. The *arity* of an operation o will be denoted $\alpha(o)$, such that $\alpha(o) = n$ if $\sigma(o) = s_1 \cdots s_n$. For constant symbols, the parameter sort is the *empty sequence*, denoted ε.

Signatures that contain unary operations only, i.e., $\alpha(o) = 1$ for all $o \in OP$, are called *graph structure signatures* as introduced by Löwe et al. [135].

Definition 3.2 (graph structure signature)**.** *[135] A signature* $SIG = (S, OP)$ *is a* graph structure signature*, if for all* $o \in OP : \alpha(o) = 1$.

In the remainder we assume that signatures do not contain *spurious* sorts, i.e., every sort is part of the parameter sort sequence of some operation symbol or acts as a target sort for some operation symbol (usually both). The reader can verify that this assumption is valid for every signature that will be constructed in this chapter.

In this chapter we assume the existence of a *global* set $\mathcal{S}$ of sorts and a global set $\mathcal{O}$ of operation symbols. Specific signatures take a subset of $\mathcal{S}$ as their sorts and a subset of $\mathcal{O}$ as their operation symbols to which they assign parameter sequence sorts and target sorts, so σ and τ are local to the signature. Later on we will be introducing bijective functions on those sets for proving that applying certain operators on signatures does not change the original signature.

Example 3.3. *An example signature is* $SIGINT = (\mathsf{int}; +, 0)$*, with typings* $+\colon \mathsf{int} \times \mathsf{int} \to \mathsf{int}$ *and* $0\colon \to \mathsf{int}$*. In our notation, the sorts of a signature are separated from the operations by a semi-colon. Intuitively, this signature describes*

the structure of integer algebras including the addition operation + having 0 *as the only constant symbol. Take* $SIGINT' = (\mathsf{int}', *, 0')$ *as another signature with* $\sigma(*) = \mathsf{int}'\,\mathsf{int}'$ *and* $\sigma(0) = \varepsilon$ *and* $\tau(*) = \tau(0) = \mathsf{int}'$*. A signature morphism* $h\colon SIGINT \to SIGINT'$ *is then defined as* $h_S(\mathsf{int}) = \mathsf{int}'$*,* $h_{OP}(+) = *$*, and* $h_{OP}(0) = 0'$*.* ■

Definition 3.4 (algebra)**.** *Let* $SIG = (S, OP)$ *be a signature. Then, a SIG-algebra* $A = (S_A, OP_A)$ *consists of*

- *a set* $S_A = \{A_s \mid s \in S\}$ *of carrier sets* A_s*, one for each* $s \in S$*,*
- *a set* $OP_A = \{op_o\colon A_{s_1} \times \cdots \times A_{s_n} \to A_s \mid o\colon s_1 \cdots s_n \to s \in OP\}$ *of operations.*

Given two SIG-algebras B *and* B'*, a SIG-algebra homomorphism* $h\colon B \to B'$ *is a family* $h = (h_s)_{s\in S}$ *of mappings* $h_s\colon A_{B,s} \to A_{B',s}$ *such that for each operation symbol* $o\colon s_1 \cdots s_n \to s \in OP$*, it holds that*

$$h_s(op_{B,o}(a_1, \ldots, a_n)) \quad = \quad op_{B',o}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \ ,$$

with $a_i \in A_{s_i}$*, for* $1 \le i \le n$*.*

We write $A : Alg(SIG)$ whenever A is a SIG-algebra. Given two algebras $A, B : Alg(SIG)$, an algebra homomorphism f from A to B is denoted $f : A \to_{Alg(SIG)} B$. The subscript indicating the specific signature is often omitted whenever this is clear from the context. Function and morphism *composition* will be denoted as usual, i.e., if $g\colon A \to B$ and $h\colon B \to C$ are morphisms [functions], then $h \circ g\colon A \to C$ is the composed morphism [function]. That is, if g and h are functions, then for all $a \in A : (h \circ g)(a) = h(g(a))$.

Example 3.5. *Given the signature* $SIGINT = (\mathsf{int}; +, 0)$ *as introduced above, an example SIGINT-algebra is* A *such that:*

- $A_{\mathsf{int}} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$*,*
- $op_+(a, b) = a + b$*, for all* $a, b \in \mathsf{int}$*,*
- $op_0() = 0$*, mapping the constant symbol to the value* 0*.*

An example algebra homomorphism $h\colon A \to A$ *maps every value from* A_{int} *to its negative value, i.e.,* $h_{\mathsf{int}}(a) = -a$*. This is indeed an algebra homomorphism since* $h_{\mathsf{int}}(0) = -0 = 0$ *and* $h_{\mathsf{int}}(op_+(a, b)) = -(a + b) = -a + -b = op_+(-a, -b) = op_+(h_{\mathsf{int}}(a), h_{\mathsf{int}}(b))$*.* ■

The special thing about graph structure signatures is that algebras of such signatures can straightforwardly, and moreover equivalently, be represented as graphs, as from Def. 2.1, i.e., as multigraphs. For this we introduce the notion of *algebra graphs*.

Definition 3.6 (algebra graph)**.** *Let $SIG = (S, OP)$ be a graph structure signature. A SIG-*algebra graph *(G, t) consists of a graph G and a pair of mappings $t = (t_N, t_E)$ with $t_N \colon N_G \to S$ mapping every node to a sort $s \in S$ and $t_E \colon E_G \to OP$ mapping every edge to an operation symbol $o \in OP$. Furthermore, it must hold that for all $n \in N_G$ and for all $o \in OP$:*

$$\sigma(o) = t_N(n) \implies \exists e \in E_G : (\mathsf{src}(e) = n \wedge t_E(e) = o) \tag{3.1}$$

Furthermore, source and target functions of G commute with parameter and target sorts of the operation symbols in OP, respectively; that is, it holds that

$$\begin{aligned} t_N \circ \mathsf{src}_G &= \sigma \circ t_E \\ t_N \circ \mathsf{tgt}_G &= \tau \circ t_E \ . \end{aligned}$$

Let A and $\hat{A}$ be two SIG-algebra graphs, and $t_A \colon A \to SIG$ and $t_{\hat{A}} \colon \hat{A} \to SIG$ be their respective mappings to SIG. A SIG-algebra graph morphism $f \colon A \to \hat{A}$ is a total graph morphism such that f commutes with t, i.e., $t_{\hat{A}} \circ f = t_A$ (see Fig. 3.3).

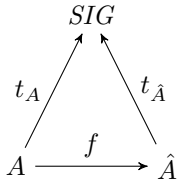


Figure 3.3: Algebra graph morphism.

The pair (t_N, t_E) can be interpreted as a *typing* of the elements in the algebra graph over the elements of the signature. When for some node n of an algebra graph $t_N(n) = s$, we say that n *is of type* s. Since we are dealing with edge-labelled graphs, the edges of algebra graphs will be labelled with the operation symbol of their type. Condition (3.1) specifies that all nodes n for which the type is a parameter sort of some operation o, n must have an outgoing edge of

type o. Stated differently, the fact that algebraic operations are total, must be reflected in algebra graphs. Note, however, that the above definition does not require algebra graphs to be deterministic. In fact, in the following we will show that SIG-algebra graphs that are deterministic are equivalent to SIG-algebras, for an arbitrary graph structure signature SIG.

3.2.2 Categories of Signatures, Algebras, and Algebra Graphs

Before we can prove the previously mentioned equivalence, we first have to introduce some other basic categories, namely that of signatures, and that of algebras of a given signature. In Appendix A, we introduce some of the basic concepts in category theory which are used throughout this chapter.

Definition 3.7 (Categories **Sig** and **Alg**(SIG))**.** *The category* **Sig** *has signatures as objects and signature morphisms as arrows. The category* **Alg**(SIG) *has SIG-algebras as objects and SIG-algebra homomorphisms as arrows, for a given signature SIG.*

For showing that **Sig** and **Alg**(SIG) are indeed categories, we have to show that all characteristics, as mentioned in Appendix A, are satisfied. For the category **Sig** we will indicate how those requirements can be shown to hold. For an arbitrary element $SIG = (S, OP) \in Obj_{\mathbf{Sig}}$, we take the identity arrow $id_{SIG} = (id_S, id_{OP})$. For all $s \in S$ we then have $id_S(s) = s$. Furthermore, for all $o \in OP$ with $\sigma(o) = s_1 \cdots s_n$ and $\tau(o) = s$ we have $\sigma(id_{OP}(o)) = id_S(s_1) \cdots id_S(s_n) = s_1 \cdots s_n$ and $\tau(id_{OP}(o)) = id_S(s) = s$. Thus, $\sigma(o) = \sigma(id_{OP}(o))$ and $\tau(o) = \tau(id_{OP}(o))$ and thus id_{SIG} is an arrow in **Sig**. Given two arrows $f\colon SIG \to SIG'$ and $g\colon SIG' \to SIG''$, the composition $g \circ f$ is then taken as the component-wise composition, i.e., $f \circ g = (g_S, g_{OP}) \circ (f_S, f_{OP}) = (g_S \circ f_S, g_{OP} \circ f_{OP})$. One can easily verify that this composition is indeed an arrow in **Sig**. Associativity of arrow composition for arbitrary arrows f, g, and h can be shown to hold as follows:

$$\begin{aligned}
(f \circ g) \circ h &= ((f_S, f_{OP}) \circ (g_S, g_{OP})) \circ (h_S, h_{OP}) \\
&= (f_S \circ g_S, f_{OP} \circ g_{OP}) \circ (h_S, h_{OP}) \\
&= ((f_S \circ g_S) \circ h_S, (f_{OP} \circ g_{OP}) \circ h_{OP}) \\
&= (f_S \circ (g_S \circ h_S), f_{OP} \circ (g_{OP} \circ h_{OP})) \\
&= (f_S, f_{OP}) \circ ((g_S \circ h_S), (g_{OP} \circ h_{OP})) \\
&= (f_S, f_{OP}) \circ ((g_S, g_{OP}) \circ (h_S, h_{OP})) \\
&= f \circ (g \circ h) \ .
\end{aligned}$$

For an arbitrary arrow $f: C \to D$, the requirement on the composition of f with the identity morphisms id_C and id_D can be shown to hold as follows:

$$\begin{aligned} f \circ id_C &= (f_S, f_{OP}) \circ (id_{S,C}, id_{OP,C}) \\ &= (f_S \circ id_{S,C}, f_{OP} \circ id_{OP,C}) \\ &= (f_S, f_{OP}) \\ &= (id_{S,D} \circ f_S, id_{OP,D} \circ f_{OP}) \\ &= (id_{S,D}, id_{OP,D}) \circ (f_S, f_{OP}) \\ &= id_D \circ f \ . \end{aligned}$$

For a given graph structure signature, we use $\mathbf{AlgGraph}^+(SIG)$ to denote the full sub-category of **Graph** having all *SIG*-algebra graphs as objects and all corresponding *SIG*-algebra graph morphisms as arrows. Within this category we can distinguish a further full sub-category consisting of all deterministic *SIG*-algebra graphs and their intermediate morphisms; this category will be denoted $\mathbf{AlgGraph}(SIG)$.

Lemma 3.8. *If SIG is a graph structure signature, then the categories* $\mathbf{Alg}(SIG)$ *and* $\mathbf{AlgGraph}(SIG)$ *are equivalent.*

Proof sketch. The proof is based on two functors, one of which turns an algebra into a graph by including all data values as nodes in the graph and operations as edges; the other functor (re)constructs the algebra from the edges. The full proof can be found in Appendix B (Section B.1). □

Remark 3.9. *Note that the above equivalence is based on algebra graphs being multigraphs. Due to the fact that operations of algebras are functional, and the additional requirement that for deterministic algebra graphs their operation and projection edges between two nodes must be unique, Lemma 3.8 also holds when* **Graph** *represents the category of simple graphs.*

3.3 Uniform Signatures

In this section we define signatures with a specific structure, called *uniform signatures* and indicate how arbitrary signatures can be transformed into an equivalent uniform signature. Uniform signatures consist of two components: a signature and a partial order. The signature component is a *graph structure signature* [135], which means that algebras of uniform signatures can equivalently be modelled as algebra graphs, according to the result from Section 3.2.

Eventually, this allows us to represent attributed graphs as one graph structure instead of a graph with an algebra component as is done in other approaches like, e.g., by Ehrig et al. [65].

For the signature component of uniform signatures we define some additional structure. The operation symbols of the signature component can be partitioned into *functional* and *projection* operation symbols. The set of data sorts can be partitioned into *structured* and *flat* data sorts. The partial order specifies an ordering relation on the operation symbols that is *total* on every set of projection operation symbols that have a common parameter sort.

Definition 3.10 (uniform signature)**.** *A* uniform signature $USIG = \langle (S, OP), \prec \rangle$ *consists of a signature* (S, OP) *and a partial ordering* $\prec$ *on the operation symbols. For the set* OP *of operation symbols there exist disjoint sets* F *and* Π *such that* $OP = F \cup \Pi$*, where*

- $F = \{f_1, \ldots, f_m\}$ *is a set of so-called* functional *operation symbols,*
- $\Pi = \{p_1, \ldots, p_n\}$ *is a set of so-called* projection *operation symbols.*

Furthermore,

- $\alpha(o) = 1$*, for all* $o \in OP$*.*

For the set S *of sorts there exists disjoint sets* U *and* D *such that* U *and* D *are exactly the set of all parameter sorts and target sorts, respectively, of the operations in* OP *.*

- $U = \{s \in S \mid \exists o \in OP : \sigma(o) = s\}$*, the set of* data *sorts;*
- $D = \{s \in S \mid \exists o \in OP : \tau(o) = s\}$*, the set of* product *sorts.*

The ordering relation $\prec$ *specifies an ordering on the elements in* Π *such that* $\prec$ *is* partial *on the set* Π*,* $\prec$ *is* total *on the set of projection operation symbols that share their parameter sort, and* $\prec$ *does not order projection operation symbols that have different parameter sorts. Formally,*

$$\forall p_1, p_2 \in \Pi : (p_1 = p_2) \vee (p_1 \prec p_2) \vee (p_2 \prec p_1) \iff \sigma(p_1) = \sigma(p_2) \ .$$

Let USIG and $USIG'$ *be two uniform signatures, a* uniform signature morphism h*:* $USIG \to USIG'$ *is a signature morphism* h*:* $USIG \to USIG'$ *that, additionally, preserves and reflects the ordering, i.e.,*

$$\forall p_1, p_2 \in \Pi : p_1 \prec p_2 \iff h_{OP}(p_1) \prec' h_{OP}(p_2) \ .$$

For every $u \in U$ we denote the subset of projection operation symbols having u as their parameter sort with Π_u, i.e.,

$$\Pi_u \quad = \quad \{p \in \Pi \mid \sigma(p) = u\} \ .$$

Note that for a uniform signature $USIG = \langle (S, OP), \prec \rangle$, the partitioning of S and OP is completely determined by τ and σ, and $\prec$, respectively. The former is based on the assumption that (uniform) signatures do not contain spurious sorts. The latter is true by definition, since $\prec$ is a partial ordering on the projection operation symbols only; the remaining operation symbols in OP thus form the set of functional operation symbols.

Remark 3.11. *The ordering of the elements in Π_u will be made explicit by an* index function *$I: \Pi \to \mathbb{N}$ mapping every projection operation symbol of a product sort u to an natural number in the range $[1, \cdots, |\Pi_u|]$. Preserving the ordering of the projection operation symbols is then equivalent to preserving their indices. In the proofs in the coming sections we will make extensive use of the indices instead of the ordering relation.*

At this point, we assume the existence of two additional disjoint global sets, namely $\mathcal{U}$ and $\mathcal{P}$, of product sorts and project operation symbols, respectively (recall the global sets $\mathcal{S}$ and $\mathcal{O}$ from Section 3.2.1). For an arbitrary uniform signature USIG, we then assume its components to satisfy the following conditions:

$$\begin{aligned} D_{USIG} &\subseteq \mathcal{S} \\ U_{USIG} &\subseteq \mathcal{U} \\ F_{USIG} &\subseteq \mathcal{O} \\ \Pi_{USIG} &\subseteq \mathcal{P} \ . \end{aligned}$$

Furthermore, we introduce two injective functions, namely $prod: \mathcal{S}^* \to \mathcal{U}$ and $proj: \mathcal{U} \times \mathbb{N} \to \mathcal{P}$. The former maps sequences of data sorts on product sorts; the latter takes as input a product sort and a natural number and returns a projection operation symbol. For the global set of data sorts and operation symbols we introduce the functions $dsort: \mathcal{S} \to \mathcal{S}$ and $oper: \mathcal{O} \to \mathcal{O}$, which are actually the identity functions on the respective sets, i.e., $dsort = id_{\mathcal{S}}$ and $oper = id_{\mathcal{O}}$, mapping every element to itself.

Likewise for arbitrary signature, we introduce a category **USig** consisting of uniform signatures.

Definition 3.12 (Category **USig**). *Uniform signatures and uniform signature morphisms form the category* **USig**.

The identity morphisms and morphism composition are defined analogously to the category **Sig**.

Example 3.13. *An example uniform signature is*

$$USIGINT = \langle(\mathsf{int}, \mathsf{int}^2, \bot; +, 0, p_1, p_2), \prec\rangle \ ,$$

with $D = \{\mathsf{int}, \bot\}$, $U = \{\mathsf{int}^2, \bot\}$, $F = \{+, 0\}$, *and* $\Pi = \{p_1, p_2\}$ *where*

- $+\colon \mathsf{int}^2 \to \mathsf{int}$,
- $0\colon \bot \to \mathsf{int}$,
- $p_1\colon \mathsf{int}^2 \to \mathsf{int}$,
- $p_2\colon \mathsf{int}^2 \to \mathsf{int}$.

Furthermore we have $\Pi_{\mathsf{int}^2} = \{p_1, p_2\}$ *with* $p_1 \prec p_2$, *and* $\Pi_\bot = \emptyset$. ■

3.3.1 Flattening Arbitrary Signatures

In this work we deal with, on the one hand, binary graphs for modelling systems and their behaviour, and, on the other hand, signatures and corresponding algebras (from which we construct attributed graphs) which we aim to represent using binary graphs as well. We already indicated that a special class of signatures, namely the class of graph structure signatures, can equivalently be represented as graphs.

The worlds of graphs and arbitrary signatures are brought together by introducing a process that translates arbitrary signatures to corresponding uniform signatures. This process is called *flattening*.

Definition 3.14 (flattening). *Let* $SIG = (S, OP)$ *be an arbitrary signature. Then,* $\mathcal{F}(SIG) = \langle(D \cup U, F \cup \Pi), \prec\rangle$ *is called the corresponding* uniform signature*, where*

- $D = S$;
- $U = \{prod(\sigma(o)) \mid o \in OP\}$,
  where $prod(\sigma(o))$ *is a fresh sort* ($\notin S$) *representing the product of the sorts in* $\sigma(o)$;

- $F = \{oper(o) \mid o \in OP\}$,
  with $\sigma(oper(o)) = prod(\sigma(o))$ *and* $\tau(oper(o)) = \tau(o)$*;*
- $\Pi = \{proj(\sigma(o), k) \mid o\colon s_1 \cdots s_k \cdots s_n \to s \in OP\}$,
  with $\sigma(proj(prod(\sigma(o)), k)) = prod(\sigma(o))$ *and* $\tau(proj(prod(\sigma(o)), k)) = s_k$.

Furthermore, the ordering $\prec$ *on the set* Π_u *is defined as the smallest ordering such that for every* $u \in U$*:*

- $\forall i, j \in 1 \ldots |\Pi_u| : i < j$ *implies* $proj(u, i) \prec proj(u, j)$.

Intuitively, flattening an arbitrary signature *SIG* results in a uniform signature $\mathcal{F}(SIG)$ in which the original sorts and operation symbols are reused and fresh product sorts and projection operation symbols are introduced. Although the functional operation symbols are directly taken from *SIG* (since $oper = id_{\mathcal{O}}$), in order to prevent from confusion, for an operation symbol $o \in OP_{SIG}$ we will denote the corresponding functional operation symbol in F_{USIG} with $\overline{o}$. For a product sort $prod(s_1 \cdots s_n)$ we introduce the shorthand notation $\overline{s_1 \cdots s_n}$; for the projection operation symbols $proj(u, k)$ of a product sort $u \in U_{USIG}$, we introduce the shorthand notation $p_{u,k}$.

From Def. 3.14 we can immediately derive the following property.

Lemma 3.15. *If SIG is a signature, then* $\mathcal{F}(SIG)$ *is a uniform signature.*

Example 3.16. *Taking the* $SIGINT = (\mathsf{int}; +, 0)$ *signature from the previous examples, the corresponding flattened signature has the following structure:*

$$\mathcal{F}(SIGINT) \quad = \quad \langle (\mathsf{int}, \overline{\sigma(+)}, \overline{\sigma(0)}; \overline{+}, p_{+,1}, p_{+,2}, \overline{0}), \prec \rangle \ ,$$

where

- $\overline{+}\colon \overline{\sigma(+)} \to \mathsf{int}$,
- $p_{+,1}\colon \overline{\sigma(+)} \to \mathsf{int}$,
- $p_{+,2}\colon \overline{\sigma(+)} \to \mathsf{int}$,
- $\overline{0}\colon \overline{\sigma(0)} \to \mathsf{int}$,
- $\prec = \{(p_{+,1}, p_{+,2})\}$.

From signature morphisms between two arbitrary signatures we can construct uniform signature morphisms. First we define how to flatten arbitrary signature morphisms, after which we will show that the resulting morphism is indeed a uniform signature morphism between the corresponding uniform signatures.

Definition 3.17. *Let $SIG = (S, OP)$ and $SIG' = (S', OP')$ be two signatures, $h\colon SIG \to SIG'$ be signature morphism, and $USIG = \langle (D \cup U, F \cup \Pi), \prec \rangle$ be the uniform signature corresponding to SIG, i.e., $USIG = \mathcal{F}(SIG)$. Then, the uniform signature morphism $\mathcal{F}(h)$ is the pair $\mathcal{F}(h) = (\mathcal{F}(h)_S, \mathcal{F}(h)_{OP})$ where, given $s \in D \cup U$, $\mathcal{F}(h)_S$ is defined as follows:*

$$\mathcal{F}(h)_S(s) = \begin{cases} h_S(s) & \text{, if } s \in D \\ \overline{\sigma(h_{OP}(o))} & \text{, if } s = \overline{\sigma(o)} \in U \text{ for some } o \in OP \end{cases} .$$

For all $o\colon s_1 \cdots s_k \cdots s_n \to s \in OP$ with $o' = h_{OP}(o)$ we have the following:

- $\mathcal{F}(h)_{OP}(\overline{o}) = \overline{o'}$ *with* $\sigma(\overline{o'}) = \overline{\sigma(o')}$ *and* $\tau(\overline{o'}) = \tau(o')$;
- $\mathcal{F}(h)_{OP}(p_{u,k}) = p_{u',k}$ *with* $u = \overline{\sigma(o)}$, $\sigma(p_{u',k}) = u' = \overline{\sigma(o')}$, *and* $\tau(p_{u',k}) = h_S(\tau(p_{u,k}))$.

The following result specifies the relation between the categories **Sig** and **USig**, as established by $\mathcal{F}$.

Lemma 3.18. *$\mathcal{F}$ is a functor from the category* **Sig** *to the category* **USig**.

Proof sketch. The proof consists of first showing that $\mathcal{F}$ produces correct morphisms. Furthermore it must commute with the identity morphisms and preserve morphism composition. The full proof is included in Section B.2. □

3.3.2 Unflattening Uniform Signatures

From a uniform signature *SIG*, we can also (re)construct a 'regular' signature, in which there is no restriction on the data sorts and operation symbols. This process will be called *unflattening* of uniform signatures; the unflattened signature of a uniform signature *SIG* will be denoted $\mathcal{U}(SIG)$.

Definition 3.19 (unflattening)**.** *Let $SIG = \langle (D \cup U, F \cup \Pi), \prec \rangle$ be a uniform signature. Then, $\mathcal{U}(SIG) = (D, OP)$ is called the corresponding unflattened signature, where*

$$OP \quad = \quad \{oper(f) \mid f\colon u \to d \in F\}$$

with $\sigma(oper(f)) = \tau(p_{u,1}) \cdots \tau(p_{u,n})$ and $\tau(oper(f)) = \tau(f)$.

When unflattening a uniform signature, we preserve the data sorts. The operation symbols in the unflattened signature are obtained by taking the preimage of the functional operation symbols under *oper*. Since *oper* is defined

as the identity function on $\mathcal{O}$, for a functional operation symbol f we have $oper(f) = oper^{-1}(f) = f$. In cases where confusion might arise about the signature to which an operation symbol belongs, we will denote the operation symbol $oper(f)$ corresponding to some functional operation symbol f with o_f.

Unflattening a uniform signature morphism $h\colon SIG \to SIG'$ results in a signature morphism between two 'regular' signatures.

Definition 3.20. *Let SIG and SIG' be two uniform signatures and $h\colon SIG \to SIG'$ be a uniform signature morphism. Then, $\mathcal{U}(h)$ is the pair $(\mathcal{U}(h)_S, \mathcal{U}(h)_{OP})$ where $\mathcal{U}(h)_S$ is defined as follows:*

- *for $d \in D_{SIG}$: $\mathcal{U}(h)_S(d) = h_S(d) \in D_{SIG'}$,*

and $\mathcal{U}(h)_{OP}$ is defined as follows:

- *for all $f\colon u \to d \in F_{SIG}$ with $f' = h_{OP}(f) \in F_{SIG'}$ we have $\mathcal{U}(h)_{OP}(o_f) = o_{f'}$ with $\sigma(o_{f'}) = h_S(\tau(p_{u,1})) \cdots h_S(\tau(p_{u,n}))$, and $\tau(o_{f'}) = h_S(\tau(f))$.*

Basically, $\mathcal{U}(h)_S$ is equal to h_S restricted to the elements of D. This is often denoted as $\mathcal{U}(h)_S = h_S{\restriction_D}$. Similarly, $\mathcal{U}(h)_{OP}$ only has images for all operation symbols in F.

Similar to the way we have shown $\mathcal{F}$ to be a functor from **Sig** to **USig**, we can show $\mathcal{U}$ to be a functor in the reverse direction, i.e., from **USig** to **Sig**.

Lemma 3.21. *$\mathcal{U}$ is a functor from the category* **USig** *to the category* **Sig**.

3.3.3 Composing Signature Flattening and Unflattening

In this section we will show that the categories **Sig** and **USig** are actually *equivalent*, based on the functors $\mathcal{F}$ and $\mathcal{U}$, which have been discussed in the previous sections. The equivalence is based on the two compositions of $\mathcal{F}$ and $\mathcal{U}$, i.e., $\mathcal{U} \circ \mathcal{F}\colon \mathbf{Sig} \to \mathbf{Sig}$ and $\mathcal{F} \circ \mathcal{U}\colon \mathbf{USig} \to \mathbf{USig}$. We will be showing that the former composition yields the identity functor on the category **Sig**, whereas the latter yields an bijective functor on **USig**.

Theorem 3.22 (**Sig** $\cong$ **USig**)**.** *The categories* **Sig** *and* **USig** *are equivalent.*

Since $\mathcal{F}$ basically only introduces new sorts (namely the product sorts) with corresponding projection operation symbols, the structure of the signature is preserved. When unflattening a flattened signature, the only thing to be reconstructed is the typing of the original operations. The total ordering of the projection operation symbols that share their parameter sort, enable this reconstruction. From this we may derive the following result.

Lemma 3.23. *$(\mathcal{U} \circ \mathcal{F})(SIG) = SIG$, for arbitrary signatures SIG.*

The reverse direction is more involved, since the product sorts and the projection operation symbols of a uniform signature $USIG$ are forgotten by $\mathcal{U}$. When assuming that we are not really interested in the actual names of the product sorts and projection operation symbols but rather in their semantics, we can show that the uniform signature $\mathcal{F}(\mathcal{U}(USIG))$ is equal to $USIG$ up to isomorphism. This is stated in the following lemma.

Lemma 3.24. *Let $USIG$ be a uniform signature. For the signature $USIG' = \mathcal{F}(\mathcal{U}(USIG))$ there exists an isomorphism $g_{USIG}\colon USIG \to USIG'$.*

At this point we have all the ingredients for showing that **Sig** and **USig** are actually equivalent categories.

Proof of Theorem 3.22. The equivalence is, obviously, based on the functors $\mathcal{F}$ and $\mathcal{U}$. In Lemma 3.23, we have shown that $(\mathcal{U} \circ \mathcal{F})(SIG) = SIG$ for all objects $SIG \in Obj_{\mathbf{Sig}}$, and therefore $\alpha_{SIG} = id_{SIG}$. For the reverse direction we have shown that there exists an isomorphism between any uniform signature $USIG$ and $\mathcal{F}(\mathcal{U}(USIG))$. Let $h\colon USIG \to USIG'$ be a uniform signature morphism and $g_{USIG}\colon USIG \to \mathcal{F}(\mathcal{U}(USIG))$ be the isomorphism for an arbitrary object $USIG \in Obj_{\mathbf{USig}}$ given by Lemma 3.24. We then have to show that the following commutativity holds for all $USIG \in Obj_{\mathbf{USig}}$:

$$(\mathcal{F} \circ \mathcal{U})(h) \circ g_{USIG} \quad = \quad g_{USIG'} \circ h \ .$$

For data sorts $d \in D_{USIG}$, the commutativity can be shown as follows:

$$\begin{array}{ll} & (\mathcal{F} \circ \mathcal{U})(h)(g_{USIG,S}(d)) \\ = & \{\ g_{USIG,S} = id_{USIG,S} \text{ for data sorts } \} \\ & (\mathcal{F} \circ \mathcal{U})(h)_S(d) \\ = & \{\ \mathcal{U} \text{ and } \mathcal{F} \text{ preserve data sorts } \} \\ & h_S(d) \\ = & \{\ g_{USIG',S} = id_{USIG',S} \text{ for data sorts } \} \\ & g_{USIG',S}(h_S(d)) \quad . \end{array}$$

For data sorts $u \in U_{USIG}$, the commutativity can be shown as follows:

$$
\begin{aligned}
& ((\mathcal{F} \circ \mathcal{U})(h))_S(g_{USIG,S}(u)) \\
= \quad & \{ \text{ let } f \in F \text{ such that } \sigma(f) = u \ \} \\
& ((\mathcal{F} \circ \mathcal{U})(h))_S(\overline{\sigma(\mathcal{U}_{OP} f)}) \\
= \quad & \{ \text{ by definition of } \sigma(\mathcal{U}_{OP}(f)) \ \} \\
& (\mathcal{F} \circ \mathcal{U})(h)_S(\overline{\tau(p_{u,1}) \cdots \tau(p_{u,n})}) \\
= \quad & \{ \ ((\mathcal{F} \circ \mathcal{U})(h))_S = h_S \text{ for data sorts and all } \tau(p_{u,k}) \text{ are data sorts } \} \\
& \overline{h_S(\tau(p_{u,1})) \cdots h_S(\tau(p_{u,n}))} \\
= \quad & \{ \ \sigma(f) = u \text{ and } \mathcal{U}(h) \text{ is a signature morphism } \} \\
& \overline{\sigma(\mathcal{U}(h)_{OP} f)} \\
= \quad & \{ \ h \text{ is a uniform signature morphism } \} \\
& g_{USIG',S}(h_S(u))
\end{aligned}
$$

Proving that the operations in *USIG* are typed correctly is tedious, but mainly follows the steps by which commutativity for the sorts has been shown. Therefore, we will not show the proof here.

The natural transformation $\beta: \mathcal{F} \circ \mathcal{U} \to ID_{\mathbf{USig}}$ we are looking for is a family of arrows $\beta_{USIG}: \mathcal{F}(\mathcal{U}(USIG)) \to USIG$, for every object $USIG \in Obj_{\mathbf{USig}}$. We have shown the existence of an isomorphism $g_{USIG}: USIG \to \mathcal{F}(\mathcal{U}(USIG))$. The inverse of every such a g_{USIG} also is an isomorphism, i.e., $g^{-1}_{USIG}: \mathcal{F}(\mathcal{U}(USIG)) \to USIG$. Now, by taking $\beta_{USIG} = g^{-1}_{USIG}$, for every object $USIG \in Obj_{USIG}$, the commutativity $\beta_{USIG} \circ h = (\mathcal{F} \circ \mathcal{U})(h) \circ \beta_{USIG'}$ can be derived from the above shown commutativity of g_{USIG} and $g_{USIG'}$. This proves the existence of a natural transformation $\beta: \mathcal{F} \circ \mathcal{U} \to ID_{\mathbf{USig}}$ and by combining this with Lemma 3.23 we may conclude that the categories **Sig** and **USig** are equivalent. □

3.4 Uniform Algebras

In the previous section we have been dealing with signatures and uniform signatures and proved the equivalence of the categories **Sig** and **USig**. This equivalence resides on a syntactical level. In this section we will move on to the semantic level and show that there is also a semantic equivalence.

In Section 3.2 we have introduced the category $\mathbf{Alg}(SIG)$, having *SIG*-algebras as objects and *SIG*-algebra homomorphisms as arrows. The next step is to define *uniform algebras* over an arbitrary uniform signature, say *USIG*.

Definition 3.25 (uniform algebra)**.** *Let $USIG = \langle SIG, \prec \rangle$ be a uniform signature with $SIG = (D \cup U, F \cup \Pi)$. Then, A is a* uniform *$USIG$*-algebra *if A is a SIG-algebra, and for all $u \in U$:*

$$\forall x, y \in A_u : (\forall\, 1 \leq k \leq n : \pi_{u,k}(x) = \pi_{u,k}(y)) \implies x = y \tag{3.2}$$

$$\forall (x_i \in A_{\tau(p_{u,i})})_{1 \leq i \leq n} : \exists y \in A_u : \forall\, 1 \leq k \leq n : \pi_{u,k}(y) = x_k \tag{3.3}$$

where $\pi_{u,k} \in \Pi_A$ represents the k-th projection operation over the product carrier set A_u.

Given two (uniform) $USIG$-algebras A and B, a (uniform) $USIG$-algebra homomorphism simply is an algebra homomorphism.

The additional condition (3.2) which we will refer to as the *functionality condition*, for reasons that will later on become clear, requires that projection operations are *jointly injective*, which intuitively means that whenever the projection operations cannot distinguish between any two elements x and y from the product carrier set A_u, then x and y must be equal, for all $u \in U$. Condition (3.3), in the sequel referred to as the *totality condition*, requires that for all $u \in U$, the corresponding product carrier set A_u contains an element for all possible combinations of elements from the projection carrier sets.

Definition 3.26 (Category $\mathbf{UAlg}(USIG)$)**.** *For a given uniform signature $USIG$, $\mathbf{UAlg}(USIG)$ denotes the category having uniform $USIG$-algebras as objects and uniform $USIG$-algebra homomorphisms as arrows.*

Example 3.27. *Let $USIGINT$ be the uniform signature of Example 3.13. A uniform $USIGINT$-algebra A could then be specified as follows.*

- $A_{\mathsf{int}} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$,
- $A_{\mathsf{int}^2} = \{\langle a, b\rangle \mid a, b \in \mathsf{int}\}$, *e.g.,* $\langle 2, 3\rangle \in \mathsf{int}^2$
- $op_{\overline{+}}(\langle a, b\rangle) = a + b$, *e.g.,* $op_{\overline{+}}(\langle 2, 3\rangle) = 5$
- $op_{\overline{0}}(\langle\rangle) = 0$,
- $\pi_{\overline{\sigma(+)},1}(\langle a, b\rangle) = a$, *e.g.,* $\pi_{\overline{\sigma(+)},1}(\langle 2, 3\rangle) = 2$
- $\pi_{\overline{\sigma(+)},2}(\langle a, b\rangle) = b$. *e.g.,* $\pi_{\overline{\sigma(+)},2}(\langle 2, 3\rangle) = 3$

As with algebras of graph structure signatures, we can likewise introduce algebra graphs over a uniform $USIG$-algebras for an arbitrary uniform signature $USIG$. Obviously, the typing then maps nodes and edges to elements of the signature component of $USIG$. We then use $\mathbf{AlgGraph}^+(USIG)$ [$\mathbf{AlgGraph}(USIG)$]

to denote the category having [deterministic] *USIG*-algebra graphs as objects and *USIG*-algebra graph morphisms as arrows. In order to extend Lemma 3.8 to uniform signatures, we need to make sure that the ordering over the projection operation symbols is properly encoded in the algebra graphs. This can be achieved by structuring the labels of the corresponding edges to contain an index. We then have the following result.

Lemma 3.28. *If USIG is a uniform signature, then the categories* **Alg**(*USIG*) *and* **AlgGraph**(*USIG*) *are equivalent.*

In later parts of this chapter we will visualize graphs of which a subgraph represents the algebra graph of some uniform algebra. To introduce our visual notation of uniform algebra graphs, Fig. 3.4 depicts a fragment of the uniform algebra from Example 3.27. The ellipse-shaped nodes represent data nodes. Those nodes are labelled with the actual data value they represent; data values can thus be interpreted as node identities. Product nodes are represented as unlabelled diamonds. The outgoing edges that are labelled with a numerical index identify the actual tuple of data values the product nodes represents. The outgoing edges labelled with operation symbols identify the data value to which the product node is mapped by that operation in the original uniform algebra.

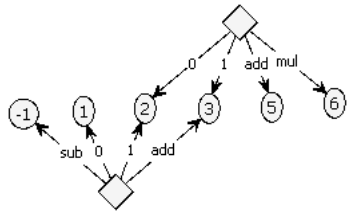


Figure 3.4: Visualizing part of the algebra graph corresponding to the uniform algebra from Example 3.27.

3.4.1 Flattening Arbitrary Algebras

The flattening of signatures has its counterpart on algebras, which then defines the semantics of the freshly introduced functional and projection operation symbols. For every operation op_o in the original *SIG*-algebra, we introduce a fresh operation $op_{\overline{o}}$ for which the semantics are practically the same, but both have different typing, i.e., $op_o \colon A_{\sigma(o)} \to A_{\tau(o)}$ whereas $op_{\overline{o}} \colon A_{\overline{\sigma(o)}} \to A_{\tau(o)}$. Flattening on algebras will be denoted with the operator $\mathcal{F}^{\mathcal{A}}$.

Definition 3.29. *Let $A = (S_A, OP_A)$ be a SIG-algebra for some signature $SIG = (S, OP)$. Then, $\mathcal{F}^{\mathcal{A}}(A) = (D_A \cup U_A, F_A \cup \Pi_A)$ is an $\mathcal{F}(SIG)$-algebra called the* flattened algebra *of A, where*

- $D_A = S_A$;
- $U_A = \{A_{\overline{\sigma(o)}} \mid o\colon s_1 \cdots s_n \to s \in OP\}$,
  with $A_{\overline{\sigma(o)}} = A_{s_1} \times \cdots \times A_{s_n}$ being the carrier set representing the Cartesian product of the sorts in $\sigma(o)$;
- $F_A = \{op_{\overline{o}} \mid o \in OP\}$,
  with $op_{\overline{o}}(\langle a_1, \ldots, a_n\rangle) \mapsto op_o(a_1, \ldots, a_n)$ being the functional operation *corresponding to o;*
- $\Pi_A = \{\pi_{u,k} \mid o\colon s_1 \cdots s_k \cdots s_n \to s \in OP, u \in U : u = \overline{\sigma(o)}\}$,
  with $\pi_{u,k}(\langle a_1, \ldots, a_k, \ldots, a_n\rangle) \mapsto a_k$, for $a_i \in A_{s_i}$, being the k-th projection operation *corresponding to the sort A_u.*

Flattening a *SIG*-algebra homomorphism $h\colon A \to A'$ requires to specify how to map the carrier sets of the flattened algebra $\mathcal{F}^{\mathcal{A}}(A)$.

Definition 3.30. *Given two SIG-algebras A and A' for some arbitrary signature SIG and a SIG-algebra homomorphism $h\colon A \to A'$, let $USIG = \mathcal{F}(SIG) = \langle (D \cup U, F \cup \Pi), \prec\rangle$ be the corresponding uniform signature and $\mathcal{F}^{\mathcal{A}}(A) = FA$ be the uniform USIG-algebra corresponding to A. The uniform USIG-algebra homomorphism $\mathcal{F}^{\mathcal{A}}(h)$ is a family of mappings $(\mathcal{F}^{\mathcal{A}}(h)_s)_{s \in (D \cup U)}$ such that:*

- *for $s \in D$ we have: $\mathcal{F}^{\mathcal{A}}(h)_s = h_s$,*
- *for $s \in U$ with $s = A_{\overline{\sigma(o)}}$ for some $o \in OP$ such that $\sigma(o) = s_1 \cdots s_n$ we have: $\mathcal{F}^{\mathcal{A}}(h)_s = ((h_{s_1} \circ \pi_{s,1}) \otimes \cdots \otimes (h_{s_n} \circ \pi_{s,n}))$.*

Intuitively, the original sorts $s \in D$ are mapped using h; the freshly introduced product sorts $s \in U$ are mapped to a sort that is constructed by the mapping of the component-sorts under h. The individual components are fetched from the product elements through the projection operations, after which they are mapped to their images using their respective h-mapping. The $\otimes$-operator ensures that the mappings of the individual components are combined properly.

We have shown how the operator $\mathcal{F}^{\mathcal{A}}$ maps arbitrary *SIG*-algebras to uniform $\mathcal{F}(SIG)$-algebras. Now, we will show that $\mathcal{F}^{\mathcal{A}}$ actually is a functor from the category $\mathbf{Alg}(SIG)$ to the category $\mathbf{UAlg}(USIG)$.

Lemma 3.31. *$\mathcal{F}^{\mathcal{A}}$ is a functor from the category $\mathbf{Alg}(SIG)$ to the category $\mathbf{UAlg}(USIG)$.*

Proof sketch. Likewise for $\mathcal{F}$, we first have to prove that $\mathcal{F}^{\mathcal{A}}$ actually produces uniform algebra morphisms, after which the proof continues with showing that $\mathcal{F}^{\mathcal{A}}$ commutes with identity morphisms and preserves morphism composition. The full proof is included in Section B.5. □

3.4.2 Unflattening Uniform Algebras

When applying the unflattening process on a uniform algebra A, we have to ensure that the semantics of the functional operations is preserved. For every functional operation $op_f \in F_A$, the set $\Pi_f \subseteq \Pi_A$ provides the parameter sorts of the corresponding operation in the unflattened algebra; the order of the parameter sorts of op_f is determined by the total ordering of the projection functions in Π_f. The unflattening procedure on uniform algebras will be denoted with the operator $\mathcal{U}^{\mathcal{A}}$.

The semantics of the operations in the unflattened algebra are equivalent to the operations of the uniform algebra they originate from. That is, elements from product sorts are decomposed in their constituent parts, taken from the flat carrier sets, by means of the corresponding projection operations, and then mapped to exactly the same value from the target sort.

Definition 3.32. *Let* $SIG = \langle (D \cup U, F \cup \Pi) \prec \rangle$ *be a uniform signature and* $A = (D_A \cup U_A, F_A, \Pi_A)$ *be a uniform* SIG*-algebra. Furthermore, let* $USIG = \mathcal{U}(SIG) = (D, OP)$ *be the unflattened signature of* SIG*. The algebra* $\mathcal{U}^{\mathcal{A}}(A) = (S_A, OP_A)$ *is called the* unflattened algebra *of* A*, where*

- $S_A = D_A$*;*
- $OP_A = \{op_{oper(f)} \mid f\colon u \to d \in F\}$*,*
  with $op_{oper(f)}(a_1, \ldots, a_n) \mapsto op_f(a)$ *where* a *is such that* $\pi_{u,k}(a) = a_k$ *for all* $1 \leq k \leq n$*.*

The well-definedness of the operations in unflattened algebras is guaranteed by the functionality and totality condition from Def. 3.25. The unflattening of algebra homomorphisms basically only remembers the mapping of the flat carrier sets. That is, for a given SIG-algebra homomorphism $g\colon A \to A'$, the corresponding $\mathcal{U}(SIG)$-algebra homomorphism $\mathcal{U}^{\mathcal{A}}(g)$ is defined as follows:

$$\mathcal{U}^{\mathcal{A}}(g) \quad = \quad g\restriction_{D_A} \ .$$

Likewise we have shown that $\mathcal{F}^{\mathcal{A}}$ is a functor from $\mathbf{Alg}(SIG)$ to $\mathbf{UAlg}(USIG)$, the following result states that $\mathcal{U}^{\mathcal{A}}$ is a functor in the reverse direction, i.e., $\mathcal{U}^{\mathcal{A}}\colon \mathbf{UAlg}(USIG) \to \mathbf{Alg}(SIG)$.

Lemma 3.33. *$\mathcal{U}^{\mathcal{A}}$ is a functor from the category* $\mathbf{UAlg}(USIG)$ *to the category* $\mathbf{Alg}(SIG)$.

3.4.3 Composing Algebra Flattening and Unflattening

In the previous sections we have shown $\mathcal{F}^{\mathcal{A}}$ to be a functor from the category $\mathbf{Alg}(SIG)$ to the category $\mathbf{UAlg}(USIG)$, and $\mathcal{U}^{\mathcal{A}}$ to be a functor in the reverse direction. In this section we will show that, based on $\mathcal{F}^{\mathcal{A}}$ and $\mathcal{U}^{\mathcal{A}}$, those categories are equivalent.

Theorem 3.34 ($\mathbf{Alg}(SIG) \cong \mathbf{UAlg}(USIG)$)**.** *Let SIG be an arbitrary signature and $USIG = \mathcal{F}(SIG)$ be the corresponding flattened signature. Then, the categories* $\mathbf{Alg}(SIG)$ *and* $\mathbf{UAlg}(USIG)$ *are equivalent.*

Flattening a SIG-algebra A preserves all the semantics of A but represents those semantics in a different form, namely by introducing product carrier sets and functional and projection operations. When unflattening a uniform algebra, say A', that was obtained by flattening another algebra, say A, we will show that the unflattened algebra $\mathcal{U}^{\mathcal{A}}(A')$ is identical to the original algebra A.

Lemma 3.35. *Let SIG be an arbitrary signature. Then, $(\mathcal{U}^{\mathcal{A}} \circ \mathcal{F}^{\mathcal{A}})(B) = B$, for any SIG-algebra B.*

In the reverse direction we are not able to reconstruct exactly the same uniform algebra, simply because their signatures are, in general, not equal but isomorphic (recall Lemma 3.24). Even in the case both algebras would be based on the same signature, it is nevertheless impossible to reconstruct the elements in the product carrier sets, since the signature does not constrain the structure of those elements; a uniform signature only requires that the elements of the product carrier sets are projected onto elements of data carrier sets such that the functionality condition is satisfied. When flattening an unflattened uniform algebra the product carrier sets are Cartesian products of data carrier sets contained in that algebra. Nevertheless, we can prove the existence of an isomorphism from any uniform $USIG$-algebra B to $\mathcal{F}^{\mathcal{A}}(\mathcal{U}^{\mathcal{A}}(B))$, for an arbitrary uniform signature $USIG$.

Lemma 3.36. *Let USIG be an arbitrary uniform signature. Then, for any uniform USIG-algebra B, there exists an isomorphism $g^{\mathcal{A}}_B \colon B \to \mathcal{F}^{\mathcal{A}}(\mathcal{U}^{\mathcal{A}}(B))$.*

We can now prove that the categories $\mathbf{Alg}(SIG)$ and $\mathbf{UAlg}(USIG)$ are equivalent, for an arbitrary signature SIG.

Proof of Theorem 3.34. In Lemma 3.35 we have shown that $(\mathcal{U}^{\mathcal{A}} \circ \mathcal{F}^{\mathcal{A}})(B) = B$ for some arbitrary *SIG*-algebra B. Based on that result, we may take the natural transformation $\alpha\colon (\mathcal{U}^{\mathcal{A}} \circ \mathcal{F}^{\mathcal{A}}) \to ID_{\mathbf{Alg}(SIG)}$ as the family of identity arrows, i.e., $\alpha_B = id_B$ for all objects $B \in Obj_{\mathbf{Alg}(SIG)}$. Let $f\colon B \to B'$ be an arbitrary arrow in $\mathbf{Alg}(SIG)$. For $\alpha_B = id_B$ and $\alpha_{B'} = id_{B'}$ it is easy to show that the requirement for α to be a natural transformation holds:

$$\begin{aligned} \alpha_{B'} \circ (\mathcal{U}^{\mathcal{A}} \circ \mathcal{F}^{\mathcal{A}})(f) &= id_{B'} \circ ID_{\mathbf{Alg}(SIG)}(f) \\ &= id_{B'} \circ f \\ &= f \circ id_B \\ &= ID_{\mathbf{Alg}(SIG)}(f) \circ id_B \\ &= ID_{\mathbf{Alg}(SIG)}(f) \circ \alpha_B \ . \end{aligned}$$

For the reverse direction we have show the existence of a natural transformation $\beta\colon (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}}) \to ID_{\mathbf{UAlg}(USIG)}$. Let *USIG* be an arbitrary uniform signature and $g^{\mathcal{A}}_B\colon B \to \mathcal{F}^{\mathcal{A}}(\mathcal{U}^{\mathcal{A}})(B)$ be the isomorphism for which we have shown the existence and construction in Lemma 3.36, for an arbitrary *USIG*-algebra B, and $h\colon B \to B'$ be a *USIG*-algebra homomorphism. We will first show that $g^{\mathcal{A}}$ commutes with h and from thereon proof the existence of a proper natural transformation β.

Let $USIG' = \mathcal{F}(\mathcal{U}(USIG)) = \langle (D' \cup U', F' \cup \Pi'), \prec' \rangle$. Then, for all operation symbols $f'\colon u' \to d' \in (F' \cup \Pi')$ we have to show that the following condition is satisfied

$$(\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h) \circ g^{\mathcal{A}}_B \ = \ g^{\mathcal{A}}_{B'} \circ h \ .$$

For all sorts $s \in D$ and all $a \in A_s$, the commutativity can be shown as follows:

$$\begin{aligned} & (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h)_s(g^{\mathcal{A}}_{B,s}(a)) \\ = \ & \{\ g^{\mathcal{A}}_{B,s} = id_{B,s} \text{ for all sorts } s \in D\ \} \\ & (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h)_s(a) \\ = \ & \{\ \mathcal{F}^{\mathcal{A}} \text{ and } \mathcal{U}^{\mathcal{A}} \text{ both preserve elements of data sorts}\ \} \\ & h_s(a) \\ = \ & \{\ g^{\mathcal{A}}_{B',s} = id_{B',s} \text{ for all sorts } s \in D\ \} \\ & g^{\mathcal{A}}_{B',s}(h_s(a)) \ . \end{aligned}$$

For all sorts $s \in U$ and all $a \in A_s$, let $s_k = \tau(p_{s,k})$ for $1 \leq k \leq n$ (note that $s_k \in D$)) and $\langle a_1, \ldots, a_n \rangle = g^{\mathcal{A}}_{B,s}(a)$ with $a_k \in A_{s_k}$. The commutativity can then be shown as follows:

$$
\begin{aligned}
& (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h)_s(g^{\mathcal{A}}_{B,s}(a)) \\
= \quad & \{ \text{ by letting } \langle a_1, \ldots, a_n \rangle = g^{\mathcal{A}}_{B,s}(a) \ \} \\
& (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h)_s(\langle a_1, \ldots, a_n \rangle) \\
= \quad & \{ \ (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h)_s = h_s \text{ for all sorts } s \in D \ \} \\
& \langle h_{s_1}(a_1), \ldots, h_{s_n}(a_n) \rangle \\
= \quad & \{ \ h \text{ is a uniform } \mathit{USIG}\text{-algebra homomorphism } \} \\
& g^{\mathcal{A}}_{B',s}(h_s(a)) \ .
\end{aligned}
$$

Next, we have to prove that the semantics of the operations in $g^{\mathcal{A}}(B')$ preserve the semantics of those in $g^{\mathcal{A}}(B)$. This proof is tedious but is based on the steps by which commutativity for the data and product carrier sets has been shown. Therefore, we will not show the proof here.

Since $g^{\mathcal{A}}_B \colon B \to (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(B)$ is an isomorphism for an arbitrary USIG-algebra B, its inverse ${g^{\mathcal{A}}_B}^{-1}$ is an isomorphism from $(\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(B)$ to B. Now, by taking $\beta_{(\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(B)} = {g^{\mathcal{A}}_B}^{-1}$, for all objects $B \in \mathit{Obj}_{\mathbf{UAlg}(\mathit{USIG})}$, the commutativity $\beta_{(\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(B)} \circ h = (\mathcal{F}^{\mathcal{A}} \circ \mathcal{U}^{\mathcal{A}})(h) \circ \beta_{B'}$ follows immediately from the above shown commutativity of $g^{\mathcal{A}}_B$ and $g^{\mathcal{A}}_{B'}$. This proves the existence of a natural transformation $\beta \colon (\mathcal{U}^{\mathcal{A}} \circ \mathcal{F}^{\mathcal{A}}) \to \mathit{ID}_{\mathbf{UAlg}(\mathit{USIG})}$, and combining this with the result from Lemma 3.35 we may conclude that $\mathbf{Alg}(\mathit{SIG})$ and $\mathbf{UAlg}(\mathit{USIG})$ are equivalent when $\mathit{USIG} = \mathcal{F}(\mathit{SIG})$ or $\mathit{SIG} = \mathcal{U}(\mathit{USIG})$. □

3.5 Uniform Attributed Graphs

Attributed graphs are graphs in which nodes (and possibly also edges) can be assigned data values as so-called *attributes*. In this section, we start by introducing a small example which will be used throughout the remainder of this chapter. Then, some usual approaches to modelling attributed graphs will be discussed. After that, we will explain how attributed graphs can be modelled based on uniform algebras and we show the close relation to the other approaches.

3.5.1 Example: Vectors

The concepts and techniques used for specifying attributed graphs and their transformation are explained through an example, in which a Vector is modelled that consists of zero or more Cells which may contain an Object. A class-diagram for this example is shown in Fig. 3.5. All instances of the Vector-class have two attributes, namely capacity and size, both of type int. All Cell objects have a single index-attribute of type int. In this diagram we have not included operations for any of the classes; they will be specified as graph productions. In this way, we basically abstract from the place where to declare and implement specific operations, i.e., we make them global. In Fig. 3.6 an example configuration is shown in which the Vector has size = 2 and capacity = 5.

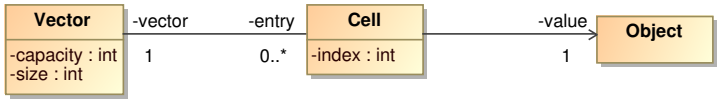


Figure 3.5: Class diagram representing the structure of the Vector example.

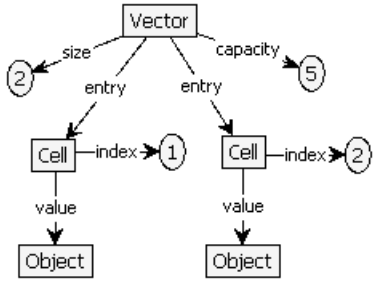


Figure 3.6: Graphical representation of a Vector containing two Objects.

In this example we consider the following three Vector operations:

insert: given an Object O, insert O in the first available Cell of the Vector. This operation is only allowed when the size of the Vector is smaller than its capacity and O is not yet included in the Vector;

delete: delete the Object contained in the last filled Cell of the Vector. The deleted object will still be available for further processing, i.e., the object

will not be deleted itself. This operation is only allowed when the size of the Vector is at least one;

doubleCap: given an Vector with capacity n, update its capacity to $2n$.

In Section 3.5.3 we will come back to this example and then give the graph productions specifying the above operations.

3.5.2 Equivalent Categories of Attributed Graphs

In the literature, several approaches for modelling attributed graphs have been proposed of which some have been shown to be closely related. Ehrig [62], for example, relates two approaches on a categorical level. The first approach, originally introduced by Löwe et al. [135], considers attributed graphs as algebras of a specific algebraic signature $ASIG = (GSIG, ASIG)$ being a graph structure signature $GSIG$ extended with a data type signature $DSIG$. They have introduced a category $\mathbf{Alg}(ASIG)$ having $ASIG$-algebras as objects and corresponding $ASIG$-algebra homomorphisms as arrows. The second approach, originally introduced by Heckel et al. [100], models attributed graphs as pairs consisting of a graph and a algebra of a suitable data type signature $DSIG$ such that the data values from the algebra are included as nodes in the graph. The category having such attributed graphs as objects and corresponding attributed graph morphisms (being pairs of graph morphisms and algebra homomorphisms) as arrows will here be denoted $\mathbf{AttrGraph}(DSIG)$. Ehrig et al. have shown these two categories to be isomorphic [62, 66].

The category $\mathbf{AttrGraph}(DSIG)$, in the sequel referred to as the *standard model* for attributed graphs, has been investigated in more detail by Ehrig et al. [65]. The authors have developed fundamental theory for the specification of *typed attributed graphs* and their transformation. Their notion of attributed graphs supports both node and edge attribution. We will include some of their definitions in which we leave out edge attributes and typing.

In this section we introduce our notion of attributed graphs as captured by the category that will be denoted $\mathbf{UAttrGraph}(USIG)$. One of the main results of this chapter is that $\mathbf{AttrGraph}(DSIG)$ and $\mathbf{UAttrGraph}(USIG)$ are equivalent categories when $DSIG = \mathcal{U}(USIG)$. This means that all transformations that can be performed in the former category have corresponding transformations in the latter, and vice versa, since transformations are described by pushouts, which are a special class of co-limits.

As discussed in Chapter 2, the literature on graph transformations clearly distinguishes between *simple graphs* and *multigraphs*, for which we introduced

the categories $\mathbf{SGraph_T}$ and **Graph**, respectively. The theory as developed by Ehrig et al. [65] is based on graph transformations on multigraphs using the DPO approach, whereas the Groove approach performs graph transformations on simple graphs using the SPO approach. Graph transformations on non-attributed graphs in the former setting can be simulated in the latter, as we have shown in other work [22]. By first considering uniform attributed graphs as labelled multigraphs, we will prove the equivalence of the categories $\mathbf{AttrGraph}(SIG)$ and $\mathbf{UAttrGraph}(USIG)$ both 'sharing' the category **Graph**. In the remaining sections, uniform attributed graphs are then build in a different framework, namely using the category $\mathbf{SGraph_T}$ of simple graphs with corresponding morphisms as its basis.

The following definition of *N-graphs* is based on the *E-graphs* as introduced in the work of Ehrig et al. [65], except that we have omitted edge attributes, i.e., N-graphs only support node-attribution.

Definition 3.37 (N-graph)**.** *An* N-graph $G = (N_R, N_A, E_R, E_A)$ *consists of sets*

- N_R *and* N_A *being disjoint sets of* regular graph nodes *and* data nodes*, respectively,*
- E_R *and* E_A *being disjoint sets of* regular graph edges *and* attribute edges*, respectively,*

together with the functions (see Fig. 3.7)

- $\mathsf{src}_R \colon E_R \to N_R$ *and* $\mathsf{src}_A \colon E_A \to N_R$*,*
- $\mathsf{tgt}_R \colon E_R \to N_R$ *and* $\mathsf{tgt}_A \colon E_A \to N_A$*,*
- $\mathsf{lab}_i \colon E_i \to Lab$*, for* $i \in \{R, A\}$*.*

Given two N-graphs G *and* G' *an N-graph morphism* $f \colon G \to G'$ *is a tuple* $(f_{N_R}, f_{N_A}, f_{E_R}, f_{E_A})$ *with* $f_{N_i} \colon N_i^G \to N_i^{G'}$ *and* $f_{E_j} \colon E_j^G \to E_j^{G'}$ *for* $i, j \in \{R, A\}$ *such that* f *commutes with all source, label and target functions.*

Attributed graphs, in this approach, consist of N-graphs combined with a suitable *SIG*-algebra, for some signature *SIG*, such that the elements of the data sorts are nodes in the graph, as defined below.

Definition 3.38 (attributed graph)**.** *Let* $DSIG = (S_D, OP_D)$ *be a signature. An* attributed graph $AG = (G, A)$ *consists of an N-graph* G *and a* $DSIG$*-algebra* A *with* $\biguplus_{s \in S'} A_s \subseteq N_G$*.*

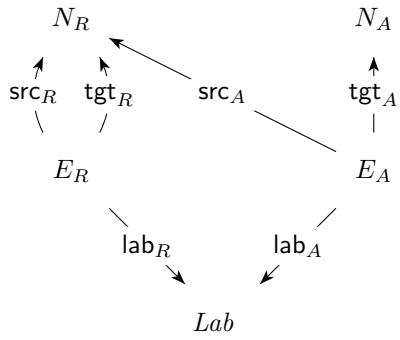


Figure 3.7: Set of nodes and edges of N-graphs, and their source, label, and target functions.

Let AG and AG' be two DSIG-attributed graphs. Then, an attributed graph morphism *$f\colon AG \to AG'$ is a pair $f = (f_G, f_A)$ such that $f_G\colon G \to G'$ is an N-graph morphism and $f_A\colon A \to A'$ is an algebra homomorphism such that the diagram in Fig. 3.8 commutes for all $s \in D_A$ where the vertical arrows are inclusions.*

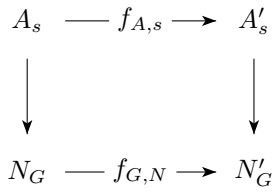


Figure 3.8: Commuting graph morphisms f_G and algebra homomorphism f_A for attributed graphs.

In the following we will define what we call *uniform attributed graphs* based on uniform *USIG*-algebras for a given uniform signature *USIG*. The graph part of uniform attributed graphs are so-called *U-graphs*.

Definition 3.39 (U-graph)**.** *A* U-graph *G is a graph such that the N_G and E_G are partitioned as follows:*

$$\begin{aligned} N_G &= N_R \cup N_D \cup N_U \\ E_G &= E_R \cup E_A \cup E_F \cup E_\Pi \ , \end{aligned}$$

with N_R, N_D, and N_U being disjoint sets of regular graph nodes, data nodes, *and* product nodes, *respectively, and E_R, E_A, E_F, and E_Π being disjoint sets of*

regular graph edges, attribute edges, functional operation edges *and* projection operation edges, *respectively. For the different edge types, the source, target, and label functions satisfy the following typing (see Fig. 3.9):*

- $\mathsf{src}_R\colon E_R \to N_R$ *and* $\mathsf{tgt}_R\colon E_R \to N_R$,
- $\mathsf{src}_A\colon E_A \to N_R$ *and* $\mathsf{tgt}_A\colon E_A \to N_D$,
- $\mathsf{src}_F\colon E_F \to N_U$ *and* $\mathsf{tgt}_F\colon E_F \to N_D$,
- $\mathsf{src}_\Pi\colon E_\Pi \to N_U$ *and* $\mathsf{tgt}_\Pi\colon E_\Pi \to N_D$,
- $\mathsf{lab}_i\colon E_i \to \mathit{Lab}$, *for* $i \in \{R, A\}$,
- $\mathsf{lab}_F\colon E_F \to \mathcal{O}$,
- $\mathsf{lab}_\Pi\colon E_\Pi \to \mathcal{P}$.

Given two U-graphs G and G', a U-graph morphism f is a graph morphism that respects the partitioning.

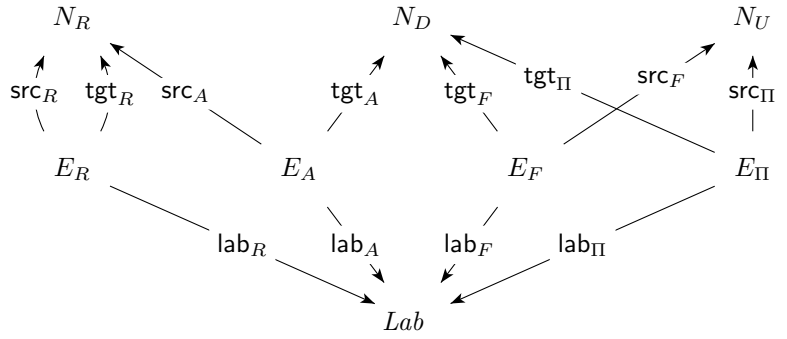


Figure 3.9: Set of nodes and edges of U-graphs, and their source, label, and target functions.

A uniform attributed graph, then, is a U-graph G combined with a *USIG*-algebra graph A, for some suitable uniform signature *USIG*, such that A is a subgraph of G.

Definition 3.40 (uniform attributed graph)**.** *Let USIG be a uniform signature. A* uniform *USIG*-attributed graph (G, A) *is a U-graph with A being a USIG-algebra graph and $A \subseteq G$, such that $N_A = N_{G,D} \cup N_{G,U}$ and $E_A = E_F \cup E_\Pi$.*

Given two uniform attributed graphs $\langle G, A\rangle$ and $\langle G', A'\rangle$, a uniform attributed graph morphism $f\colon \langle G, A\rangle \to \langle G', A'\rangle$ is a U-graph morphism, such that $f{\restriction}A$ is a USIG-algebra graph morphism.

By combining Def. 3.40 with Def. 3.39, one can easily verify that for a uniform attributed graph $\langle G, A\rangle$ it holds that A is a *full subgraph* of G. This can be formalized as follows.

Proposition 3.41. *Let USIG be an arbitrary uniform signature and (A, t) be a uniform USIG-algebra graph. Then, a pair $\langle G, A\rangle$ is a uniform attributed graph iff there exists an embedding morphism $\eta_G\colon A \to G$ and $\nexists e \in (E_G \setminus E_A) : t(\mathsf{src}(e)) \in D_{USIG} \vee t(\mathsf{tgt}(e)) \in U_{USIG}$.*

In the sequel, when stating that G is a uniform *SIG*-attributed graph, we implicitly assume a tuple $\langle G, A\rangle$ where A is the *SIG*-algebra graph included in G; likewise, a uniform attributed graph morphism $f\colon \langle G, A\rangle \to \langle G', A'\rangle$ will be denoted as $f\colon G \to G'$, accompanied by the statement that it is a uniform *SIG*-attributed graph morphism. If the signature is not relevant or clear from the context it will be omitted.

Proposition 3.42. *If $f\colon G \to G'$ is a uniform attributed graph morphism, then the diagram (1) in Fig. 3.10 is a pullback.*

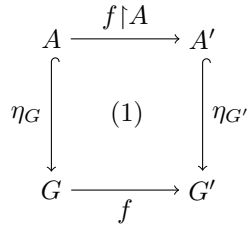


Figure 3.10: Uniform attributed graph morphism.

In order to make the embedding of the algebra graph in the uniform attributed graph more explicit, we introduce the notion of a *graph embedding*, being a pair (G, G^-) of graphs such the latter is a subgraph of the former.

Definition 3.43 (graph embedding, reflection)**.** *Let* $\mathbf{G}$ *be a sub-category of* **Graph**. *A* graph embedding *over* $\mathbf{G}$ *is a pair (G, G^-) such that $G^- \in Obj_{\mathbf{G}}$ and $G^- \subseteq G \in$* **Graph**. *If G and H are graph embeddings, then a* reflection *from G to H is a graph morphism $h\colon G \to H$ such that for all $n \in N_G$, $h_N(n) \in N_{H^-}$ implies $n \in N_{G^-}$, and for all $e \in E_G$, $h_E(e) \in E_{H^-}$ implies $e \in E_{G^-}$.*

A graph embedding G is said to be glued over *a graph $G^{--} \subseteq G^-$ if for all $e \in (E_G \setminus E_{G^-})$ and incident nodes $n \in \{\mathsf{src}(e), \mathsf{tgt}(e)\}$, $n \in N_{G^-}$ implies $n \in N_{G^{--}}$.*

Intuitively, this means that if an embedding G is glued over a graph G^{--}, nodes in G^{--} are only allowed to be connected to nodes outside G^{-}. In our context, for instance, we do not want to allow product nodes to be used as attributes. That is, our embeddings will be glued over the subgraph of the algebra graph consisting of data nodes only.

Furthermore, we introduce the notion of an *embedding functor* for graph categories. An embedding functor then maps a graph, say G, to a subgraph of G.

Definition 3.44 (embedding functor)**.** *Let* $\mathbf{G}$ *be a sub-category of* **Graph**. *Then a functor* $\mathcal{E}\colon \mathbf{G} \to \mathbf{Graph}$ *is called an* embedding functor *if*

- $\mathcal{E}(G) \subseteq G$ *for all* $\mathbf{G}$*-graphs* G*, and*
- $\mathcal{E}(f) = f{\restriction}\mathcal{E}(G)$ *for all* $\mathbf{G}$*-graph morphisms* $f\colon G \to H$.

Now, we can introduce the category $\mathbf{REmb}(\mathbf{G})$ having graph embeddings as objects and reflections (as from Def. 3.43) as arrows. Similarly, we can introduce $\mathbf{REmb}(\mathcal{E})$ with $\mathcal{E}\colon \mathbf{G} \to \mathbf{Graph}$ an embedding morphism, to denote the full sub-category of $\mathbf{REmb}(\mathbf{G})$ consisting of embeddings (G, G^{-}) glued over $\mathcal{E}(G^{-})$. By now letting *USIG* be an arbitrary uniform signature and $\mathcal{E}_{USIG}\colon \mathbf{AlgGraph}(USIG) \to \mathbf{dGraph}$ be the embedding functor mapping every *USIG*-algebra graph (G, t) to the discrete sub-graph consisting of the nodes $\{n \in N_G \mid t(n) \in D_{USIG}\}$, our category of uniform attributed graphs with embedded deterministic algebra graphs can be defined as follows:

$$\mathbf{UAttrGraph}(USIG) \quad = \quad \mathbf{REmb}(\mathcal{E}_{USIG}) \ . \tag{3.4}$$

Attributed Graphs versus Uniform Attributed Graphs

Starting from the category $\mathbf{AttrGraph}(SIG)$, we will prove its equivalence to the category $\mathbf{UAttrGraph}(USIG)$, given that $SIG = \mathcal{U}(USIG)$, in several steps. Fig. 3.11 gives an overview of the main categories we have introduced in this chapter so far. It includes the functors $\mathcal{F}$, $\mathcal{U}$, $\mathcal{F}^{\mathcal{A}}$, and $\mathcal{U}^{\mathcal{A}}$, which are depicted as solid arrows. The dashed arrows denote dependencies between different categories. For example, the category $\mathbf{Alg}(SIG)$ depends on the category **Sig** since it is based on a single object from that category. The category $\mathbf{AttrGraph}(SIG)$ likewise depends on the category $\mathbf{Alg}(SIG)$; the dependency with **Graph** exists because every object in $\mathbf{AttrGraph}(SIG)$ contains an N-graph component which is an object in **Graph**. The category $\mathbf{UAttrGraph}(USIG)$ has analogous dependencies with $\mathbf{UAlg}(USIG)$ and

Graph, although the way the objects in **UAttrGraph**(*USIG*) are constructed from a graph and a uniform *USIG*-algebra differs from the way this is done for the objects in the category **AttrGraph**(*SIG*). Nevertheless, the similarity between both ways of constructing (uniform) attributed graphs and the equivalence of the categories **Alg**(*SIG*) and **UAlg**(*USIG*) strongly suggests the existence of a close relation between the two categories of attributed graphs. We will now prove that this relation is an equivalence as well.

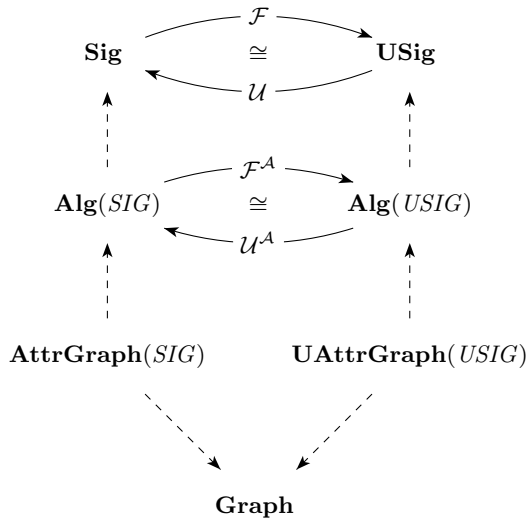


Figure 3.11: Overview of the different categories.

Theorem 3.45. *Let SIG be an arbitrary signature, and USIG be an arbitrary uniform signature. Then, the category* **AttrGraph**(*SIG*) *is equivalent to the category* **UAttrGraph**(*USIG*) *when* $USIG = \mathcal{F}(SIG)$*:*

$$\mathbf{AttrGraph}(SIG) \cong \mathbf{UAttrGraph}(USIG) \ .$$

The equivalence between the categories will be proven in a number of steps. First, we defined the following functor.

Definition 3.46. *Let* $\mathcal{D}\colon \mathbf{C} \to \mathbf{dGraph}$ *be a functor to discrete graphs. The category of* $\mathcal{D}$*-attributed graphs* **AttrGraph**($\mathcal{D}$) *is defined by*

- *objects* (G, C) *where* G *is a graph and* C *is an object of* $\mathbf{C}$*, such that* $\mathcal{D}(C) \subseteq G$*;*
- *arrows* $f\colon (G_1, C_1) \to (G_2, C_2)$ *such that* $f = (f_G, f_C)$ *where* f_G *is a graph morphism and* f_C *is an arrow from* $\mathbf{C}$*, such that* $\mathcal{D}(f_C) = f_G{\restriction}\mathcal{D}(dom(g))$*, i.e.,* f_G *and* f_C *agree upon the discrete graph.*

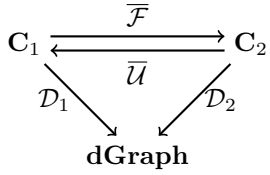
Different instantiations of the category $\mathbf{C}$ in the above definitions result in different categories of attributed graphs. Some examples of useful functors $\mathcal{D}$ are:

- $\mathcal{A}_{SIG}\colon \mathbf{Alg}(SIG) \to \mathbf{dGraph}$ for an arbitrary signature $SIG = (S, OP)$, mapping every SIG-algebra to the discrete graph, say G, in which $N_G = \bigcup_{s \in S} A_s$;
- $\mathcal{A}_{USIG}\colon \mathbf{Alg}(USIG) \to \mathbf{dGraph}$ for an arbitrary uniform signature $USIG$, mapping every $USIG$-algebra to the discrete graph, say G, in which $N_G = \bigcup_{d \in D} A_d$;
- the functor $\mathcal{E}_{USIG}\colon \mathbf{AlgGraph}(USIG) \to \mathbf{dGraph}$ introduced above, given some uniform signature $USIG$.

Using the above, the standard model of attributed graphs is then given by $\mathbf{AttrGraph}(\mathcal{A}_{SIG})$.

A next ingredient to prove Theorem 3.45 is the notion of *source equivalent* functors.

Definition 3.47. *Two functors* $\mathcal{D}_i\colon \mathbf{C}_i \to \mathbf{dGraph}$*, for* $i = 1, 2$*, are* source equivalent *if there exists functors* $\overline{\mathcal{F}}\colon \mathbf{C}_1 \to \mathbf{C}_2$ *and* $\overline{\mathcal{U}}\colon \mathbf{C}_2 \to \mathbf{C}_1$ *which establish an equivalence between* $\mathbf{C}_1$ *and* $\mathbf{C}_2$ *and such that, moreover, the following diagram of functors commutes:*



Examples of source equivalent functors are $\mathcal{A}_{SIG}$, $\mathcal{A}_{\mathcal{F}(SIG)}$, and $\mathcal{E}_{\mathcal{F}(SIG)}$, for an arbitrary signature SIG. Similarly, the functors $\mathcal{A}_{\mathcal{U}(USIG)}$, $\mathcal{A}_{USIG}$, and $\mathcal{E}_{USIG}$ are source equivalent, for an arbitrary uniform signature $USIG$.

We need source equivalent functors to further close the equivalence gap between the standard model and our notion of attributed graphs. The next theorem will provide a number of steps. It states that replacing the algebra graph parameter in the standard model by a source equivalent one does not change the category.

Theorem 3.48. *If* $\mathcal{D}_i\colon \mathbf{C}_i \to \mathbf{dGraph}$*, for* $i = 1, 2$*, are two source equivalent functors, then* $\mathbf{AttrGraph}(\mathcal{D}_1)$ *and* $\mathbf{AttrGraph}(\mathcal{D}_2)$ *are equivalent categories.*

This is shown by functors between $\mathbf{AttrGraph}(\mathcal{D}_1)$ and $\mathbf{AttrGraph}(\mathcal{D}_2)$ that coincide with $\mathcal{D}_1$ and $\mathcal{D}_2$ on the algebra component and with the identity functor on the graph component.

The equivalence bridge between the two categories of attributed graph is now closed by the following result.

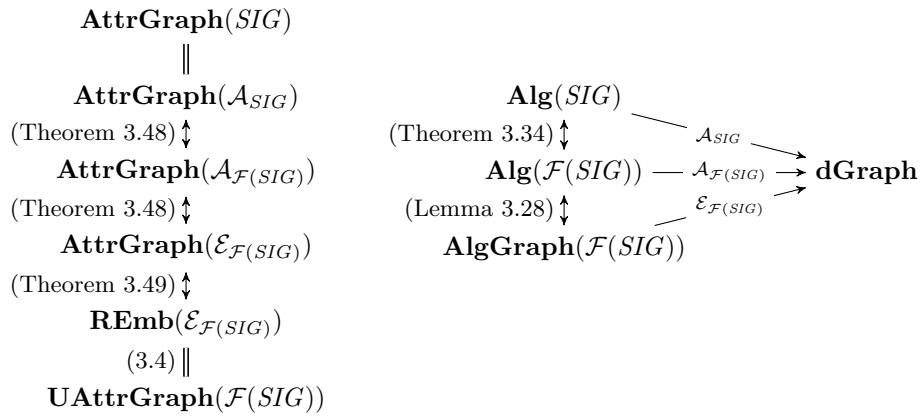
Theorem 3.49. $\mathbf{AttrGraph}(\mathcal{E}_{USIG})$ *and* $\mathbf{UAttrGraph}(USIG)$ *are equivalent categories, for an arbitrary uniform signature* $USIG$*.*

Proof sketch. The intuition behind the proof is that $\mathbf{AttrGraph}(\mathcal{E}_{\mathcal{F}(SIG)})$ represents the category of attributed graphs, as from the standard model, over a uniform signature such that only the data nodes of the flattened algebra are included in the graph, whereas the objects in $\mathbf{UAttrGraph}(\mathcal{F}(SIG))$ represent attributed graphs including the entire (deterministic) algebra graph. They thus only differ in including the remaining algebra graph elements in the graph explicitly. The full proof is included in Appendix B (Section B.8). □

We now have all the ingredients to proof the equivalence between the categories $\mathbf{AttrGraph}(SIG)$ and $\mathbf{UAttrGraph}(USIG)$.

Proof of Theorem 3.45. The equivalence follows from a chain of equivalences sketched in the following diagrams. In the above diagrams, $\leftrightarrow$ denotes equivalence relations between categories and $\to$ denotes a functor. The left chain contains the actual steps of the proof; the diagram on the right presents the justification for applying Theorem 3.48. □

The above steps can be applied in a similar way when, in Theorem 3.45, we would have required that $SIG = \mathcal{U}(USIG)$, although some of the proofs would have been a bit more involved since we then have to rely on isomorphisms instead of identities.



3.5.3 Uniform Attributed Graph Transformations

From this point on, we work with uniform attributed graphs in the simple graph setting. That is, we let **Graph** be the category of simple graphs and build the required categories on top of that. For instance, the category **AlgGraph**(SIG) then represents the category of deterministic simple SIG-algebra graphs.

In this section, we take a closer look at how the transformations of uniform attributed graphs are specified in the Single Pushout approach (recall Section 2.2.1). In the standard approach, attributed graph productions are based on term algebras with a designated set of *variables*. As mentioned before, one of the advantages of our approach is that we do not require to introduce variable names for specifying uniform attributed graph productions. Nevertheless, we need some way to specify that the values of some attributes in such productions can *freely* be assigned a value by an actual matching. We therefore introduce the notion of a *free algebra graph*, which is an algebra graph containing free data nodes, i.e., data nodes that have not been assigned a specific data value. Formally, this means that such data nodes have no incoming edges representing constant operations. A uniform attributed graph production $p\colon L \to R$ is then defined as a usual SPO graph production in which L and R are uniform attributed graphs with embedded free algebra graphs, with the additional constraint that p restricted to the algebra graph of L is an isomorphism.

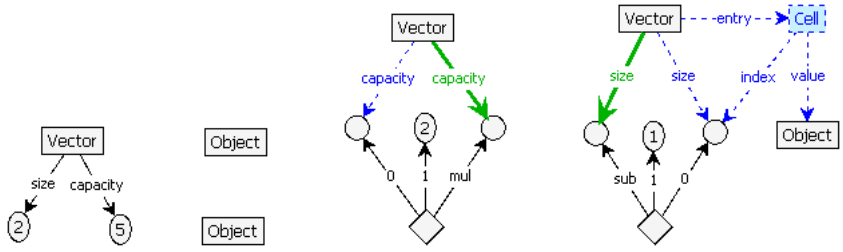
Definition 3.50 (uniform attributed graph production)**.** *Let SIG be a uniform signature, and A and B be free SIG-algebra graphs. Then, a* uniform attributed graph production $\langle L, A\rangle \xrightarrow{p} \langle R, B\rangle$ *consists of uniform attributed graphs* $\langle L, A\rangle$

and $\langle R, B\rangle$ and a (partial) uniform attributed graph morphism p such that $p\restriction A$ is an isomorphism.

In practice, the left-hand-side and the right-hand-side of such graph productions share the (free) algebra graph, i.e., $p\restriction A$ is actually the identity morphism. In uniform attributed graph productions, free data nodes are visualized as unlabelled ellipses. As one would expect, uniform attributed graph production systems now consist of a set of uniform attributed graph productions and a single uniform attributed graph as its start graph.

In the sequel we will show some further examples of uniform attributed graphs and their transformations. For those to be interpreted properly, the reader should be aware of how we deal with a number of (implementation and visualization) issues. In Section 3.7 we elaborate on some of those issues.

Example 3.51. *The uniform attributed graph production system specifying the example from Section 3.5.1 is shown in Fig. 3.12. It contains the three rules, one for each operation, and a start graph G_0 modelling the empty Vector surrounded by three Objects that are not (yet) contained in the Vector. The rule specifying the insert-operation has already be shown in Fig. 3.2.*



(a) start graph G_0 **(b)** doubleCap-operation. **(c)** delete-operation.

Figure 3.12: Uniform attributed graph production system for the Vector example.

To ensure that matchings of uniform attributed graph productions can effectively be computed, we have to put some constraints on the free algebra graphs as they appear in the such productions. Those constraints and the way matchings are computed once those constraints are fulfilled are discussed in Section 3.7. Direct derivations on uniform attributed graphs are then constructed as usually done in the SPO approach.

In Fig. 3.13 we have depicted a direct derivation on attributed graphs. The rule being applied (in the upper row) specifies the doubleCap-operation in the traditional way, i.e., by separately showing the left-hand-side and the right-hand-side. The rule morphism is suggested by the two dimensional placing of the graph elements. The matching of the non-algebra graph elements is trivial. We have not shown the part of the algebra graph included in the rule since for the lower part of the figure it does not have any added value. The result of the direct derivation is that the node representing the integer value 1 that was shared in the host graph G in referenced only from the Cell node in the target graph H. Furthermore, although the integer value 2 was not included in G (since it was not referenced) in appears in H since there it is referenced by the Vector node.

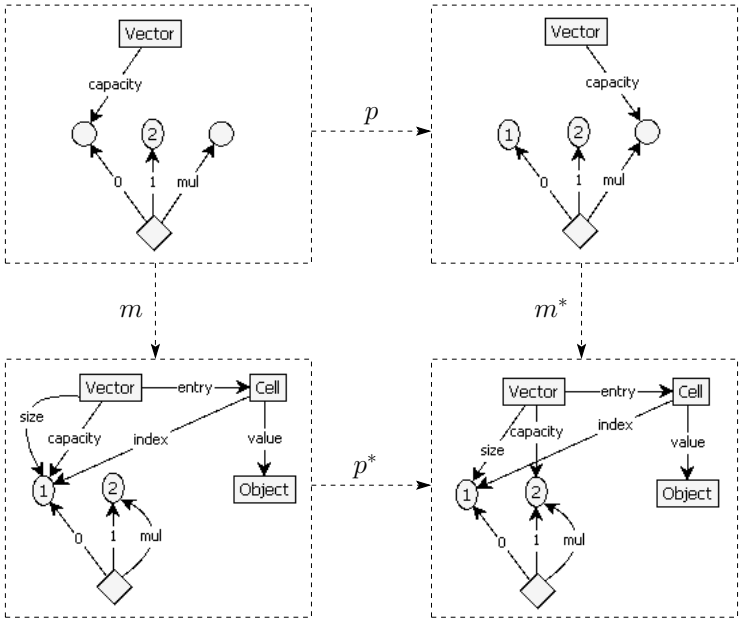


Figure 3.13: Example of a direct derivation of a uniform attributed graph.

Convention. In the sequel we write 'attributed graph (transformation)' whenever we mean 'uniform attributed graph (transformation)', unless explicitly stated differently.

3.6 Abstraction on Uniform Attributed Graphs

A major advantage of our approach to modelling attributed graphs is the straightforward way in which we can specify algebra abstraction for attributed graphs. We introduce abstraction morphisms between attributed graphs. Such an abstraction morphism will leave the regular graph structure unchanged but specify which elements of the algebra graph will be merged. In general, this results in attributed graphs based on non-deterministic algebra graphs, although there are special cases of abstract algebra graphs that are deterministic, e.g., algebra graphs of *point algebras*.

3.6.1 Abstraction Morphisms

Our approach to attributed graph abstraction is based on graph morphisms only. That is, abstractions on attributed graphs are completely specified by attributed graph morphisms. Due to the uniformity of our approach, abstraction morphisms are usual attributed graph morphisms the additional constraint that the diagram described by the attributed graph morphism and the embeddings is a pushout.

Definition 3.52. *Let $h\colon A \to B$ be an arrow in the category* $\mathbf{AlgGraph}^{+}(SIG)$*, with SIG an arbitrary uniform signature. An* attributed graph abstraction α_h *is a family of attributed graph morphisms $\alpha_h^G\colon \langle G, A\rangle \to \langle H, B\rangle$, for all uniform attributed graphs $\langle G, A\rangle$, such that the diagram (1) in Fig. 3.14 is a pushout. The morphism α_h^G is then called an* abstraction morphism.

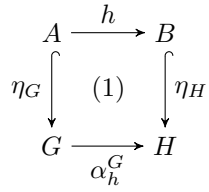


Figure 3.14: Abstraction morphism.

Using this definition, it is straightforward to see that the (non-)injectivity of the algebra graph morphism determines to what degree abstraction is performed. If h is non-injective, some elements of the algebra graph are merged. By Def. 3.6, those nodes must be of the same type, i.e., elements of the same carrier set.

Merging two nodes implies that the number of elements of that type decreases, unless the target domain is still infinite. In specific cases of h, infinite domains might be reduced to finite ones. For non-injective algebra graph morphisms h such that the size of $dom(h)$ is finite, the corresponding abstraction morphisms are said to be *proper* abstraction morphisms. That is to say, all morphisms in α_h *reduces* the size of the algebra graph and therefore also the size of the individual attributed graph. In the case h is an algebra graph isomorphism, α does not decrease the size of the attributed graph which basically means that the level of precision is preserved, or, stated differently, no real abstraction has occurred. Needles to say, we are mainly interested in cases where α is a proper abstraction morphism. The following example depicts a specific non-injective algebra graph morphism h.

Example 3.53. *Let us return to the integer algebra as introduced in Example 3.5. Suppose we depict the deterministic algebra graph, say A, of the uniform algebra that is obtained by flattening the integer algebra. An excerpt of A is shown on the left in Fig. 3.15 including some elements of type* int*, some of type* $\overline{\mathsf{int}\,\mathsf{int}}$*, and some projection operation edges. A possible abstraction, as suggested before, is obtained by mapping all* strictly *positive integers (i.e., excluding 0) to the 'abstract' value* pos*, all strictly negative values to the 'abstract' value* neg*, and 0 to* zero*. The product nodes are mapped to tuples of abstract values component-wise. For instance, the tuple $\langle -2, -1\rangle$ will be mapped to $\langle \mathsf{neg}, \mathsf{neg}\rangle$. The algebra graph $\hat{A}$ on the right in Fig. 3.15 depicts the excerpt from the algebra graph to which the element of A are mapped. When adding the operation edges that represent the addition operation, both algebra graphs still seem deterministic for the excerpt shown. If we, however, would have included the tuples $\langle -2, 1\rangle$, $\langle -2, 2\rangle$, and $\langle -2, 3\rangle$ in the algebra graph A, it is easy to see that the algebra graph $\hat{A}$ becomes non-deterministic, since the single tuple $\langle \mathsf{neg}, \mathsf{pos}\rangle$ to which all three 'concrete' tuples would be mapped, has three outgoing edges for the addition operation.*

The following example will illustrate how attributed graphs based on the algebra graph A from the previous example can be mapped to abstract attributed graphs based on the algebra graph $\hat{A}$.

Example 3.54. *In Fig. 3.16 five attributed graphs are shown. The graphs in the upper row are attributed graphs based on the algebra graph A as from Example 3.53; the lower two graphs are attributed graphs based on the algebra graph $\hat{A}$. It is easy to see that the graph $\langle H_1, \hat{A}\rangle$ is an abstraction on the graph $\langle G_1, A\rangle$. For the graphs $\langle G_2, A\rangle$ and $\langle G_3, A\rangle$, it is easy to verify that, based*

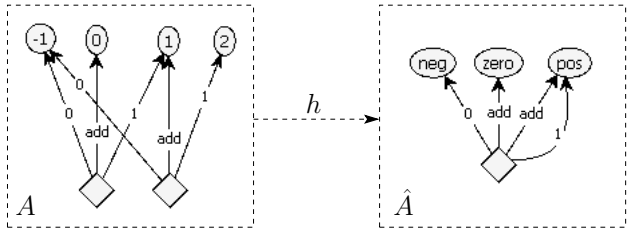


Figure 3.15: Two algebra graphs and an algebra graph morphism.

on the algebra graph morphism $f: A \to \hat{A}$, they are both mapped to the same attributed graph $\langle H_2, \hat{A} \rangle$. Note that there are many more attributed graphs that are abstracted to the same graph $\langle H_2, \hat{A} \rangle$, namely all the attributed graphs with a single A-node having an x-attribute with a value greater than zero.

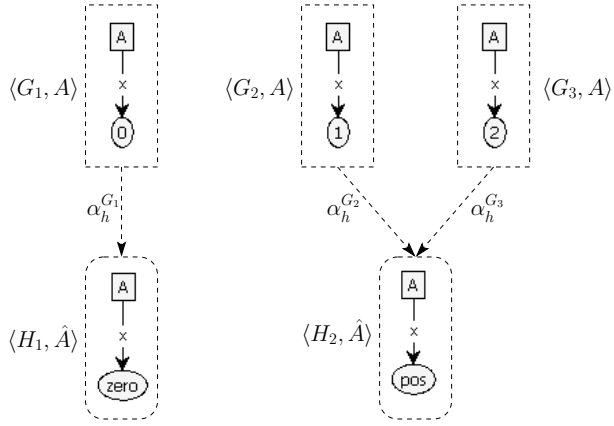


Figure 3.16: Applying abstractions on attributed graphs.

Although in this work we are not interested in the reverse direction, i.e., concretion of attributed graphs, we will shortly discuss how concretion can be defined. For an attributed graph $\langle G, B \rangle$, with B being a *SIG*-algebra graph, and a *SIG*-algebra graph morphism $f: A \to B$, we define the set of attributed graphs $\langle H, A \rangle$ that abstract to $\langle G, B \rangle$ under f as the set of concretizations of $\langle G, B \rangle$ under f; each such $\langle H, A \rangle$ can then said to be an f-concrete attribute graph of $\langle G, B \rangle$.

3.6.2 Concrete versus Abstract Transformations

In this thesis we analyze graph transition systems that are generated by repeatedly applying graph transformations. Based on abstraction morphisms, we can perform graph transformations on abstract attributed graphs. The question then arises, how concrete transformations relate to abstract transformations. For this question to be answered, we take a closer look at how matches to abstract attributed graphs are constructed. Actually, an *abstract match* is the composition of a matching and an abstraction morphism. For example, let $h\colon B \to C$ be an algebra graph morphism and $m\colon \langle L, A\rangle \to \langle G, B\rangle$ be a matching for some attributed graph production $p\colon \langle L, A\rangle \to \langle R, A\rangle$ to an attributed graph $\langle G, B\rangle$. The abstract match of m, denoted $\hat{m}$, is the composition of m and α, i.e., $\hat{m} = \alpha \circ m$.

In the next theorem we show that for every direct derivation $G \overset{p,m}{\Longrightarrow} H$ of an attributed graph production $L \overset{p}{\rightarrowtail} R$ and a matching $m\colon L \to G$ together with an abstraction morphism $\alpha_h^G\colon G \to \hat{G}$ there exists a direct derivation $\hat{G} \overset{p,\hat{m}}{\Longrightarrow} H'$. Moreover, the graph H' reached by this direct derivation is exactly the target graph of the abstraction morphism $\alpha_h^H\colon H \to \hat{H}$, i.e., $H' = \hat{H}$.

This construction has a lot in common with one of the main results for graph transformation systems, namely the *Embedding Theorem* [45, 69]. Roughly speaking, this theorem states sufficient and necessary conditions under which transformations can be extended to larger contexts. These results do not directly apply in our context, simply because we consider a different graph category, which is based on simple graphs. Therefore, we rephrase the embedding theorem and prove it for our specific context.

Theorem 3.55. *Let $p\colon L \to R$ be an attributed graph production, and G be an attributed graph with algebra graph A. Furthermore, let $h\colon A \to \hat{A}$ be an algebra graph morphism. Then, for every direct attributed graph derivation $G \overset{p,m}{\Longrightarrow} H$ there exists a direct attributed graph derivation $\hat{G} \overset{p,\hat{m}}{\Longrightarrow} \hat{H}$, where $\hat{G} = \alpha_h^G(G)$, $\hat{H} = \alpha_h^H(H)$, and $\hat{m}$ is the abstract counterpart of m, i.e., $\hat{m} = \alpha_h^G \circ m$.*

For proving the above theorem we will first prove that, under certain conditions, embeddings are stable under pushout. Therefore, we give a characterization of pushouts in our specific graph category.

Lemma 3.56. *Let G, H_1, and H_2 be graphs, and $f_1\colon G \to H_1$ and $f_2\colon G \to H_2$ be (partial) graph morphisms. Then, the following steps describe the pushout of the span $H_1 \overset{f_1}{\leftarrowtail} G \overset{f_2}{\rightarrowtail} H_2$ constructively.*

- *for $i = 1, 2$ let $\bar{f}_i: G \to \bar{H}_i$ be a total extension of f_i, defined by adding a distinct fresh node n' to H_i and setting $\bar{f}_i(n) = n'$ for each $n \in N_G \setminus dom(f_{N,i})$. Hence, $\bar{H}_i = (\bar{N}_i, \bar{E}_i)$ such that $\bar{N}_i$ extends N_i with the fresh nodes, and $\bar{E}_i$ extends E_i with the fresh edges implied by the totality of $\bar{f}_i$.*
- *Let $\bar{N} = \bar{N}_1 \cup \bar{N}_2$ be the union of the extended node set $\bar{N}_1$ and N_2, and likewise $\bar{E} = \bar{E}_1 \cup \bar{E}_2$. Let $\simeq \subseteq \bar{N} \times \bar{N}$ be the smallest equivalence relation such that $\bar{f}_{N,1}(n) \simeq \bar{f}_{N,2}(n)$ for all $n \in N_G$; and likewise for edges.*
- *Let $W = \{X \in \bar{N}/_{\simeq} \mid X \subseteq N_1 \cup N_2\}$, and for $i = 1, 2$ define $g_{N,i}: N_i \to W$ such that for all $n \in N_i$*

$$g_{N,i} : n \mapsto [n]_{\simeq} \text{ if } [n]_{\simeq} \subseteq N_1 \cup N_2 \ .$$

- *Let $F = \{([n]_{\simeq}, a, [n']_{\simeq}) \mid [(n, a, n')] \in E_1 \cup E_2\}$; moreover, for $i = 1, 2$, define $g_{E,i}: E_i \to F$ such that for all $(n, a, n') \in E_i$*

$$g_{E,i} : (n, a, n') \mapsto ([n]_{\simeq}, a, [n']_{\simeq}) \text{ if } [(n, a, n')]_{\simeq} \subseteq E_1 \cup E_2 \ .$$

- *Let $H = (W, F)$; then $g_i: H_i \to H$ are morphisms for $i = 1, 2$.*

The graph H together with the morphisms g_1 and g_2 then is the pushout, i.e., the span (f_1, f_2) forms the pushout diagram shown in Fig. 3.17.

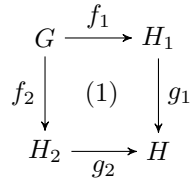


Figure 3.17: Pushout diagram.

The object H constructed in Lemma 3.56 has been proven to be indeed the pushout since it satisfied the required co-limit property [162]. From the above definition we can prove that if G^- is a subgraph of G which is preserved by f_1, i.e., $f_1 \restriction G^-$ is isomorphic, the image of the subgraph G^- is also preserved by the morphism g_2, i.e., $g_2 \circ f_2 \restriction G'$ is an isomorphism as well.

Lemma 3.57. *Given the pushout diagram of Fig. 3.17 and a sub-graph G^- of G. If f_1 preserves G', i.e., $f_1{\restriction}G'$ is isomorphic, then $g_2 \circ f_2{\restriction}G'(G)$ is isomorphic as well.*

Proof. For any node $n \in N_{G^-}$ we have that $n \in dom(f_1)$, due to $f_1{\restriction}G^-$ being isomorphic. The morphism f_2 being total implies $n \in dom(f_2)$. Therefore, it holds that $\bar{f}_1(n) \simeq \bar{f}_2(n)$. The previous facts hold likewise for all edges $e \in E_{G^-}$.

Let $K = f_2(G^-)$. For every two distinct nodes $s, t \in N_K$ with $s' = f_2^{-1}(s)$ and $t' = f_2^{-1}(t)$ we have to prove that s and t are mapped injectively by g_2. Since $f_1{\restriction}G^-$ is isomorphic it holds that $f_1(s) \neq f_1(t)$ and therefore $\bar{f}_1(s) \neq \bar{f}_1(t)$. Putting all this together we may conclude that $(g_2 \circ f_2)(s) = (g_1 \circ f_1)(s) \neq (g_2 \circ f_2)(t) = (g_1 \circ f_1)(t)$ and thus s and t will be mapped injectively by g_2. The injectivity requirement can likewise be shown for any two distinct edges $e, e' \in E_K$.

Elements that are fresh in H_2 are, due to the pushout construction, also mapped injectively by g_2. This can be derived from the fact that such elements are only equivalent to themselves in the equivalence relation $\simeq$.

Thus, we may conclude that g_2 is injective on all elements in H_2 that are either fresh in H_2, i.e., not in $cod(f_2)$, or in $cod(f_2{\restriction}G^-)$. □

Using Lemma 3.57, we can prove Theorem 3.55 as follows.

Proof of Theorem 3.55. See Fig. 3.18 for a diagram containing all the elements (graphs and morphisms) needed. Eventually, we want to prove that the morphisms $\hat{p}^* \circ \alpha_h^G \circ m$ and $\alpha_2 \circ m^* \circ p$ are commuting and, moreover, describe a pushout.

At first, we are given the attributed graph production $p\colon \langle L, F\rangle \to \langle R, F\rangle$ and an attributed graph $\langle G, A\rangle$ with a matching $m\colon L \to G$. The graph H is constructed such that the commuting square (1) is a pushout. From Proposition 3.41 we obtain the embedding morphism $\eta_G\colon A \to G$.

The commuting square (3) being a pullback and m being a total morphism implies that f is a total morphism. By applying the result from Lemma 3.57 we may conclude that A is also a subgraph of H, i.e., there exists an embedding morphism η_H such that $\eta_H = \eta_G \circ p^*$.

From the abstraction morphism α_h^G we obtain the graph G', i.e., $G' = cod(\alpha_h^G)$. Due to Lemma 3.57 we have that $\hat{\eta}_G$ is an embedding.

By now constructing the embedding pushout of the diagram $\hat{A} \xleftarrow{\alpha} A \xrightarrow{\eta_H} H$, this results in the graph H' with $\hat{\eta}_H$ being an embedding and $\alpha_2 = \alpha_{h,H}$ (similar to $\alpha_1 = \alpha_{h,G}$). We then have two morphisms $\alpha_2 \circ p^*$ and $\hat{\eta}_H$. Due to fact that

(2) has been constructed as a pushout, we obtain the unique morphism $\hat{p}^*$ such that $\hat{p}^* \circ \hat{\eta}_G = \hat{\eta}_H$ and $\hat{p}^* \circ \alpha_1 = \alpha_2 \circ p^*$. In fact, α_2 can be shown to be α_h^H.

For proving that the commuting square (4) describes a pushout, we rely on the fact that the commuting squares (2) and $(2 + 4)$ are pushouts. By decomposition, it holds that (4) also describes a pushout. By composing the pushouts (1) and (4) we have that $(1 + 4)$ is a pushout, i.e., $(\hat{p}^* \circ \alpha_1 \circ m, \alpha_2 \circ m^* \circ p)$ is a pushout. □

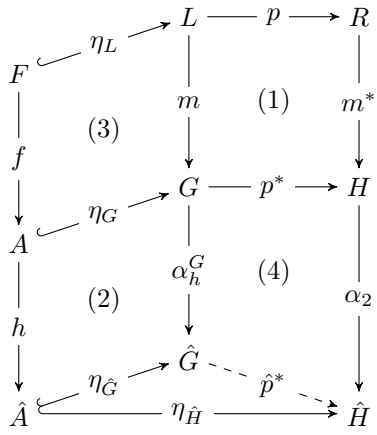


Figure 3.18: Concrete versus abstract graph transformations.

Simulation Relation

Whenever individual states of two distinct transition systems have similar behaviour, e.g., in terms of outgoing transitions, the literature on *simulation relations* between transition systems provides a lot of theory on how to define and exploit such similarities (see, e.g., [146, 141]). One often used result is that this LTL properties are reflected by the abstraction, i.e., if an LTL property holds for the abstract transition system, that it also holds for the concrete one. Here, we will shortly indicate how a simulation between two graph transition systems T_1 and T_2 can be defined for a given algebra graph morphism $h\colon A \to B$.

Definition 3.58. *Let $h\colon A \to B$ be an algebra graph morphism, and $P_1 = (G_1, \mathcal{R})$ and $P_2 = (G_2, \mathcal{R})$ be two graph production systems. Furthermore, let $T_1 =$*

$(S_1, \rightarrow_1, G_1)$ and $T_2 = (S_2, \rightarrow_2, G_2)$ be the graph transition systems generated by P_1 and P_2, respectively. The relation $\precsim \subseteq S_1 \times S_2$ is called a simulation *if for all $(s_1, s_2) \in \precsim$ the following condition is satisfied for all productions $p \in \mathcal{R}$ and corresponding matchings m:*

$$s_1 \xrightarrow{p,m} s_2 \in \rightarrow_2 \quad \textit{implies} \quad \exists s_2' \in S_2 : s_2 \xrightarrow{p,(\alpha_{h,s_1} \circ m)} s_2' \in \rightarrow_2 \ \wedge (s_1', s_2') \in \precsim$$

where $(\alpha_{h,s_1} \circ m)\colon L_p \rightarrow s_2$ is the abstract matching corresponding to the matching $m\colon L_p \rightarrow s_1$.

If such a simulation relation exists, then T_2 *simulates* T_1. From the above theorem we can immediately conclude that, for a given algebra graph morphism, $h\colon A \rightarrow \hat{A}$, there exists a simulation relation between the graph transition systems generated by the attributed graph production systems $P = (G, R)$ and $\hat{P} = (\hat{G}, R)$, where $\hat{G} = \alpha_{h,G}(G)$. When extending the semantics of α_h to graph production systems, we can state the following result.

Theorem 3.59. *Let $h\colon A \rightarrow \hat{A}$ be an algebra graph morphism and $P = (G, \mathcal{R})$ be an attributed graph production system. Then, the graph transition system T_P is simulated by the graph transition system $T_{\hat{P}}$ generated from the graph production system $\hat{P} = (\hat{G}, \mathcal{R})$ with $\hat{G} = \alpha_{h,G}(G)$, denoted $T_P \precsim T_{\alpha_h(P)}$.*

Proof. This can be proved by inductively applying Theorem 3.55. □

We will illustrate the result of applying the abstraction technique as described in this section on the Vector-example by comparing the graph transition system of the actual example with the graph transition system obtained by applying the abstraction explained in Example 3.53.

Example 3.60. *The original Vector-example gives rise to an infinite state space, as we have pointed out earlier. When starting with a graph containing two Objects that can be inserted in the Vector, part of the state is shown in Fig. 3.19. The solid arrows represent actual rule applications. The final dashed arrow represents the fact that the state space continues, without explicitly showing the reachable graphs (which would not be possible anyway). Note that the start graph has two outgoing insert-arrows. Each of them represents the application of the insert rule on a distinct Object, though both applications result in an isomorphic graph, since we cannot distinguish between the two Objects. Fig. 3.20 depicts the state space in which the integer domain only contains three values, namely* neg, zero, *and* pos. *This results in a finite state space. Nevertheless, the latter state space is a simulation of the former since every transition*

of the former is included in the latter. All transitions of the former transition system that represent the application of the doubleCap *rule (there are infinitely many) are simulated by the single* doubleCap *transition in the latter transition system.*

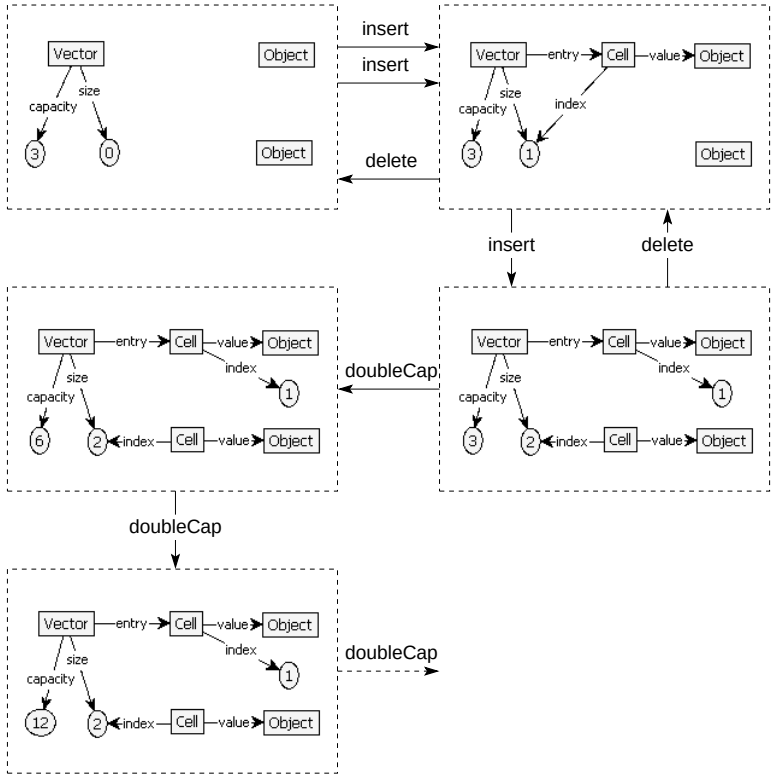


Figure 3.19: Finite part of the infinite graph transition system of the original Vector-example.

3.6.3 Non-Determinism and Case Merging

Whenever an algebra graph morphism $h\colon A \to \hat{A}$ is non-injective the algebra graph $\hat{A}$ might contain product nodes with multiple equally labelled outgoing

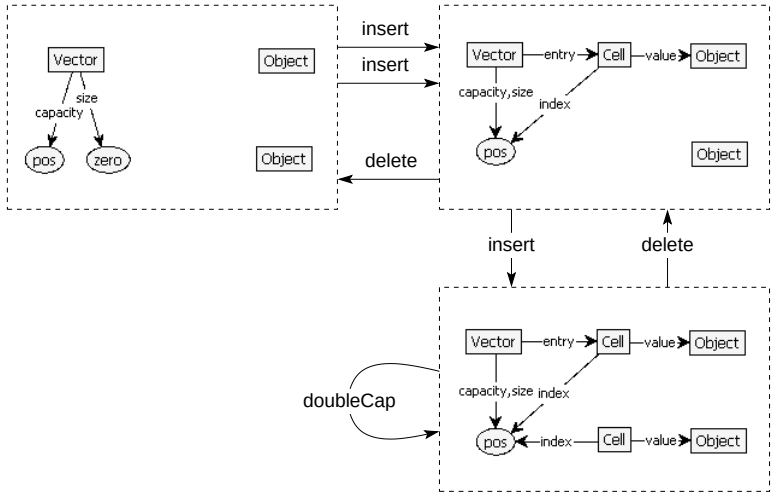


Figure 3.20: Complete graph transition system of the Vector-example with abstraction.

functional operation edges. This yields the possibility of having multiple matchings for specific operations on product nodes. From this we must conclude that the abstract graph transition system might contain more transitions than the original graph transition system. Stated differently, the graph transition system based on $\hat{A}$ is an *over-approximation* of the graph transition system based on A. Additional matchings might arise because of two reasons:

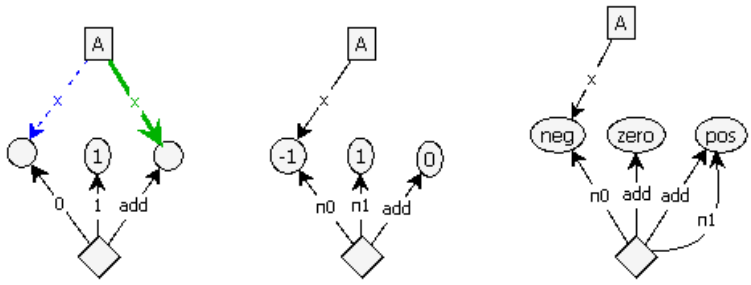
non-determinism: non-injective abstraction morphisms might introduce non-determinism in the algebra graph;

case merging: non-injectivity of abstraction morphisms might introduce matchings of rules that did not occur in the original case.

In the following example, we illustrate the first case, i.e., how non-injective algebra graph morphisms introduce new matchings due to non-determinism in the algebra graph. In Section 3.7 we will discuss our visualization of attributed graphs in more detail.

Example 3.61. *Suppose we have two graph production systems P and $\hat{P}$, such that the left graph of Fig. 3.15 depicts an excerpt of the algebra graph A of the graphs generated by P and the right graph depicts an excerpt of the algebra graph*

$\hat{A}$ of the graphs generated by $\hat{P}$, where $h\colon A \to \hat{A}$ is as in Example 3.53. Let p be an attributed graph production shown in Fig. 3.21(a) and G and $\hat{G} = \alpha_h^G(G)$ be attributed graphs as shown in Fig. 3.21(b) and Fig. 3.21(c), respectively. Then, p has exactly one matching for G which results in the concrete transformation as depicted in Fig. 3.22. When taking $\hat{G}$ as the host graph, there exists two distinct abstract matchings for p for which the transformations are shown in Fig. 3.23.



(a) Rule. **(b)** Concrete graph. **(c)** Abstract graph.

Figure 3.21: An attributed graph production, and two attributed graphs.

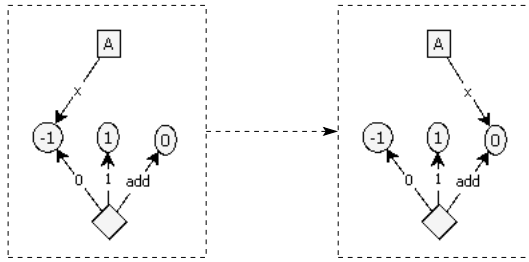


Figure 3.22: Concrete transformation.

The second reason why additional matchings might be introduced occurs in graph production systems that contain rules that distinguish between different values for a specific attribute. Suppose we have a graph production system that contains two rules for which the applicability depends on the value of some Boolean flag. That is, one rule is applicable if the flag is `false`, the other is applicable if the flag is `true`. If we then have an abstraction morphism that merges the two values `true` and `false`, both rules will be applicable, whereas

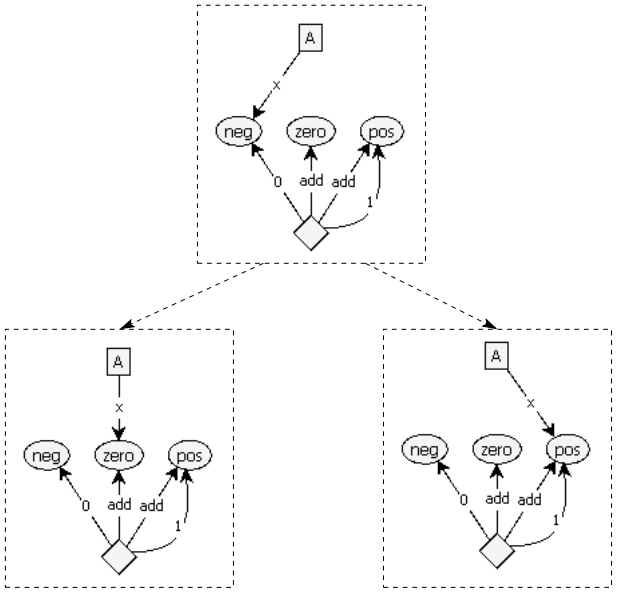


Figure 3.23: Two abstract transformations.

in the original case, only one rule would have been applicable. Note that in this case, the new algebra graph might still be deterministic.

When verifying abstract transition systems, paths (i.e., sequences of transitions) might include abstract transitions that do not reflect concrete transitions. If such paths do not satisfy the particular property being verified, this results in so-called *false negatives*. In our context, for checking whether an abstract counter-example reflects an actual execution of the system, we have to check whether we can construct a concrete path that corresponds to the abstract counter-example. In the field of *counter-example guided abstraction-refinement* (see, e.g., [34, 35]), though, false negatives are used as instruments for refining the abstraction such that the next verification iteration will not include those paths.

In Chapter 4 we will shortly discuss how this abstraction technique can be of use when specifying the semantics of object-oriented languages by attributed graph production systems.

3.6.4 Abstraction to Final Algebras

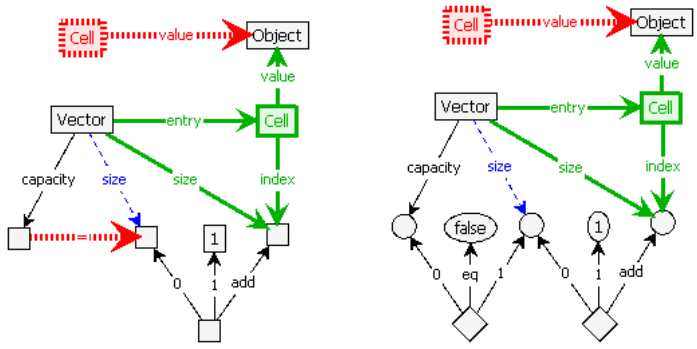
In our framework of performing abstraction on attributed graphs, a special abstraction morphisms maps the algebra graph to the algebra graph representing the *final algebra* of the signature under consideration. Final algebras contain singleton carrier sets for every sort in the signature. All operations in the final algebra take those single elements as parameters and map them to the single element of the carrier set of the target sort [71]. Algebra graphs representing final algebras are therefore guaranteed to be deterministic. In the context of verifying attributed graph production systems, final algebras can be used as the coarsest possible abstraction. Although on the semantic level final algebras are thus not of much use, in Chapter 4 we will show that when applying graph transformations for specifying semantics of programming languages, final algebras are useful for generating state spaces on which static analysis can be applied.

3.6.5 Negative Application Conditions

When specifying negative application conditions (NACs) in attributed graph productions, we have to be careful to make sure that abstraction morphisms do not reduce the applicability of rules with NACs. Stated more formally, Theorem 3.55 should continue to hold in the presence of NACs. This can be achieved by requiring that the elements of the algebra graphs may not be part of any NAC. Suppose we want to specify that an Object can only be inserted in a Vector if the size of the Vector is not equal to its capacity, assuming that its size is invariantly less or equal than its capacity. When naively specifying this with a NAC, the rule would require injectivity of the matching concerning the data nodes representing the size and the capacity. Using the Groove notation, the injectivity constraint can be specified by a NAC as shown in Fig. 3.24(a).

Although this rule could have been applicable in the original case, it cannot be applied on an attributed graph in which both values are represented by the same abstract value. Such situations can be avoided by specifying the inequality of the values through the corresponding algebraic operation. For integers this would mean to specify that the applying the equality operation on the tuple containing both values must return `false`. Thus, the rule depicted in Fig. 3.24(b) is one way of correctly specifying the constraint.

Suppose that the abstraction morphism merges the two distinct nodes that represent the size and the capacity of the Vector. Using the rule in which the constraints on the attributes is specified in term of algebraic operations, the



(a) Incorrect. **(b)** Correct.

Figure 3.24: Specifying inequality of attribute values.

result from Theorem 3.55 still applies from which we may then conclude that any matching is preserved by any abstraction morphism.

3.7 Implementation Issues

In this section we will shortly discuss some implementation issues that arise due to our formalization of attributed graphs and the fact that algebra graphs can potentially be infinite. First we list the set of algebras that are supported by the Groove Tool Set and discuss how this set can be extended. Thereafter, we introduce a shorthand notation for fixing attribute values, and discuss how to deal with infinite algebra graphs, e.g., when visualizing attributed graphs.

3.7.1 Pre-defined and User-defined Algebras

At the time of writing, the Groove Tool Set provides support for attributed graphs in which attributes are instances of the primitive data types Integer, String, and Boolean. For integer calculation, the usual operations like for instance addition, subtraction, multiplication, and division are supported. Furthermore, the integer algebra also includes operations to compare different values, such as, e.g., $<$, $\leq$, $>$, $=$, etc. The String algebra currently only implements operations for concatenating strings and comparing strings for equality.

The set of algebras can manually be extended. For this, one has to implement a number of Java classes, one of which defines the interface of new algebra, i.e.,

its corresponding signature, and another of which implements that interface defining the actual algebraic operations (and constants). The new algebra must finally be registered as being supported by the Groove Tool Set in order to be used in actual attributed graphs and their transformation specifications.

3.7.2 Fixing Attribute Values

In our implementation as well as the visualization of uniform attributed graph productions, we have introduced a shorthand notation for assigning specific data values to free elements. Formally, the correct way to specify this is to include an edge representing the constant symbol pointing from the *empty tuple* to the free element that must be assigned the semantics of that constant. The shorthand notation and visualization then include the constant edge as a self-edge of that free element. The formally correct notation and our shorthand notation are shown in Fig. 3.25(a) and Fig. 3.25(b), respectively, for an example in which a node labelled A has a flag-pointer to the Boolean constant `true`.

(a) Formally correct. **(b)** Shorthand.

Figure 3.25: Notations for specifying fixed semantics to free elements.

3.7.3 Dealing with Infinite Algebra Graphs

Due to the that uniform algebra graphs might be infinite, one could question how Groove deals with such infinite structures. In particular, how are infinite algebra graphs of uniform algebras with infinite data domains stored internally, and can matchings for arbitrary uniform attributed graph productions effectively be computed?

Needles to say, Groove does not explicitly store the entire algebra graph of the uniform algebra under consideration (even when it is finite). The basic idea is that Groove only stores the algebra graph elements that are part of the

regular graph structure, i.e., data nodes that are reached via attribute edges of the graph. Algebra graph elements that are further required in the coarse of the transformation process, are created *on-demand*.

For the second question to be answered positively, we need to put two constraints on the free uniform algebra graphs that are used in uniform attributed graph productions:

1. target nodes of projection operation edges are also target nodes of an attribute or functional operation edge;
2. there are no cyclic dependencies between product nodes.

If these constraints are fulfilled, a matching of a uniform attributed graph production is constructed by first matching the regular graph structure (thus including the attribute edges) and then extending this 'sub-match' (if possible) such that also all remaining algebra graph elements are mapped. For product nodes of which all components (reached by the outgoing projection operation edges) have been assigned a semantics (either in the production itself or by the 'sub-matching'), the target node of all outgoing functional operation edges can be determined. If operations are *nested*, i.e., the result of some operation on some product node is used as a component of another product node, a *dependency graph* is constructed. This dependency graph consists of all projection operation edges and reversed functional operation edges. Due to the second constraint, such dependency graphs are acyclic; the first constraint then guarantees that the leaves of such dependency graphs have fixed semantics, from which the semantics of the remaining nodes can be determined. Stated differently, if the above requirements are met, matchings can always effectively be computed.

3.7.4 Visualizing Attributed Graphs

As we have mentioned before, attributed graphs formally include the entire algebra graph. Usually this gives rise to infinite structures. When visualizing attributed graphs and attributed graph productions we therefore select a finite part of the algebra graph that is meaningful to be shown. For attributed graphs we have chosen only to show the data values that are referenced by attribute edges from the graph. Nodes representing data values are depicted as ellipses. In attributed graph productions we only show the part of the algebra graph that is needed for completely specifying the transformation. This includes the free elements for which the semantics will be fixed by either the matching or by including constant symbols from the signature (as illustrated in Fig. 3.25(b)).

In cases where constraints are specified through algebraic operations on product nodes, we also include those product nodes together with the outgoing projection operation edges.

3.8 Conclusion

3.8.1 Summary

In this chapter we have introduced a new approach to modelling attributed graphs and specifying attributed graph transformations in a *uniform* way, i.e., the attributed graph transformation is completely specified and performed in the graph formalism.

A major advantage of our approach is that attributed graph transformations are completely specified by graph morphisms. This means that we do not need to introduce a variable assignment function or include a set of equations of the variables for specifying attributed graph productions or direct attributed graph derivations. Another advantage of having uniform attributed graphs is the ease with which we can perform attributed graph transformations on different levels of abstraction, with respect to the underlying algebra. By specifying proper abstraction morphisms we have shown that direct derivations are preserved by the abstraction. That is, all direct derivations in the 'concrete' graph transformation system (GTS) have a corresponding direct derivation in the 'abstract' GTS. We have also shown that proper abstraction morphisms might give rise to direct derivations in the abstract GTS that are not reflected in the original GTS. A disadvantage of our approach is that transformation specifications, as currently visualized in our approach, are visually not very attractive. Especially when specifying transformations in which multiple algebraic operations are applied, the rule-graphs may become large and less intuitive. This, however, is mainly a matter of concrete versus abstract syntax; introducing hyperedge-like structures could partly alleviate this issue.

We have shown that our approach is equivalent to other approaches for transforming attributed graphs by taking the approach introduced by Ehrig et al. [65] as a starting point. We have defined flattening and unflattening operators, both on the level of signatures and on the level of actual algebras. By flattening an arbitrary signature SIG we obtain a *uniform signature* $\mathcal{F}(SIG)$; by flattening an arbitrary SIG-algebra A we obtain a uniform $\mathcal{F}(SIG)$-algebra $\mathcal{F}^{\mathcal{A}}(A)$. For the reverse direction we have defined unflattening functors $\mathcal{U}$ (for uniform signatures) and $\mathcal{U}^{\mathcal{A}}$ (for uniform algebras). Based

on the these functors, we have shown that the categories **AttrGraph**(SIG) and **UAttrGraph**($USIG$)$_D$ are equivalent when either $USIG = \mathcal{F}(SIG)$ or $SIG = \mathcal{U}(USIG)$. From this equivalence we may conclude that all categorical (co-)limit constructions that exists in **AttrGraph**(SIG) also exists for the corresponding elements in **UAttrGraph**($USIG$).

3.8.2 Discussion

Edge-attribution. One of the limitations of our approach is that it does not support edge attribution in a straightforward way, in contrast to the approach proposed by, for example, Ehrig et al. [65]. The Groove Tool Set has been set up as a lightweight and easy to use graph transformation tool mainly targeted towards the generation and verification of state spaces of systems specified through graph production systems. One of the fundamental choices in its design is that edges do not have identities. That is, edges are identified by means of their source node, label, and target node. In other work [22] we have shown that arbitrary graph structures can be translated to graph structures in which the edges of the original graph are represented by nodes having outgoing source and target edges, yielding equivalent results in terms of direct derivations. One could argue that edge-attribution is often applied as a technique to more elegantly model entities of the system under consideration, although without edge-attribution, the formalism is equally expressive.

Special Algebra Graphs. Our way of applying abstractions on uniform attributed graphs is by specifying so-called abstraction morphisms, which maps a specific algebra graph to another algebra graph such that the latter is less precise than the former. Using this approach, it is not possible to use abstraction domains in which concrete values are mapped to multiple abstract values, since such mappings cannot be specified as homomorphisms. An often used example is the partitioning of the integer domain as depicted in Fig. 3.26, where

- the value all represents all integers, i.e., $\mathsf{all} = \mathbb{Z}$,
- $\mathsf{pos} = \{n \in \mathbb{Z} \mid n \geq 0\}$,
- $\mathsf{neg} = \{n \in \mathbb{Z} \mid n \leq 0\}$,
- $\mathsf{spos} = \{n \in \mathbb{Z} \mid n > 0\}$,
- $\mathsf{sneg} = \{n \in \mathbb{Z} \mid n < 0\}$,
- the value zero represents the integer value 0.

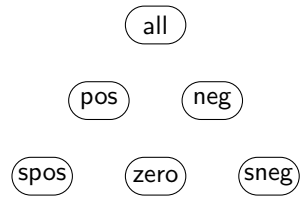


Figure 3.26: An example abstract integer domain.

The abstract value all generalizes all other abstract values, whereas the value neg generalizes the values sneg and zero, and the value pos generalizes the values spos, and zero. The algebra graph over this abstract domain cannot be obtained be any algebra graph morphism from the algebra graph representing the regular integer algebra. One could, however, specify the algebra graph over the above abstract domain by hand (since it is finite) and then perform graph transformations using that algebra graph.

Galois Connections. One can verify that our abstraction morphisms with corresponding *concretization morphisms* actually specify *Galois connections* [48]. The elements of the abstract lattice are then *sets* of abstract graphs in which every set contains all attributed graphs over a fixed abstract algebra graph. However, as we are not interested in computing fixpoints of computations but rather in the transition relation among individual abstract graphs within the same set of abstract graphs, this does not provide useful theory in the current framework.

4 Semantics Through Graph Transformations

4.1 Introduction

Since the Unified Modeling Language (UML) [144] has been accepted as a standard for modelling software program artifacts, it is applied in different phases of the software engineering process such as e.g. *requirements engineering*, and *system design*. Until recently, the focus has mainly been on using UML models for documentation-like purposes only. Using the UML in this way does not provide any means for combating the major problems of, e.g., maintenance and evolution. That is, UML itself does not facilitate automatic transformations between models at different levels of abstraction or their synchronization when introducing small or major modifications.

The introduction of the Model Driven Architecture (MDA) framework [143, 86, 118, 138] caused a *paradigm shift*, which made researchers develop new approaches in which software artifacts were no longer only used for static purposes but rather as models being subject to *transformations*. Specific transformations might actually produce model or code that can be executed. In the MDA framework, models and model transformations are central concepts; the models are specified in diverse (modeling and programming) software languages (SLs), and the model transformations define relations between these languages. That is, a transformation specifies the transformation from a model in one language (often called the *source* language) into a model in another language (the *target* language). By specifying the transformations at the level of the language (in contrast to the level of the models in that language), the transformation specifi-

cation supports the transformation of any model in the source language. Model transformations are intended to be correctness preserving: they should not introduce errors or essential changes. This, however, can be guaranteed only if the meaning of the SLs involved is defined with sufficient precision. Unfortunately, this is often lacking: many SLs have a well-defined syntax but only an *informal semantics*, e.g. described by text or, in the case of a programming language, by a compiler.

On the longer-term, we aim at developing intuitive, though formal, ways in which all aspects of SLs, besides their concrete syntax, can be defined in a consistent and rigorous manner. As a common formal foundation we use graphs and graph transformations, which we believe to be powerful enough to capture all relevant SL aspects. Furthermore, the research that has been carried out in the field of graph transformations over the past four decades offers many theoretical results that can practically be applied in our context.

In this thesis we focus on the one hand on *specifying* the behaviour of systems by means of graph transformations and on the other hand on *verifying* the correctness of such systems. We have developed a small programming language called Taal including all ingredients to be truly object-oriented [197] such as classes, objects, inheritance, and methods. Alternatively, we could have taking an existing object-oriented programming language such as, e.g., Java [93]. The advantage of starting with a fresh small programming language is that we do not (yet) have to deal with more complex concepts such as exception handling and multi-threading. By extending the language with such features incrementally, we are able to investigate the effect on the semantics specification. One of the disadvantages is that we cannot directly apply our approach to moderate-sized systems written in existing programming languages like Java.

In our approach, states (or snapshots) of Taal programs are modelled as graphs. By specifying the execution semantics of Taal as graph transformation rules, we can simulate the execution of the program through performing graph transformations. Such simulations result in transition systems that contain paths representing all the possible executions of the program under simulation. Those transition systems can then be input to various analysis procedures. A particular kind of analysis is the application of verification methods such as *model checking*. By verifying whether all (or some) execution paths of the program satisfy particular properties, one obtains (more) insight in the (correctness of the) behaviour of the program.

In the context of MDA, one could be interested in whether a program (or model) in some programming (or modelling) language that is obtained by some transformation from another program (or model) in the same or, even more

interesting, another language, correctly preserves some specific characteristics of the original program. If semantic models, such as, e.g., transition systems, of the behaviour of both programs are at hand, one can apply formal methods for investigating what kind of relation there exists between both programs.

Overview of the Chapter

This chapter describes a formal approach to specifying operational semantics of programming languages. We define the semantics of a toy programming language called Taal in which we distinguish between the control flow semantics and the operational semantics. Although the control semantics of the language have major impact on the execution semantics, we mainly focus on the latter and only shortly discuss the former. In Section 4.5 and Section 4.6, we set up the basis for the simulation of the execution of Taal programs. That is, we introduce concepts such as *syntax graphs*, *flow graphs*, and *program graphs*. In Section 4.7 we then discuss the basic principles behind the simulations and discuss the actual transformation rules that specify the dynamic semantics of Taal.

This chapter is based on [111, 112].

4.2 Approach

As mentioned before, we have developed the artificial object-oriented programming language Taal. Taal programs are specified textually. From the textual program we generate a graph model, the *Flat Abstract Syntax Graph* (fasg), by first performing the well-known and well-studied processes of *parsing* and *static analysis*, after which we perform a so-called *flattening* transformation. In Section 4.5, we will describe some interesting issues of the flattening transformation. In this chapter we mainly focus on the transformations being performed from the fasg onwards, since they have been implemented as graph production systems. For a detailed description on how we transform a textual program into its fasg, the interested reader is referred to [111].

Figure 4.1 gives an overview of the processes involved in this work. Firstly, we play the role of a *language designer* by defining the concrete and abstract syntax of Taal. The former is defined as an EBNF grammar which is fully included in Appendix C; the meta model of the latter is defined as a UML class diagram and will be discussed in Section 4.3. Furthermore, we have defined the

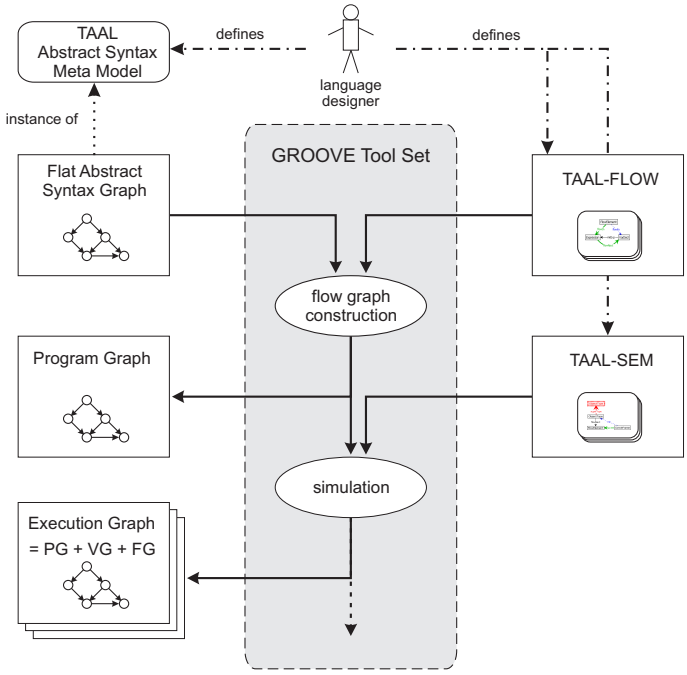


Figure 4.1: From an FASG to its simulation.

flow and execution semantics of Taal in terms of sets of graph transformation rules, in Fig. 4.1 denoted TAAL-FLOW and TAAL-SEM, respectively.[1]

An FASG is formalized as a *plain graph*, i.e., a labelled directed graph as described in Chapter 2. This means that our graphs have no hierarchy and that we do not distinguish between edge types by means of different shapes but through edge-labels instead. From a FASG we construct a *Program Graph* (PG) by including the control flow semantics explicitly in the graph structure. This process is called *flow graph construction* as shown in Fig. 4.1. This is done by applying the rules in TAAL-FLOW to the FASG. This set of rules is guaranteed to terminate and the result of this process is a unique PG. Details on this

[1]Some of the rules in TAAL-FLOW and TAAL-SEM will be discussed in this chapter. The full set of transformation rules and some example Taal programs together with their corresponding Groove graph models are available from http://www.cs.utwente.nl/~kastenbe/taal.

transformation and the resulting PG are discussed in Section 4.6.

A PG can be inspected manually to get an impression on how the generated flow graphs look like. Here, a PG is subject to another graph transformation process called *simulation*, which results in a simulation of the original program. The simulation results in a set of *Execution Graphs* (EGs). Basically, an EG is a PG enriched with dynamic information that represents the actual state of the program. In traditional compiler terminology, a snapshot of the program consists of a denotation of the *heap* and the *stack*; in our terminology these concepts are represented by the *Value Graph* (VG) and the *Frame Graph* (FG), respectively.

EGs reside on a different level from the artifacts we have seen so far. Each of the artifacts starting from the textual program to the corresponding PG are all unique *representations* of the same program at different levels of abstraction. In the simulation phase, single graph transformation steps simulate execution steps of the actual program, which means that each transformation step describes a state transition. The set of all reachable EGs thus specifies all the possible states the program can be in. The dashed arrow in Fig. 4.1 indicates that the result of the simulation can serve as input for further processing or analysis. Some types of analysis will be discussed in Section 4.8; one analysis technique, namely model checking, will be the main topic of Chapter 5.

4.3 Abstract and Concrete Syntax

An important aspect in (programming) language development is the specification of the *abstract syntax*. The abstract syntax of a language defines the language constructs and how those constructs relate to each other. For a program to be expressed in the language, a *concrete syntax* is required. For the concrete syntax of a programming language, we often distinguish between a textual and graphical syntax. Many programming languages such as e.g. Java [93] or C [116] are accompanied by a textual syntax, for which the exact structure is specified for example in an (E)BNF grammar. Modelling languages such as e.g. the UML [144] are mainly graphical, although they may also include textual languages for specifying additional semantic constraints. The UML, for example, is extended with the Object Constraint Language (OCL) [196].

For Taal we specify the abstract syntax by means of UML class diagrams. In the coming paragraphs we will introduce the abstract syntax through three sub-diagrams covering *types*, *statements*, and *expressions*.

4.3.1 Types

In Taal we support the use categories of three types: ObjectType, PrimitiveType, and NullType. How these types are related to each other and how they are composed is depicted in Fig. 4.2. We will shortly discuss each of the concepts from Fig. 4.2, together with their associations.

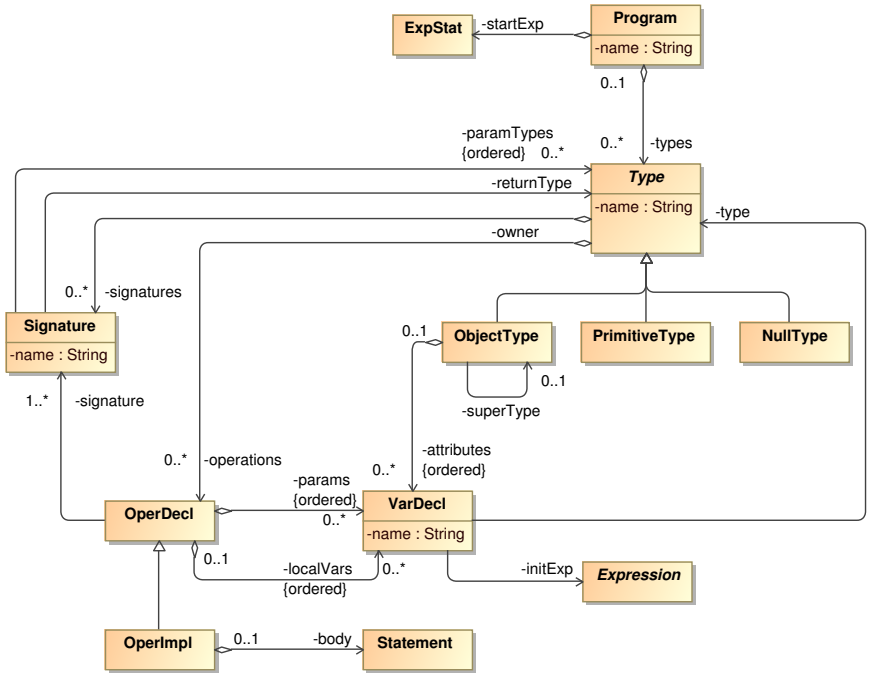


Figure 4.2: Meta-model of the Taal abstract syntax – Types.

Program. The class Program represents the whole program. In a program multiple data-structures can be declared and referred to. The data-structures being declared are the types of the program represented by the types-association with the class Type. When a program has multiple types, these types are ordered, reflecting the order in which they appear in the code. The fact that a program needs to start somewhere, requires every program to have a single

start-expression. Therefore, the class Program has a startExp-association with class ExpStat, which will be discussed later on.

Type. Typing is a very important concept in object-oriented languages. The purpose of class Type is to model typing at a generic level. The class Type has three subclasses, namely ObjectType, PrimitiveType and NullType. The Type class has an association with class OperDecl called operations representing a (possibly empty) set of operations which is the object-oriented way: associate operations with type declarations. Type is abstract, which is indicated in the diagram by the fact that its name is in italics.

ObjectType. Class ObjectType is a specialization of class Type. It enables the programmer to define new types in the program. Class ObjectType has a superType-association with itself modelling inheritance relations between instances of this class. Class ObjectType has an association called attributes with class VarDecl representing the instance variables.

PrimitiveType. Taal includes a number of primitive data types such as integers, reals, and strings. Those are represented by the class PrimitiveType.

NullType. Class NullType represents the type of a special value: the *null value*. This type is a *singleton* class, which means that during a run of the program, there can only exist a single instance which is shared if used at multiple places simultaneously. The special thing about this class is that it *conforms to* any other class. That is, the singleton null-instance can be used as an instance of any other class, although method calls and attribute references on the null-instance result in (run-time) errors.

OperDecl. Operations that are declared for ObjectTypes are captured by the class OperDecl. It explicitly refers to the Type-class it belongs to by means of an association called owner. Every operation may require multiple parameters in order to be executed. This is modelled by the params-association with the class VarDecl. An operation may also have local declared variables. Therefore, the localVars-association has been introduced. Note that both associations params and localVars are ordered. The order of the parameters partially identify the operation; local variables are ordered for reasons that will become clear in Section 4.6. In order to track the referred operation implementation at run-time it is needed to assign a Signature to every operation.

OperImpl. The meta-model explicitly distinguishes between abstract and concrete operations. Abstract operations, i.e. operations that are only declared and do not have an actual implementation, are instances of the OperDecl class; concrete operations are instances of the OperImpl class. The OperImpl class, therefore, has an association with Statement called body.

Signature. The class Signature stores the elements that uniquely identify an operation, i.e. the types of the parameters (this is an ordered list as), the return type, and the name of the operation. In Section 4.7, it will become clear that method signatures play a crucial role in the execution phase that is often referred to as *dynamic method lookup*.

VarDecl. Class VarDecl represents the declaration of a variable, which can be either an object variable (also called instance variable), a local variable in some method, or a formal method parameter. Every variable is of some type which is captured by the type-association with class Type. Furthermore, this class has an association with class ObjectType called owner representing that every instance variable belongs to at most one object. The association initExp with class Expression represents the initialization of the instance variable at declaration. Note that this association is not optional.

4.3.2 Statements

Taal facilitates the use of a variety of statements as depicted in Fig. 4.3. The different statements and their associations with other language constructs will be discussed in this section. The classes in the gray dashed rounded rectangles have been or will be discussed in more detail in the class diagrams for types (see Section 4.3.1) and expressions (see Section 4.3.3) as indicated by the bold text in those rectangles.

Statement. The class Statement is the super class of all specific statements that can occur in a Taal-program. It has six subclasses, namely ConditionalStat, ExpStat, ReturnStat, BlockStat, WhileStat, and AssignStat. The Statement class itself cannot be instantiated, i.e. it is an *abstract* class, which is indicated in the diagram by italic typeface.

BlockStat. The class BlockStat represents a (possibly empty) block of statements. This is modelled by the association subStats with class Statement. This

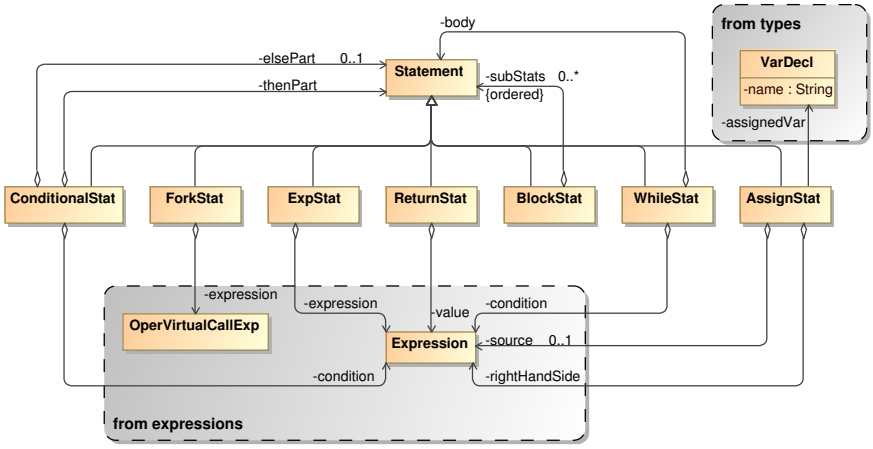


Figure 4.3: Meta-model of the Taal abstract syntax – Statements.

association is ordered which enables sequential execution of statements in the same block. For simplicity and minimization of implementation effort, we have chosen not to support local variables inside BlockStats. This does not limit the expressiveness of the language, although it might influence the usage of run-time resources.

AssignStat. The class AssignStat represents the statements in which a particular evaluation of an expression will be assigned to some variable. The variable that will be assigned a new value, might be the attribute of some ObjectType-instance. This is represented by the association with class Expression called source. The variable that will be assigned is referred to by the association with class VarDecl called assignedVar. The actual value to be assigned to the variable is the evaluation of the expression on the right side of the assignment operator. This expression is referred to by the rightHandSide-association. Note that the assignedVar and rightHandSide component are obligatory, whereas the source component is optional.

ConditionalStat. The class ConditionalStat represents the well-known if-then or if-then-else construction in programming languages. It consists of two required, and one optional element. The first required element is the condition which

determines where to continue the program. This is modelled by the condition-association with class Expression. The second required element is a statement representing the then-part which will be executed when the conditional-expression evaluates to `true`. This is modelled by the thenPart-association with class Statement. There is another association with this class called elsePart, which may or may not exist for a particular conditional statement.

WhileStat. The class WhileStat represents a while-statement. It holds a reference to an instance of class Expression by the condition-association that figures as the condition, as well as a reference to an instance of Statement that figures as the body to be executed in case the condition returns `true`. Both components are obligatory, although the Statement may be an empty BlockStat.

ExpStat. The class ExpStat represents an expression used as a statement, i.e. the resulting value of the expression is discarded. The actual Expression is referred to with an expression-edge. Typical examples are *object creation* and *method invocation*.

ForkStat. The class ForkStat represents a statement which introduces parallelism. It references an OperVitualCallExp (see Section 4.3.3) by an expression-association for which the simulation will be guided by a separate thread (in parallel). This will be explained in further detail in Section 4.7.

ReturnStat. The class ReturnStat models a statement that indicates the points at which a method successfully terminates execution and optionally (as specified by the multiplicity 0..1) returns a value.

4.3.3 Expressions

The different expressions supported in Taal are shown in Fig. 4.4. In the following paragraphs we will shortly explain each of them separately.

Expression. The class Expression is the superclass of all supported expressions in Taal. Similar to the Statement-class, the Expression class is abstract (again indicated by the italic typeface) and can therefore not be instantiated. Every Expression is of a certain type as represented by the obligatory resultType association with class Type.

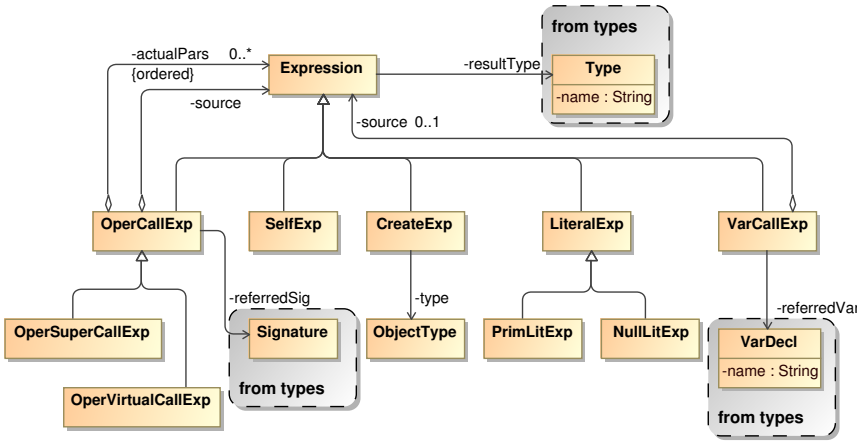


Figure 4.4: Meta-model of the Taal abstract syntax – Expression.

CreateExp. The class CreateExp represents an expression creating a new object. It has a link with the ObjectType of which the newly created object will be an instance.

LiteralExp. Instances of the class LiteralExp represent literal expressions. The class LiteralExp cannot be instantiated, i.e. it is an abstract class. Literal expression occur as instances of one of its subclasses PrimLitExp or NullLitExp. Every LiteralExp has a corresponding value as represented by the value-association.

PrimLitExp. Instances of the PrimLitExp class represent literal expressions of some primitive type, i.e. they have PrimitiveType as their resultType.

NullLitExp. Instances of the NullLitExp class refer to the single instance of the NullType.

OperCallExp. The class OperCallExp represents a reference to an operation of a type. Each OperCallExp-instance may have actual parameters. This is represented by the association between OperCallExp and Expression called actualPars. An OperCallExp instance also has an association to the Signature of the operation of which it is an invocation. Note that an OperCallExp does not reference

an OperDecl or OperImpl. The actual OperImpl that provides an implementation for the called operation cannot statically be determined and is therefore looked up dynamically, i.e., at run-time; the process of dynamic method lookup is discussed in detail in Section 4.7.3.

The class OperCallExp has an optional source-association to Expression. This construction is needed, for example, when a method is called on a variable of some ObjectType or on the return value of some operation (if that operation returned an instance of some ObjectType). The class OperCallExp is abstract, which is indicated in the diagram by the italic typeface.

OperVitualCallExp. The class OperVitualCallExp represents a usual invocation of an operation. With usual, we mean that the process of method lookup will start from the actual type of the object.

OperSuperCallExp. A special type of a method invocation is supported through the class OperSuperCallExp. This class represents method invocations for which the method lookup process starts at the supertype of the current object.

SelfExp. Expressions of type SelfExp represent references to the current object itself. This is useful, for example, in cases when a method has a parameter with the same name as one of the attributes of the ObjectType for which the operation is defined.

VarCallExp. The class VarCallExp represents a reference to a variable. Each VarCallExp-instance has an association with the attribute, local variable, or formal parameter of an operation. In the metamodel this is indicated by the association with class VarDecl called referredVar. Instances of VarCallExp optionally have a source-expression of type Expression. This is needed when referring to an attribute of a specific object.

4.3.4 TAAL EBNF Grammar

The previous sections focussed on how we defined the abstract syntax of Taal. This immediately raises the question, how the concrete syntax of Taal has been defined and how it relates to the abstract syntax specification. Here, we will shortly discuss the just mentioned issues.

The concrete syntax of Taal has been defined by an EBNF grammar. The entire Taal grammar specification has been included in Appendix C. A textual

Taal program is, as usual, *syntactically correct* if it conforms to the grammar. Many issues that could make a Taal program *semantically incorrect* are revealed during static analysis performed by the compiler.

Listing 4.1 shows some fragments from the grammar of which the relation with the class diagrams discussed in the previous sections will be discussed. The first part specifies the local structure of the Program-concept. Grammar elements such as, e.g., <PROGRAM_START>, <PROGRAM_END>, and <CURLY_OPEN> represent pure syntactic elements needed by the compiler to correctly parse the program. Those elements have no counterparts in the abstract syntax. The element ParsedExpression represents the expression that starts up the program. In the abstract syntax, this is reflected by the startExp-association between the classes Program and Expression. The names for the associations in the abstract syntax are hard-coded in the Taal compiler. There exist compiler generators, such as, e.g., ANTLR [147], that support the inclusion of such names in the grammar specification.

```
ParsedProgram ::=
  <PROGRAM_START> <STRING>
  <CURLY_OPEN> ParsedExpression <CURLY_CLOSE>
  ( ParsedTypeDecl )*
  <PROGRAM_END>;

...

ParsedStatement ::=
  ParsedExpression [ <ASSIGN> ParsedExpression ] <SEMICOLON>
  | ParsedReturnStat <SEMICOLON>
  | ParsedConditionalStat
  | ParsedWhileStat
  | ParsedBlockStat

...

ParsedWhileStat ::=
  <WHILE> ParsedExpression <DO>
  ( ParsedStatement )*
  <ENDWHILE>

...

PROGRAM_START ::= "program"
PROGRAM_END   ::= "endprogram"
WHILE         ::= "while"
ASSIGN        ::= ":="
STRING        ::=
  ["a"-"z", "A"-"Z", "_"]
  ( ["a"-"z", "A"-"Z", "0"-"9", "_" ] )*
```

Listing 4.1: Grammar excerpt of the Taal EBNF grammar.

The second part defines the types of statements that can occur in any Taal program. In the Taal grammar, square brackets specify optionality of elements. The second line of this part thus states the a particular kind of statement consists of only a `ParsedExpression`, which in the abstract syntax is represented by the ExpStat. A `ParsedExpression` followed by `<ASSIGN>`, i.e., the assign-operator `:=`, and another `ParsedExpression` is captured by the concept AssignStat in the abstract syntax. Other types of statements, as we have seen in Section 4.3.2, are for instance the ReturnStat, included in the grammar as the non-terminal `ParsedReturnStat`, and the WhileStat, specified in the grammar through the non-terminal `ParsedWhileStat`.

One final example of a non-terminal is the `ParsedWhileStat`. The abstract syntax indicates that every `while`-statement requires an Expression (its condition) and a Statement (its body). The third part of Listing 4.1 shows the excerpt of the Taal grammar which specifies that, indeed, an occurrence of a `ParsedWhileStat` requires an `ParsedExpression` and a (possibly empty) sequence of `ParsedStatements`.

Terminals in the grammar are sequences of characters that will be treated by the compiler as atomic elements. Some example terminals are listed in the final part of Listing 4.1.

4.3.5 Remarks

At this point we want to make a few remarks concerning the design decisions made for Taal. At the moment, we only support single-file programs. That is, the start-expression and the classes referenced in the program must be specified in a single file. Furthermore, we also do not provide any means of structuring the program components. One could think of a package structure as provided by Java. Another simplification is that we have left out some constructs that are common in many (object-oriented) programming languages such as, e.g., the declaration and usage of *class variables*, also called *static variables*, and *static methods*. We do not regard those as being of high interest in the context of Taal. We are confident that the above features can easily be incorporated in Taal.

Another feature of the object-oriented paradigm that is not supported by Taal is *encapsulation*, i.e. the concept of information hiding at the level of objects. Stated differently, we do not provide any means of hiding object attributes and methods to prevent referencing them outside the object's scope. Java, for example, does provide encapsulation functionality through visibility keywords such as `public`, `protected`, and `private` that can be attached to

attributes and methods.

A final remark concerns the way classes can be initialized in Taal. Currently we only support object creation using a CreateExp expression which does not take any parameters. This is similar to the default *object constructor* as provided by Java. This constructor cannot be included explicitly in the source code of a Taal program. We will see in Section 4.7 that this design decision requires object initialization and method invocation to be handled differently, whereas other languages, such as e.g. Java, deal with object initialization as invocations of special constructor methods that are generated by the compiler.

4.4 Two Example Programs

For further explaining our approach, we will use two example Taal programs. The textual representation of this programs are shown in Listing 4.2 and Listing 4.3. The first program mainly includes usual statements as one could expect in any program to be written in Taal; the second program is included since it contains a fork statement. In Section 4.7 we will see how the (expected) difference between simulating both programs is reflected in our approach. The intuition behind the semantics of the language is that a Taal program has similar meaning as a corresponding Java program. The first program specifies the types Vase, Flower, and Rose, with Rose being a subtype of Flower. Its initial expression creates a fresh Vase-instance and calls the changeFlower-method with a fresh Rose-instance as its only parameter. The effect of the changeFlower-method is to cut-off pieces from the given Flower-instance as long as it does not "fit" the Vase-instance. The second example program specifies the type Amoeba which can initialize its own attribute child through the doit-method. The cloning process is guided be a second thread as indicated by the keyword fork. Later on it will become clear how this can lead to parallelism in the program.

```
program vase

  { new Vase().changeFlower(new Rose()) }

class Vase
  myFlower: Flower;
  height: Integer := 20;

  changeFlower(newFlower: Flower)
  locals tempVase: Vase := new Vase();
  {
    tempVase.myFlower := self.myFlower;
    while newFlower.length.largerThan( height.plus(15) ) do
      newFlower.cut();
```

```
    endwhile
    self.myFlower := newFlower;
  }

  getColor(): String {
    return myFlower.getColor();
  }
endclass

class Flower
  color: String := 'yellow';
  length: Integer := 50;

  getColor(): String {
    return color;
  }

  cut() {
    length := 35;
  }
endclass

class Rose extends Flower
  myColor: String := 'red';

  getColor(): String {
    color := myColor;
    return super.getColor();
  }

  cut() {
    length := length.minus(5);
  }
endclass

endprogram
```

Listing 4.2: Flowers in a Vase.

```
program amoebaWorld

  { new Amoeba().doit() }

class Amoeba
  child: Amoeba;

  doit() {
    fork self.clone();
    child := null;
  }
  clone() {
    child := new Amoeba();
  }
endclass

endprogram
```

Listing 4.3: Amoeba-world.

4.5 Flat Abstract Syntax Graphs

On the one hand, Flat Abstract Syntax Graphs (FASGs) are instances of the meta-models we have shown in Section 4.3.1 through Section 4.3.3. On the other hand, since plain graphs do not provide any means of directly including concepts such as, e.g., inheritance and ordered relations which do occur in the meta-models, they have to be encoded in FASGs through additional graph elements. In this section we will shortly discuss some of the issues that arise when flattening UML diagrams to so-called *type graphs*. The main issues that are of major importance when specifying the operational semantics are, as mentioned above:

- how to incorporate the notion of *inheritance* in plain graphs such as to be able to benefit from it when specifying the semantics of the language;
- how to model *ordered* associations in plain graphs, which is required for correctly specifying the language semantics.

The coming paragraphs give an informal description of how those issues have been resolved. For a more formal specification of the flattening transformation in our approach, the interested reader is referred to [112]. Although in our context we only focus on the two above mentioned issues, Kleppe and Rensink [117] give a rather complete formalization of UML class and object diagrams based on graph models.

The first issue to resolve is the fact that in plain graphs we do not have different types of nodes or edges for modelling different syntax element types or associations between those types and instances thereof. Instead, we introduce typing of elements through edge labels. Node types can then be modelled by self-edges.

4.5.1 Inheritance

Concerning the inheritance structure available in UML and used in Section 4.3 for specifying the abstract syntax of Taal, we ideally want to specify semantics for language constructs as generally as possible. For example, suppose we have the inheritance structure as shown in Fig. 4.5.

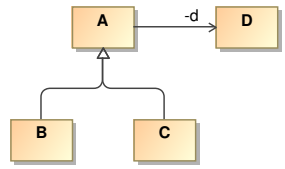


Figure 4.5: Example inheritance structure in UML.

For cases in which specific transformation do not distinguish between the two subclasses of class A, we want to specify a single transformation rule that applies on both B- and C-instances. In the usual setting this can only be achieved by including the reflexive and transitive closure of the superType-association in the rule. The main drawback of this approach is that transformation rules become slightly bigger and, more importantly, less intuitive.

An alternative solution is to transform the meta-model to a type graph in which every node is labelled with all names of its superclasses. Furthermore, also all the associations a specific class, say A, has with other classes must be pushed down to all the subtypes of A. Essentially, this is the same as building the *closure* of the original meta-model, as proposed by Ehrig et al. [64]. The above inheritance structure then results in the type graph as shown in Fig. 4.6.

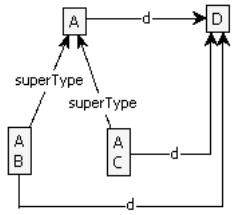


Figure 4.6: Pushing down node- and edge-inheritance in plain graphs.

Using this approach, the graphs become more complex and less elegant to visualize. This is not a major concern to us in this work for two reasons. On the one hand, we focus on the easy of specifying the semantics of a (programming) language and not so much on the shape of the actual models that reside on the different levels; on the other hand, the graphs of even small programs can already be fairly large, and, eventually, we are not really interested in visualizing the graphs but rather in the transition systems that arise from simulating a program.

Finally, the Groove Tool provides functionality of leaving out undesired labels in the visualization of the graphs without effecting the actual graphs on which the transformations are performed.

4.5.2 Ordered Associations

The second issue we discuss concerns the way we model ordered associations between syntax elements. We illustrate our approach by means of the artificial meta-model shown in Fig. 4.2.

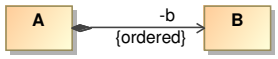


Figure 4.7: Modelling an ordered association in UML.

Modelling ordered associations in plain graph will be achieved by introducing a label set Lab^{U2G} which is disjoint with the set of labels Lab that is used in the graphs elsewhere, i.e. $Lab^{\mathrm{U2G}} \cap Lab = \emptyset$. The special labels used for modelling the ordered associations properly are orderFirst and orderNext. An orderFirst-edge will be pointing to the first element in the ordered list; any element itself will have an orderNext-edge referring to the next element in the list. The source element of the ordered association has an edge to all elements of the list labelled with the name of the association. Essentially, this all boils down to transforming the meta-model from Fig. 4.7 into the type graph shown in Fig. 4.8.

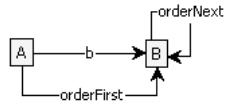


Figure 4.8: Modelling an ordered association in plain graphs.

The validity of this transformation depends on the fact that the association in Fig. 4.7 is a *composition*, which means that B-instances can only be associated with at most one A-instance at a time. If the association would be a regular association, we would need some additional graph elements that specify the origin of the ordered association.

4.5.3 The Flower and Vase Example

In order to give an impression on how FASGs look like, Fig. 4.9 depicts the part of the FASG from the textual program from Listing 4.2 modelling the Flower type declaration with its color and length attributes.

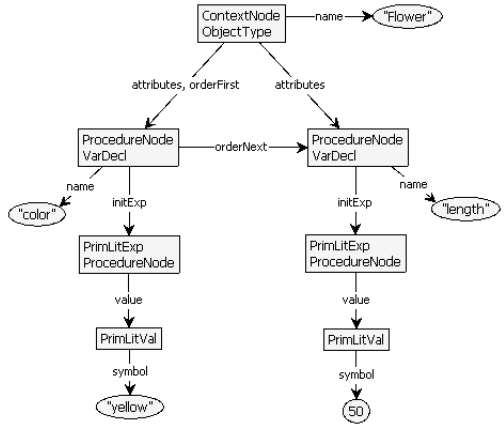


Figure 4.9: FASG of the Flower type declaration.

4.6 Program Graphs

For any language, the flow of control over the statements and expressions of a program is part of the language semantics. In the case of *imperative* languages, execution of the program is determined by the sequential appearance of elements in the textual program. In some cases, parts of the program are skipped or executed multiple times. Although the FASG provides sufficient information to uniquely determine the next program piece to be executed, specifying the execution semantics of the language can be done more intuitively, and simulating the actual program can be performed more efficiently if the sequential execution relation between language constructs is *hard-coded* in the graph structure. This can easily be explained by considering the WhileStat construct. The (flow) semantics of a WhileStat can be specified in natural language as follows:

1. evaluate the condition;

2. (a) if the condition evaluates to `true`, execute the body and continue at step 1;
   (b) if the condition evaluates to `false`, continue with the statement after the WhileStat.

The difficulty is not so much to direct control from step 1 to either step 2a or step 2b, but rather the transition from step 2a back to step 1. For correctly determining the graph element that represents the condition of the WhileStat, we need to be able to identify the WhileStat itself. Especially in cases when multiple WhileStat instances are nested, we have to make sure that we identify the 'closest' WhileStat. Things can get more complicated when the last statement of the body is contained in a BlockStat which itself is again nested in yet another BlockStat. That is to say, in general there is no *a priori* bound on the level to which BlockStats are allowed to be nested. This problem occurs for the last statement of every BlockStat. Those statements do not have a direct reference to the statement to be executed after the BlockStat they are contained in.

Another problem of a similar kind occurs for statements that are composed from an Expression which itself may by composed of further sub-expressions. Suppose, for example the following assignment statement (assuming that all variables are declared and correctly typed):

```
a.b := c.d();
```

The rightHandSide of this AssignStat consists of an OperVitualCallExp having a reference to the Signature of the method `d()` and a source expression which is an VarCallExp-instance having a reference to the VarDecl belonging to `c`. Instead of `c.d()`, we could have written `c.d.e.f()` as well, indicating that, again, there is no *a priori* bound on the level of nested VarCallExp-instances.

Both examples can be summarized by stating that it is not possible to simulate any Taal program with production rules in which the level of nesting (BlockStats or VarCallExps for the above examples) is fixed. In our approach there are at least two ways to deal with this problem which will be discussed in more detail:

1. using regular expressions in the rules;
2. by separating the simulation of the program in a control flow generation phase and an actual execution simulation phase.

Ad 1. The idea behind the first alternative is to include regular path expressions in the semantics rules that can match an arbitrary depth of nesting.

By furthermore including negative application conditions (*NAC*s, for short), one can also specify, for example, that the matched BlockStat is the first (when going 'up' in the syntax graph) having an next statement to be executed. That is, a BlockStat, obviously, itself can also be the last statement of another BlockStat it is nested in. Ad 2. The second alternative splits the execution phase in two sub-phases: a first phase in which control flow information is added to the syntax graph more explicitly, by means of special labelled *control flow edges*. The result of this phase is then the start graph of the actual execution simulation phase. We will first show the advantages and disadvantages of the first alternative and then discuss why we have chosen the second one.

The main advantage of the first alternative is that, based on the specification of the abstract syntax of the language, the semantics can be specified by a single set of production rules, and simulating any program only consists of repeatedly applying those production rules on the FASG that models the actual program. The main disadvantage of this approach is that the production rules tend to be quite large and complex, and therefore hard to understand; one can easily make mistakes when constructing the semantics rules, which makes it hard to have confidence in the correctness of the rules. As a result of the size and complexity of the rules, especially in the presence of regular expressions, the simulation phase is slowed down significantly since searching for rule applications consumes (much) more time.

The second approach reflects the steps that are performed by any compiler. The disadvantage here is that one has to construct two sets of graph production rules: one set containing rules generating the control flow graph elements and a second set consisting of rules defining the actual execution semantics. Applying the first rule-set on a well-formed FASG results in what will be called a *Program Graph* (PG, for short). The rules specifying the execution semantics will then be much smaller compared to the other approach. An execution semantics rule will, typically, only match the syntax element to which the program counter is pointing and the syntax element to be simulated next. As a result, the semantics rules are much easier to understand and specifying those rules is less error-prone. Furthermore, since flow graph construction can be done once and for all, i.e., it is independent of the execution semantics, simulating the program will be (much) more efficient.

In the remainder of this section we will give a detailed description of the structure of Program Graphs (PGs), focusing on the syntax elements that are part of so called *flow graphs* and the different roles they can have.

4.6.1 Flow Graphs

As mentioned above, the PG of a Taal program is the starting point of its execution simulation. Basically, the PG combines two kinds of structures:

- the FASG described in Section 4.5, modelling the required elements of the concrete syntax in terms of nodes and edges representing abstract syntax elements,
- *flow graphs*, which model the sequential execution relation between executable statements.

Given the FASG of a Taal program, we construct the corresponding PG by applying graph production rules which add control flow information to the FASG. Traditionally [83], flow graphs are directed graphs in which every node, except for the *final node* (or *end node*), represents the execution of some program instruction. The other nodes of a flow graph are partitioned into *procedure nodes* and *predicate nodes*. The difference is that a procedure node has exactly one successor node, whereas a predicate node has multiple successors (usually two) and depending on the evaluation of some (Boolean) condition, one successor is taken.

In our approach we define flow graphs in a similar way, distinguishing procedure nodes, predicate nodes, and *context nodes*. The context nodes serve as both start and end nodes of the flow graphs.

Definition 4.1 (flow graph)**.** *A* flow graph *is a directed graph consisting of three types of nodes (also called* flow elements*), namely* procedure*,* predicate *and* context nodes*, connected by different types of successor-edges. Procedure and predicate nodes represent executable statements; the context nodes represent the start and end point of each flow graph; the successor-edges represent the sequential relation between between statements. For every flow graph the following properties hold:*

- *a flow graph has exactly one context node which has one outgoing successor-edge;*
- *every procedure node has exactly one outgoing successor-edge;*
- *every predicate node has exactly two outgoing successor-edges.*

A flow graph can best be understood in relation to a *locus of control*, which is a node of the Execution Graph (discussed in detail in Section 4.7) standing for a thread of execution. Control is said to be *at* a node of the flow graph. Such

a node represents the next syntax element to be simulated and can, in compiler terminology, be compared to the instruction to which the *program counter* is pointing. In our formalism, the program counter will be represented by a special edge labelled pc of which the target node represents the syntax element to be simulated. In Section 4.7.2 we will further elaborate on the appearance of the program counter in our formalism. The successor-edges indicate where control should go after it leaves the current syntax element. In our approach we distinguish three different kinds of successor-edges, by using the labels flowNext, flowTrue, and flowFalse.

Every flow graph will have a single context node that serves as both its start and its end node. This yields *cyclic* flow graphs which is a bit unusual. This cyclic property is discussed in more detail in Section 4.7. The general structure of flow graph is depicted in Fig. 4.10. We say that every syntax element that can serve as either a procedure, predicate, or context node is a FlowElement. In the coming paragraphs, we will discuss each of the subclasses in more detail.

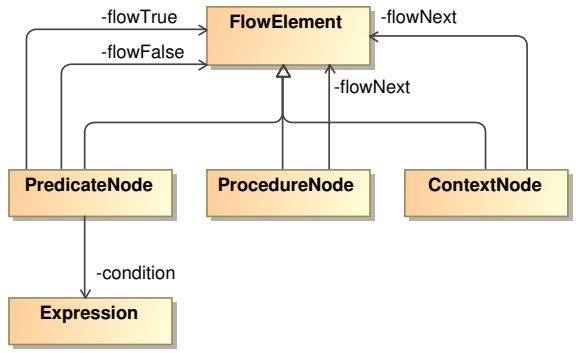


Figure 4.10: Flow graph meta-model.

Procedure Nodes. Procedure nodes represent statements after which it is deterministic which statement to execute next. The syntax elements that can serve as a procedure node are shown in Fig. 4.11. One element that seems a bit out of place here is the VarDecl-element since it is neither a Statement nor an Expression. When discussing the context nodes it will become clear why we model it as a procedure node.

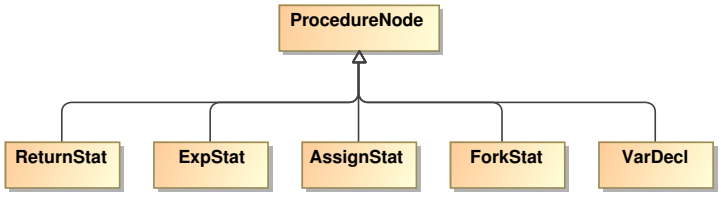


Figure 4.11: Procedure nodes.

Predicate Nodes. Statements that are represented by predicate nodes are related to a condition, which will be evaluated to either `true` or `false`. The actual value of the condition determines which branch will be taken. In this paper we only consider predicate nodes with two outgoing successor-edges: one for the case the condition is evaluated to `true` and one for the case the condition is `false`. As Fig. 4.12 shows, in Taal there are only two kinds of statements for which a conditional expression needs to be evaluated: the WhileStat and the ConditionalStat (if-then and if-then-else).

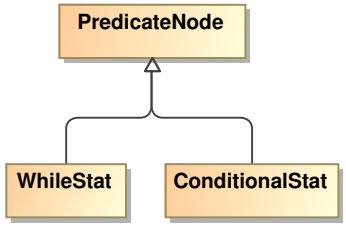


Figure 4.12: Predicate nodes.

Context Nodes. Fig. 4.13 shows what types of nodes can appear as context nodes. In special cases, such as an ObjectType without attributes, a flow graph can exist of *only* the context node. In those cases, the context node still has a successor-edge, which in this case points back to the context node, making the cyclic property even clearer.

We will discuss each of the different context nodes in more detail.

Program context: Program flow graphs control the startup of the program being modelled. In TAAL, program startup is modelled by the execution

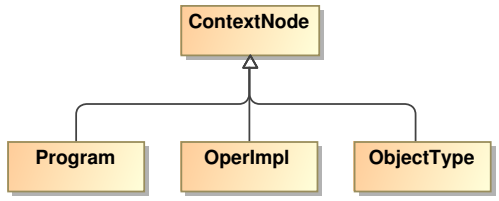


Figure 4.13: Context nodes.

of the start expression of the program. This expression may require other kinds of statements to be executed. A program's initial expression will typically be built up from an expression which instantiates an object on which a method will then be called (this is the case in the example shown in Section 4.3). A program graph always contains exactly one flow graph at Program context.

ObjectType context: ObjectType flow graphs are traversed when an object is instantiated. Object creation will be discussed in detail in Section 4.7. Now it becomes clear why the VarDecl-element is a subclass of ProcedureNode: instantiating an object means that its instance variables need to be created and assigned their initial value. Instance variables declarations are represented by VarDecl-elements and therefore VarDecl-elements are part of ObjectType flow graphs. A program graph contains a ObjectType flow graph for each ObjectType being specified in the original program.

OperImpl context: OperImpl flow graphs control the execution of the body of operations. The execution of an operation can be split up into a number of phases: (1) evaluating the parameters, (2) calling the operation, (3) looking up the corresponding implementation, (4) passing the actual parameters, (5) instantiating the local variables and (6) executing the body. An OperImpl flow graph only takes care of the last two phases. The first and second phase are part of another flow graph in which the operation is called. The third and fourth phase will be discussed in detail in Section 4.7.3. A program graph contains a OperImpl flow graph for each operation that has been implemented in the original program, including the operations implemented in the standard library.

Remark. There is one structural constraint on flow graphs as they appear in our work, that cannot be specified in the meta-model shown in Fig. 4.10. For any FlowElement, incoming flowNext-edges and incoming flowTrue-edges are mutually exclusive. If a FlowElement has an incoming flowTrue-edge this means that it is part of a BlockStat which will only be executed if a particular condition evaluates to `true`. Since Taal does not provide plain *goto* statement that could enable to jump to any point in the program, as for example available in the C programming language [116], there is no other way to reach this FlowElement during simulation of the program.

4.6.2 Flow Graph Construction

Although in this chapter, the focus is on defining the actual execution semantics, we will briefly give some insight in how PGs are constructed from FASGs by applying the rules in taal-flow (recall Fig. 4.1), and discuss some of the important characteristics of this process. First we describe the general principle on which this process is based, after which we will discuss some actual rules. An overview of all the flow graph construction rules is given in Appendix C.

Eventually, the control flow semantics of all flow elements must have been constructed. The generation of control flow graphs can be done in a *top-down* or *bottom-up* fashion with respect to the abstract syntax graph. We have specified the graph production rules to generate control flow in a top-down fashion. This makes the specification of the rules simpler and results in more intuitive rules, for reasons that are similar to why we have chosen to have a separate control flow generation phase.

Flow graphs are constructed at three distinct contexts, as discussed above. Constructing flow graphs in a top-down fashion is achieved by specifying that every flow element triggers the construction of the flow semantics of its sub-flow elements, starting from the context nodes. The actual "trigger" is represented by a special edge labelled flowIn. In some cases, for example for the WhileStat, we need an intermediate flow element which serves as a temporary bridge between syntax elements for which insufficient information is locally available to determine the correct flow of control. Such a temporary bridge is formed by a node labelled FlowTmp. When also specifying the structure of flow graphs during construction, the meta-model from Fig. 4.10 should be extended with the above mentioned elements, as shown in Fig. 4.14. In the following paragraphs we will briefly discuss some flow graph construction rules.

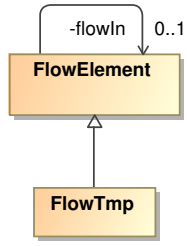
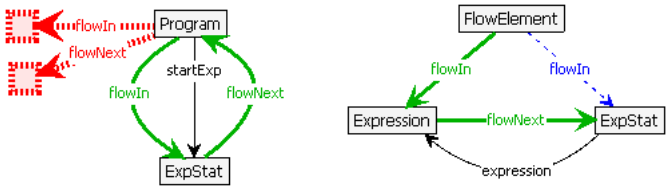


Figure 4.14: Extension to the meta-model from Fig. 4.10

Program. The construction of the Program flow graph starts with application of the rule depicted in Fig. 4.15(a). This rule specifies that if the only Program-node of a Taal program has no outgoing flowIn-edge and no outgoing flowNext-edge, the flow graph construction process has not yet started nor finished, respectively. In that case, the top-down flow graph construction process is started by creating a flowIn-edge to the ExpStat reached by the startExp-edge, and a flowNext-edge in the reverse direction. The created flowIn-edge now triggers the rule that defines the control flow semantics of the ExpStat. This rule is depicted in Fig. 4.15(b). This rule also clearly indicates how flow graphs are constructed using a top-down approach.

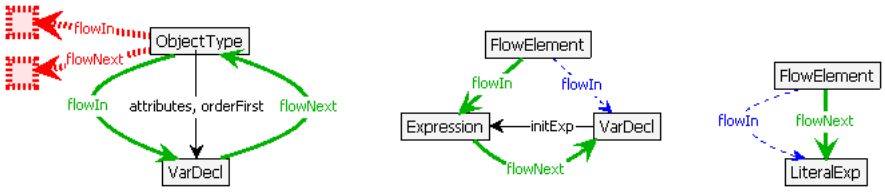


(a) Program. **(b)** ExpStat.

Figure 4.15: Flow graph construction rules (part I).

Object Types. The flow graph of an ObjectType consists of the attribute declarations and their initial expressions. When an ObjectType is instantiated, the order in which its attributes are initialized reflects the order in which they appear in the original program. The rule that initializes the flow graph creation of an ObjectType-instance, depicted in Fig. 4.16(a), is very similar to the one for a

Program-instance. An application of the ObjectType-rule triggers the creation of the control flow semantics of an attribute declaration attached to the matched ObjectType-instance, as represented by a VarDecl-instance. Again, the rule for VarDecl flow elements, depicted in Fig. 4.16(b), obeys the top-down approach by "passing through" the flowIn-edge to the Expression representing the initial expression of this attribute declaration. Note that this rule explicitly requires the existence of an initial expression for every declared attribute. The rule shown in Fig. 4.16(c) specifies that for LiteralExp-instances, the flow graph construction process terminates, since these are considered the *leaves* in the syntax graph with respect to the control flow semantics of Taal. That is, when constructing the control flow semantics of a LiteralExp-instance, the flowIn-edge is replaced by a flowNext-edge and no new flowIn-edge is created.



(a) ObjectType. (b) VarDecl. (c) LiteralExp.

Figure 4.16: Flow graph construction rules (part II).

While Statement. In order to give some insight in how we deal with cases in which the flow of control may branch, we discuss the rule that defines the control flow semantics of the WhileStat. The rule is depicted in Fig. 4.17. Before the WhileStat itself will be simulated, its condition must first be evaluated. A straightforward way to achieve this would be to redirect the flowIn-edge to the Expression pointed to by the condition-edge. The problem now arises that the condition must also be reevaluated whenever the body of the WhileStat has been executed. Solving this in the usual way would mean to also create a flowIn-edge from the body-Statement of the WhileStat to its condition. This would then result in the conditional Expression having two incoming flowIn-edges. However, a flow element with two incoming flowIn-edges gives rise to two matching of the corresponding flow graph construction rule. To ensure that the conditional expression has exactly one incoming flowIn-edge, we introduce a temporary FlowTmp node. The syntax elements after which control should flow to the Expression, now first point to the FlowTmp node with usual

flowNext-edges. As soon as the outgoing flowIn-edge of the FlowTmp nodes has been replaced by a flowNext-edge, the FlowTmp node will be merged with its flowNext-successor, thereby ensuring that all its incoming successor-edges point to the proper syntax element. A similar issue arises for the body of the WhileStat. When the condition of the WhileStat evaluates to `true`, control must flow along the flowTrue-edge, triggering the simulation of the body of the WhileStat. Triggering the flow graph construction of the body of the WhileStat, however, requires corresponding Statement to have an incoming flowIn-edge. Again, we introduce a FlowTmp node as temporary bridge between the flowTrue-edge that will eventually point to the first syntax element of the body and the flowIn-edge that triggers the body's flow graph construction process. Finally, note that the flowNext-edge from the WhileStat to its "next" flow element is replaced by a flowFalse-edge.

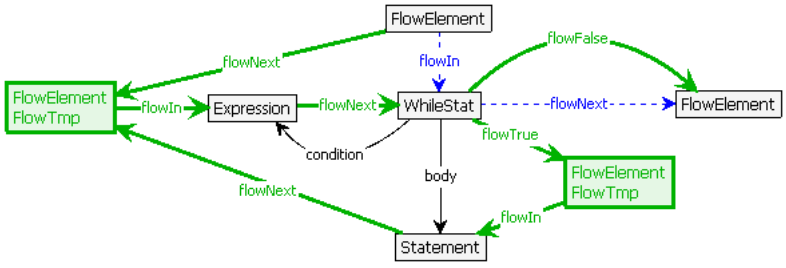


Figure 4.17: Flow graph construction rules (part III).

Connectedness of Flow Graphs

Although one possibly expects that program execution is modelled by one single flow graph, a PG contains several flow graphs at the different contexts. The different flow graphs are not directly connected to each other. They will be semantically connected to each other when simulating the program. For instance, the Program flow graph is connected to an ObjectType flow graph by executing a CreateExp expression; the Program flow graph will also be connected to an OperImpl flow graph, when a specific operation of the instantiated ObjectType is called. OperImpl flow graphs can also be connected to each other. This occurs when one OperImpl flow graph contains a method call to another operation (potentially to itself in a recursive fashion). The connection between flow graphs will be discussed in more detail in Section 4.7.3.

Confluence and Termination

We have specified a set of flow graph construction rules, in the sequel referred to as TAAL-FLOW, that properly enriches the FASG of any Taal program with flow graphs, thus resulting in a Program Graph (PG). Since each Taal program should correspond to exactly one PG, we must guarantee that the graph production system consisting of the the set TAAL-FLOW and an arbitrary FASG *terminates* and is *confluent*.

Essentially, in MDA [143] terminology, the transformation of an FASG of a Taal program into the corresponding PG can be interpreted as a *model transformation*. Typically, a model transformation is defined by specifying how every language construct in the *source language* translates to a language construct in the *target language*. When all instances of the source language constructs have been properly translated, the model transformation has finished and the result is often a unique model of the target language.

Although we have not formally proven termination and confluence of TAAL-FLOW (when applied to an arbitrary FASG), we will now intuitively discuss why we are confident that this is indeed the case. Let us first get convinced of the fact that TAAL-FLOW is confluent. If we can prove that every pair of simultaneous rule applications, i.e., rule applications sharing their host graph, are *local confluent*, we may conclude (global) confluence of the whole graph transformation system, due to standard theory on rewrite systems (see, e.g., [173]). As mentioned above, flow graphs are constructed at different contexts. More importantly, those individual flow graphs are mutually disconnected. From this we may conclude that the construction of the individual flow graphs are independent of each other. That is, any pair of rule applications involved in the construction of distinct flow graphs are local confluent. The next step is to consider rule applications that together construct a single flow graph. Due to the fact that flow graphs are constructed in a top-down fashion, the number of flowIn-edges typically determine the number of rule applications. Initially, there are no flowIn-edges. The rules for each of the context nodes introduce flowIn-edges. Most of the rules "pass through" this trigger down the syntax graph, thereby triggering a single successive rule application. Examples of this have been shown in Fig. 4.15(b) and Fig. 4.16(b). The control flow semantics of some Taal features, however, are slightly more complex since such features consists of multiple sub-components. An example of such a feature is the WhileStat of which the flow graph construction rule has been shown in Fig. 4.17. An application of this rule introduces two flowIn-edges, one for each sub-component, namely its condition and its body. Since both sub-components correspond to

syntactically disjoint parts of the program (and therefore involve disjoint parts of the FASG), the rule applications that construct the sub-flow graphs for the WhileStat's condition and body are independent. Summarizing, since all pairs of simultaneous applications of rules in TAAL-FLOW are local confluent, we may conclude that any graph transformation system in which TAAL-FLOW is applied to an arbitrary FASG, is confluent.

Argumenting why applying TAAL-FLOW to an arbitrary FASG terminates, is a bit more straightforward. There are at least two ways to do this. Since the flow graph construction process propagates in a top-down fashion with respect to the syntax graph, the process has finished if all the leaves of the syntax graph have been processed. The fact that every TAAL program is represented by a finite FASG then indicates that the flow graph construction process terminates somewhen. Alternatively, we could keep track of the number, say n, of syntax elements that have not yet been processed. Obviously, from the start all syntax elements (which are finitely many, suppose N) still have to be processed, i.e., $n = N$. When $n = 0$, all syntax elements have been processed and the flow graph construction has finished. Since every application of a rule in TAAL-FLOW construct the local flow graph for the syntax element under transformation, in the target state the value of n has decreased by one. We may therefore conclude that it takes finitely many steps to fully construct all flow graphs and thus the corresponding PG. Stated differently, the PG construction process terminates.

4.6.3 The Flower and Vase Example

Applying TAAL-FLOW to the graph representing the FASG of the example from Listing 4.2 produces a graph transition system in which every path ends in the same final state (due to the fact that TAAL-FLOW is confluent and terminates), which represents the graph modelling the PG. Since the rule applications creating flow graph elements for different flow graphs do not interfere with each other, they can be interleaved in any possible order. As mentioned above, they constitute a *globally confluent* graph transition system, which intuitively means that all paths will finally reach the same state, which in our case happens to be a *final* state in which no rule is applicable anymore. This state represent the PG of the program to be simulated. Due to TAAL-FLOW being confluent it suffices to execute only a single sequence instead of generating the entire graph transition system.

Fig. 4.18 again shows the FASG of the Flower type declaration with additional control flow edges that have been created during the control flow construction process (compare with Fig. 4.9). In this figure we can clearly see that during

simulation, control will reach the initial expression of the instance variables just before it reaches the syntax element representing their declaration.

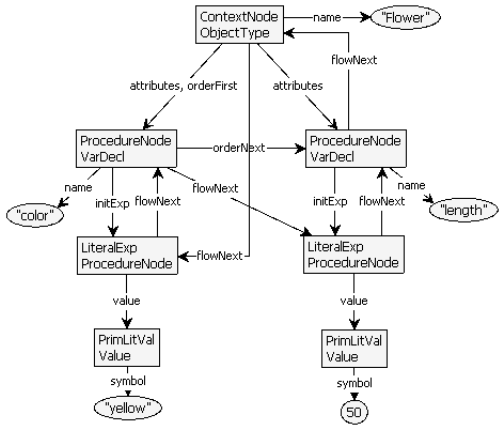


Figure 4.18: Excerpt from the PG showing the flow graph for type Flower.

4.7 Execution Graphs

The simulation of the program is represented by a sequence of *Execution Graphs* (EGs). This sequence is generated by applying a set of graph production rules, modelling the operational semantics of Taal to the PG of the actual program. That is, the PG is the start graph of the simulation. Every intermediate graph reached during transformation represents a particular state of the program.

During simulation, we need some graph structure which keeps track of the objects that have been created and the values of their attributes as well as intermediate results of algebraic calculations that will be performed. Furthermore, we have to introduce some notion of a *call stack* which stores the information about what method is called from where. Summarizing, an EG then combines three kinds of information:

- the PG, described in Section 4.6, which provides the required static semantics;
- a *Value Graph*, modelling the data part of the current state; in compiler terms, this roughly corresponds to the *heap*

- a *Frame Graph*, modelling the process part of the current state; in compiler terms, this corresponds to the *stack*.

In the remainder of this section we will discuss the structure of the value graph and the frame graph. Thereafter, we discuss the rules which specify the actual operational semantics of Taal. For this, we categorize the different rules according to different processes that occur during program simulation such as e.g. *object creation* and *method invocation*.

4.7.1 Value Graphs

The part of an Execution Graph (eg) that forms the Value Graph (vg) consists of graph elements that model the run-time data values which partially identify the program state and the 'memory locations' at which those values are stored. In Fig. 4.19 the meta-model of vgs is shown. In the following paragraphs we will discuss the concepts that are introduced in this meta-model.

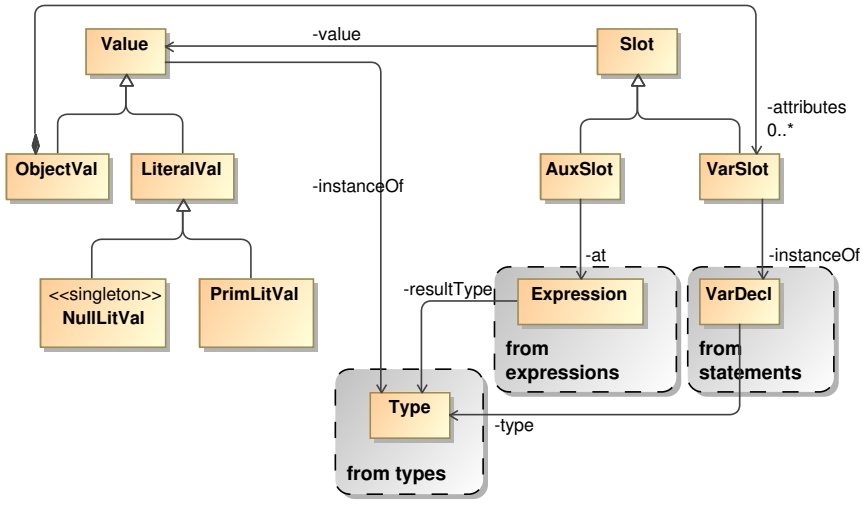


Figure 4.19: Value graph meta-model.

Value. The class Value stands for a concrete value of an arbitrary type. The type of each value is fixed and indicated by an instanceOf-edge leading from

the Value-node to the corresponding Type-node (already introduced in Section 4.3.1). Note that this refers to primitive values as well as the null-value and object-instances; in fact the instanceOf-edge could be specialized to lead from ObjectVal to ObjectType, etcetera. The class Value is abstract.

ObjectVal. The class ObjectVal is a specialization of Value used to model values of ObjectType. An object value may have attributes, which are modelled by VarSlots referenced through the attributes-association. Each ObjectVal-node has precisely one VarSlot (see below) for each attribute of its ObjectType (which are given by the attributes-association of the ObjectType). In other words, for each ObjectVal-node there is a one-to-one correspondence between the VarSlots appearing as attributes-targets, and the VarDecl-nodes appearing as attributes-targets of the the corresponding ObjectType.

NullLitVal. The class NullLitVal stands for the one and only null value, i.e., the unique value of the NullType. In other words, there will always be precisely one instance in the Execution Graph.

PrimLitVal. An instance of the class PrimLitVal stands for a value of some primitive type. Conceptually there are infinitely many of them. Of course we cannot represent all those; instead, an Execution Graph will contain precisely those primitive values that actually occur as values anywhere in the current state.

Slot. Instances of the class Slot contain a Value, occurring anywhere in the state snapshot, as referenced by the value-edge. There are two kinds of slots: VarSlots, which correspond to variables declared in the program, and AuxSlots, which are temporary slots used to store intermediate values during evaluation of expressions. This class is abstract.

VarSlot. The class VarSlot is a specialization of class Slot, viz. a container corresponding to a variable in the program. Above we have already seen that objects have corresponding VarSlot-instances for all their attributes. Similarly, in the Frame Graph we will see that formal parameters and local variables also give rise to VarSlot-instances. For each instance of these kinds of VarSlot, there is a corresponding VarDecl.

AuxSlot. An instance of the class AuxSlot is a container holding a temporary value during evaluation of an expression. These traditionally correspond to stack locations. AuxSlots have no corresponding VarDecl; instead, each AuxSlot refers to the sub-expression for which it holds a value, via an at-edge.

An example value graph is given in Fig. 4.20. This shows three objects with attributes, one of static type Flower and dynamic type Rose and two of (static and dynamic) type Vase. One of the Vases holds a reference to the Rose with its myFlower-attribute. The actual values of the attributes are represented by PrimLitVals with associated actual values. Note that one of the myFlower-attribute of one of the Vases has not yet been initialized, i.e., the myFlower-attribute has value null.

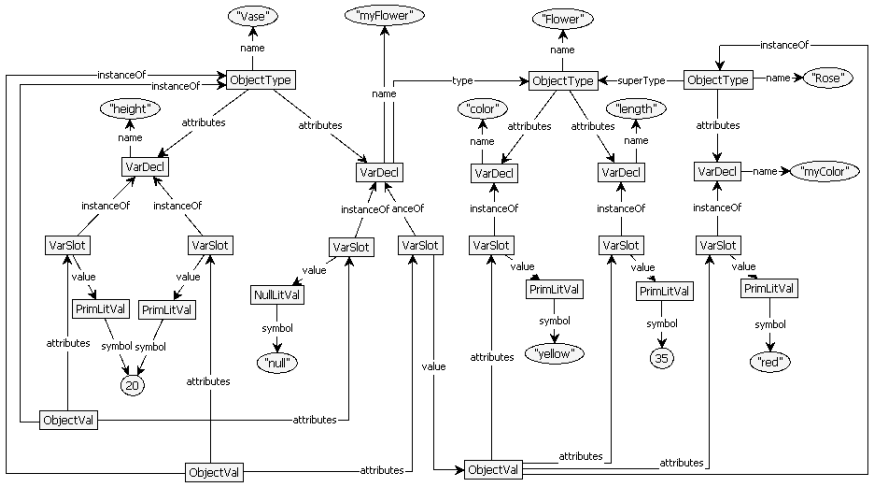


Figure 4.20: Fragment of the Value Graph reached after executing the program from Listing 4.2.

4.7.2 Frame Graphs

Execution Graphs also comprise graph elements that reflect the call stack of the program at any point during simulation. Those elements together form what we will call the *Frame Graph* (FG). The Frame Graph meta-model is shown in Fig. 4.21. It essentially introduces only one new type of node, namely Frame. Every Frame-instance directs the execution of a flow graph. The three

types of ContextNodes thus give rise to three different subtypes of Frames: the ProgramFrame, the OperFrame, and the ConstrFrame. Each of the Frame-types will be discussed in more detail in the following paragraphs.

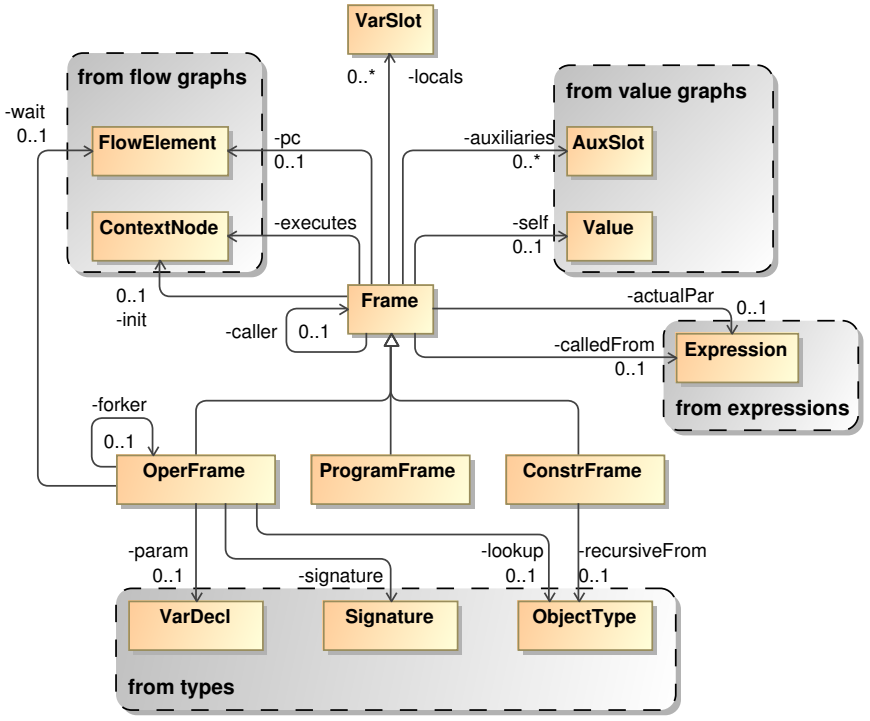


Figure 4.21: Frame graph meta-model.

The class Frame is the main type for execution frame nodes. In general, a Frame controls the execution of the code at a particular ContextNode. In terms of the program graph, that code can be found as the control flow graph appended to the relevant ContextNode. Each Frame has an executes-reference to the corresponding ContextNode.

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 the ContextNode. 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.

In fact, each frame that does not correspond to a Program has a calling frame, referenced by a caller-edge. Moreover, where the caller gives the semantic calling context, there is also a *syntactic* calling context, viz. the location in the flow graph of the caller where its pc-edge was pointing when the current Frame was invoked. This syntactic context is stored in a calledFrom-reference. When the nested frame has finished, it is deleted and a pc-edge is re-created in the calling frame, using the calledFrom-reference to determine the correct location.

A further important aspect of Frame-instances is that they can have local and auxiliary *variables*. The former are instances of the node type VarSlot, discussed above as part of the value graph; they are referenced through a one-to-many locals-association. (Actually, the current version of Taal has been defined such that not all types of Frame have local variables – in fact, only OperFrames do – but we regard this as a coincidence.) Auxiliary variables are instances of AuxSlot, also discussed above, referenced to by auxiliaries-edges. A Frame may have a self-edge to the value that is the context of the operation being executed. This is usually an ObjectVal, but for built-in operations it may be a primitive value; hence the type is Value. (Again, in TAAL, self-edges only occur for OperFrames and ConstrFrames.)

Finally, there is an auxiliary edge labelled actualPar that is used for the purpose of passing parameters at operation calls; see Section 4.7.3 below.

ProgramFrame. The class ProgramFrame is a subclass of Frame controlling the execution of the entire program. To denote that fact, every instance has an executes-edge to the (unique) Program-node. A ProgramFrame is only active when the program starts; the initial statement usually creates an object and invokes a method upon it. When control returns to the ProgramFrame, the program is finished and terminates as we will see in Section 4.7.3.

OperFrame. The class OperFrame is the subclass of Frame that controls the execution of operations. The signature of the operation being executed is known at compile-time for every operation; in this graph this is indicated by a signature-edge. The actual operation implementation being executed (which is looked up dynamically, see Section 4.7.3 below) is indicated by an executes-edge to the corresponding OperImpl-instance. Furthermore, an OperFrame is always *called* from another frame, and hence has a caller-edge to its calling frame.

Furthermore, in order to be able to reconstruct the pc-edge in the calling frame after this one terminates, the OperFrame also records the FlowElement from which it was called, through a calledFrom-edge. OperFrames that are introduced by a ForkStat for introducing parallelism references the OperFrame from which it is forked by a forker-edge. Since we do not allow two operations to be executed on the same object in parallel, a forked operation has to *wait* until its forker Frame has finished execution. This is modelled by the wait-edge. More detail on this will be discussed in Section 4.7.3. In addition, there are a number of auxiliary edges, which are used during the process of method invocation; see Section 4.7.3 below.

- lookup, used to signal the phase of dynamic method lookup, from the start of the method invocation until the appropriate method implementation has been found (see Section 4.7.3);
- init, used to signal the phase of parameter passing, which takes place after method lookup but before the actual execution of the method body (see Section 4.7.3);
- param, which is used during the parameter passing phase to point at the formal parameter being initialized (see Section 4.7.3). The corresponding actual parameter is indicated by the actualPar-edge at the calling frame; see above.

There are several consistency requirements on the local graph structure of an OperFrame:

- all the VarSlot-instances reachable through locals have an instanceOf-edge pointing to a VarDecl that is one of the localVars or param of the corresponding OperImpl;
- the FlowElement pointed to by calledFrom is reachable through a chain of flow-edges from the ContextNode to which the caller-Frame has an instanceOf-edge.

ConstrFrame. The class ConstrFrame is the subclass of Frame controlling the initialization of an object, or in Java terms the execution of a constructor. It shares most of the characteristics of an OperFrame, except that its executes-edge is not pointing to an OperImpl but to an ObjectType. This implies that there are actually no locals-edges, since constructors in Taal cannot have local variables or formal parameters. In addition, a ConstrFrame can also have an auxiliary init-edge, used temporarily when initializing an object. This will be discussed

in more detail in Section 4.7.3 below. When initializing some type which has a supertype, ConstrFrames are stacked using the caller-edge. Finishing initializing for one type, the edges calledFrom en recursiveFrom (which are mutually exclusive) determine whether to continue initialization of a subtype or whether to return the resulting ObjectVal.

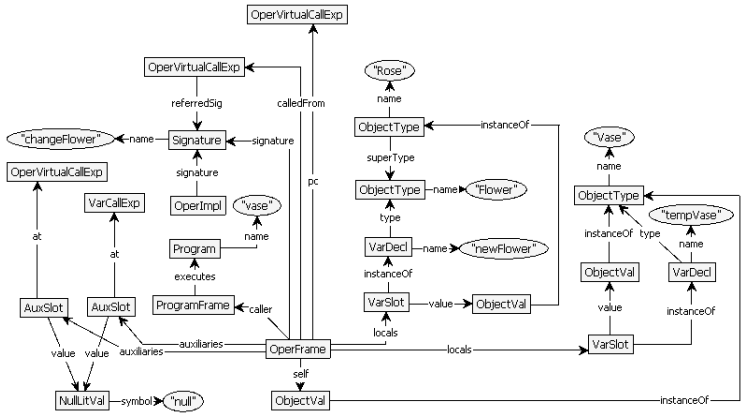


Figure 4.22: Fragment of the Frame Graph reached during simulation.

4.7.3 Program Simulation

The actual simulation of any Taal program is based on a set of graph production rules that specify the operational semantics of the language constructs available in Taal. Each such rule basically mimics the effect of the syntax element in terms of the Value and Frame Graph. For instance, object creation and variable assignment are reflected in the value graph, whereas method invocation is reflected mainly in the frame graph.

In the coming paragraphs we will discuss the transformation rules describing the operational semantics of the most important language construct. The remaining language constructs have simulation rules that have a lot in common with one of the discussed rules. We distinguish the following categories of rules:

- *statement execution*, which occurs when the pc-edge points to a Statement node;
- *expression evaluation*, which occurs when the pc-edge is pointing to an

Expression node. For most types of Expressions, its evaluation mainly involves the removal and creation of AuxSlots;

- *flow graph termination*, which occurs when the pc-edge point to a ContextNode of any flow graph;
- *object creation*, which is triggered by an CreateExp;
- *method invocation*, which is triggered by an OperCallExp.

4.7.4 Statement Execution

This section contains the description of transformation rules that specify the operational semantics for a number of Statement-types. The rules are shown in Fig. 4.23.

AssignStat. The effect of an AssignStat is to make a variable (modelled by a VarSlot) point to a pre-computed value, namely the rightHandSide of the assignment. We have to distinguish between local and instance variables. In either case, the assignedVar (possibly together with the AuxSlot at the source-referenced Expression) uniquely identifies a VarSlot-instance; this receives the value of the AuxSlot at the rightHandSide. The AuxSlot-instances involved are subsequently discarded. In Fig. 4.23(a) we have shown the rule corresponding to the case of a local variable.

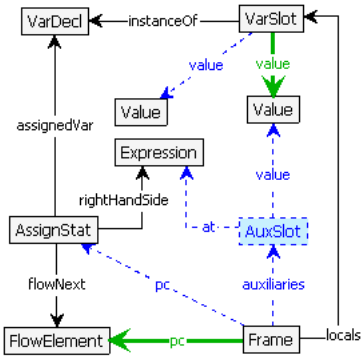
ExpStat. The semantics of an ExpStat merely consists of discarding the AuxSlot at the Expression pointed to by the expression-edge and moving the pc-pointer forward. The rule is not shown since it is not very interesting.

PredicateNode. When the program counter reached a PredicateNode (i.e. either a ConditionalStat or a WhileStat), its condition has already been evaluated. This means that the semantics of the PredicateNode consists of propagating the pc-edge along the proper flow-edge. That is, the pc-edge will be redirected to the executable element reached along the flowTrue-edge if the condition evaluated to `true`, and otherwise to the syntax element that is reached along the flowFalse-edge. Fig. 4.23(c) depicts the rule for the `true`-case.

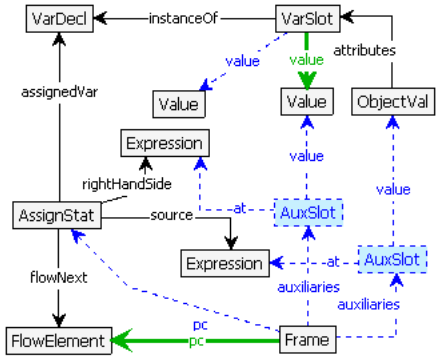
ReturnStat. When executing a ReturnStat, the flow graph of the current frame is terminated, which is always an OperFrame. (A ConstrFrame implicitly terminates upon the end of initialization, and a ProgramFrame upon reaching the

end of the program; both are signalled by the pc-edge reaching the corresponding ContextNode, and not by an explicit ReturnStat.) We combine the detailed description with the discussion of method invocation process.

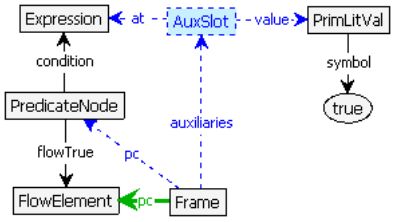
VarDecl. Essentially, when simulation reaches a VarDecl, a VarSlot-instance has to be created and the value of an initial expression has to be assigned to it. Like with the VarCallExp and AssignStat, we have to distinguish between instance variables and local variables. This can be detected by investigating the context: if the VarDecl is among the localVars of the OperImpl associated with this Frame, it is a local variable, otherwise it must be one of the attributes of the ObjectType of the Value referenced by self. (Note that in either case there is guaranteed to be an initial expression, which indeed has been evaluated before control reaches the VarDecl.) Fig. 4.24(a) shows the rule for local variables; Fig. 4.24(b) shows the rule for instance variables.



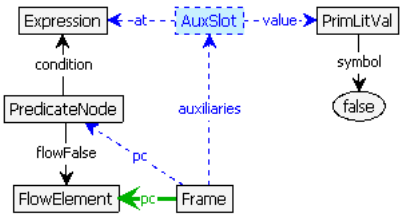
(a) AssignStat – local variable.



(b) AssignStat – attribute.

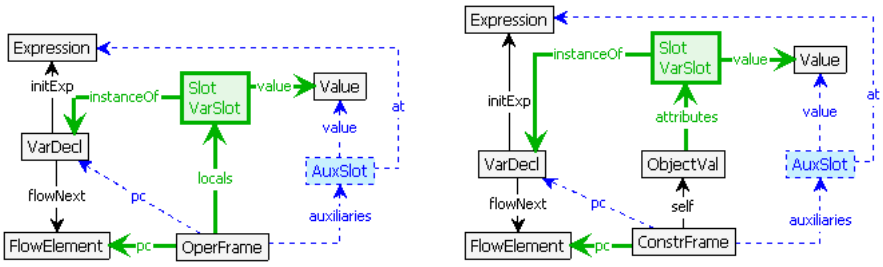


(c) PredicateNode – `true`.



(d) PredicateNode – `false`.

Figure 4.23: Simulation rules for some Statement-types.



(a) VarDecl – local variable. **(b)** VarDecl – attribute.

Figure 4.24: Simulation rules for VarDecl.

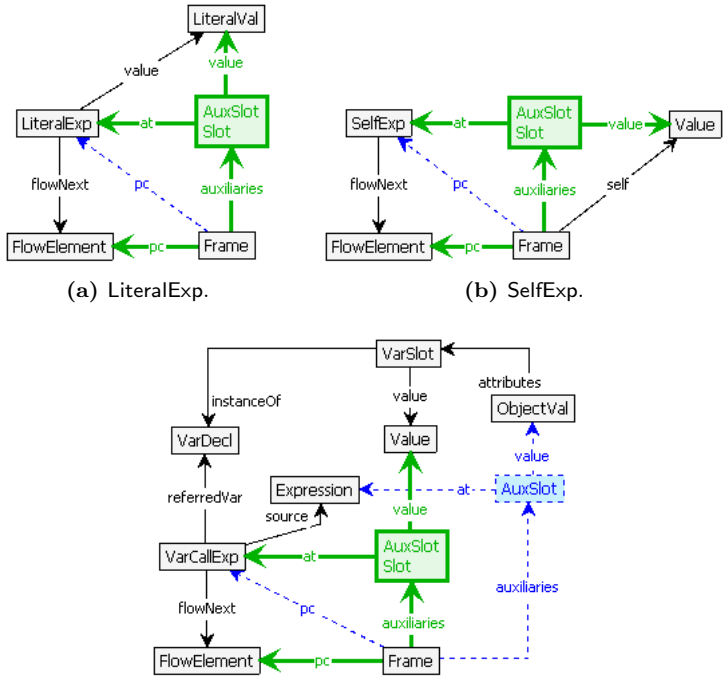
4.7.5 Expression Evaluation

An important choice is to store intermediate values, resulting from the evaluation of sub-expressions, in AuxSlot-instances instead of explicitly modelling a stack-like structure. AuxSlots are connected on the one hand to the sub-expression in question (through an at-edge), and on the other hand to the intermediate value (through a value-edge). Apart from this, whenever an expression is evaluated, the pc-edge is moved to the next FlowElement in the control flow graph. The rules of Expression-types that are part of one of the other categories (i.e. object creation or method invocation) will be discussed later on; the rules for the Expression-types discussed here are shown in Fig. 4.25.

LiteralExp. The semantics of a LiteralExp consists of creating a fresh AuxSlot for the expression and assigning the value identified by the LiteralExp to it. This one rule captures the semantics of both subtypes of LiteralExp, being NullLitExp and PrimLitExp. Furthermore, the pc-pointer is moved forward. The rule is shown in Fig. 4.25(a).

VarCallExp. The semantics of a VarCallExp depends on whether it represents a call to an instance variable or a local variable in the scope of some method. In either case the referredVar identifies a unique VarSlot; execution of the VarCallExp consists of creating a fresh AuxSlot for the expression and assigning the value of the referredVar to it. Furthermore, the pc-pointer is moved forward. In Fig. 4.25(c) we have shown the rule in case the VarCallExp refers an instance variable.

SelfExp. When executing a SelfExp a auxiliary AuxSlot is created containing the value (which is always an instance of ObjectVal) and the program counter is moved forward.



(a) LiteralExp. (b) SelfExp.

(c) VarCallExp – attribute

Figure 4.25: Simulation rules for some Expression-types.

4.7.6 Flow Graph Termination

In Section 4.6 we have raised the issue that every flow graph of a Program Graph is cyclic since the ContextNode of every flow graph is both its start node and end node. The way we deal with the cycles is by having the convention that when simulation reaches a ContextNode this means that the flow graph of this ContextNode has been terminated. In the upcoming paragraphs we will see what the consequences of this convention are when starting the simulation of any flow graph. Here we will discuss the semantics of the three types of ContextNodes.

Program. When control reaches the Program-instance, this indicates that the program has terminated. Nothing happens any more, and we need no transformation rule. Note that we have not implemented any form of garbage collection. On the other hand, since the Program itself has no local variables, and all auxiliary expression values have been consumed, the only execution graph nodes are Value-nodes with, in the case of ObjectVals, VarSlot-attributes. See also Fig. 4.20.

OperImpl. The only case in which control (in the form of a pc-edge) can reach an OperImpl-node is if the operation does not contain any explicit ReturnStat. This, in turn, is only possible if the type of the operation is NullType; so we are safe in treating this situation as a kind of implicit ReturnStat that returns NullLitVal. We show the rule when discussing method invocation.

ObjectType. Control reaches an ObjectType-instance after a new object of that type has been created and initialized; it signals the termination of a ConstrFrame. There are actually two cases: the ObjectType in question may be the actual type of the ObjectVal just created, or it may be a supertype. We discuss the details as part of the object creation process.

4.7.7 Object Creation

Object creation is one important point where the execution of programs with and without inheritance differ. The following paragraphs discuss how we define the process of object creation. Traditionally, in object-oriented languages, object construction is a two-pass affair: first, space is allocated for the object and its instance variables (or in other words, the object and its variables are *created*), and subsequently the instance variables are initialized. However, to avoid the need for distinguishing the state of a variable in between its creation and initialization, it is usually specified that at the time of its creation, an instance variable already receives a *default value*. (Note that it is not decidable whether a variable is accessed before being explicitly initialized.) In turn, the fact that variables receive default initial values turns their explicit (re-)initialization from an absolute necessity into a practical convenience: it is, in principle, always possible to defer initialization to an ordinary method. Indeed, the Java Language Specification [93] states that the execution of a constructor body is internally implemented as a special init method.

In Taal, on the other hand, we have taken a more simplistic approach,

which avoids both the need for default initial values and the need for two consecutive passes. All attributes have an initializing expression; forward references to uninitialized variables are forbidden.[2] In this setting, we can at the same time construct locations for the variables and assign initial values to those locations, provided we take care that this process starts at the top of the inheritance hierarchy. This results in the following subprocesses.

Allocation. The actual object creation occurs when control reaches a CreateExp-instance. A ConstrFrame and an ObjectVal are created straight away. The ObjectVal is referenced through self from the ConstrFrame. Moreover, the fresh ConstrFrame has an init-pointer to the ObjectType, to indicate the fact that we are initializing an instance of this type. The corresponding rule is shown in Fig. 4.26(a).

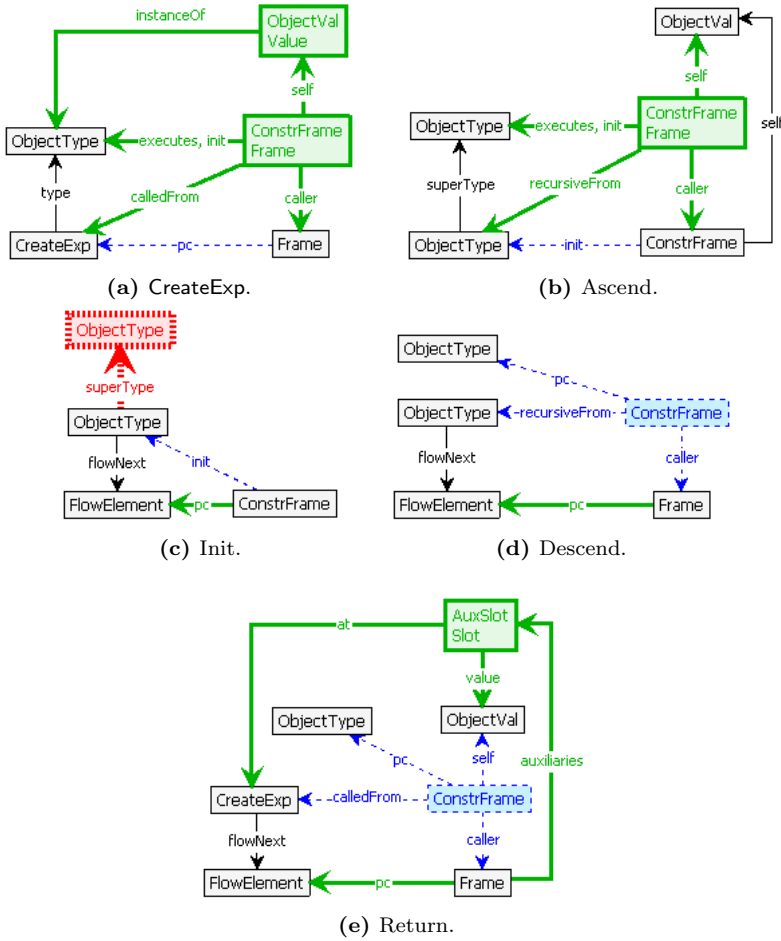
Initialization. A ConstrFrame-instance with an init-edge to an ObjectType will be treated in either of the following two ways, depending on whether the ObjectType has a super type or not. If it has a super type, then a new ConstrFrame is created recursively for that, but with the same self. The new ConstrFrame has a recursiveFrom-pointer to the subtype for which it continues initialization. If the ObjectType has no super type, then execution is started, by replacing the init-edge with a pc-edge pointing to the first FlowElement reachable from the ObjectType. The subsequent simulation rules will compute initial values and assign them to newly instantiated AuxSlot-instances for the ObjectVal.

Termination. A ConstrFrame terminates when the pc-edge has arrived (back) at the ObjectType. The frame is discarded and a pc-edge is (re)created at the caller frame. Just as for initialization, there is a case distinction, depending on whether the current frame was called recursively from a sub-type or directly from a CreateExp.

- If the current ConstrFrame was invoked recursively, there is a recursiveFrom-edge to the ObjectType that models the sub-type from which it was called. This means that initialization now proceeds back down the inheritance hierarchy to that sub-type, and the calling Frame is also a ConstrFrame, which already has a self-edge to the underlying object. In this case no return value is required. The rule specifying this case is show in Fig. 4.26(d).

[2]Note that this is a constraint that is generally undecidable and hence cannot be enforced at compile time.

- If the current ConstrFrame was called directly, there is a calledFrom-edge to a CreateExp. This means that the underlying object, pointed to by self, should be returned to the caller, in the same way as in a ReturnStat (see below). The rule specifying this case is shown in Fig. 4.26(e).



(a) CreateExp. **(b)** Ascend.

(c) Init. **(d)** Descend.

(e) Return.

Figure 4.26: Simulation rules for object creation.

4.7.8 Method Invocation

Method calls are among the basic expressions of a program. Their effect is entirely within the frame graph: the execution of a method call creates a new OperFrame-node, associates it with an OperImpl-instance (which is part of the program graph), passes the actual to the formal parameters, and finally turns over control to it.

Frame Creation. The creation of the OperFrame itself is straightforward: the more difficult parts follow only later. Once the frame is created, control is transferred to it, but not yet in the form of a pc-edge; instead the corresponding method implementation has to be found first. This phase is modelled by an outgoing lookup-edge to the class in which the method is sought. We assume that for every called method an implementation exists either in the class or in one of its super-classes. This can be checked at compile-time.

There are two versions of method invocation: virtual and super. In the first case the lookup starts at the dynamic type of the target object; in the second case it starts at the super-type of the type in which the current method is located. The two cases are shown in Fig. 4.27(a) and Fig. 4.27(b), respectively.

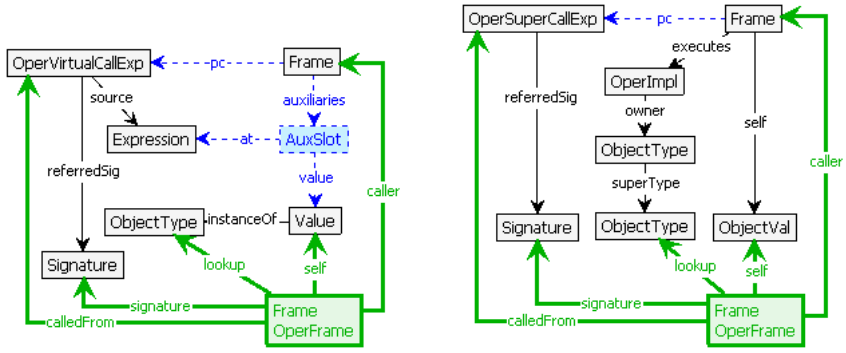


Figure 4.27: Simulation rules for method calls.

Method Lookup. The association of an OperImpl-instance with a freshly created OperFrame is called *method lookup*. In programs without inheritance, method lookup is static. That is, the compiler is able to decide which OperImpl is to be

associated with a given method call, purely on the basis of the Signature. Not so, however, in the presence of inheritance, where the dynamic type of the "target" (the object asked to execute a method) is a factor in determining the relevant method implementation. All the compiler is able to determine statically is the *signature* of the method to be executed (see Section 4.3). This is complemented by a dynamic method lookup protocol whereby classes are queried (as it were) in succession whether they have implemented a method with the given signature, in which case that definition becomes the one to execute; if not, the query is passed on to the next higher class in the inheritance hierarchy. (This part of the protocol could be optimized statically, since each class can store a map from those method signatures implemented somewhere in their super-classes to the "most concrete" implementation; this obviates the need for recursively passing on queries.) The rules for method lookup propagation and resolution are shown in Fig. 4.28.

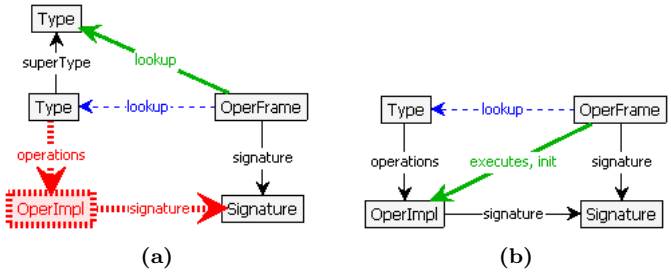


Figure 4.28: Simulation rules for method lookup.

Parameter Passing. Another complication is the need to pass the parameters from the calling Frame to the called, newly created OperFrame. This essentially involves an assignment of all actual parameter values (which are stored in AuxSlot-instances associated with auxiliaries-edges to the calling frame) to the corresponding formal parameters (which are VarSlot-instances associated with locals-edges to the called frame). The correspondence of actual to formal parameters is given only through the *ordering* of the parameters. For that reason, the parameter passing phase is modelled by *four* transformation rules, shown in Fig. 4.29: for starting, continuing and ending the phase, as well as for the case where the number of parameters is zero. To keep track of the current actual and formal parameters during propagation, we have introduced special

actualPar-edges and param-edges pointing to, respectively, the current actual parameter (an Expression) and the current formal parameter (a VarDecl).

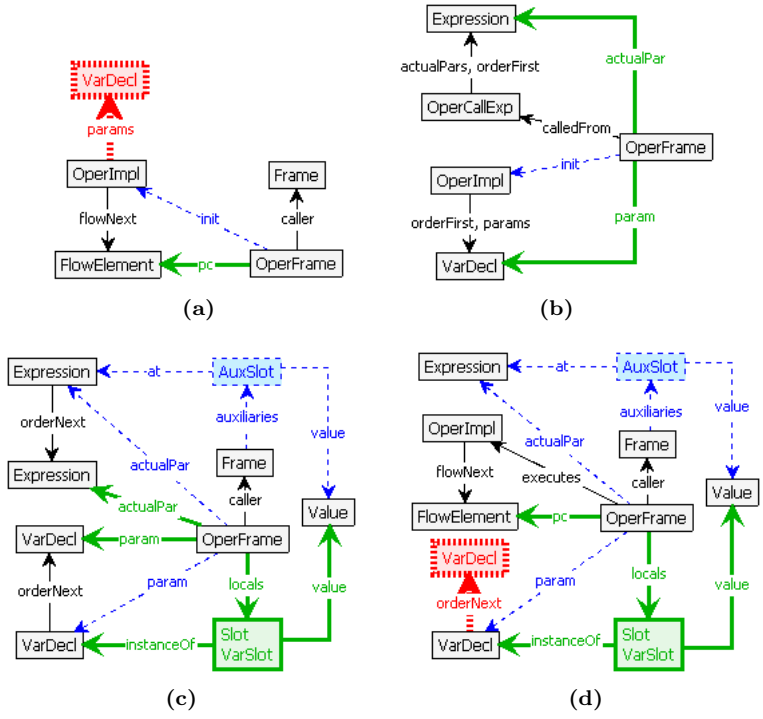


Figure 4.29: Simulation rules for parameter passing.

Return. The final process of a method invocation is to return the value corresponding to the actual invocation. Also this process consists of several steps:

- the value of the AuxSlot attached to the value-referenced Expression will be assigned to a new AuxSlot at the calledFrom-referenced OperCallExp of the calling frame;
- the AuxSlot of the returned value has to be discarded;
- all local variables of the method, represented by VarSlot-instances pointed to by locals-edge, must be garbage collected;

- control must be returned to the caller Frame properly, i.e. a pc-edge is created from the caller Frame to the next FlowElement with respect to the expression from which the current method is calledFrom;
- the current OperFrame-instance is discarded.

Since discarding the locals is a repetitive process, it requires a separate rule; the main rule is only applicable if no more locals-edges exist. The two rules are shown in Fig. 4.30(a) and Fig. 4.30(b), respectively.

A special case occurs if the method has type NullType and no explicit ReturnStat: then instead control eventually reaches the OperImpl-node. The actions necessary to deal with this are analogous to those for the ReturnStat, except that this time the value to be returned is not taken from some expression; instead, it is always NullLitVal. The rules specifying this case are shown in Fig. 4.30(c) and Fig. 4.30(d).

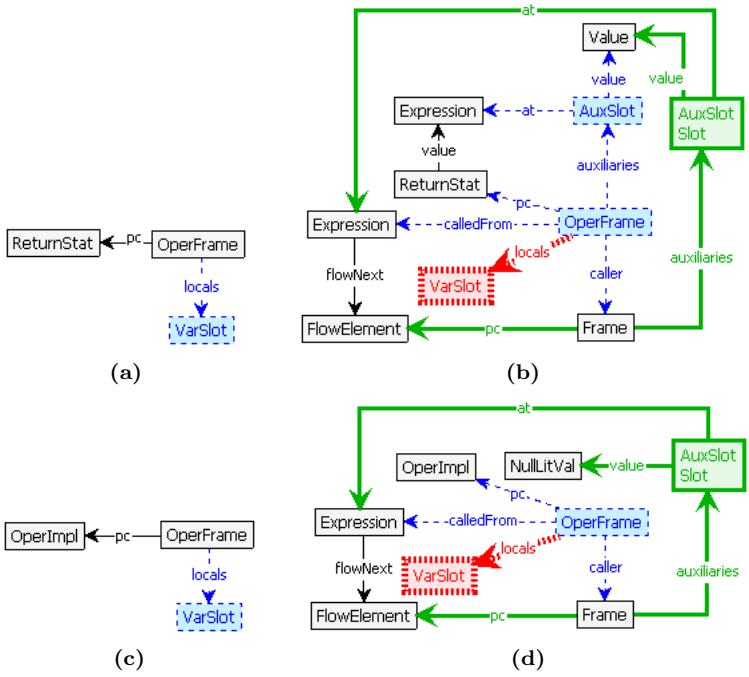


Figure 4.30: Simulation rules for method return.

Parallelism. As mentioned before, parallelism is introduced by a ForkStat. The execution of a ForkStat is comparable to that of an OperVitualCallExp. The pc-edge is replaced by a lookup-edge which guides the process of method lookup given the proper Signature. This is shown in Fig. 4.31(a). The process of parameter passing for a forked method call is also similar to normal method calls, except that after passing the last parameter (or in the case the method has an empty list of formal parameters), a wait-edge is created instead of a pc-edge. This is necessary for preventing from situations in which multiple methods are executing on the same object, which could otherwise result in incorrect results due to different interleavings of statement executions of the running methods. As soon as the forker Frame has finished it execution, the forked method call is allowed to start its execution. This is depicted in the rule shown in Fig. 4.31(b). Termination of a forked method call is represented by either an explicit ReturnStat or by having the pc-edge pointing to the corresponding OperImpl node. The former case is treated like discussed before; for the latter case we have specified a special rule which is shown in Fig. 4.31(c).

Primitive Type Operations. Method calls to operations of primitive types will be dealt with in a special way. We assume that every primitive type provides implementations for the operations declared. Therefore, we do not need to perform method lookup. Basically, such operations consist only of applying the algebraic operation on the current object and the values in the list of actual parameters. Those values are all easily accessible using context information of the OperCallExp. The resulting value can then be determined using the *algebra graph* as explained in Chapter 3. For this to work, transformation rules have to be specified for all the primitive type operations that are supported in the language. In Fig. 4.32 we have shown the transformation rule for the "greater than" operation of the primitive type Integer.

4.7.9 Examples

In the following paragraphs we show the result of simulating the example programs shown in Listing 4.2 and Listing 4.3. Since the first program consists of only a single thread, one would expect the corresponding Graph Transition System (GTS) to contain a single path from the start state (i.e. the Program Graph) to the end state (i.e. the final Execution Graph in which the program counter is located at the Program node). In Fig. 4.33, the GTS is shown. One issue that will immediately draw the readers attention is the branching in the final part of the GTS. When taking a closer look at the rules that cause the

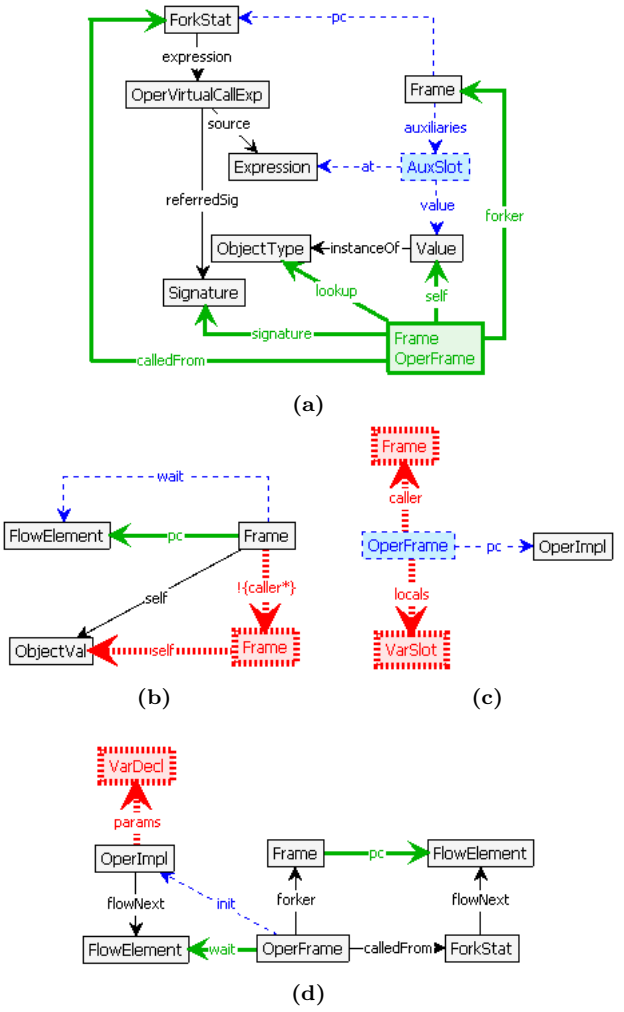


Figure 4.31: Simulation rules for introducing parallelism.

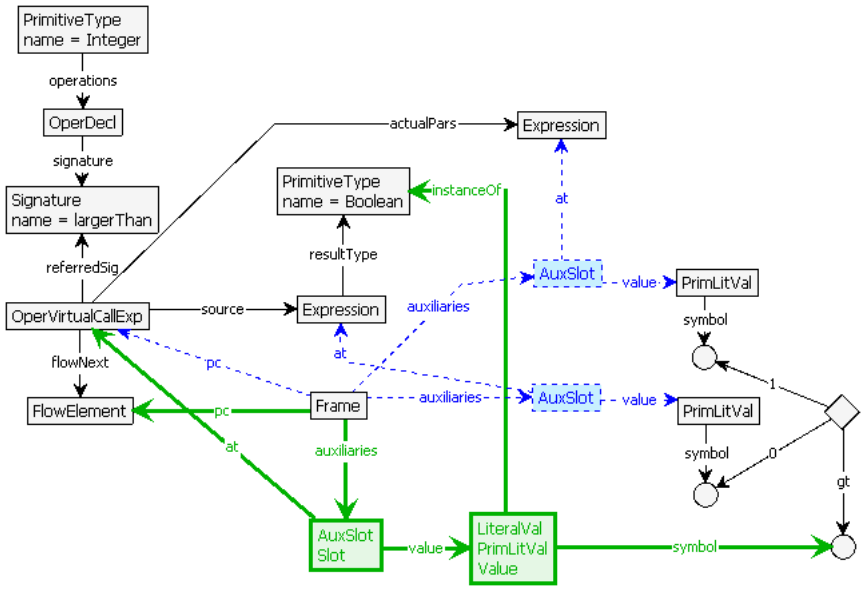


Figure 4.32: Simulation rule for a specific primitive method call.

branching, we can see (although not in the figure — it is simply not readable) that this involves rules that perform some local garbage collection, e.g. removing local variables of a method just before executing its return statement.

When comparing the textual program with the rule applications as appearing in the GTS, we can also get some more insight in which parts of the GTS model particular processes such as e.g. object creation and method invocation. In the GTS we have grouped some rule applications by surrounding them with a red dashed box. Each box is furthermore accompanied by a short description of what process the rule applications in that box constitute. The GTS in Fig. 4.34 shows two red boxes that contain the execution steps of the same simulation of the clone-method. Due to parallelism, its simulation appears twice in the state space. The only difference is that the right box represents the simulation that is preceded by a simulation step of an ExpStat-instance which happens to be the last simulation step of the remaining program. If there would have been more simulation steps to be performed, the GTS would contain (much) more branching.

The program from Listing 4.3 yields a graph transition system, shown in

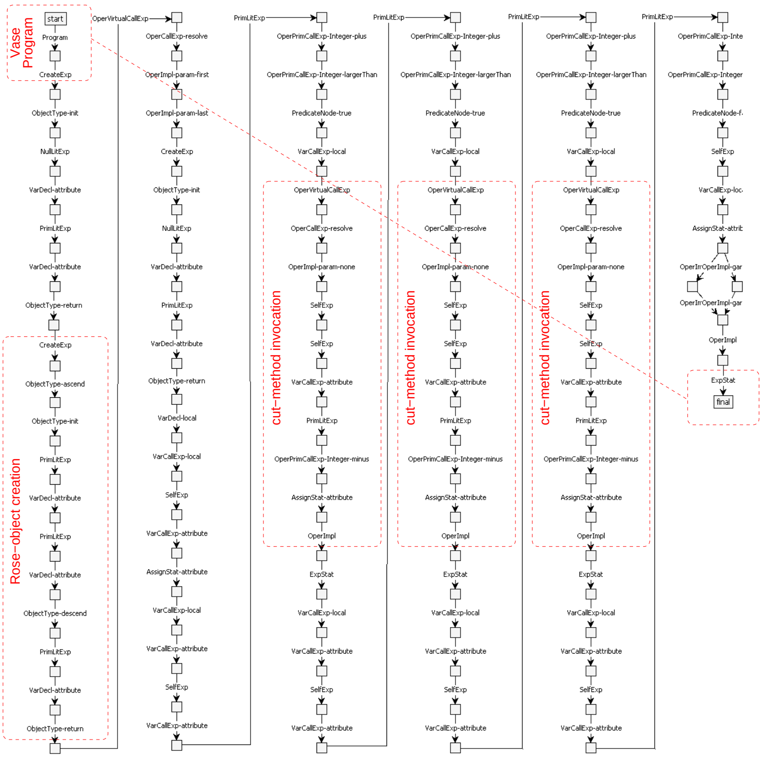


Figure 4.33: Graph transition system modelling the simulation of the program shown in Listing 4.2.

Fig. 4.34, in which branching occurs approximately halfway the simulation of the program. This branching is due to the parallelism introduced by the fork statement. We can clearly see that the simulation steps performed by both threads are independent based on the many diamonds that occur in the state space.

4.8 Analysis on TAAL Programs

This work mainly focussed on developing a formal and intuitive semantics for a object-oriented programming language. In the previous section we have shown

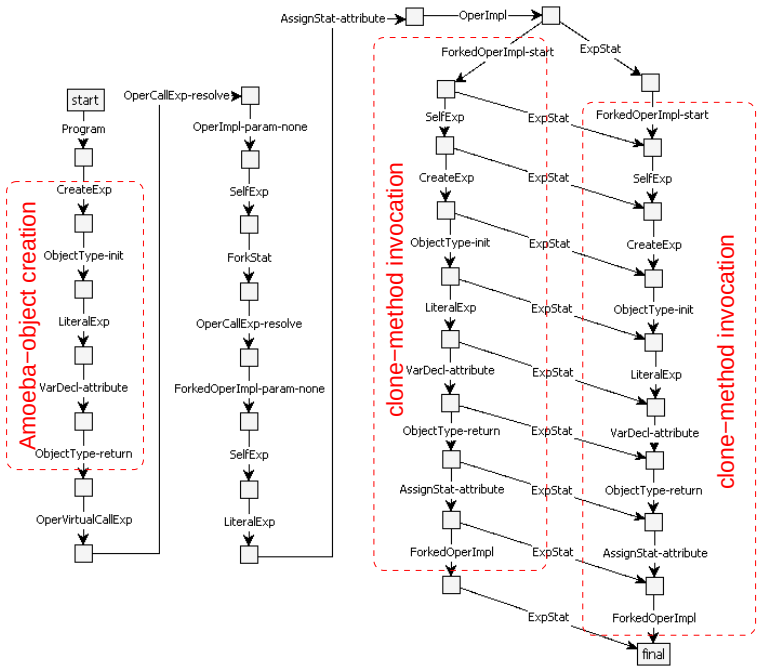


Figure 4.34: Graph transition system modelling the simulation of the program shown in Listing 4.3.

some results of simulating actual Taal programs, i.e., the graph transition systems produced by the simulation. The next step would be to perform various kinds of analysis on those transition systems.

A first analysis could be to determine whether the simulation of a specific Taal program produces a finite transition system. That is, given an a priori fixed amount of resources, e.g., time and memory, does the simulation produce all the reachable states of the program. Due to the isomorphism check in Groove, the fact that a simulation produces a finite transition system does not mean that the program always terminates. Whenever simulation reaches a state that is isomorphic to some state seen before, the transition representing the current rule application will point back to that isomorphic state. This might introduce loops in the program's simulation. The program can potentially execute such loops infinitely often and therefore produce non-termination execution

paths.

A second possible kind of analysis is of syntactic nature. During the simulation of the program, the transformation rules could annotate the parts of the program that have been executed. Provided that the simulation produces a finite transition system, one could investigate which part of the program has not been involved in the simulation. Currently, Taal does not provide any means of interaction with some environment or language constructs for including some form of randomness. Therefore, the outcome of the simulation of a Taal program is deterministic and parts of the programs that have not been reached during simulation can be regarded as *dead code*, i.e., code that will not be executed by any simulation of that particular program.

Since the transitions in the graph transition systems produced by Groove carry information about which rule has been applied, we can analyze the simulation of Taal programs on a more semantic level. That is, we can perform analysis on the temporal ordering of applicability of specific rules. In Chapter 5 we will further elaborate on this kind of analysis and distinguish between rules that modify the graphs on which it is applicable and those that do not.

As we have mentioned in Section 4.1, in the context of MDA, research is carried out on proving model transformations being behaviour-preserving. Engels et al. [79] have implemented a model transformation from UML Activities to Taal programs. That is, from the graph encoding of a UML Activity the model transformation produces an fasg that represents the corresponding Taal program. Engels et al. [81] have studied the semantics of UML Activities and implemented them in Groove graph transformation rules. By then comparing the transition system generated for a UML Activity diagram and the transition system of the corresponding Taal program, Engels et al. have shown that the ordering of actions in the former coincide with the order of corresponding methods in the Taal program.

4.9 Extensions

4.9.1 Graph-Based Semantics for the .NET Intermediate Language

In the previous sections we have given a formal semantics to an artificial object-oriented language called Taal. In order to get more insight in the generality of our approach, a master project [176] has been carried out under our supervision in which the above described approach has been applied to programs compiled to the .NET Intermediate Language (.NET IL) [139]. For this study, we have

chosen the .NET IL since it 'represents' a set of programming languages. That is, the basic idea behind the .NET Framework is to provide a simple though powerful framework that is able to execute programs written in many different languages by one single virtual machine. Basically, the .NET IL is a language which provides an intermediate format for a set of languages, among which C♯, J♯, VB .NET, Eiffel, and Perl, which have many features in common. For specifying the dynamic semantics of the .NET IL, we applied essentially the same steps as discussed in previous sections, but using a different (abstract) syntax. In the following paragraphs we will shortly discuss the main differences with the way our approach has been applied to Taal. For more details, the interested reader is referred to the MSc thesis [176].

Instructions versus Statements

In Fig. 4.35 we have shown a excerpt from the meta-model of the .NET IL as taken as a starting point in this study.

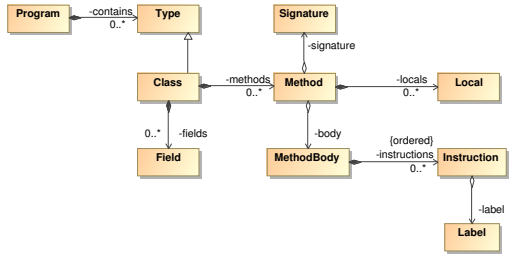


Figure 4.35: Excerpt from the meta-model of the .NET IL abstract syntax.

When comparing this diagram with the one depicted in Fig. 4.2 there are quite some commonalities. For example, both diagrams contain a class Program modelling the "root" of the program under simulation. Both languages feature the concept Type. Other concepts that serve comparable purposes are Method and OperImpl, and Class and ObjectType. Whereas in Taal there is only one concept VarDecl that provides means to specify variables at both object and method level, in the .NET IL there is a distinction between Fields and Locals, where obviously Fields represent instance variables and Locals represent local method variables.

A major difference is the way the basic executable elements are incorporated. In Taal, this has been done by distinguishing different types of Statements and

Expressions (see Fig. 4.3 and Fig. 4.4, respectively). In the .NET IL the basic executable elements are Instructions. An Instruction in the .NET IL can be compared with a Java byte code instruction, which resides on a lower level of abstraction than the Statements and Expressions as provided in the (concrete and abstract) syntax of Taal. Typical Instructions only involve moving or copying data values from one place in memory to another or perform simple algebraic operations on the elements that are on top of the *execution stack*. In graph terms, the former case corresponds to replacing or creating edges that reference specific data nodes in the graph. Whereas Taal comprises around twenty different types of Statements and Expressions, the .NET IL includes more than 200 different types of Instructions. Specifying the semantics of the entire .NET IL using the Taal approach thus requires to specify a separate transformation rule for every single Instruction. We have focussed on a subset of these instructions, leaving out instructions concerning, for instance, exception handling, parallelism, boxing and unboxing, and type conversion [139].

Control Flow Graph Construction

Another difference between applying our approach to Taal and .NET IL is that in the latter we omitted the separate phase in which the syntax graph is enriched with explicit control flow information. This choice is based on the observation that for many instructions, the instruction to be executed next actually is the instruction that is syntactically the next in the program. In those cases, the next instruction to execute is reachable through graph elements that model the syntactic sequential ordering relation. Typical instructions that have non-trivial flow semantics are so called *branch* instructions. Some example instructions that have non-trivial control flow semantics are `br` (simple branch), `bne` (branch on non-equal), and `brtrue` (branch on `true`) [139]. From Fig. 4.35 we can see that every Instruction has a unique Label referenced through the label-relation. Branch instructions, furthermore, have an explicit reference to the Label of the instruction to branch to (if the branch-condition evaluates to `true`) referenced by a target-relation (not shown in the diagram). By resolving the label identifier during static analysis, the Instruction belonging to the target Label of the branch Instruction can at run-time easily be determined.

Explicit Stacks

The last difference between applying our approach to .NET IL and Taal we will discuss here, involves the way *partial results*, i.e., evaluations of sub-expressions,

are included in run-time graph structures (called *Execution Graphs*). In Taal, we have introduced the Frame Graph in which

- local variables are attached to the specific type of Frame by means of locals-edges;
- *partial results*, i.e. evaluations of sub-expressions, are located at the particular syntax element.

For the .NET IL we have chosen to stay closer to the traditional approach by including an explicit stack structure in the run-time graphs. In the traditional approach, every frame is accompanied by a stack structure on which, among other things, the actual values of formal parameters are placed, space is reserved for the local variables, and partial results are stored temporarily. Including stack structures explicitly in the Execution Graphs, allows for more intuitive transformations rules for specifying the semantics of instructions that load or store values from and to formal parameters and local variables. In this study we have investigated some alternative ways to do this, ending up with the concept of a *shared stack* in which all frames originating from a single thread share the same stack. We have only considered .NET IL programs that give rise to single-threaded simulations; allowing for multi-threaded simulations then requires to extend the semantics graph production system with rules creating fresh stack structures whenever a new thread is started up.

4.9.2 Control Flow Specification Language

So far, we have been specifying the (static and dynamic) semantics of programming languages by graph transformation rules in a *manual* fashion. When considering more exotic programming language constructs (and combinations thereof) the flow of control through a program can become rather complex and highly non-trivial. Listing 4.4 shows a small example code pattern with non-trivial control flow.

```
while (true) {
  try {
    ...
    break;
    ...
  } finally {
    ...
  }
}
```

Listing 4.4: Example code pattern with non-trivial control flow.

In another master project [175, 174] carried out under our supervision, we have developed the Control Flow Specification Language (CFSL, for short). The CFSL is a specification language in which one can specify the control flow semantics of programming languages that feature many of the usual programming constructs. It supports the specification of control flow semantics for advanced constructs such as, e.g., `for` and `while` statements containing `break` or `continue` statements, and even more complex programming constructs such as `try-finally-catch` as available in, e.g., JAVA.

Fig. 4.36 gives an overview of the processes involved in this project. In the role of a *meta language designer*, we have defined the (concrete and abstract) syntax of the CFSL and its semantics, again, through a set of graph transformation rules, denoted CFSL-SEM in Fig. 4.36. The basic idea behind the CFSL is that the control flow semantics of the Language Under Development (LUD) are specified as Control Flow Specification Graphs (CFSGs, for short). Every CFSG is an instance of the CFSL meta model and conforms to the abstract syntax meta model of the LUD. The set of CFSGs then have to be specified by the language designer who defined the LUD and its semantics.

From the set of all CFSGs (i.e., one for each language construct) and the transformation rules in CFSL-SEM, GROOVE generates a set of transformation rules, denoted LUD-FLOW, which specifies the control flow semantics of the LUD. Stated differently, given a single CFSG of some language construct, say C, and the set of transformation rules CFSL-SEM, GROOVE generates a (possibly singleton) set of graph production rules which specifies the construction of control flow graphs for any C-instance in the abstract syntax graph of the LUD program. Applying the transformation rules in LUD-FLOW to the Abstract Syntax Graph of some LUD program P then results in the corresponding PG of P. Whether the PGs obtained this way are subject to further analysis such as, e.g., simulation, is outside the scope of this Master project, as indicated by the dashed arrow in the figure.

CFSL Meta-model

In this project we distinguish three types of control flow, namely *sequential*, *conditional*, and *disruptive* flow. Each type of control flow is facilitated through one or several constructs in the CFSL. All available elements in the CFSL and their associations are shown in the meta-model of the CFSL (Fig. 4.37), i.e., the model to which every CFSG has to conform. We will shortly discuss the most important concepts and relate them to the different types of control flow just mentioned.

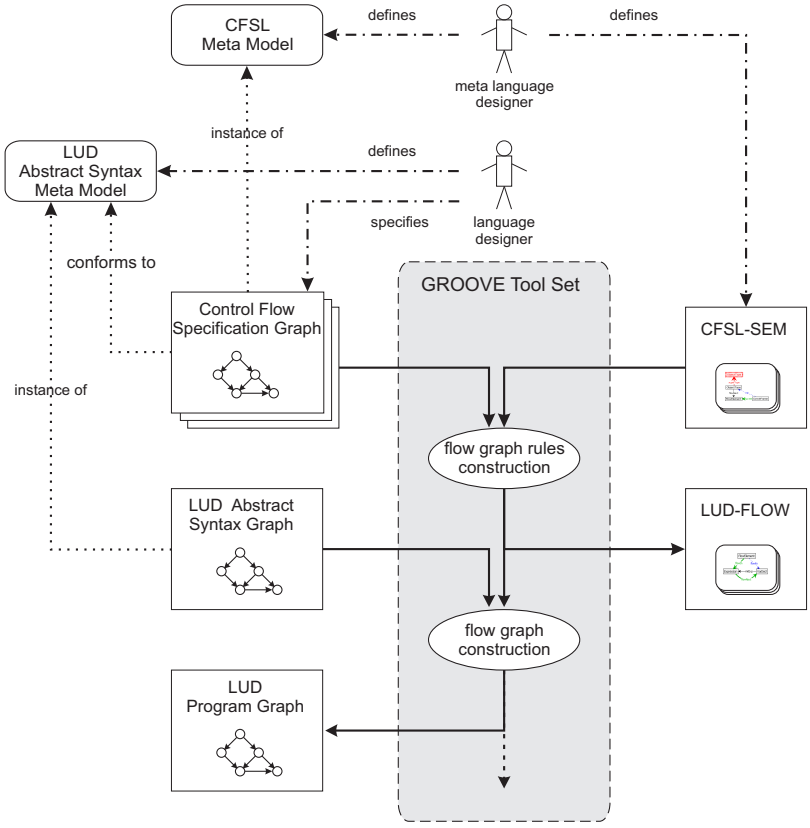


Figure 4.36: From CFSL specifications to control flow graphs.

Every CFSG specifies the control flow semantics for a specific language construct; language constructs are represented by AbstractSyntaxElements (in the sequel referred to as ASE). The ASE can be seen as a generic node representing all elements of the ASG (which all happen to be nodes) to which control can be transfered during execution; it is similar to the FlowElement in Taal. As a rule of thumb, we can say that for every type of ASE we specify a single CFSG. The outgoing entry and an exit-edges identify the point at which the actual execution of that ASE starts or ends, respectively.

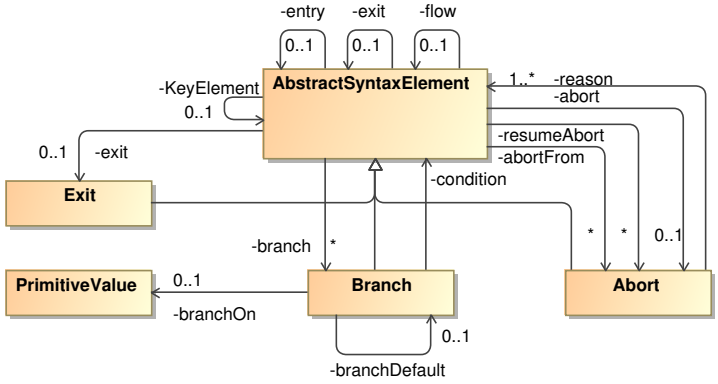


Figure 4.37: The meta-model of the CFSL.

Sequential flow represents the type of control flow that reflects the appearance of statements in the program. Statements that are executed subsequently are connected by edges labelled flow. *Conditional flow* refers to the type of control flow that is based on the evaluation of some expression, as in the case of the `while` statement. For each potential outcome of the evaluation of the expression, we introduce a Branch-node. Those Branch-nodes are connected (1) to the KeyElement through a branch-edge, (2) to the original expression by means of a condition-edge, and (3) to the corresponding value with a branchOn-edge. The branchDefault-edge represents the branch that is taken when no other branches apply. The last type of control flow is the *disruptive flow*. A statement is said to introduce disruptive flow of control if it is the cause of an *abrupt termination* (in contrast to what is often called *successful termination*). Disruptive flow is modelled through an Abort node to which control flows along an abort-edge. Every Abort node has a reason-edge pointing to the ASE that caused the abrupt termination. In some cases of disruptive flow it cannot immediately be determined at which statement to continue the program. For these situations we introduce the abortFrom and resumeAbort-edges. The details of how this is handled are out of scope of this thesis. The interested reader is referred to [174].

Example CFSG

We will now give an impression on how the control flow semantics of the `while` statement in Java would be specified in CFSL. For this, we start at the very

first beginning, namely the BNF rule for the `while` statement. We require that the grammar is enriched with *role names* for the different elements that form the context of a `while` statement; those role names will reappear in the CFSG. The WhileStatement BNF rule could then look as follows:

```
WhileStatement ::=
    <WHILE>
    <LPAR> condition:Expression <RPAR>
    body: Statement
```

The elements `<WHILE>`, `<LPAR>`, and `<RPAR>` represent the `while`-keyword and the parentheses enclosing the condition, and are only part of the concrete syntax; they will not be part of the ASG. This BNF rule gives rise to a local ASG as show in Fig. 4.38.

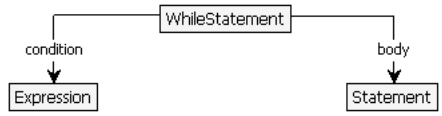


Figure 4.38: Local ASG of the `while` statement.

The CFSG of the WhileStatement, which is shown in Fig. 4.39, includes elements of all three types of control flow. First of all, the entry and the exit of the execution of the WhileStatement are specified. The execution of a WhileStatement starts with evaluating its condition. Thus, the node pointed to by the condition-edge will be the WhileStatement's entry. Its exit is represented by an unlabelled node. If the WhileStatement terminates successfully, control will flow to this node. The sequential flow is specified by the flow-edges from Expression to WhileStatement and from Statement to Expression. That is, after evaluating the condition, control will always flow to the WhileStatement, and after executing the body, control will always flow to the condition, which will then be reevaluated.

The conditional flow is specified by means of two Branch nodes, since the Expression has exactly two possible outcomes, namely `true` and `false`. On `true`, the body is executed another time; on `false`, the WhileStatement has terminated successfully and control will flow to the its exit.

There are cases in which the execution of the body of a WhileStatement is disrupted. We will discuss two of those cases, being the execution of a break

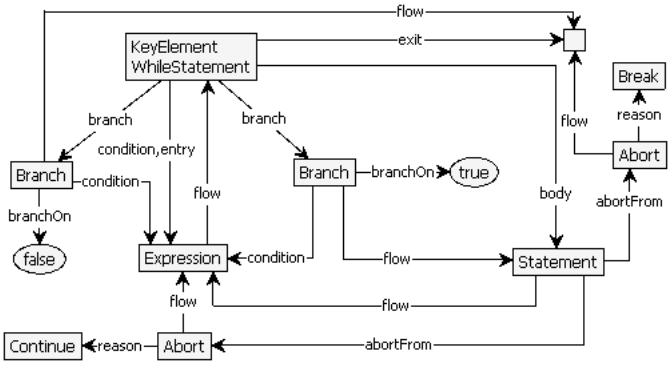


Figure 4.39: CFSG for the JAVA `while` statement.

and a continue statement. This is specified by two Abort nodes having incoming abortFrom-edges originating from the body. An Abort node must have an outgoing reason-edge by which it keeps track of the AbstractSyntaxElement that caused the abruptive flow. For the WhileStatement, both the break and the continue statement refer to the body as the reason for abruptly terminating the WhileStatement. Both cases, however, differ in the way control is specified to continue. A break statement causes control to flow to the exit of the WhileStatement, whereas a continue statement directs control to the condition which will then be reevaluated to determine whether the body of the WhileStatement has to be executed (at least) once more.

4.9.3 Remarks

The extensions discussed in this section point out that, on the one hand, the approach applied for specifying the semantics for TAAL is general and can in principle be applied to any programming language. On the other hand, we have shown that the approach can also be applied on a higher level of abstraction. That is, the CFSL can be regarded as a meta-language for which we have specified the semantics in terms of graph transformation rules by hand. Whereas, usually, a graph production systems generates graphs, the full transformation of a CFSG results in a set of final graphs which should be interpreted as graph transformation rules. Here we encounter another advantage of the GROOVE approach in which graph transformation rules are specified as graphs themselves.

4.10 Conclusion

4.10.1 Summary

In this chapter we have shown how to use graph transformations for specifying the control flow and operational semantics of programming languages. The graph transformation formalism offers a number of advantages. First, the visual presentation of the graph transformation rules provide an intuitive understanding of the semantics. Second, formal verification techniques become available. Furthermore, the graph transformation rules offer the possibility to include in one mathematical structure, the graph, information on both the run-time system and the program that is being executed. Traditional approaches to operational semantics (e.g. [198, 24, 1, 150, 25, 49]) often need to revert to inclusion in the syntax definition of run-time concepts, e.g. inclusion of the concepts of location to indicate a value that may possibly change over time. This seems to be an artificial manner of integrating parts of the language definition, i.e. of the abstract syntax and the semantic domain, that can be avoided using graph transformation rules. Finally, in graph transformation rules, context information can be included more naturally and uniformly than for example when using SOS-rules [151, 198].

The example language Taal that we have developed comprises some of the fundamental aspects of object-oriented programming languages, like inheritance, including dynamic method look-up, and object creation. The structure of our solution, and the ease with which it can be applied to others languages as shown in Section 4.9, makes us confident that the approach can be extended to real-life software languages in the object-oriented paradigm:

- All the transformation steps (parsing, static analysis, flow generation and simulation) are structured according to the concepts in the abstract syntax. This lends a modularity to the definitions that is independent of the language being defined.
- The structure of the Flow and Execution Graphs is generic, in the sense that the elements therein are not specific to Taal; rather, they capture the essential aspects of imperative, object-oriented languages.

4.10.2 Related Work

A traditional approach to specifying language semantics in an operational way is called *structural operational semantics* [151], or SOS for short. Applying the

SOS approach to specifying the semantics of programming languages require a solid understanding of advanced logics and rewrite systems. We believe that the graph transformation approach fits better to the expertise and experience of the average software engineer, which would enable him or her to also be involved in higher level activities than actual code writing.

Although other work has been presented that used graphs and graph transformation rules for (parts) of language definitions (e.g., [39, 78]), none of these reach the same level of completeness. In [39], Corradini et al. use graph transformations to formalize the semantics of a realistic programming language: they address a fairly large fragment of Java. Technically, the difference is that they interpret method invocation *unfolding*, meaning that the *program graph* changes dynamically. This obviates the need for the frame graph, at the price of having program-dependent rules (namely, one per method implementation). That is, their approach specifies what is often called big-step semantics, whereas in our approach we specify the semantics of the language in the smallest possible steps, namely per language construct. Program-dependent rules specifying big-step semantics are very likely to yield more efficient simulations than program-independent rules specifying small-step semantics, since the former provide far less points of interaction if multiple threads are operating in parallel. On the other hand, subtle errors often occur due to incorrect interleavings of atomic execution steps in the original program. One could think of two threads executing in parallel, both executing the same or different methods that reads and writes a global variable. If both method calls can only be performed as atomic steps, as suggested by the Corradini-approach, such errors will remain unrevealed. Our approach might require more resources (e.g., time and memory) to simulate the program, at least it will examine all possible interleavings of the most atomic steps. Another difference is that Corradini et al. do not provide any tool support, and in that sense theirs is a more theoretic exercise. In contrast, our results have successfully been applied by Engels et al. [79] for proving that their implementation of a model transformation from UML Activities to corresponding Taal programs is behaviour-preserving, as discussed in Section 4.8.

Another, less directly, related source of research is on defining dynamic semantics of (UML-type) design models, where also the idea of using graph transformations has been proposed (see, e.g., [78, 126, 127, 191]). Furthermore, in [78], Engels et al. present ideas on how to use collaboration diagrams, interpreted as graph transformations rules, for defining SL semantics.

4.10.3 Discussion

Parallelism. One of the advantages of applying our approach for specifying operational semantics on a self-defined language is that one can start with basic language constructs and investigate the effect of extending the language. Originally, Taal has been specified as programming language without parallelism, i.e. without the language construct ForkStat. Introducing new language features involves syntactic and semantic efforts. In this work we have focussed on the semantic efforts. That is, in what sense do the graph production systems have to be modified in order to support the new language features. For parallelism, we only needed to specify two additional rules for generating control flow information and seven additional rules for simulating the semantics (of which four involve the parameter passing process in a similar way as for usual method calls). The other rules did not require any modification. Though, it would be naive to conclude from this one extension that our approach to specifying operational semantics allows language extension in a modular fashion. Introducing a exception handling mechanism will very likely be much more involved.

Abstract Interpretation of Primitive Data Types. In the previous chapter we have shown that our approach provides easy ways of performing attributed graph transformations based on abstract algebras. We have also mentioned the special case in which the actual algebra is the final algebra of the specific signature(s). A final algebra, intuitively, has a singleton carrier set for every sort in the signature. Any operation op_o for some operation symbol $o\colon s_1 \cdots s_n \to s$ in the signature will then return that single value from the carrier set A_s. When applying such abstractions on the simulation of Taal programs, this yields interesting results, especially in cases where the continuation of the simulation depends on the evaluation of some expression, e.g. the simulation of a WhileStat or ConditionalStat. The condition will then always evaluate to this single abstract value. As a result, all the rules that take care of the different outcomes of the evaluation will then be applicable, since those concrete values will also be mapped to this one abstract value. Suppose we simulate the semantics of a WhileStat. The rules for both possible evaluations, being `true` and `false`, which would usually be applicable in a mutual exclusive way, are then both applicable. This means that the simulation continues along a separate branch in the state space for every possible concrete outcome of the evaluation. As mentioned before, the results of such abstract simulations can be useful, for example, for *dead code* analysis.

Nested Quantification in Graph Transformations. Currently, some aspects of the semantics that involve multiple elements to be equally treated are specified for a single element at a time. For example, the process of parameter passing required four separate rules distinguishing between the cases of no parameters, the first, every next, and last parameter to be passed (see Fig. 4.29). In the case of garbage collecting local variables of methods, a rule (Fig. 4.30(a) and Fig. 4.30(c)) has been specified that removes a single local variable every time it matches. For every local variable, the simulation contains a separate execution step for every local variable to be garbage collected. The work done on *nested quantification* in graph transformations (see e.g. [160, 124]) enables to specify such processes for all those elements in a single rule. In the case of the local variables, garbage collecting them can easily be specified by universally quantifying the locals-edge and referenced VarSlot and even be included in the rules shown in Fig. 4.30(b) and Fig. 4.30(d). In the case of parameter passing this is a bit more complicated. The way the abstract syntax of the language is currently defined does not provide sufficient means for specifying the process of parameter passing by a universal quantified rule. This can only be achieved when the expressions representing the actual parameters are explicitly referencing the VarDecls of the formal parameters. If such an abstract syntax is available, the rule shown in Fig. 4.40 could specify the entire process of parameter passing. This rule includes a single level of universal quantification. The special node labelled ∀ indicates which elements should be matched universally.

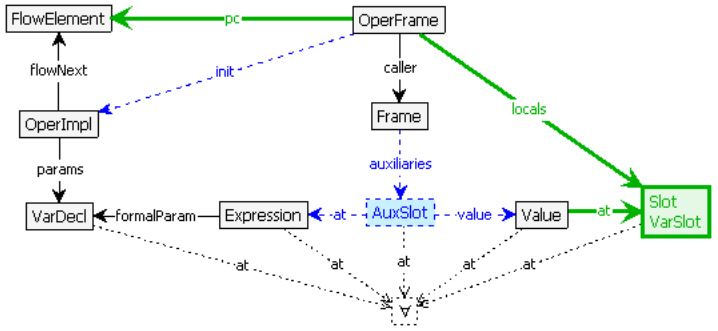


Figure 4.40: Nested quantified rule specifying garbage collecting local variables.

Correctness. Using our approach to specifying static and dynamic (e.g., control flow) semantics of (programming) languages, questions arise concerning the way

correctness can (formally) be proven. For Taal, for example, one could question whether the rules in taal-sem generate graphs that are indeed Execution Graphs. Likewise, for the cfsl, it would be interesting to investigate how one could prove that the rules in cfsl-sem generate graph production systems that indeed generate correct flow graphs.

One of the basic issues is that Groove performs graph transformations in an *untyped* setting. That is, Groove does not require an explicit specification of a so-called *type graph* [44] to which all graphs and transformation rules have a proper *typing morphism*. If such a type graph, say TG, could be constructed for a given graph transformation system, TG could (automatically) be compared with the type graph (or meta model) that defines the (abstract) syntax of the language. However, this would only partially solve the problem. Another part of the problem is due to the fact that termination of graph transformation systems is, in general, undecidable. There are some results by Ehrig et al. on identifying termination criteria for *layered* graph transformation systems [63]. Those termination criteria require that the transformation rules are assigned to different layers based on their effect on shared graph elements. For the graph transformation systems we are dealing with, such a layering does (often) not exists.

Nevertheless, in case of the cfsl, the correctness problem has partially been tackled. As usual with UML-like meta models, the meta-model from Fig. 4.37 is accompanied by a number of constraints on the combination of different elements. Some of the constraints have been specified as graph transformation rules. If one such a constraint is violated, the corresponding rule is applicable which then immediately terminates that branch of the transformation process. For an overview on which constraints have been formalized as graph transformation rules, the interested reader is referred to [174].

5 Model Checking Graph Production Systems

5.1 Introduction

In the previous chapter we have shown that the graph transformation framework provides formal and intuitive means of specifying the semantics of object-oriented languages. When modelling the behaviour of software systems in general, and *concurrent systems* (in which multiple processes operate in parallel) more specifically, the graph transformation framework has been proven very powerful (see, e.g., [70]), and new application domains such as, e.g., various types of *wireless sensor networks* [166], are currently under investigation.

In this chapter we will focus on verifying state spaces generated from arbitrary graph production systems using the verification technique called *model checking*. Since model checking has been introduced in the 1980s [74, 32, 155, 33], numerous model checking tools have been developed, among others, SPIN [104], JPF [194], SLAM [11], BLAST [19], MAGIC [30], and SMV [137]. As mentioned in Section 1.2, model checking techniques are applied on a *model* of the system. Such models can be specified in dedicated languages such as, e.g., PROMELA for SPIN [104], or in more general-purpose languages such as, e.g., JAVA source or byte-code for Java PathFinder [194]. Model checkers can furthermore be characterized in terms of (1) the techniques used to explore or traverse the state space and (2) the way individual (or sets of) states are stored. With respect to the two just mentioned characteristics, there is a distinction between *explicit-state* model checkers and *symbolic* model checkers. The difference is that the former category of model checkers store states individually (often

as so-called *bit vectors*), whereas the latter uses specific models to store *sets* of states (using so-called *Binary Decision Diagrams* [26]). Explicit-state model checkers are often used in practice since they provide useful diagnostics about why systems are not correct; symbolic model checkers have been shown to be successful for the verification of system in which state spaces are finite but may be huge (see, e.g., [28]).

When representing system states as graphs and system behaviour as graph productions, the next step is to generate the system's state space, on which we can apply standard model checking techniques to verify whether the system satisfies specific properties. In this work we distinguish three different basic approaches to model checking, namely *sequential*, *on-the-fly*, and *bounded* model checking. In [113] we have proposed a fairly straightforward approach to verify graph production systems. There, we focussed on graph productions that give rise to finite state spaces and applied the sequential model checking approach. This means that we first generate the full state space and perform the verification process subsequently. In this approach, properties are specified in the branching temporal logic *CTL* [32] and the model checking procedure was based on the standard backward-state traversal algorithm [37, 17]. This approach provided a proof-of-concept and gave some insight in the advantages and disadvantages of the basic idea of model checking graph production systems.

In general, however, termination of graph production systems is undecidable (cf. Section 4.10), and thus the state spaces they generate may be infinite. Therefore, the approach from [113] cannot be applied to arbitrary graph production systems. An obvious alternative approach could be to verify such systems using the *on-the-fly* model checking approach, in which the state space is verified *while* it is generated. This would, however, only partially solve the problem since such algorithms might dive into correct, though infinite, parts of the state space while other (possibly finite) parts of the system contain small counter-examples.

To solve this, we propose an algorithm that combines a known algorithm for on-the-fly model checking with basic ideas from *bounded model checking* [21, 20]. That is to say, the algorithm iteratively generates ever-larger parts of the system's state space and performs on-the-fly verification on those parts. Which parts to generate will be specified through *boundary conditions* that are updated in an iterative fashion as long as no system-executions violating the system requirements have been identified.

In this work, properties are specified as formulae in the linear time temporal logic *LTL* (for Linear Temporal Logic) [133]. We have chosen to formalize properties as *LTL* formulae for the availability of a large body of research results in terms of model checking approaches and corresponding algorithms, especially

in the sub-field of on-the-fly model checking. However, choosing *LTL* puts a strong restriction on the properties we can specify. More precisely, *LTL* provides means to reason about properties holding in specific states along system executions but does not provide means to reason about how certain states could be reached in terms of actions performed by the system. Although this kind of information is provided by graph transition systems, it is lost in the translation from graph transition systems to *Kripke structures*. The same remark holds for the approach proposed in [113], where properties are specified as formulae in the branching temporal logic *CTL*.

One major advantage of the framework we propose is that transitions are computed only once and those results are reused whenever explored states are revisited. This is advantageous since computing transitions is one of the main bottlenecks in the performance of the graph transformation framework. This is due to the computational complexity of computing rule applications and state isomorphism checking. The main disadvantage of our algorithm is that it cannot guarantee *completeness*. That is, we cannot guarantee that the algorithm always finds a counter-example if there exists one. This is due to the fact that not all paths (and thus also not all counter-examples) have a finite representation. Whenever there exists a finitely representable counter-example, a so-called *reachable accepting cycle*, our algorithm is guaranteed to find it with a finite amount of resources.

Overview of the Chapter

This chapter is structured as follows. In Section 5.2 we briefly recall the basic idea, strengths, and weaknesses of the model checking technique and discuss our categorization of the basic approaches to model checking. Section 5.3 introduces the required concepts for *LTL* model checking and discusses the relation with the graph transformation framework. Next, in Section 5.4 we recall the automata theoretic approach to *LTL* model checking.

Section 5.5 then continues with introducing the ingredients used in the new on-the-fly bounded model checking algorithm, which is discussed in Section 5.6. This section also includes some analysis on the time-complexity of the algorithm based on different growth factors of systems on which the algorithm can be applied. In Section 5.7 we discuss some implementation issues and report on experimental results. This chapter finishes with some concluding remarks.

This chapter is based on [113, 115].

5.2 Model Checking

Verification techniques aim at providing (semi-)automatic means to determine whether a system is correct or faulty. In Section 1.2 we have mentioned the two most important verification techniques being *theorem proving* (see, e.g., [58]) and *model checking* [37, 4]. As should be clear by now, in this work we focus on applying (existing) model checking techniques to verify the behaviour of systems specified as graph production systems.

To recall, the most important positive (+) and negative (–) characteristics of the model checking technique are the following:

+ it provides *automatic* means to determine whether a model of a system satisfies a set of requirements specified, e.g., as (temporal) logic formulae;
+ if the system does not satisfy the property under verification, a *counter-example* is provided that falsifies that property;
– it suffers from the *state-explosion problem*.

The general approach to model checking is depicted in Fig. 5.1. Before the actual model checking procedure can be performed by the Model Checker (in the gray-shaded rectangular box), the following two processes have to be carried out:

- *modelling*: from the system we construct a model that specifies the behaviour of the system. This construction can be done manually or automatically. A way to manually specify a model of the system is to specify its behaviour using well-known specification languages such as, among others, CCS [140, 141], CSP [103, 141], or Promela [104];
- *formalizing*: the property to be verified is specified in some formal language. Typically, properties are specified as formulae in some (temporal) logic such as *CTL* or *LTL*.

Assume M is a model of some system and φ as the property to be verified for this system, the model checking procedure produces a verdict on whether M *satisfies* φ, denoted by $M \vDash \varphi$. Stated differently, whether M *is a model of* φ (hence the term 'model checking'). If the verdict is positive, the model is said to satisfy the property; if the verdict is negative, the model is said to *falsify* the property.

In this work, we distinguish three different basic approaches to model checking, being:

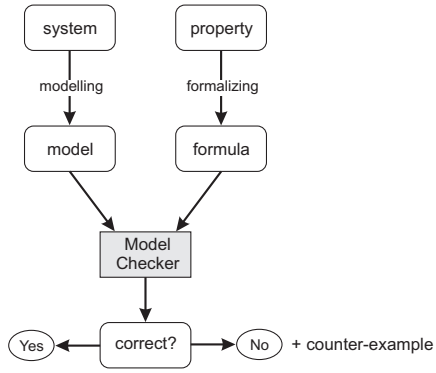


Figure 5.1: The general model checking approach.

- *sequential* model checking;
- *on-the-fly* model checking;
- *bounded* model checking.

In the following sections we will briefly discuss the major features of the different approaches.

Sequential Model Checking

Sequential model checking is the most straightforward approach. The basic idea is to first generate the entire state space of the system and then perform the verification procedure sequentially. This approach is depicted in Fig. 5.2. Note that this picture is a refinement of Fig. 5.1. Needless to say, this approach only applies to finite state spaces.

In [113] we proposed to apply this approach to model check graph production systems generating finite state spaces. Obviously, the use of this approach is fairly limited since many systems have huge and often even infinite state spaces. Nevertheless it provided some insight in the strengths and weaknesses of the general approach of model checking graph production systems.

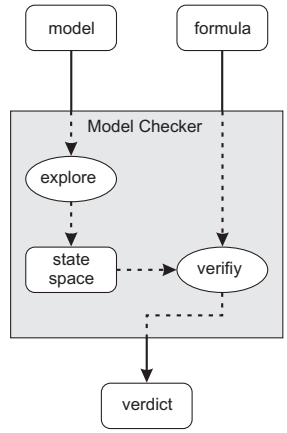


Figure 5.2: The sequential model checking approach.

On-The-Fly Model Checking

A more sophisticated approach to model checking is based on the idea of verifying the system model on-the-fly, i.e., during its generation. Figure 5.3 depicts the interplay between the verification and the exploration tasks in this approach. This approach has originally be introduced by Courcoubetis et al. [47]. The main advantage of this approach is that counter-examples can be identified without the need to first fully generate the state space of the system model. In fact, using this approach we only examine the system executions that potentially produce undesired behaviour. Parts of the state space that are guaranteed not to contain a counter-example do not even have to be considered. Therefore, this approach can also be applied in cases where the state space is potentially infinite. A number of different on-the-fly model checking algorithms have been developed, one of which will be discussed in detail in Section 5.4.

Bounded Model Checking

More recently, bounded model checking has been introduced by Biere et al. [21, 20], and applied as a technique for iteratively verifying finite state systems symbolically. Figure 5.4 gives a schematic overview of this approach. The basic principle behind bounded model checking is to generate part of the state space, represent that part as a *satisfiability problem*, and search for solutions efficiently

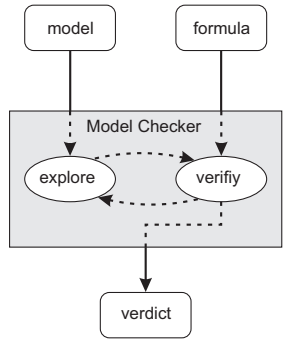


Figure 5.3: The on-the-fly model checking approach.

using SAT solvers. Which part of the state space to generate and verify is usually specified by a *boundary condition*. Typically, the boundary condition restricts the maximal length of execution paths allowed in one iteration. If the current iteration does not yield a counter-example, this maximal length is increased and the verification procedure is restarted. This process is repeated until (1) a counter-example has been found, (2) the problem becomes intractable, or (3) a so-called *completeness threshold* has been reached. In the first case, a bug has been found which can then be investigated and (hopefully) repaired. In the second case, the state space becomes too large to, e.g., fit in memory, and further analysis is impossible or does not pay off. Reaching the completeness threshold means that the system can be considered (sufficiently) correct.

5.3 LTL Model Checking: Ingredients

In this section we recall the basic ingredients for model checking systems specified as *Kripke structures* [123] against properties specified in the linear time temporal logic *LTL* [133]. We then show how this framework can be applied in the context of graph transformations. For this we define how to construct a Kripke structure for a system of which the behaviour is specified by a graph production system.

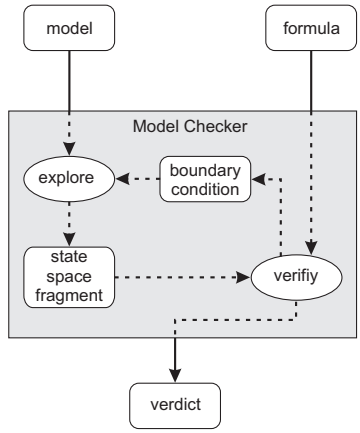


Figure 5.4: The bounded model checking approach.

5.3.1 Kripke Structures

Kripke structures are transition systems in which states are labelled and every state has at least one successor, i.e., the transition relation is total.

Definition 5.1 (Kripke structure)**.** *Given a set AP of atomic proposition, a* Kripke structure $K = \langle S, \rightarrow, s_0, L \rangle$ *consists of:*

- *a set S of* states*;*
- *a total* transition relation $\rightarrow \subseteq S \times S$*, i.e.,* $\forall s \in S : \exists s' \in S : (s, s') \in \rightarrow$*;*
- *an* initial state s_0*;*
- *a* labelling function $L\colon S \rightarrow 2^{AP}$ *which maps each state to a subset of atomic propositions holding in that state.*

Given a Kripke structure K we are interested in the *paths* (or *runs*) generated by K. A path is an infinite sequence $\rho = s_0, s_1, \ldots$, of states. For a path $\rho = s_0, s_1, s_2, \ldots$, we use the following shorthand notations:

- $\rho(i)$ denotes the i-th element of ρ, e.g., $\rho(0) = s_0$
- ρ^i denotes the suffix of ρ starting at $\rho(i)$. e.g., $\rho^1 = s_1, s_2, \ldots$

The set of all paths generated by a Kripke structure $K = (S, \rightarrow, s_0, L)$ from some state $s \in S$ is denoted $Path_K(s)$ and can then be defined as follows:

$$Path_K(s) \quad = \quad \{\rho \in S^\omega \mid \rho(0) = s \wedge \forall i \geq 0 : \rho(i) \rightarrow \rho(i+1)\} \ .$$

The totality of the transition relation guarantees that every finite path can be extended to an infinite one.

5.3.2 Linear Temporal Logic

In the *linear time* temporal logics *LTL* [133], formulae are interpreted over all linear runs of the system. This means that at every time instant only a single possible future will be considered, in contrast to branching time logics such as *CTL* [32] where potentially multiple futures are considered at every time instant. In the following, we shortly recall the syntax and semantics of *LTL*.

Syntax. Given AP as the set of *atomic propositions*, the syntax of *LTL* formulae is as follows:

- for every $p \in AP$, p is a *LTL* formula — *atomic propositions*
- if ϕ and ψ are *LTL* formulae, then so are:
  - $\neg\phi$, — *boolean negation*
  - $\phi \vee \psi$, — *boolean disjunction*
  - $\mathsf{X}\,\phi$, — *temporal next*
  - $\phi\,\mathsf{U}\,\psi$. — *temporal until*

From the listed boolean and temporal operators, the following operators can be derived as indicated:

- $\mathsf{tt} \equiv \phi \vee \neg\phi$, — *true*
- $\mathsf{ff} \equiv \neg\mathsf{tt}$, — *false*
- $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, — *boolean conjunction*
- $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$, — *boolean implication*
- $\phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, — *boolean equivalence*
- $\mathsf{F}\,\phi \equiv \mathsf{tt}\,\mathsf{U}\,\phi$, — *temporal eventually*
- $\mathsf{G}\,\phi \equiv \neg\mathsf{F}\,\neg\phi$. — *temporal globally*

At an intuitive level, the boolean operators have their usual meaning. The temporal operators might require some additional explanation. The formula $\mathsf{X}(\phi)$ is true in the current state if ϕ is true in the successor state. The formula $\phi\mathsf{U}\psi$ is true in the current state if eventually ψ will hold and in the meanwhile

ϕ is true continuously. The temporal F-operator can be used to specify reachability, i.e., $\mathsf{F}\phi$ is true in the current state if there exists a reachable state on the path in which ϕ is true. With the G-operator, safety properties can easily be specified, i.e., $\mathsf{G}\phi$ is true if ϕ is true now and in all the successor states.

The duality between F and G reflects the duality between reachability and safety properties: a system satisfies a safety property ϕ if there is no reachable state in which $\neg\phi$ is satisfied.

Semantics. *LTL* formulae are interpreted over linear runs of a Kripke structure. A run ρ over a Kripke structure K can then be seen as a function $\rho\colon I\!N \to \mathbf{2}^{AP}$, which assigns truth values to the elements of AP at each position of the run. Let ρ be a run, $p \in AP$ be an atomic proposition, and ϕ, ψ be *LTL* formulae. The satisfaction relation $\models$ is then defined as below:

$$
\begin{array}{lll}
\rho \models p & \text{iff} & p \in L(\rho(0)) \\
\rho \models \neg\phi & \text{iff} & \rho \not\models \phi \\
\rho \models \phi \vee \psi & \text{iff} & \rho \models \phi \text{ or } \rho \models \psi \\
\rho \models \mathsf{X}\phi & \text{iff} & \rho^1 \models \phi \\
\rho \models \phi \mathsf{U} \psi & \text{iff} & \exists j \geq 0 : (\rho^j \models \psi \ \wedge \ \forall 0 \leq k < j : \rho^k \models \phi) \quad .
\end{array}
$$

5.3.3 From Graph Production Systems to Kripke Structures

As explained in Section 2.2, every graph production system gives rise to a graph transition system, in which states represent the reachable graphs and transitions represent the actual rule applications producing those graphs. In order to decide whether a graph production system satisfies an *LTL* formulae we define how to translate graph transition systems to Kripke structures. In the following, let $P = (\mathcal{R}, I)$ be a graph production system, $T_P = (S_T, \rightarrow_T, I)$ be the graph transition system of P, and $K_P = (S_K, \rightarrow_K, s_0, L_K)$ be the Kripke structure corresponding to T_P.

First of all, we have to define the set AP of atomic propositions. Furthermore, we have to define a translation from (finite or infinite) graph transition systems to Kripke structures. Obviously, the states of the graph transition system will also be states of the corresponding Kripke structure, i.e., $S_K = S_T$. However, we have to be cautious when defining the labelling function and the transition relation.

Atomic Propositions

When specifying the behaviour of a system in term of graph transformations, the level of granularity of the transformation rules indicate to what degree we are interested in the possible behaviours of the system. Therefore, it is straightforward to take the *rule names* as atomic building blocks for formally specifying the required behaviour of the system. That is to say, for a graph production system P, the set of atomic proposition AP is equal to the set of names N_p (recall Section 2.2) of the graph productions $p \in \mathcal{R}_P$:

$$AP \quad = \quad \{N_p \mid p \in \mathcal{R}_P\} \ .$$

Labelling Function

The next step is to define the labelling function L_K, which maps every state s to the set of atomic propositions holding in s. Graph transition systems provide information about which rules (possibly none) are applicable in all states. Based on that we can define the labelling function. That is to say, a state s will be mapped to the set of rule names of rules that are applicable in that state. This can be formalized as follows.

$$L_K(s) \quad = \quad \{N_p \mid \exists m \in \mathcal{M}, s' \in S : s \xrightarrow{p,m} s'\} \ .$$

Transition Relation

For defining the transition relation $\rightarrow_K$, we partition the rules $\mathcal{R}$ in so-called *conditions* and *rules*. A graph production is said to be a *condition* if it does not specify any changes. Formally, a graph production $p\colon L \rightarrow R$ is a condition if $L = R$ and $p = id_L$, i.e., p is the identity morphism of L. The set of all conditions of a graph production system P will be denoted $\mathit{conditions}(P)$. Thus,

$$\mathit{conditions}(P) \quad = \quad \{p\colon L \rightarrow L \in \mathcal{R}_P \mid p = id_L\} \ .$$

The transition relation $\rightarrow_K$ can now be defined as follows:

$$\begin{aligned} \rightarrow_K \quad = \quad & \{(s, s') \mid \exists p, m : s \xrightarrow{p,m} s' \wedge p \notin \mathit{conditions}(P)\} \cup \\ & \{(s, s) \mid \nexists p, m, s' : s \xrightarrow{p,m} s' \wedge p \notin \mathit{conditions}(P)\} \ . \end{aligned} \tag{5.1}$$

The first part of the union in (5.1) states that all transitions that represent

applications of actual rules are also included as transitions in $\rightarrow_K$. Conditions, intuitively, only check whether a graph includes some special graph pattern; they do not specify any changes. Nevertheless, applications of conditions appear as self-loops in the graph transition system, although they do not specify actual behaviour of the system. Therefore, those transitions are excluded from $\rightarrow_K$.

The second part ensures the totality of $\rightarrow_K$. That is, for every state that is a *deadlock state* in the graph transition system T_P, $\rightarrow_K$ includes a *loop transition* to that state itself. By this, we ensure that all finite behaviours of the original graph transition system are extended to infinite ones in the corresponding Kripke structure.

Remark 5.2. *We must be careful with interpreting the labelling function. A state s being labelled with some condition means that s satisfies the condition; a state s being labelled with some rule name N_p for some rule p, means that s satisfies the* preconditions *of p. The labelling function does not say anything about actual applications of the rules.*

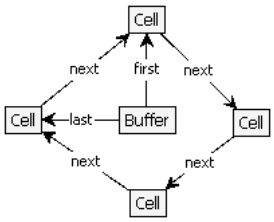
5.3.4 Example: Circular Buffer

We explain the above translation through the circular buffer example, from Section 2.2.2. Fig. 5.6 again depicts the graph transition system T of this example. For the corresponding Kripke structure, we highlight the most important things. The initial state of the Kripke structure is pointed to by the sourceless arrow. Unlike the transitions of the graph transition system, the transitions of the Kripke structure are unlabelled. Furthermore, the transition in T representing the application of the empty-rule does not appear in K_T, since in fact is a condition. The labelling function is included in Fig. 5.6(b) by attaching the sets of atomic propositions to the corresponding states.

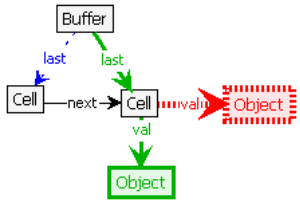
A typical property that can be verified for this example is whether it is eventually possible to either put an object in the buffer or get an object from the buffer. This is specified in *LTL* as follows:

$$\mathsf{GF}(\mathsf{put} \vee \mathsf{get}) \ .$$

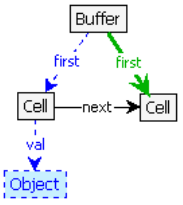
It is easy to see that this property is indeed satisfied by the Kripke structure of Fig. 5.6(b).



(a) Initial graph: the empty Buffer.



(b) The put-rule.



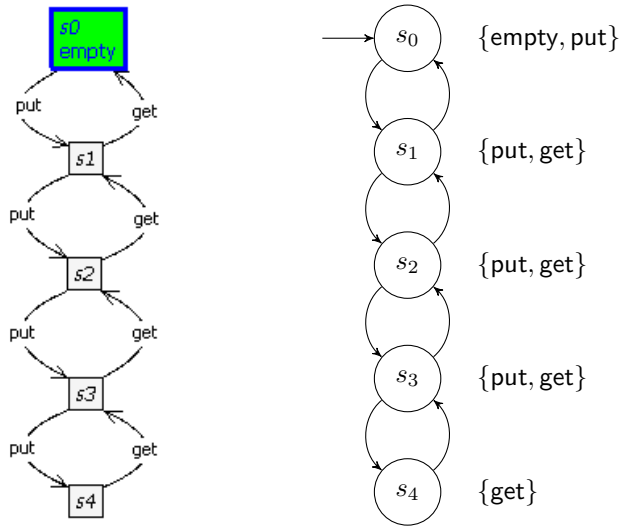
(c) The get-rule.

Figure 5.5: The graph production system specifying the circular buffer.

5.4 On-The-Fly LTL Model Checking

When implementing the model checking technique aiming at the verification of *LTL* formulae, an often used approach is the *automata theoretic approach*, originally introduced by Vardi and Wolper [189]. The basic idea is to translate the model checking problem to the problem of *emptiness checking* of the language of some specific automaton, typically a *Büchi automaton* [27]. As mentioned in Section 5.1 we have chosen to verify properties specified as *LTL* formulae since there is a large body of mature results on this topic and straightforward and easy-to-implement algorithms exist. One such an algorithm is the *nested depth first search* (NDFS, for short) algorithm, originally introduced by Courcoubetis et al. [47].

Before diving in to the details of the automata theoretic approach, we first introduce Büchi automata and related concepts such as the *language* of a Büchi automaton and the *product* of two Büchi automata. Finally, this section presents a recently proposed variant of the NDFS algorithm used to determine whether the language of a given Büchi automaton is empty, namely the algorithm proposed by Schwoon and Esparza [171].



(a) Graph transition system T.

(b) Kripke structure K_T.

Figure 5.6: Graph transition system T and corresponding Kripke structure K_T.

5.4.1 Büchi Automata

In the automata theoretic approach to model checking, systems and properties are modelled as Büchi automata. Different types of Büchi automata have been developed and applied in the literature, varying in what is labelled (states or transitions) and which *runs* are accepted (see, e.g., [89, 82, 184]). In our framework, Büchi automata have labelled transitions and the acceptance condition is specified in terms of a *single* set of designated states, the so-called *accepting states*.

Definition 5.3 (Büchi automaton)**.** *A Büchi automaton* $B = (S, \Sigma, \rightarrow, s_0, F)$ *consists of:*

- *a set* S *of* states*;*
- *an* alphabet Σ*;*
- *a* transition relation $\rightarrow \subseteq S \times \Sigma \times S$*;*
- *an* initial state $s_0 \in S$*;*

- *a set* $F \subseteq S$ *of* accepting states.

For a Büchi automaton B, we furthermore introduce the following concepts. A *run* of B on an infinite word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a sequence $\rho = s_0, s_1, \ldots$, where s_0 is the initial state and $(s_i, a_i, s_{i+1}) \in \rightarrow$, for all $i \geq 0$. A run $\rho = s_0, s_1, \ldots$, is *accepting* if there are infinitely many indices $i \geq 0$ such that s_i is an accepting state. The *language* of a Büchi automaton B, denoted $\mathcal{L}(B)$, is the set of all words that have an accepting run in B. Formally,

$$\mathcal{L}(B) \quad = \quad \{w \in \Sigma^\omega \mid w \text{ is accepted by } B\} \ .$$

In Def. 5.3, we do not require a Büchi automaton to have a finite set of states, since later on we will be working with Büchi automata that are potentially infinite. In the infinite case, accepting runs might contain infinitely many *distinct* accepting states. In the finite case, accepting runs can be characterized more specifically. The acceptance criterion on a run $\rho = s_0, s_1, \ldots$, then translates to requiring that ρ must contain at least one accepting state of the Büchi automaton infinitely often. Every accepting run can then be divided into three parts $\rho_1 \rho_2 \rho_3$ such that ρ_2^ω is an *accepting cycle* reachable from the initial state.

Definition 5.4 (reachable accepting cycle)**.** *Let $B = (S, \Sigma, \rightarrow, s, F)$ be a Büchi automaton. An infinite sequence $\rho = s_0, s_1, s_2, \ldots$ is an* accepting cycle *if there exist a natural number k such that for all indices $j \geq 0$ it holds that $s_{j+nk} = s_j$, for all $n \in \mathbb{N}$, and ρ contains at least one accepting state. An accepting cycle ρ is* reachable *if s_0 is reachable from the initial state s.*

The natural number k represents the *length* of the accepting cycle, i.e., the number of states. If $\rho = s_0, s_1, s_2, \ldots$ is an accepting run for which $i = 2$ and $k = 3$, ρ can be finitely represented as depicted in Fig. 5.7. Intuitively, such accepting runs look like *lassos*.

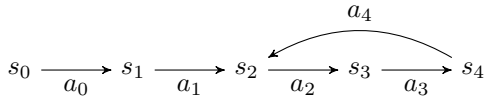


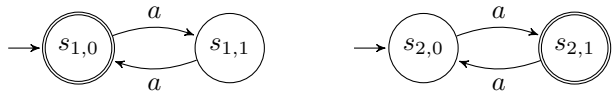
Figure 5.7: Accepting run $\rho = s_0, s_1, s_2, \ldots$ where ρ^2 is a reachable accepting cycle of length three.

Based on Def. 5.3, we define the *product* of two Büchi automata (sharing the same alphabet, say Σ) as follows (regardless of whether the automata are finite or infinite).

Definition 5.5 (product)**.** *Let* $B_1 = (S_1, \Sigma, \rightarrow_1, s_{1,0}, F_1)$ *and* $B_2 = (S_2, \Sigma, \rightarrow_2, s_{2,0}, F_2)$ *be two Büchi automata. The* product *of* B_1 *and* B_2*, denoted* $B_1 \times B_2$*, is defined as* $B_1 \times B_2 = (S, \Sigma, \rightarrow, s_0, F)$*, where*

- $S = S_1 \times S_2$,
- $\rightarrow \subseteq S \times \Sigma \times S$ *defined by* $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ *if* $s_1 \xrightarrow{a}_1 s_1'$ *and* $s_2 \xrightarrow{a}_2 s_2'$,
- $s_0 = (s_{1,0}, s_{2,0})$,
- $F = F_1 \times F_2$.

As pointed out by Choueka [31], for products constructed as from Def. 5.5, it is in general not the case that the language of the product is equal to the intersection of the languages of both components, i.e., in general, $\mathcal{L}(B_1 \times B_2) \neq \mathcal{L}(B_1) \cap \mathcal{L}(B_2)$. For example, Fig. 5.8 depicts two Büchi automata B_1 and B_2 with $\mathcal{L}(B_1) = \mathcal{L}(B_2) = a^\omega$, although $\mathcal{L}(B_1 \times B_2) = \emptyset$.



(a) Büchi automaton B_1. **(b)** Büchi automaton B_2.

Figure 5.8: Two Büchi automata for which Def. 5.5 does not produce the correct product.

When restricting to cases in which one of the Büchi automata consists of accepting states only, we can show that $\mathcal{L}(B_1 \times B_2) = \mathcal{L}(B_1) \cap \mathcal{L}(B_2)$. This is stated in the following result.

Proposition 5.6. *Let* B_1 *and* B_2 *be two Büchi automata. Then,* $F_1 = S_1$ *implies* $\mathcal{L}(B_1 \times B_2) = \mathcal{L}(B_1) \cap \mathcal{L}(B_2)$.

Proof. The proof consists of showing that there exists an inclusion in both directions. That is,

1. $L(B) \subseteq L(B_1) \cap L(B_2)$,
2. $L(B_1) \cap L(B_2) \subseteq L(B)$.

where $B = B_1 \times B_2$.

From $S_1 = F_1$ we obtain $F_B = S_1 \times F_2$. Let $w = a_0 a_1 \ldots$, be an arbitrary word in $L(B_1) \cap L(B_2)$, i.e., w is accepted by both B_1 and B_2. Then, there are accepting runs $\rho = s_0, s_1, \ldots$, and $\rho' = r_0, r_1, \ldots$, for B_1 and B_2,

respectively, such that $s_i \xrightarrow{a_i}_1 s_{i+1}$ and $r_i \xrightarrow{a_i}_2 r_{i+1}$, for $i \geq 0$. From this we may conclude that there exists a sequence $\bar{\rho} = (s_0, r_0), (s_1, r_1), \ldots$, such that $(s_i, r_i) \xrightarrow{a_i} (s_{i+1}, r_{i+1})$, for $i \geq 0$. Since ρ' is accepted by B_2, it contains infinitely i such that $r_i \in F_2$. And thus, the sequence $\bar{\rho}$ contains infinitely i such that $(s_i, r_i) \in F_B$. Therefore, $\bar{\rho} \in L(B)$.

Now, let $w' = a_0 a_1 \ldots$, be an arbitrary word in $L(B)$, i.e., w' is accepted by B. By definition, there exists an accepting run $\rho = (s_0, r_0), (s_1, r_1), \ldots$ for B with $(s_i, r_i) \xrightarrow{a_i} (s_{i+1}, r_{i+1})$, for $i \geq 0$. By construction we know that the sequence $\rho_1 = s_0, s_1, \ldots$ is a run in B_1 and by definition, any run in B_1 is accepting thus $w' \in L(B_1)$. Furthermore, the sequence $\rho_2 = r_0, r_1, \ldots$, is a run in B_2. Since ρ is an accepting run in B it contains infinitely many i such that $(s_i, r_i) \in F_B$. By definition, the B_2-components of those states are in F_2. Therefore, ρ_2 contains infinitely i such that $r_i \in F_2$ and is therefore an accepting run in B_2 and thus $w' \in L(B_2)$. Since $w' \in L(B_1)$ and $w' \in L(B_2)$ we have $w' \in L(B_1) \cap L(B_2)$.

Due to commutativity of product and intersection, this property holds symmetrically when $S_2 = F_2$. □

5.4.2 The Automata Theoretic Approach

The basic idea of tackling the model checking problem using the automata-theoretic approach is to translate it to the problem of checking whether the language of some specific automaton is empty. In this approach, the system is assumed to be modelled as a Büchi automaton, say B_{sys}. The language of this automaton, denoted $\mathcal{L}(B_{sys})$, is the set of all behaviours of the system. As mentioned before, we deal with system requirements that are expressed as *LTL* formulae. It has been shown that for every *LTL* formulae ϕ there exists a Büchi automaton, say B_ϕ, such that B_ϕ accepts exactly the infinite behaviours that satisfy ϕ [190]. There exist many approaches for translating arbitrary *LTL* formulae to Büchi automata accepting the corresponding infinite behaviours; see, e.g., [89, 177, 82, 87].

Suppose we want to verify whether a system modelled as a Büchi automaton B_{sys} satisfies an *LTL* formula ϕ. The automata theoretic approach prescribes to determine whether all words in $\mathcal{L}(B_{sys})$, i.e., all accepted behaviours of the system, are also included in $\mathcal{L}(B_\phi)$. Formally,

$$\mathcal{L}(B_{sys}) \subseteq \mathcal{L}(B_\phi) \ . \tag{5.2}$$

Stated differently, we have to determine whether the system does not include

behaviours that are not allowed by ϕ. This can be formalized by taking the *complement* of $\mathcal{L}(B_\phi)$, denoted $\overline{\mathcal{L}(B_\phi)}$; the model checking problem is then reduced to determining whether:

$$\mathcal{L}(B_{sys}) \cap \overline{\mathcal{L}(B_\phi)} = \emptyset \ , \tag{5.3}$$

Equivalently, we can construct the Büchi automaton $B_{\neg\phi}$ accepting all infinite behaviours of the negation of ϕ, and solve the emptiness problem of the intersection:

$$\mathcal{L}(B_{sys}) \cap \mathcal{L}(B_{\neg\phi}) = \emptyset \ . \tag{5.4}$$

The emptiness problem itself is then solved by constructing a Büchi automaton, say B, such that $\mathcal{L}(B) = \mathcal{L}(B_{sys}) \cap \mathcal{L}(B_{\neg\phi})$, and then determine whether $\mathcal{L}(B) = \emptyset$. In fact, the Büchi automaton B is constructed as the product of B_{sys} and $B_{\neg\phi}$, i.e., $B = B_{sys} \times B_{\neg\phi}$.

5.4.3 The Schwoon and Esparza NDFS Algorithm

Determining whether the language of a Büchi automaton B is (non-)empty, regardless whether B is constructed as the product of two Büchi automata, boils down to identifying a word for which there exists an accepting run. For finite Büchi automata, an accepting run exists if and only if there exists a non-trivial *strongly connected component* (SCC, for short) that is reachable from the initial state and contains at least one accepting state. An SCC of a graph G is a maximal subgraph G' such that there exists a path from any node to any other node in G'. Stated differently, the finitely representable language of a Büchi automaton is non-empty if there exists an accepting state, say s, that is both reachable from the initial state and from itself.

The algorithm we propose in Section 5.6 for verifying arbitrary graph production systems is based on the nested depth-first-search (NDFS, for short) algorithm as described by Schwoon and Esparza [171], which optimizes the original NDFS algorithm introduced by Courcoubetis et al. [47]. We will refer to the algorithm by Schwoon and Esparza as the *SE*-algorithm. Algorithm 1 depicts the skeleton of the *SE*-algorithm for a given Büchi automaton $B = (S, \Sigma, \rightarrow, s_0, F)$. The set of successor states of a state s is denoted $succs(s)$. In the course of the algorithm, states are either coloured *white*, *cyan*, *blue*, or *red*; the colours should be interpreted as follows:

a state s is coloured white if s is freshly generated;

a state s is coloured cyan if s's successors are not yet fully explored;

a state s is coloured blue if the blue search for s has finished and s has not yet been reached in a red search;

a state s is coloured red if s has both been reached in the blue and red search.

In NDFS algorithms, the outer DFS (here called the *blue* search) explores the state space. When an accepting state and all its successor states are fully explored, a second DFS (*red* search) is started. The aim of the red search is to identify accepting runs in the Büchi automaton. By properly keeping track of a *search stack*, the *SE*-algorithm can also produce an actual counter-example when reporting an accepting cycle.

Proposition 5.7. *If the SE-algorithm reports a cycle, then some reachable accepting state belongs to a non-trivial SCC. Also, when a reachable accepting state belongs to a non-trivial SCC, the SE-algorithm will report a cycle.*

5.5 Model Checking Graph Production Systems

As mentioned in Section 5.1, termination of graph production systems is, in general, undecidable. The sequential approach to model checking graph production system as we proposed in [113], can therefore often not be applied. Techniques like on-the-fly model checking can yield useful results, even if the state space of the graph production system under consideration is infinite. Nevertheless, there are cases in which this approach does not help either. The verification procedure could for instance dive into a correct, though infinite, part of the state space. In such cases, the procedure will never terminate and therefore also not produce a counter-example although other, finite, parts of such state spaces might very well contain (small) counter-examples. One way to partially solve this problem is to include some form of *randomness* in the exploration strategy to make sure that different runs of the verification procedure explore the same state space, though possibly in a different order. Repeating the verification procedure often enough would then reduce the chance of incorrectly concluding that the system satisfies its requirements. Brim et al. [23] have shown that applying randomization for deciding which states to store indeed improves the verification procedure, in the average case. Grosu and Smolka [94] proposed an *LTL* model checking algorithm that walks through the state space using a technique called *random sampling*. Their experiments have shown that their approach is fast, memory-efficient, and extremely scalable.

An alternative solution to the above mentioned problem is to apply ideas from bounded model checking (recall Section 5.2). Instead of fully exploring one

Algorithm 1 Nested DFS algorithm by Schwoon and Esparza [171].

```
procedure Check(s_0)
    DFS_blue(s_0);
    report system ok;
end procedure

procedure DFS_blue(s)
    s.colour := cyan;
    for all t ∈ succs(s) do
        if t.colour = cyan ∧ (s ∈ F ∨ t ∈ F) then
            report cycle;
        else if t.colour = white then
            DFS_blue(t);
        end if
    end for
    if s ∈ F then
        DFS_red(s);
        s.colour := red;
    else
        s.colour := blue;
    end if
end procedure

procedure DFS_red(s)
    for all t ∈ succs(s) do
        if t.colour = cyan then
            report cycle;
        else if t.colour = blue then
            t.colour := red;
            DFS_red(t);
        end if
    end for
end procedure
```

branch of the state space at the risk of missing small counter-examples in other branches, the bounded model checking approach ensures that counter-examples that are *finitely representable* will always be identified with a finite amount of resources when using *proper* boundary conditions.

This raises the following three questions:

- When are counter-examples finitely representable?
- How should we deal with non-finitely representable counter-examples?
- What are *proper* boundary conditions?

The first question has been answered in Section 5.4.1, by introducing the notion of a *reachable accepting cycle*: a counter-example is not finitely representable if it does not represent a reachable accepting cycle. Concerning the second question, in the current framework (i.e., explicit-state model checking without abstraction) we cannot deal with counter-examples that are not finitely representable, simply because we cannot identify them with a finite amount of resources. In this work we therefore restrict to identifying finitely representable counter-examples. Concerning the third question, proper boundary conditions are boundary conditions that generate ever-larger, though *finite*, sub-state spaces. The use of proper boundary conditions is essential in our framework.

The algorithm we propose for model checking arbitrary graph production systems is based on the principles of the algorithm by Courcoubetis et al. Actually, we take the improved version proposed by Schwoon and Esparza [171] as our starting point. We have adapted the algorithm to model check state spaces in an iterative fashion, such that in subsequent iterations ever-larger, though finite, sub-state spaces are verified.

In this section we introduce some basic concepts on which our on-the-fly bounded model checking algorithm relies, namely *boundary conditions* and *approximation sequences*. We also discuss how to obtain a Büchi automaton accepting all infinite behaviours of a system that is specified as a graph production system.

5.5.1 Boundary Conditions

As discussed before we will combine on-the-fly with bounded model checking. This means that the state space will be verified in an iterative fashion. In each iteration a finite sub-state space will be verified in an on-the-fly manner. *Boundary conditions* determine which transitions belong to specific sub-state spaces

and which do not. The idea is that after verifying a sub-state space without finding a path that falsifies the checked property, a bigger part of the state space will be verified. That is, the boundary is updated after every successful iteration.

A boundary condition consists of a sequence of sets of transitions such that every element in the sequence is a superset of the previous element. If **Trans** denotes the set of all transitions, the concept of boundary condition can be formalized as follows.

Definition 5.8 (boundary condition)**.** *A boundary condition is a sequence* $b_1, b_2, \ldots$ *with* $b_i \in \mathbf{2}^{\mathbf{Trans}}$*, for* $i \geq 1$*, such that* $b_i \subseteq b_j$ *for all* i *and* j *with* $1 \leq i < j$*.*

A boundary condition $b = b_1, b_2, \ldots$ *is a* proper *boundary condition if all elements of* b *are finite, and for all* $t \in \mathbf{Trans} : \exists i : t \in b_i$*.*

In the sequel, "increasing" or "updating" the boundary condition b means that we replace the element from b used for exploration by its successive element in b. In Section 5.7.3, we elaborate on appropriate boundary conditions when applying our model checking algorithm to graph production systems.

5.5.2 Approximation Sequences

The base assumption for our algorithm to produce correct results is that the boundary conditions give rise to partial state spaces that are finite. Such partial state spaces will be called *initial fragments*. The basic idea is that increasing the boundary condition results in a sequence of ever-larger sub-state spaces, such that for every pair of subsequent sub-state spaces, the smaller one is fully contained in the bigger one. This notion of full containment can formally be defined as follows.

Definition 5.9 (initial fragment)**.** *Let* B_1 *and* B_2 *be two Büchi automata.* B_1 *is an* initial fragment *of* B_2*, denoted* $B_1 \sqsubseteq B_2$*, if the following conditions are satisfied:*

- $s_{0,1} = s_{0,2}$*;*
- $S_1 \subseteq S_2$ *and* S_1 *is finite;*
- $\rightarrow_1 \subseteq \rightarrow_2$*, such that all states in* S_1 *are* $\rightarrow_1$*-reachable;*
- $F_1 = F_2 \cap S_1$*.*

The following result follows immediately from the above definition.

Lemma 5.10. *The initial fragment relation $\sqsubseteq$ is a partial order.*

Proof. We have to prove that $\sqsubseteq$ is reflexive, transitive, and anti-symmetric. These properties follow immediately from Def. 5.9. □

Based on the initial fragment relation we introduce the concept of an *approximation sequence* for Büchi automata.

Definition 5.11 (approximation sequence)**.** *Let B be a Büchi automaton. An* approximation sequence *for B is a sequence $B_1, B_2, \ldots$, of Büchi automata such that*

- $B_i \sqsubseteq B_j$, *for* $1 \leq i \leq j$;
- $\bigsqcup_{i \geq 1} B_i = B$.

where $\bigsqcup_{i \geq 1} B_i$ denotes the union of all initial fragments B_i of B.

Given a proper boundary condition $b = b_1, b_2, \ldots$ for some Büchi automaton B, one can easily verify that b gives rise to an approximation sequence $B_1, B_2, \ldots$ for B, in which B_i contains all states reachable by b_i-transitions, i.e., transitions that are selected by the boundary function b_i, for all $i \geq 0$. Based on the boundary condition $b_1, b_2, \ldots$, we can define the set of *border transitions* of the initial fragment B_i, denoted $BT_B(B_i)$, including those transitions t of which the source state is contained in B_i but that are not selected by b_i, i.e., $t \notin b_i$. Formally,

$$BT_B(B_i) \quad = \quad \{t \in \rightarrow_B \mid \mathsf{src}(t) \in S_i \ \wedge \ t \notin b_i\} \ .$$

Fig. 5.9 visualizes the notion of border transitions. Assume that the triangles represent the continuations of the gray states. Then, the white states, the bullet-state (representing an accepting state) and the solid transitions (contained in the gray drop-like area) together form an initial fragment, say B_i, of the entire Büchi automaton, say B. The dashed transitions represent the border transitions of B_i.

Note that we do not require that for an approximation sequence $B_1, B_2, \ldots$, border transitions of some initial fragment, say B_i, are included in B_{i+1} (see Section 5.7.4). It might even be the case that the target state of a border transition of some initial fragment B_i is included in B_i since that state might be reachable from the initial state via transitions that are selected by the boundary function under consideration.

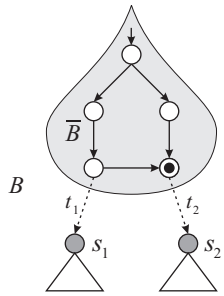


Figure 5.9: Initial fragment $\bar{B} \sqsubseteq B$ with border transitions t_1 and t_2.

5.5.3 From Graph Production Systems to Büchi Automata

In the context of *LTL* model checking using the automata theoretic approach, the system must be modelled as a Büchi automaton. We could define a translation from graph transition systems to Büchi automata in a similar way as we have done for Kripke structures (recall Section 5.3.3). However, in the literature, the relation between Kripke structures and Büchi automata has already been worked out thoroughly; see, e.g., [37]. In fact, there is not so big a difference between Kripke structures and Büchi automata. Basically, in our context (in which Büchi automata have labelled transitions) we observe the following differences:

- Kripke structures include a labelling function over states and have unlabelled transitions, whereas Büchi automata have labelled transitions and unlabelled states;
- Büchi automata include a designated set of accepting states, whereas in Kripke structures there is no such a distinction between states.

Despite the above differences, a Kripke structure $K = (S, \rightarrow, s_0, L)$ over a set AP of atomic propositions directly corresponds to a Büchi automata $B = (S', \Sigma, \rightarrow', \iota, F)$, with $\iota \notin S$, where all states are accepting (i.e., $F = S' = S \cup \{\iota\}$) and the alphabet is the powerset of the atomic propositions (i.e., $\Sigma = \mathbf{2}^{AP}$). The transition relation $\rightarrow'$ is then defined as follows:

$$\rightarrow' \quad := \quad \{(\iota, a, s_0) \mid L(s_0) = a\} \ \cup \ \{(s, a, s') \mid (s, s') \in \rightarrow \ \wedge L(s') = a\} \ .$$

Remark 5.12. *For graph production systems that produce finite state spaces, our model checking problem is decidable. Since termination of graph production*

systems is undecidable, this affects the decidablity of our problem. For non-termination graph production systems of which all counter-examples are lasso-like, our algorithm is a semi-decision *algorithm. This means that whenever a counter-example exists, our algorithm will find it with finitely many resources. Otherwise, our algorithm continues forever. In general, our algorithm is not even a semi-decision algorithm since counter-examples that are not lasso-like cannot be identified with finitely many resources.*

5.5.4 Example: Circular Extensible Buffer

To make the example of the circular buffer a bit more interesting, we include the extend-rule of which individual applications extend the Buffer with one fresh Cell in case the Buffer is completely filled. This rule is depicted in Fig. 5.10. One can easily verify that the state space of the new circular buffer becomes infinite. A part of the graph transition system is shown in Fig. 5.11. The gray colour of the state $s18$ at the right bottom of the figure denotes the fact that this state has not been explored.

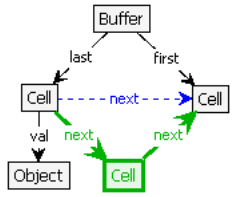


Figure 5.10: Graph transformation rule extending a full Buffer.

Here, it would be interesting to check a property concerning the way the Buffer is extended. For example, requiring that from a state in which the buffer can be extended, we can always eventually put in element in or get an element from the Buffer, can be expressed in *LTL* as follows:

$$\mathsf{G}(\mathsf{extend} \rightarrow \mathsf{F}(\mathsf{put} \vee \mathsf{get})) \ . \tag{5.5}$$

Indeed, the above property is satisfied by the circular extensible buffer since in every state reached by an extend-application, we can in fact both put an element in and get an element from the Buffer.

Another property to check is the following, requiring that whenever the

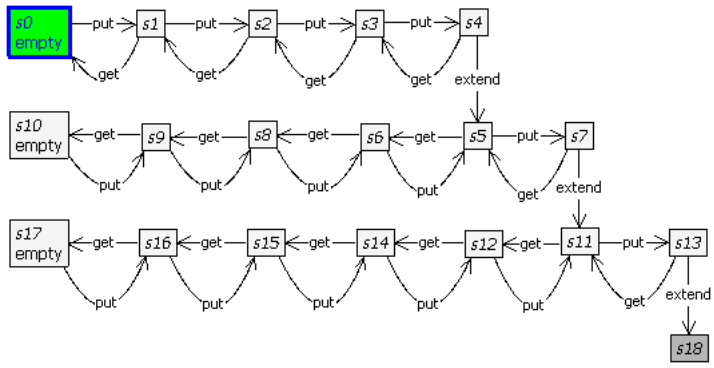


Figure 5.11: Part of the infinite state space of the circular extensible buffer.

Buffer is extended, we will eventually reach a state in which the Buffer is empty:

$$\mathsf{G}(\mathsf{extend} \rightarrow \mathsf{F}\ \mathsf{empty}) \ . \tag{5.6}$$

The graph transition system depicted in Fig. 5.11, however, contains infinite paths that do not satisfy this property. Executing the following sequence of transitions results in such a path:

$$\mathsf{put}\ \mathsf{put}\ \mathsf{put}\ \mathsf{put}\ \mathsf{extend}\ \mathsf{get}\ \mathsf{put}\ \mathsf{get}\ \mathsf{put}\ \cdots$$

That is to say, the above path is a counter-example for property (5.6).

5.6 On-the-Fly Bounded Model Checking Algorithm

The algorithm we propose for on-the-fly bounded model checking of graph production systems is shown in Algorithm 2 and Algorithm 3 and extends the *SE*-algorithm (Algorithm 1). In the sequel we will refer to the algorithm obtained by combining Algorithm 2 and Algorithm 3 as the SE^+-algorithm (indicating that is a small extension of the original *SE*-algorithm). By specifying proper boundary conditions, the SE^+-algorithm generates an approximation sequence and verifies every initial fragment on-the-fly. The new Check-procedure ensures that the algorithm continues as long as initial fragments have been successfully verified (i.e., without finding counter-examples) and boundary transitions have been encountered. The SE^+-algorithm abstracts from the actual boundary

condition by iterating over the initial fragments of the approximation sequence generated by the boundary condition under consideration.

The SE^+-algorithm assumes a given proper boundary condition $b_1, b_2, \ldots$ for a (possibly infinite) Büchi automaton $B = (S, \Sigma, \rightarrow, s_0, F)$. The integer i identifies the current boundary function b_i used to check whether transitions must be traversed or not. We then say that the SE^+-algorithm is in the i-th iteration for verifying B. Obviously, the integer i also indicates the number of initial fragments that have successfully been verified, i.e., in which no reachable accepting cycle has been detected. Furthermore, a flag *continue* is used to determine whether a next iteration is required. This flag is set to `true` if a border transition is encountered. For the SE^+-algorithm to be correct, the state-colouring of the previous stage should be undone after every iteration (line 5 in Algorithm 2).

The comments on the right-hand-side of Algorithm 3 indicate the places where the blue and red search of the SE^+-algorithm deviate from the original SE-algorithm of Algorithm 1. In the new blue and red search (Algorithm 3), the SE^+-algorithm iterates over the outgoing transitions of the given state s. If the transition (s, a, s') is selected by the current boundary function (line 4 and line 23 for the blue and red search, respectively, in Algorithm 3), i.e., $(s, a, s') \in b_i$, it is processed as in the original SE-algorithm (lines 5–9 and lines 24–29, respectively). Otherwise, $\langle s, a, s' \rangle$ is a border transition of B_i. Then, in the blue search the flag *continue* is set to `true` (line 11), and in the red search such transitions are not considered.

Algorithm 2 New Check procedure.

```
 1: procedure CHECK(s_0)
 2:     bool continue := true;
 3:     int i := 0;
 4:     while continue do
 5:         resetColours();
 6:         continue := false;
 7:         i := i + 1;
 8:         DFS_BLUE(s_0, i);
 9:     end while
10:     report system ok;
11: end procedure
```

Algorithm 3 Adapted blue and red search.

```
 1: procedure DFS_BLUE(s, i)                                  // additional parameter
 2:     s.colour := cyan;
 3:     for all ⟨s, a, s'⟩ ∈ → do                             // updated
 4:         if (s, a, s') ∈ b_i then                          // added
 5:             if s'.colour = cyan ∧ (s ∈ F ∨ s' ∈ F) then
 6:                 report cycle;
 7:             else if s'.colour = white then
 8:                 DFS_BLUE(s', i);
 9:             end if
10:         else
11:             continue := true;                            // added
12:         end if
13:     end for
14:     if s ∈ F then
15:         DFS_RED(s, i);                                   // updated
16:         s.colour := red;
17:     else
18:         s.colour := blue;
19:     end if
20: end procedure

21: procedure DFS_RED(s, i)                                  // additional parameter
22:     for all ⟨s, a, s'⟩ ∈ → do                             // updated
23:         if (s, a, s') ∈ b_i then                          // added
24:             if s'.colour = cyan then
25:                 report cycle;
26:             else if s'.colour = blue then
27:                 s'.colour := red;
28:                 DFS_RED(s', i);                          // updated
29:             end if
30:         end if
31:     end for
32: end procedure
```

5.6.1 Correctness and Relative Completeness

The correctness of the SE^+-algorithm is partially guaranteed by the correctness of the original SE-algorithm (recall Proposition 5.7. Since the SE^+-algorithm applies the SE-algorithm on finite parts of the (possibly infinite) state space, every accepting cycle reported by our algorithm is indeed an accepting run in that finite sub-state space.

Theorem 5.13 (soundness)**.** *Let B be a Büchi automaton and $b = b_1, b_2, \ldots$, be a proper boundary condition. Suppose we apply the SE^+-algorithm to B using the boundary condition b. If the SE^+-algorithm reports the existence of an accepting cycle in the i-th iteration then there exists a reachable accepting cycle that is entirely contained only in all components B_k of the approximation sequence over B under b, for $k \geq i$, and in B itself.*

Proof. Assume the SE^+-algorithm reports the existence of a accepting cycle in the i-th iteration. The correctness of the SE-algorithm also applies to the SE^+-algorithm which implies that there indeed exists a reachable accepting cycle in B_i, i.e., the i-th component of the approximation sequence $B_1, B_2, \ldots$, generated by b. Since it successfully verified all initial fragments up to $i-1$, it holds that the accepting cycle includes at least one transition t such that $t \in b_i$ and $t \notin b_j$, for all $j < i$. Therefore, the accepting cycle will first be identified in the i-th iteration. The fact that b is a proper boundary condition then guarantees that the accepting cycle is entirely contained in all components B_k, for $k \geq i$. Obviously, the accepting cycle is then also contained in B. □

As we have mentioned in Remark 5.12, our algorithm for model checking arbitrary graph production systems cannot always decide with finitely many resources whether there are counter-examples or not. Nevertheless, whenever there exists a lasso-like counter-example (i.e., it contains a reachable accepting cycle) the SE^+-algorithm is guaranteed to find it with a finite amount of time.

Theorem 5.14 (relative completeness)**.** *Let B be a Büchi automaton and $b = b_1, b_2, \ldots$, be a proper boundary condition. Suppose we apply the SE^+-algorithm to B using the boundary condition b. If there exists a reachable accepting cycle, the SE^+-algorithm will find it with a finite amount of resources.*

Proof. Suppose there exists a reachable accepting cycle ρ. Then, there exists an $i \geq 1$ such that ρ is completely contained in the component B_i of the approximation sequence over B under b. The general assumption that B_i is finite and the completeness of the SE-algorithm implies that the SE^+-algorithm will identify ρ as a reachable accepting cycle. □

Since there is only a finite amount of time and memory available for the verification procedure, we either predefine the maximal number of iterations to be performed, or stop the verification procedure manually after a certain number of iterations. Thus, our algorithm can guarantee completeness only if the automaton is finite and within the chosen upper bound.

5.6.2 Complexity Analysis

In this section we take a closer look at how the run-time performance of our algorithm depends on the complexity of computing and traversing transitions. For example, in the graph transformation framework, computing a transition includes finding a matching for some rule and constructing the target state. In the presence of isomorphism checking, we also need to determine whether the target state is isomorphic to some other state seen before. Traversing a transition then consists of setting the current state to the target state of the selected transition. We also perform some analysis on the actual time-gain when reusing exploration results based on different growth functions of the approximation sequences. Although reusing exploration results often pays off, knowing the exact gain can be useful, for instance, for deciding whether it is worth investing effort in developing and implementing proper algorithms. At the end of this section, we elaborate on the effect of the 'speed' of approximation.

This section is based on the following assumptions:

- the cost of computing and traversing transitions are *constant* (and will be denoted c and t, respectively);
- in the classical case of model checking, we have $c \approx t$;
- in the graph transformation framework, we have $c \gg t$.

Although this is a very coarse abstraction, the analysis performed in the following paragraphs provides useful insights.

5.6.2.1 Reusing Computation Results

In a naive setting, i.e., where transitions are recomputed for every next step of the approximation, the time-complexity of verifying a Büchi automaton B up to the n-th iteration ($n \geq 1$) of an approximation sequence $B_1, B_2, \ldots$ over B, can be specified as follows:

$$\mathcal{T}(\mathit{verify}(B, n)) \quad = \quad (2t + c) \cdot \sum_{i=1}^{n} |B_i| \tag{5.7}$$

where $|B_i|$ is the size, i.e., the number of transitions, of the initial fragment B_i. Since every transition might be traversed in both the blue and the red search, the traversal cost per transition must be doubled.

If Algorithm 3 would reuse the exploration results of initial fragments that have been successfully verified, it has to traverse the existing transitions (from previous iterations) and compute the outgoing transitions only for unexplored states reached by border transitions and fresh states up to the next boundary. Thus, the cost can then be split into two components: on the one hand the cost for computing all explored transitions once, and on the other hand the cost for traversing the previous initial fragment. The time-complexity of the algorithm in the optimized setting, i.e., where computation results are reused, can thus be specified as follows (the star denotes the fact that we consider the optimized setting):

$$\mathcal{T}(\mathit{verify}^*(B,n)) \quad = \quad c \cdot |B_n| + 2t \cdot \sum_{i=1}^{n} |B_i| \ . \tag{5.8}$$

The difference in order of magnitude of t and c determines whether the right-hand-side sum of (5.8) mainly depends on its first component, its second component, or equally on both components.

If traversing and computing a transition are equally expensive, for (5.7) both t and c have a comparable effect on the overall complexity of the verification procedure. In such cases, reuse of exploration results does not significantly pay off. In the graph transformation framework, determining whether a rule is applicable is a *graph matching problem*: we need to identify an occurrence (or the occurrences) of the condition of the rule (which is a graph itself) in the *host graph*. This results in c being potentially much bigger then t. Reusing the exploration results from previous initial fragments might then have a significant positive effect on the overall performance of the algorithm.

The actual time-gain can be determined in an *absolute* and a *relative* fashion. Here, we focus on the latter, since from that we can easily deduce the former. The relative gain, denoted $\mathcal{G}$, is computed as follows:

$$\mathcal{G} \quad = \quad \frac{\mathcal{T}(\mathit{verify}(B,n)) - \mathcal{T}(\mathit{verify}^*(B,n))}{\mathcal{T}(\mathit{verify}(B,n))} \tag{5.9}$$

If the above equation approaches to 1, this means that the time requirements of the algorithm when reusing computation results are negligble with respect to the setting in which transitions are computed over and over again.

By filling in the general expressions for both time-complexities in (5.9), we get the following reduced expression:

$$\begin{aligned} \mathcal{G} &= \frac{(2t+c)\cdot\sum_{i=1}^{n}|B_i| - (c\cdot|B_n| + 2t\cdot\sum_{i=1}^{n}|B_i|)}{(2t+c)\cdot\sum_{i=1}^{n}|B_i|} \\ &= \frac{c\cdot(\sum_{i=1}^{n}|B_i| - |B_n|)}{(2t+c)\cdot\sum_{i=1}^{n}|B_i|} \end{aligned}$$

Even if $c \gg t$, the factor with which the state space grows in every next iteration can still diminish the benefit of reusing computation results. We distinguish between cases in which the size of initial fragments grows linearly, quadratically (and in general polynomially), or exponentially. In Section 5.7 we perform some experiments on graph transformation systems that produce state spaces with a linear or exponential growth factor.

Linear Growth Factor. In cases where the size of initial fragments grows *linearly*, we have

$$|B_i| = g_0 + g_1\cdot(i-1) \tag{5.10}$$

for $1 \leq i \leq n$, where $g_0 = |B_1|$ and g_1 is the linear *growth factor*. The summation $\sum_{i=1}^{n}|B_i|$ can be rewritten to a simple expression using a well-known geometric sequence as follows:

$$\begin{aligned} \sum_{i=1}^{n}(g_1\cdot(i-1)) &= g_1\cdot\sum_{i=1}^{n}(i-1) \\ &= g_1\cdot\sum_{i=1}^{n}(i-1) \\ &= g_1\cdot\sum_{i=1}^{n-1} i \\ &= g_1\cdot\frac{n(n-1)}{2} \quad . \end{aligned}$$

Combining (5.7) with the result of the above rewriting yields the following for-

mula:

$$\mathcal{T}(\mathit{verify}(B,n)) \quad = \quad (2t+c)\cdot\left(g_0\cdot n + g_1\cdot\frac{n\cdot(n-1)}{2}\right) \quad . \qquad (5.11)$$

This means that under the assumptions that (1) computing transitions is much more expensive than traversing them and (2) the state space grows linearly in the number of iterations, re-exploring the state space every next iteration results in a time complexity of the algorithm which is in the order $c\cdot g_1\cdot\frac{n^2}{2}$, where g_1 is the growth factor and n is the number of iterations.

The relative time-gain can now be computed by filling in the expression for $|B_n|$ as from (5.10) and the simplified expression for $\sum_{i=1}^{n}|B_i|$ in the reduced form of (5.9). This results in the following expression for the time-gain in the linear case, denoted $\mathcal{G}_{lin}$:

$$\begin{aligned}\mathcal{G}_{lin} &= \frac{c\cdot(\sum_{i=1}^{n}|B_i| - |B_n|)}{(2t+c)\cdot\sum_{i=1}^{n}|B_i|}\\ &= \frac{c\cdot\left(g_0\cdot n + g_1\cdot\frac{n\cdot(n-1)}{2} - g_0 - g_1\cdot(n-1)\right)}{(2t+c)\cdot\left(g_0\cdot n + g_1\cdot\frac{n\cdot(n-1)}{2}\right)}\end{aligned}$$

For sufficiently large n, the above expression approaches $\frac{c}{2t+c}$. For the graph transformation framework, where we assumed $c \gg t$, this fraction is approximately 1. The quotient of the reuse time-complexity with respect to the recompute time-complexity approaches $\frac{2t\cdot g_1\cdot n + c\cdot g_1}{(2t+c)\cdot g_1\cdot n}$. Now, the assumption that $c \gg t$ implies that this fraction is approximately 0.

Quadratic Growth Factor. In cases where the size of initial fragments grows *quadratically* in the number of iterations, a similar analysis can be done based on the following formula:

$$|B_i| \quad = \quad g_0 + g_1\cdot(i-1) + g_2\cdot(i-1)^2 \qquad (5.12)$$

where, again, $g_0 = |B_1|$, g_1 is the linear growth factor, and g_2 is the quadratic growth factor. Applying similar reductions yields the following time complex-

ity[1]:

$$\mathcal{T}(\mathit{verify}(B,n)) \quad \approx \quad (2t+c)\cdot(g_0\cdot n + g_1\cdot\frac{n^2}{2} + g_2\cdot\frac{1}{3}\cdot n^3) \tag{5.13}$$

In this case, the relative gain, denoted $\mathcal{G}_{quad}$, is

$$\mathcal{G}_{quad} \quad \approx \quad \frac{c\cdot(g_0\cdot n + g_1\cdot\frac{n^2}{2} + g_2\cdot\frac{n^3}{3}) - c\cdot(g_0 + g_1\cdot(n-1) + g_2\cdot(n-1)^2)}{(2t+c)\cdot(g_0\cdot n + g_1\cdot\frac{n^2}{2} + g_2\cdot\frac{n^3}{3})} \tag{5.14}$$

Similar to (5.12), (5.14) approaches $\frac{c}{2t+c}$, for sufficiently large n.

Generalizing the above analysis, one can verify that if the size of subsequent initial fragments grows polynomially, i.e.,

$$|B_i| \quad = \quad p(i,k)$$

where $p(i,k)$ is a polynomial over i of order k for $k \in \{0,1,2,\ldots\}$, the assumption that $c \gg t$ implies that the relative time-gain when reusing exploration results is approximately 100% when the number of iterations is sufficiently large.

Exponential Growth Factor. In cases where the size of subsequent initial fragments grows *exponentially* in the number of iterations, we have

$$|B_i| \quad = \quad g_0 + g_1\cdot x^{i-1} \tag{5.15}$$

for $i > 1$, where x is the exponential growth factor or *base factor* with $x > 1$, and thus

$$\mathcal{T}(\mathit{verify}(B,n)) \quad = \quad (2t+c)\cdot(n\cdot g_0 + g_1\cdot\sum_{i=1}^{n} x^{i-1}) \tag{5.16}$$

We can express the relative gain in the exponential case, denoted $\mathcal{G}_{exp}$, as follows:

$$\mathcal{G}_{exp} \quad = \quad \frac{c\cdot(n\cdot g_0 + g_1\cdot\sum_{i=1}^{n} x^{i-1}) - c\cdot(g_0 + g_1\cdot x^{n-1})}{c\cdot(n\cdot g_0 + g_1\cdot\sum_{i=1}^{n} x^{i-1})} \tag{5.17}$$

[1]For this reduction we used the fact that $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

Using the well-known geometric sequence $\sum_{i=1}^{n} x^i = \frac{x^{n+1}-x}{x-1}$, the numerator of the above fraction can be reduced as follows:

$$\begin{aligned}
& c \cdot \left(n \cdot g_0 + g_1 \cdot \sum_{i=1}^{n} x^{i-1} \right) - c \cdot \left(g_0 + g_1 \cdot x^{n-1} \right) \\
= \; & c \cdot \left(g_1 \cdot \sum_{i=1}^{n} x^{i-1} - g_1 \cdot x^{n-1} + n \cdot g_0 - g_0 \right) \\
= \; & c \cdot \left(g_1 \cdot \left(\sum_{i=1}^{n} x^{i-1} - x^{n-1} \right) + (n-1) \cdot g_0 \right) \\
= \; & c \cdot \left(g_1 \cdot \sum_{i=1}^{n-1} x^{i-1} + (n-1) \cdot g_0 \right) \\
= \; & c \cdot \left(g_1 \cdot \left(x^0 + \sum_{i=1}^{n-2} x^i \right) + (n-1) \cdot g_0 \right) \\
= \; & c \cdot \left(g_1 \cdot \left(1 + \frac{x^{n-1} - x}{x-1} \right) + (n-1) \cdot g_0 \right)
\end{aligned}$$

For n sufficiently large, the numerator of (5.17) approximates to $c \cdot g_1 \cdot x^{n-2}$, whereas the denominator of (5.17) approximates to $c \cdot g_1 \cdot x^{n-1}$. Thus, it follows that for sufficiently large n, (5.17) approximates to $\frac{1}{x}$. This means that the relative gain does not depend on the number of iterations, but only on the base factor x. Intuitively, this means that the larger the base factor, the smaller the relative gain of reusing exploration results.

5.6.2.2 Speed of Approximation

Another way of influencing the performance of the overall verification procedure requires some knowledge of the system. If counter-examples are likely to occur in later steps of the approximation, one could tune the algorithm by increasing the size of the next approximation. Intuitively, this means that, for the finite case, the approximation sequence contains fewer components. Stated differently, it takes fewer iterations to verify the sub-state space containing the error, and therefore many transitions are traversed less often.

Suppose we have two approximation sequences $\rho = B_1, B_2, \ldots$, and $\rho' = B'_1, B'_2, \ldots$ with $B'_i = B_{5i}$. If B_7 contains a counter-example (and thus B'_2

also), then for the approximation sequence ρ the algorithm iterates over all B_i, for $1 \leq i \leq 6$, which will all be verified successfully. When verifying B_7, the algorithm will find the counter-example. When taking ρ' as the approximation sequence, the algorithm will find the counter-example in B'_2 in which case it traverses the transitions of all B_i, for $1 \leq i \leq 5$ (at most) three times, instead of (at most) twice during every iteration as for ρ. The down-side of this approach is that in searching through $B'_2 = B_{10}$, the algorithm might also compute and traverse transitions that are only included in B_k, for $k > 7$, and would therefore only be considered for ρ if B_7 had not contained a counter-example. This is especially dangerous in case the growth is exponential.

5.7 Implementation and Experiments

We have implemented both the *CTL* model checking algorithm from [37, 17] and the on-the-fly bounded model checking algorithm for *LTL* (Algorithm 2 and Algorithm 3) in the graph transformation tool Groove. In [113] we reported on some performance statistics comparing Groove with the model checker Spin [104]. In this section we discuss some implementation issues. Furthermore, we elaborate on how to specify proper boundary conditions in the graph transformation framework in general, and for the Groove graph formalism specifically. Finally, we report on various experiments (including additional experiments with the *CTL* model checking algorithm) and summarize our observations on those experiments.

5.7.1 From LTL Formulae to Büchi Automata

For our experiments with the on-the-fly bounded algorithm we have used the ltl2ba library which implements the algorithm proposed by Gastin and Oddoux [87], using the classical approach as one of the steps to eventually produce rather small Büchi automaton in an efficient way. ltl2ba produces Büchi automata in which the transitions are labelled with conjunctions over the set AP of atomic propositions. When constructing the product of such a Büchi automaton with the Büchi automaton representing the system's state space, this causes a mismatch, since both Büchi automata have different alphabets. This mismatch can conceptually be solved by defining a new transition relation $\rightarrow'$ for the system automaton which contains a set of transitions between two successive states s and s' in the system's state space, one for each subset of rule names for which there exists an outgoing transition. Then, the product can be

constructed as from Def. 5.5.

5.7.2 State Colouring

As in any implementation of the NDFS algorithm, state objects are extended with some additional bits to effectively store state-colours, allowing efficient colour-checking. As specified in Algorithm 2, state-colours are reset before verifying every next approximation step. In practice, this is achieved by choosing a fresh colour scheme for the next iteration. The basic algorithm as presented in Algorithm 2 starts every next iteration from the initial state. Successfully verifying some initial fragment, say B_i, means that the blue search has terminated for all states of B_i. All states of B_i will thus be coloured either blue or red in the current colour scheme, depending on whether they have only been part of the blue search or also be subject to the red search, respectively. Therefore, it is sufficient for the algorithm to use two copies of the original colour scheme; the algorithm switches the colour scheme every next iteration.

5.7.3 Boundary Conditions for GPSs

When verifying graph production systems specified in Groove, there are different ways of naturally specifying boundary conditions such that state spaces up to those boundaries can (formally) be proven finite. The finiteness of such state spaces is partly due to the fact that Groove generates graphs up to isomorphism [161].

We distinguish the following two approaches to specify proper boundary conditions.

- As mentioned in Section 5.1, the *size* of a state plays a central role in boundary specifications. In the graph transformation context, the boundary could be specified in term of the size of the graphs that are allowed to be generated. Since Groove performs graph transformation on so-called *simple graphs*, defining the size of the graph as the number of nodes it contains, yields proper boundary conditions. That is to say, rules that only increase the number of edges can only generate a finite number of different states, since creating an edge between two nodes of that are already connected with a similar edge will cause the edges to be merged.
- Alternatively, the boundary can be specified by selecting the subset of rules which do not increase the size of the graph. In the former case, increasing the bound would mean to increase the size of graphs that are

allowed to be generated; in the latter case, increasing the bound would mean to (temporarily) allow rules outside the set of selected rules. If further knowledge of the graph production system is available, the subset of allowed rules can also be specified manually. Examples of such cases are subsets of rules that are designed (and possibly proven) to be terminating.

5.7.4 Experiments

We have performed some experiments on various graph production systems, each with their own characteristics such as, e.g., a high or low degree of symmetry or dynamism, and whether they generate finite or infinite state spaces. We analyze the results afterwards. All experiments were run on a 2.2 GHz Intel Centrino processor with 2 GB of memory; the Java Virtual Machine was initialized with 512 MB of memory and could use up to 1 GB.

TAAL Programs

In Chapter 4 we have developed an object-oriented language called Taal for which the (dynamic) semantics have been defined in terms of graph production systems taal-flow and taal-sem. The graph production system taal-flow defines a model transformation which transforms arbitrary Flat Abstract Syntax Graphs (fasgs) into the corresponding Program Graph (pg). The execution semantics of Taal is defined by taal-sem for which we will verify some properties. In this experiment we compare the performance of the *CTL* and *LTL* model checking algorithms implemented in Groove. Although *CTL* and *LTL* are expressively incomparable, there are some properties that can be expressed in both logics (see, e.g., [131, 75, 187]). In particular, by omitting the path quantifiers in the temporal formulae verified in the following experiment, we obtain equivalent *LTL* formulae. In order to make a fair comparison, we therefore focus on properties that can be expressed in both *CTL* and *LTL*.

Experiment 1. In this experiment we perform some analysis on the simulation of a Taal program given in Listing 5.1, which is similar to the one from Listing 4.2. The main changes are that the start expression calls the method fill() which introduces some non-determinism, and the method cut() of class Tulip which effectively does not change the length-attribute, thus introducing non-terminating behaviour. These main changes are shown in Listing 5.1. By varying the initial value of the length of a Flower, we generate state spaces of different sizes. We report on the time required to verify the state space against a number of

properties. For *CTL*, the total time is equal to the time needed to generate the state space increased with the time taken for the actual verification; for *LTL*, the verification is performed during state space exploration.

```
program vase

  { new Vase().fill() }

class Vase
   ...
  fill() {
    fork changeFlower(new Rose());
    fork changeFlower(new Tulip());
  }
   ...
endclass

class Tulip extends Flower
  myColor: String := 'orange';
   ...
  cut() {
    length := length.minus(0);
  }
   ...
endclass
...
endprogram
```

Listing 5.1: The fill()-method of class Vase.

Interesting properties we can check are listed below, specified as *CTL* formulae. Here, we focus on properties that are satisfied to compare generation times for the different approaches. The results are listed in Table 5.1 in which the first column indicates the initial length of any Flower-instance; the fourth column includes the time required to only generate the corresponding state space, i.e., without performing any kind of verification.

- For every method call (call) a corresponding implementation will eventually be found (resolve):

$$\mathsf{AG}(\mathsf{call} \rightarrow \mathsf{AF}(\mathsf{resolve})) \tag{5.18}$$

- Every object creation expression (create) will eventually result in an actual object (created) and going up the inheritance hierarchy for initialization (ascend) is complemented by going down the hierarchy (descend):

$$\mathsf{AG}((\mathsf{create} \rightarrow \mathsf{AF}(\mathsf{created})) \wedge (\mathsf{ascend} \rightarrow \mathsf{AF}(\mathsf{descend}))) \tag{5.19}$$

- The (properly nested) conjunction of (5.18) and (5.19):

$$\mathsf{AG}((\mathsf{create} \rightarrow \mathsf{AF}(\mathsf{created})) \wedge (\mathsf{ascend} \rightarrow \mathsf{AF}(\mathsf{descend})) \wedge (\mathsf{call} \rightarrow \mathsf{AF}(\mathsf{resolve}))) \tag{5.20}$$

length	States	Transitions	Time (ms)	Property	Time (ms)		Quotient
					CTL	*LTL*	*LTL* / *CTL*
45	4540	11396	11090	(5.18)	11125	14138	1.27
				(5.19)	11162	14535	1.30
				(5.20)	11195	15127	1.35
50	6120	15396	14693	(5.18)	14743	19882	1.34
				(5.19)	14796	20312	1.37
				(5.20)	14843	20809	1.40
55	7700	19396	18493	(5.18)	18561	25004	1.34
				(5.19)	18623	25748	1.38
				(5.20)	18688	26052	1.39
60	9280	23396	22366	(5.18)	22446	30280	1.34
				(5.19)	22521	30960	1.37
				(5.20)	22609	33435	1.47
65	10860	27396	26431	(5.18)	26528	37540	1.41
				(5.19)	26622	38014	1.42
				(5.20)	26724	38696	1.44
85	17180	43396	46814	(5.18)	46979	92848	1,97
				(5.19)	47155	> 100000	>2
				(5.20)	47327	> 100000	>2

Table 5.1: Results of experiment 1.

Circular Extensible Buffer

The example of the circular buffer with the extend-rule (recall Section 5.5.4) generates a very straightforward, though infinite, state space. For this example we will perform the following experiments in which the initial state represents the empty buffer of size three.

Experiment 2. In this experiment the state space of the circular buffer is generated iteratively using boundary conditions that limit the size of the graphs that are allowed to be generated. The experiment is repeated for different values for the *initial* and the *step* size. We fix the number of maximal iterations to 60. In Table 5.2 we report on the total number of states and transitions and the time required to generate them with and without reusing the exploration results from previous iterations. The property to be verified is the following:

$$\mathsf{G}(\mathsf{put} \rightarrow \mathsf{F}(\mathsf{get})) \tag{5.21}$$

stating that whenever we can put an element in the buffer, we will always reach a state in which we can get an element from the buffer. The property is satisfied in all experiments listed in Table 5.2.

Max. Itrs.	Boundary Initial	Boundary Step	States	Transitions	Time (ms) Restart	Time (ms) Reuse	Quotient
60	5	1	580	160	8168	406	20.1
60	5	2	2002	3886	36996	2186	16.9
60	5	3	4356	8634	111259	7885	14.1
60	10	1	652	1240	9827	454	21.6
60	10	2	2196	4268	42499	2437	17.4

Table 5.2: Results of experiment 2.

Experiment 3. We now generate the same state space as in the previous experiment, but using a different type of boundary condition. In this experiment we initially forbid applications of the extend rule. Whenever the state space generated by applications of the put and get rule has not resulted in a counter-example, we allow exploration paths containing a single extend-application. If neither of those exploration paths results in an accepting cycle, exploration paths containing up to two extend-applications are verified. This experiment is repeated for different upper bounds with respect to the number of extend-applications contained in single exploration paths. Here we also verify property (5.21). The results are contained in Table 5.3. The column headed "Itrs." states the different upper bounds.

Itrs.	States	Transitions	Time (ms) Restart	Time (ms) Reuse	Quotient
10	106	190	562	146	3.8
25	451	850	2223	401	5.5
50	1526	2950	14649	1325	11.0
75	3226	6300	66924	3803	17.6
100	5551	10900	181968	8003	22.7

Table 5.3: Results of experiment 3.

Leader Election Protocol

We have implemented the leader election protocol for unidirectional rings as proposed by Dolev et al. [56] as a graph production system in Groove; the rules and the start graph are included in Appendix D. At startup, every process

picks a unique number non-deterministically after which they send messages along the ring in a synchronous way. Depending on the content of the messages, a process decides whether it can still become the leader or whether it should become passive which means that it only passes through messages. A single run of the protocol should always end in a situation in which a single process has been identified as the leader. We have included a transformation rule (or actually a condition) called leader which is enabled in a state when there exists a leader. Additionally, we have extended the protocol with some rules that can change the initial configuration of the ring (i.e., increase or decrease the number of processes in the ring) as long as none of the processes has picked its unique number. This yields an infinite state space. Furthermore, by using a boundary condition that initially forbids applications of the increase rule, we can verify the protocol iteratively with respect to the number of participating processes.

Obviously, we are interested in whether all executions of the system eventually reach a state in which a leader has been elected, i.e., in which the leader condition is satisfied. This is specified by the following property

$$\mathsf{GF}(\mathsf{leader}) \tag{5.22}$$

Although the original protocol specification is correct, extending it with the increase and decrease rules introduces infinite executions that will never reach a state in which a leader has been elected. An example of such a execution is the path along which the ring is extended indefinitely, i.e., the path consisting of increase transitions only. For our implementation of the extended protocol we are actually interested in whether all executions in which the actual protocol starts (that is when the first message is sent along the ring) a leader will eventually be elected. A process sending its first message is modelled by a rule named init. The property we want to verify can thus be specified as follows:

$$\mathsf{G}(\mathsf{init} \rightarrow \mathsf{F}(\mathsf{leader})) \tag{5.23}$$

The path consisting of increase transitions only does not falsify property (5.23) although the behaviour is clearly unintended.

Experiment 4. This experiment consists of verifying a correct implementation of the leader election protocol against property (5.23). The boundary condition is specified by selecting all rules except the increase rule. The ring initially consists of two processes, thus in the fourth iteration the ring consists of five processes. We report on the time to verify the state space in which no counter-example

will be identified.

Experiment 5. For this experiment we have introduced an (artificial) error in the leader election protocol. The error occurs depending on the size of the ring. The rule causing the error is included in Appendix D (Section D.3). We perform the experiment for implementations in which the error occurs for ring sizes equal to five, six, and seven. We report on the time taken to find the error and the number of states and transition that were explored.

	Experiment 4			Experiment 5		
Itrs.	**States**	**Transitions**	**Time (ms)**	**States**	**Transitions**	**Time (ms)**
2	181	317	207			
3	958	2135	517			
4	6900	19258	2194			
5	62770	215592	19863	1023	2240	624
6	696194	2873353	1156156	6995	19395	2186
7				62914	215793	19546
8				969415	2873547	1183328

Table 5.4: Results of experiment 4.

5.7.5 Observations

The first observation from experiment 1 is that the *LTL* model checking algorithm is clearly more time consuming than the *CTL* algorithm (when the system is correct and its state space finite). When comparing the results for cases where the size of the state space doubles, we observe the following:

- from length $= 45$ to length $= 60$, the net verification time increases from 35 to 80 milliseconds for *CTL* and from 3048 to 7914 milliseconds for *LTL*;
- from length $= 50$ to length $= 65$, the net verification time increases from 50 to 97 milliseconds for *CTL* and from 5189 to 11109 milliseconds for *LTL*.

From the above marks it furthermore becomes clear that the time requirements of both the *CTL* and *LTL* model checking algorithms grow linearly with the size of the state space, which is to be expected from the classical complexity analysis on *CTL* and *LTL* (see, e.g., [188]).

Experiments 2 and 3 clearly indicate the potential benefit of reusing exploration results. From the figures of experiment 3 we can furthermore observe that the cost when re-exploring the state space are (about) quadratic in the size

of the initial fragments; when re-using exploration results, the cost is (about) linear in the size of the initial fragments. A comparison of the figures on experiment 4 and 5 indicates that the combination of on-the-fly and bounded model checking is indeed useful in practice: finding the counter-example in a ring of n processes does not require to fully generate the corresponding state space.

From experiments 1, 4 and 5 we observe that the time requirements of their last sub-experiments grew unproportionally compared to the other sub-experiments (for experiment 1 only in the *LTL* case). This is very likely due to the fact that in those last sub-experiments much time is spend on *garbage collection*.

A final observation is that the average time needed to generate a single state in the different experiments varies a lot. For example, for the circular buffer experiments this number is much higher than for the leader election protocol experiments, as depicted in Table 5.5. This can be explained by the fact that the circular buffer example has a much higher degree of symmetry compared to the leader election protocol. Therefore, much time is spent on isomorphism checking. If we would not check for isomorphic states, the state space would grow in the order n^3 instead of n^2, and the state space size penalty would be worse than the penalty for isomorphism checking.

Experiment	**Total**			**Avg. time**
	states	**transitions**	**time (ms)**	**per state**
Circular Buffer	20646	39378	27046	1.31
Leader Election Protocol	141741	474730	45137	0.32

Table 5.5: Average generation times per state for various experiments.

Summarizing, the conclusions are:

- the implementation of the *CTL* model checking algorithm is clearly less time consuming than the implementation of the *LTL* algorithm;
- in the graph transformation framework, reuse of computation results pays off significantly in many cases, in accordance with the complexity results of Section 5.6.2;
- for systems with highly symmetric state spaces, the average time to generate a single state is higher than for systems with less symmetry (or no symmetry at all); nevertheless, the isomorphism checking penalty is often much less then the state space size penalty when not performing isomorphism checking; see also [164, 158].

5.8 Conclusion

5.8.1 Summary

In this chapter we proposed an on-the-fly bounded model checking algorithm, called the SE^+-algorithm, for verifying arbitrary graph production systems. It combines a known algorithm for on-the-fly model checking with ideas from bounded model checking. Using proper boundary conditions, the SE^+-algorithm verifies state spaces generated from graph production systems in an iterative fashion. The partial correctness of the algorithm is based on the use of proper boundaries. We proposed some boundary conditions that can be applied to any graph production system, for example boundary conditions that limit the state space based on the size of the graphs that are allowed to be generated. Alternatively, one can define system-specific boundary conditions by selecting a subset of rules with specific characteristics that guarantee the sub-state spaces of all iterations to be finite.

In the context of graph transformation, computing transitions is much more expensive than traversing them as we have shown through a number of experiments. We have performed some complexity analysis for verifying state spaces iteratively with and without reusing exploration results for cases where state spaces grow linearly and exponentially (generalizing to polynomial growth), and exponentially. We concluded that for polynomially growing state spaces, the relative gain always approaches 100%, when the number of iteration is sufficiently large, independently of the order of the polynomial. In the exponential case, we have shown that the relative gain when reusing exploration results tends to $\frac{1}{x}$, where x is the exponential growth factor.

We have implemented the SE^+-algorithm in the graph transformation tool Groove. In [113] we already reported on some small experiments comparing the performance of Groove with the model checker Spin [104] with respect to state space generation. In this chapter we performed a number of combined experiments on the *CTL* and *LTL* model checking algorithm to compare their mutual performance with respect to the verification of actual properties. The results of those experiments indeed indicate that the time-complexity of both algorithms is linear in the size of the state space. We also experimented with correct and faulty implementations of the leader election protocol by Dolev et al. [56] from which we concluded that the SE^+-algorithm has practical advantages due to its on-the-fly nature. Clearly, further experiments must be conducted to get further insight in the performance and limits of the approach presented in this chapter.

The main disadvantage of the SE^+-algorithm, which is inherent to this form of explicit-state model checking infinite state spaces, is that we cannot guarantee completeness. That is to say, counter-examples that cannot be captured as reachable accepting cycles will never be identified.

Although this chapter presented an algorithm for on-the-fly bounded model checking of graph production systems, this algorithm can be applied to other types of rule-based systems as well. Examples of such other types of rule-based systems are term (graph) rewrite systems [173] and Petri nets [156].

5.8.2 Related Work

The main difference between our approach to on-the-fly bounded model checking and the usual one [21, 20] is that we perform the verification procedure on the model that has been generated instead of translating it to a corresponding *satisfiability problem*. Although there exist highly optimized SAT solvers that can solve satisfiability problems very efficiently, we do not see a straightforward way of translating our graph transition systems (including their internal graph structures) into a corresponding satisfiability problem. This may be a topic of future research.

A totally different approach to verifying infinite state systems that is often applied, is by using *abstraction* techniques. By abstracting from specific details of the system, infinite state spaces might be reduced to finite ones. If the applied abstraction is to coarse, counter-examples might be introduced that do not represent actual system executions. Such counter-examples might then serve to refine the original abstraction such that those counter-examples will not occur anymore. In Chapter 7 we will discuss different abstraction techniques in more detail.

Work closely related to ours as described in this chapter is that of König and Kozioura [120]. They have proposed to verify graph transformation systems by approximating them using Petri nets, i.e., their method is based on abstraction principles. They mainly aim at verifying whether all reachable graphs satisfy specific structural properties, i.e., they focus on the satisfaction of *reachability* or *safety* properties. Our algorithm is focussed on verifying the infinite behaviours of systems, i.e., we also include *liveness* properties, and thus both approaches are partly complementary.

Schmidt and Varró [169, 192] have introduced the CheckVML approach to model checking visual modelling languages (based on graph transformations). They aim at exploiting off-the-shelf model checking tools like Spin [104]. For this, they translate a graph transformation system into an equivalent Promela

specification after which Spin can perform the formal analysis. A detailed comparison between our approach and the CheckVML approach appeared in [164].

5.8.3 Discussion

Optimizing the Basic Algorithm. We see two obvious ways of optimizing the basic algorithm, namely by recognizing *pocket states* and by starting next iterations from source states of border transitions.

States of initial fragments that are guaranteed not to be part of any counter-example could be treated specially. Typical cases are states that are part of correct *terminal strongly connected components* (TSCC, for short), i.e., a strongly connected component from which no other component is reachable and which itself does not contain a counter-example. Such sub-state spaces can be called *pockets*, and their constituent states can thus be called *pocket states*. Additionally, states of which all successor states are pocket states can be said to be pocket states as well. Trivial cases of correct TSCCs are non-accepting states with only an outgoing final transition, i.e., deadlock states of the original system. Schwoon and Esparza [171] argued that marking states that will never be part of any accepting cycle does asymptotically not pay off. In their setting, the NDFS algorithm is applied on the state space only once. In our setting, where the state space is verified iteratively, marking pocket states potentially improves the performance of the algorithm since every next iteration may benefit from it. For example in the leader election protocol, every sub-state space that represents an actual election for some specific ring configuration can be shown to consists of pocket states only.

If the algorithm successfully verified some B_i of an approximation sequence $B_1, B_2, \ldots$, it could continue by taking the source state of border transitions of B_i as the starting point for verifying B_{i+1} instead of the initial state. This is advantageous if the part of the state space reachable from such states is (much) smaller than B_{i+1}. Fig. 5.12 depicts such a case. The dashed lines (numbered i, $i+1$, and $i+2$) represent the boundaries of subsequent initial fragments. Fig. 5.12 indicates that target states of border transitions of B_i do not necessarily have to be included in B_{i+1}. The correctness of the algorithm can only be preserved if the algorithm resets the state-colouring every time it starts from another border state. This optimization can be combined with the previous one by not resetting the colour of pocket-states.

We have performed some small experiments for verifying our implementation of the leader election protocol using the basic algorithm equipped with the above

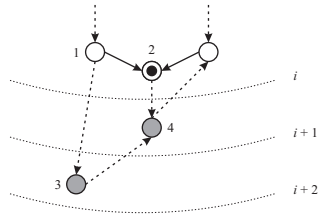


Figure 5.12: Example border transitions. From state 4, only a small part of B_i is reachable; note that the transition from state 1 to state 3 is a border transition of both B_i and B_{i+1}.

mentioned optimizations. The algorithm identified states that formed trivial TSCC and all states always leading to such pocket states. Due to the exponential growth of the state space, we could not observe any significant performance improvements. Further research and more experiments are necessary to get better insight in the performance of our (optimized) algorithm on systems with different characteristics.

Comparison. Concerning our implementation of the on-the-fly bounded model checking algorithm we proposed in this chapter, we are not aware of any tool with which it can objectively be compared, since many of the well-known explicit-state model checkers use, e.g., a different system specification language, other ways of representing system states, or totally different state space exploration techniques.

Linear Temporal Logic. As mentioned in Section 5.1 the choice of *LTL* as the formalism for expressing properties to be verified, partially cuts down the advantages provided by the graph transformation framework. Although graph transition systems provide information about which rule has been applied to get from one state to another state, propositional temporal logics like *CTL* and *LTL* do not include modal operators to use that kind of information. In fact, individual graph transitions also provide information about which graph elements of the source state are mapped to target elements of the target state. This information is captured in the graph morphism attached to single graph transitions. Using this type of information requires to consider modal and quantified logics. such as the modal μ-calculus [121] and *QCTL* (for Quantified *CTL*; see, e.g., [84]). Modal logics like the modal μ-calculus [121] include *modal operators* like

$\langle a \rangle \phi$ (and its dual $[a]\phi$), where a is some action of the program, expressing that after performing an action a in some (all) reached state(s) ϕ holds. Quantified logics such as $QCTL$ (for Quantified CTL) provide further means to reason about evolution of entities occurring within individual states; see, e.g., [84]. We will elaborate on these types of extensions in further detail in Chapter 7.

6 Dynamic Partial Order Reduction Using Probe Sets

6.1 Introduction

Although explicit state model checking is, by now, a well-established technique for verifying concurrent systems and the previous chapter indicated its value in the context of rule-based systems in general and graph transformation system more specifically, one major problem to cope with is the *state explosion problem*. In traditional concurrent systems, there are two main causes of this problem: (1) different orderings of independent actions which, for instance, stem from different processes that execute concurrently and (2) huge or even infinite data domains of state variables. In Chapter 3 we have touched one specific technique for dealing with the latter cause, i.e., through abstraction. In this chapter we focus on the former cause.

A strong recent trend is the extension of model checking techniques to *software* systems. Software systems have, besides the above mentioned problems, the additional problem of unpredictable dynamics, for instance in the size of the data structures, the depth of recursion and the number of threads.

Typically, the number of components in concurrent software systems is fairly large, and the actions performed by those components, individually or together (in case of synchronization), can be interleaved in many different ways, which caused the state space to grow excessively. A popular way of tackling this problem is by using so-called *partial order reduction* techniques. Many of those techniques are based on the idea that, in a concurrent model of system behaviour

based on *interleaving semantics*, different orderings of independent actions, e.g., steps taken by concurrent components, can be treated as *equivalent*, in which case not all possible orderings need to be examined.

In the literature, a number of algorithms have been proposed based on this technique; see, e.g., [185, 186, 92, 91, 85]. These are all based upon variations of two core techniques: *persistent* (or *stubborn*) *sets* [185, 91] and *sleep sets* [91]. In their original version, these techniques are based on two important assumptions:

- the number of actions is finite and *a priori* known;
- the system consists of a set of concurrent processes; the orderings that are pruned away all stem from interleavings of actions from distinct processes.

Due to the dynamic nature of software, the domain of (reference) variables, the identity of method frames and the number of threads are all impossible to establish beforehand; therefore, the number of (potential) actions is unbounded, meaning that the first assumption is no longer valid. This has been observed before by others, giving rise to the development of *dynamic* partial order reduction; e.g., [85, 95]. As to the second assumption, there are types of formalisms that do not rely on a pre-defined set of parallel processes but which do have a clear notion of independent actions. In the context of graph transformations, not only is the size of the graphs that can be generated unbounded (and so the first assumption fails) but also there is no general way to interpret such systems as sets of concurrent processes, and so the second assumption fails as well.

In this chapter, we present a new technique for partial order reduction, called *probe sets*, which is different from persistent sets and sleep sets. Rather than on concurrent processes, we rely on abstract *enabling* and *disabling* relations among actions, which we assume to be given. Like persistent sets, probe sets are subsets of enabled actions satisfying particular local (in)dependence conditions. Like the existing dynamic partial order reduction techniques, probe sets are optimistic (as opposed to conservative methods), in that they underestimate the paths that actually have to be explored to find all relevant behaviour. The technique is therefore complemented by a procedure for identifying *missed actions*.

We show that probe set reduction preserves all traces of the full transition system system modulo the permutation of independent actions. Moreover, we show that the probe set technique is capable of reducing systems in which there are no non-trivial persistent sets, and so existing techniques are bound to fail.

However, the critical part is the missed action analysis. In principle, it is possible to miss an action whose very existence is unknown. To show that the detection of such missed actions is nevertheless feasible, we further refine our

setting by assuming that actions work by manipulating (reading, creating and deleting) *entities*, in a rule-based fashion. For instance, in graph transformation, the entities are graph elements, i.e., nodes and edges. Thus, the actions are essentially rule applications. Missed actions can then be conservatively predicted by overestimating the applicable rules.

Overview

The structure of this chapter is as follows. In Section 6.2 we briefly recall the basic ideas of partial order reduction. We further discuss the main difference between static and dynamic approaches, and explain the basic concept of persistent sets. In Section 6.3 we then introduce our general framework consisting of abstract enabling and disabling relations. There, we also show how this general framework can be instantiated in the context of so-called entity-based systems. Section 6.4 then introduces the required ingredients for the approach we propose and proves its correctness. In Section 6.5 the actual probe set algorithm is explained by discussing how the ingredients from Section 6.4 are put together. Section 6.6 discusses one way to apply the new approach in the context of graph transformations. Section 6.7 finishes this chapter with a summary of the main results, and some concluding remarks.

This chapter is based on [114].

6.2 Partial Order Reduction

One of the main causes of the state explosion problem is that the different actions that can occur in a system can be ordered in many different ways. Many of the executions of such systems, however, represent behaviours that are very similar. This is due to the fact that in many cases the majority of the actions do not influence each other or, stated differently, are *pair-wise independent*. The basic observation is that applying pair-wise independent actions in different orders has the same overall effect. That is, the states reached by different orderings of such actions are equivalent (or isomorphic in the case of graph transformation systems).

Partial order reduction techniques aim at selecting a *subset* of all enabled actions for every state explored. Some of such techniques ensure that this selection guarantees that the reduced system reflects all properties to be verified by introducing a nothing of *visibility*. An action is then said to be *invisible* if its execution does not affect the value of propositional variables under consideration.

An example of such an approach is the *ample set* approach (see, e.g., [37]). We focus on techniques that select subsets of enabled actions ensuring that every execution of the actual system is somehow represented in the reduced system. Selecting a suitable subset is then based on an *equivalence relation* over system executions; the equivalence relation is, in turn, based on a dependency relation between the actions that can occur in the system. In general, if one system execution, say v, can be obtained from another one, say w, by repeatedly switching the order of two subsequent and independent actions, then v and w are equivalent, denoted $v \simeq w$. Thus, if T_S is the set of all system executions of some system S, then partial order reduction techniques typically aim at selecting a set $T'_S \subseteq T_S$ such that for all $v \in T_S$, there exists a $w \in T'_S$ with $v \simeq w$.

The key to partial order reduction is to select *per state* a subset of the enabled transitions in that state in some way. The variation between different partial order reduction techniques is due to different ways to select such a subset. Many partial order reduction approaches are based on constructing so-called *persistent* (or *stubborn*) *sets* [185, 186, 91], or *ample sets* [37]. As mentioned in Section 6.1, the persistent set approach is based on a number of assumptions which do not hold for the type of systems we work with.

In the following sections we will first recall the basic idea behind the persistent set approach and depict one of the main disadvantages of that approach. Thereafter, we will further discuss the main difference between partial order reduction techniques that determine subsets of enabled transitions *statically* and those that perform the selection *dynamically*.

6.2.1 Persistent Sets

One popular approach to static partial order reduction that has originally been introduced by Godefroid [91] is based on ensuring that eventually the subsets of enabled actions explored from all the states are *persistent*. The definition of persistent sets is based on a *reflexive* and *symmetric dependency relation*, often denoted D. If Act is the set of all actions of some concurrent system, then $D \subseteq Act \times Act$ is a *valid dependency* relation if and only if for all actions $a_1, a_2 \in Act$, $(a_1, a_2) \notin D$ (i.e., a_1 and a_2 are *independent*) implies that the following two conditions are satisfied for all states s:

enabledness: if a_1 is enabled in s and $s \xrightarrow{a_1} s'$, then a_2 is enabled in s if and only if a_2 is enabled in s';

commutativity: if a_1 and a_2 are enabled in s, then there exists a unique state s' such that $s \xrightarrow{a_1 a_2} s'$ and $s \xrightarrow{a_2 a_1} s'$.

Based on the above notion of (in)dependency between actions, persistent sets of enabled actions can be defined as follows.

Definition 6.1 (persistent set)**.** *A set $A \subseteq Act$ of actions enabled in some state s is* persistent in *s iff for all non-empty sequences of actions*

$$s = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots \xrightarrow{a_{n-1}} s_n \xrightarrow{a_n} s_n$$

from s in the non-reduced transition system and including only actions $a_i \notin A$, for $1 \leq i \leq n$, a_n is independent of all actions in A.

Basically, selecting a persistent set at a state s requires one to have (or somehow derive) sufficient information about *future* exploration paths that start in s.

One of the main disadvantages of this approach is that it is sensitive to the shape of the state space. Suppose we apply the persistent set approach to the state space shown in Fig. 6.1, where the transitions a_i and b_i, for $1 \leq i \leq 3$, represent actions taken by different processes. For this example, diamonds correspond to pair-wise independent actions. Thus we have, $(a_i, a_{i+1}), (b_i, b_{i+1}) \in D$, for $i = 1, 2$ and $(a_1, b_3), (a_2, b_2), (a_3, b_1) \in D$. In state q the actions a_1 and b_1 are enabled and are independent of each other. However, both sets $\{a_1\}$ and $\{b_1\}$ are not persistent in q_0. When selecting $A = \{a_1\}$ as candidate persistent set in state q_0, it is easy to see that the sequence $q_0 \xrightarrow{b_1} q_2 \xrightarrow{b_2} q_5 \xrightarrow{b_3} q_9$ contains only actions outside A, but b_3 and a_1 are pair-wise dependent; exploring either a_1 or b_3 from state q_5 disables the other action. Similarly, when selecting $A = \{b_1\}$ as a persistent set in state q_0, there is a dependency between a_3 and b_1. The only set A of enabled actions that is persistent in q_0 is the trivial persistent set, i.e., the set $A = \{a_1, b_1\}$ of all actions enabled in q_0. One can easily verify that in this small example for all states we can only select the trivial persistent sets, which results in no reduction at all.

6.2.2 Static versus Dynamic Partial Order Reduction

Many partial order reduction techniques determine the subsets of enabled transitions to be explored statically. This means that at the time of exploring a state, such a subset has to be chosen in such a way that they guarantee *a priori* not to rule out any relevant execution of the system. When exploring the remaining part of the state space, those subsets of already explored states will not change: subsets are chosen once and cannot be extended later on. In many

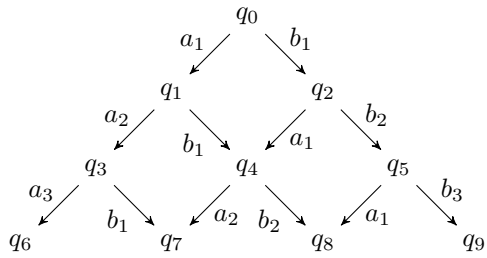


Figure 6.1: Triangular shaped state space.

cases, actions are only indirectly dependent of each other. Especially for complex system, which often consist of a vast amount of actions performing various operations on shared or local variables, many actions are only indirectly dependent of each other. Therefore, a priori selecting the smallest subset of enabled transitions from which all system traces can still be generated is often very difficult, if not impossible. The usual approach is thus to stay on the safe side and *over-approximate* the selection. Obviously, for most cases this does not result in the maximal possible reduction. Nevertheless, many such approaches obtain reasonable reduction results in terms of the number of states and transitions explored and the amount of resources such as, e.g., time and memory, required for exploration.

The reduction might be increased when the selection of enabled transitions to be explored would at first be *under-approximated* and later on be updated in case some system traces have been ruled out. This is the basic idea behind performing partial order reduction in a dynamic fashion. Initially, the selection could, for instance, be based on local dependency relations between the enabled transitions only. Further analysis for missed system traces will then in later stages cause some selections to be extended. Although the reduction may be more effective, the analysis for missed system traces obviously requires additional resources. And thus, one cannot simply conclude that dynamic partial order reduction techniques will always outperform comparable static ones.

Although Flanagan and Godefroid recently introduced a partial order reduction algorithm for model checking software systems that computes persistent sets dynamically [85], most of the assumptions on the underlying formalism of Godefroids original approach [91] have endured, especially that the number of actions is finite and known in advance. Whether persistent sets are computed statically or dynamically does not affect the disadvantage of the original ap-

proach as mentioned above. In the following sections we will use an example generating a similar transition system, when compared to Fig. 6.1, to indicate that our dynamic partial order reduction approach, i.e., using probe sets, is not sensitive for the shape of the state space.

6.3 Stimulation, Disabling, and Reduction

One of the basic assumptions in the persistent set approach to partial order reduction is that the concurrent system consists of a number of processes execution concurrently. This approach relies on a reflexive and symmetric (in)dependency relation. In addition, all actions from a single process are by definition dependent. Our focus is on contexts where there is no such a notion of a process. In graph transformation systems actions (corresponding to rule applications) are not related to specific processes but instead describe the behaviour of the system from a global or integrated point of view. Dependency relations will then be defined based on the effect of actions only. In this section we therefore introduce basic concepts on which our approach relies.

In our approach, we assume a countable universe of *actions* denoted Act; individual actions will be denoted $a, b, \ldots$ Whereas (as recalled above) the persistent set approach is based on a single reflexive and symmetric (in)dependency relation, we introduce two types of dependencies between actions, namely *stimulation* and *disabling*.

stimulation: $a \triangleright b$ denotes that a stimulates b;

disabling: $a \blacktriangleleft b$ denotes that a disables b.

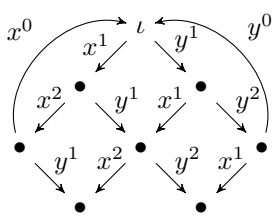
The stimulation relation is *irreflexive*, i.e., for all $a \in Act$ it holds that $a \not\triangleright a$. The intuition is that a stimulates b if the effect of a fulfills (part of) the preconditions of b that were not fulfilled before (i.e., that occurrence of b cannot be executed directly before a). The action a disables b if its effect violates part of b's precondition (i.e., that occurrence of b cannot be executed directly after a). If two actions a and b neither stimulate nor disable each other they are said to be *independent* and executing either $a{\cdot}b$ or $b{\cdot}a$ is equivalent. In the theory of *event structures* (e.g., [198]), $\triangleright$ roughly corresponds to a notion of (direct) *cause* and $\blacktriangleleft$ to *asymmetric conflict* (e.g., [132]).[1] In terms proposed by Janicki and Koutny [107], $\triangleright$ corresponds to *weak causality* (denoted $\nearrow$). Two

[1] This analogy is not perfect, since events can occur only once whereas our actions can in principle re-occur.

simultaneously enabled actions that are furthermore independent, can then be said to be *commutative* (denoted $\rightleftharpoons$). That is to say, they can happen in any order, but not simultaneously. Fig. 6.2 shows an example program in speudo code, its corresponding transition system, and the complete dependency table.

```
do [x+y<3] ->
       x := (x+1)%3;
or [x+y<3] ->
       y := (y+1)%3;
od
```

(a)



(b)

	x^1	x^2	x^0	y^1	y^2	y^0
x^1	◀	▷				◀
x^2		◀	▷		◀	
x^0	▷		◀		▷	▷
y^1			◀	◀	▷	
y^2		◀			◀	▷
y^0		▷	▷	▷		◀

(c)

Figure 6.2: A non-deterministic process (a), its transition system (b) and the stimulus and disabling relations (c). Action x^i [y^i] assigns i to x [y], with pre-condition $x+y < 3$.

Sequencing actions results in *words*, which will be denoted $v, w \in Act^*$. The empty word is denoted ε. For a word w, the set of actions in w is denoted A_w. With respect to stimulation and disabling, not all words represent possible computations of the actual system. To make this precise, we define an *influence* relation over words.

Definition 6.2. *Let $v, w \in Act^*$ be two words. Then v* influences *w, denoted $v \rightsquigarrow w$, if and only if the following condition is satisfied:*

$$\exists a \in A_v, b \in A_w : a \rhd b \ \vee \ b \lhd a \ .$$

Influence can be positive or negative. For instance, in Fig. 6.2, $x^1 \cdot x^2 \rightsquigarrow y^1 \cdot y^2$ due to $y^2 \lhd x^2$ and $x^1 \cdot y^1 \rightsquigarrow x^2 \cdot y^2$ due to $x^1 \rhd x^2$, whereas x^1 and $y^1 \cdot y^2$ are independent.

Obviously, not all words represent possible executions of the system. Therefore, we distinguish between words that can potentially be executed by a system and those that can not. The former will be called *feasible*; the latter will be called *infeasible*.

Definition 6.3 (word feasibility)**.** *A word w is* feasible *if for all sub-words $a \cdot v \cdot b$ with $b \rightsquigarrow a$, it holds that $v = v_0 \cdot c_1 \cdot v_1 \cdot c_2 \cdots c_n \cdot v_n$, with $n \geq 0$, $c_i \in Act$ for $1 \leq i \leq n$ and $v_i \in Act^*$ for $0 \leq i \leq n$ such that $a \rightsquigarrow c_1 \rightsquigarrow c_2 \rightsquigarrow \cdots \rightsquigarrow c_n \rightsquigarrow b$.*

Intuitively, feasibility of a word means that for any backwards influence between actions there must exist a chain of forward influences.

For many purposes, it suffices to interpret words up to permutation of independent actions. Words that are equal up to permutation of independent actions are said to be *equivalent*. We define this equivalence as a binary relation over arbitrary words.

Definition 6.4. *$\simeq \subseteq Act^* \times Act^*$ is the smallest reflexive and transitive relation over arbitrary words such that $v{\cdot}a{\cdot}b{\cdot}w \simeq v{\cdot}b{\cdot}a{\cdot}w$ if $a \not\leadsto b$.*

Some properties of this equivalence, such as the relation with feasibility, are expressed in the following proposition.

Proposition 6.5. *Let v, w, w_1, and w_2 be words. Then,*

1. *If v is feasible and $v \simeq w$, then w is feasible;*
2. *$\simeq$ is symmetric over the set of feasible words;*
3. *$v{\cdot}w_1 \simeq v{\cdot}w_2$ if and only if $w_1 \simeq w_2$.*

Proof. The proof is included in Appendix E. □

It should be noted that, over feasible words, the setup now corresponds to that of (Mazurkiewicz) *traces*, which have a long tradition; see, e.g., [136, 52]. The main difference is that our underlying notion of influence, built up as it is from stimulation and disabling, is more involved than the symmetric binary dependency relation that is commonly used in this context (e.g., as in the static persistent set approach [91]) — hence for instance the need here to restrict to feasible words before $\simeq$ is symmetric.

We also define two prefix relations over words, the usual "hard" or "strong" one, which expresses that one word is equal to the first part of another, and a "weak" prefix up to permutation of independent actions.

Definition 6.6 ((weak) prefix)**.** *Let v, and w be words. Then,*

- *v is called a* prefix of *w, denoted $v \preceq w$, iff $\exists u : v{\cdot}u = w$;*
- *v is called a* weak prefix of *w, denoted $v \precsim w$, iff $\exists u : v{\cdot}u \simeq w$.*

It is not difficult to see that both relations $\preceq$ and $\precsim$ restricted to feasible words, are partial orders.

6.3.1 Transition systems

In previous chapters we have seen different types of transitions systems such as, e.g., *graph transition systems*, and more advanced transition system-like formalisms such as, e.g., *Kripke structures*, and *Büchi automata*.

In this chapter we deal with transitions systems over an alphabet Act of action names. Let $S = (Q, \rightarrow, \iota)$ be a transition system with $\iota \in S$ and $\rightarrow \subseteq Q \times Act \times Q$, we then require the following additional constraints. For all $q, q_1, q_2 \in Q$:

1. all traces are feasible; i.e., $\iota \xrightarrow{w} q$ implies w is feasible;
2. the system is deterministic up to independence; i.e., $q \xrightarrow{w_1} q_1$ and $q \xrightarrow{w_2} q_2$ with $w_1 \simeq w_2$ implies $q_1 = q_2$;
3. all out-degrees are finite; i.e., $enabled(q) = \{a \in Act \mid \exists q \xrightarrow{a} q'\}$ is a finite set.

The second constraint implies (among other things) that the actions in Act are fine-grained enough to deduce the successor state of a transition entirely from its source state and label. For convenience, we introduce some further notations:

$$\begin{array}{rcl} q \vdash w & :\Leftrightarrow & \exists q' : q \xrightarrow{w} q' \\ q \uparrow w & := & q' \quad \text{such that } q \xrightarrow{w} q' \ . \end{array}$$

The expression $q \vdash w$ states that q *enables* w, and $q \uparrow w$ should be read as w *after* q, i.e., the state reached from q after w has been performed. Clearly, $q \uparrow w$ is defined (uniquely, due to determinism) iff $q \vdash w$. The following algebraic properties may help to strengthen the reader's intuition:

$$\begin{array}{rcl} q \vdash v{\cdot}w & \Leftrightarrow & q \vdash v \ \wedge \ (q \uparrow v) \vdash w \\ q \uparrow (v{\cdot}w) & = & (q \uparrow v) \uparrow w \ . \end{array}$$

In addition to determinism modulo independence, we define the notion of dependency consistent and dependency complete transition systems.

Definition 6.7 (dependency consistency and completeness)**.** *A transition system S is called* dependency consistent *if it satisfies the following properties for all $q \in Q$:*

$$q \vdash a \ \wedge \ a \rhd b \implies q \nvdash b \tag{6.1}$$

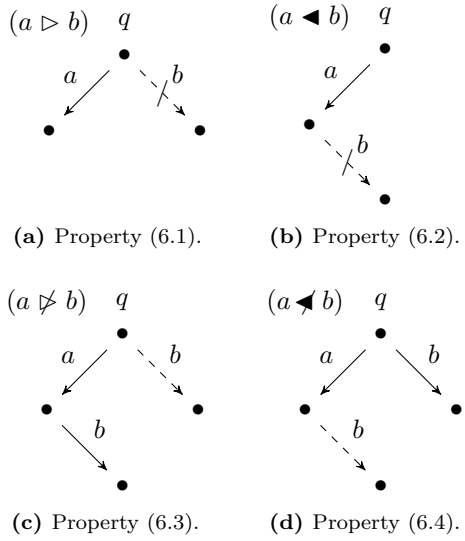
$$q \vdash a \ \wedge \ a \blacktriangleleft b \implies q \nvdash a{\cdot}b \ . \tag{6.2}$$

S is called dependency complete *if it satisfies:*

$$q \vdash a{\cdot}b \;\wedge\; a \not\rhd b \;\;\Longrightarrow\;\; q \vdash b \tag{6.3}$$

$$q \vdash a \;\wedge\; q \vdash b \;\wedge\; a \not\blacktriangleleft b \;\;\Longrightarrow\;\; q \vdash a{\cdot}b \;. \tag{6.4}$$

Dependency consistency puts constraints on the *non-existence* of transitions as implied by the *dependency* relations. For instance, property (6.1) requires that if an action a is enabled in state q and a stimulates another action b, i.e., $a \rhd b$, then b must not be enabled in q. Dependency completeness, on the other hand, requires the *existence* of transitions based on *independency* relations. Dependency consistency and completeness are illustrated in Fig. 6.3.



(a) Property (6.1). **(b)** Property (6.2).

(c) Property (6.3). **(d)** Property (6.4).

Figure 6.3: Illustration of the consistency and completeness properties of Def. 6.7. The (negated) dashed arrows are implied by the others, under the given (in)dependency relations.

The following property states an important consequence of dependency completeness, namely that weak prefixes of traces are themselves also traces.

Proposition 6.8. *If S is a dependency complete transition system, then $q \vdash w$ implies $q \vdash v$ for all $q \in Q$ and $v \precsim w$.*

Proof. We actually prove the proposition for $v \simeq w$; this is trivially extended to $v \precsim w$. Due to the definition of $\simeq$, it suffices to regard the case where $w = v_1 \cdot a \cdot b \cdot v_2$ and $v = v_1 \cdot b \cdot a \cdot v_2$ with $a \not\leadsto b$.

If $q \vdash w$ then $q \xrightarrow{v_1} q_1 \xrightarrow{a \cdot b} q_2 \xrightarrow{v_2}$, where $a \not\triangleright b$ and $b \not\blacktriangleleft a$. Due to (6.3), it follows that $q_1 \vdash b$, hence due to (6.4), we have $q_1 \vdash b \cdot a$. But then $q_1 \xrightarrow{b \cdot a} q_2$ due to determinism, hence we are done. □

In this work we aim at *reducing* dependency complete transition systems to smaller transition systems (having fewer states and transitions), which are no longer dependency complete but from which the original transition system can be reconstructed by completing it w.r.t. (6.3) and (6.4). We now define this notion of reduction formally.

Definition 6.9 (reduction)**.** *Let R, S be two dependency consistent transition systems. We say that R* reduces *S if $Q_R \subseteq Q_S$, $\rightarrow_R \subseteq \rightarrow_S$, $\iota_R = \iota_S$, and for all $w \in Act^*$, it holds that*

$$\iota_S \vdash_S w \implies \exists v \in Act^* : w \precsim v \ \wedge \ \iota_R \vdash_R v \ .$$

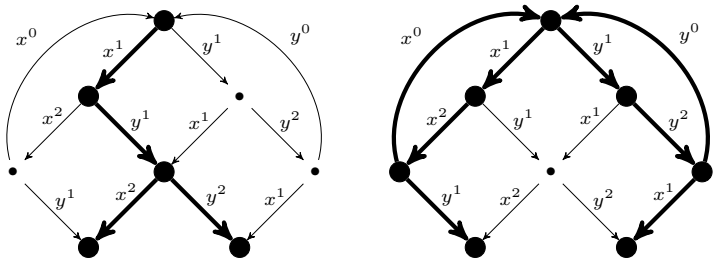
Intuitively, the above definition states that if R reduces S, then every trace w of S is also "included" in R, which means that there exists a trace v in R such that w is a (weak) prefix of v. That is to say, the actions of w are all included in v, possibly in a different order (respecting the dependencies) and possibly separated by additional actions that are independent of all remaining actions. Essentially, our definition of reduction coincides with Godefroid's notion of a *trace automaton* [90, 91]. We will often characterize a reduced transition system only through its set of states Q_R.

Figure 6.4 shows two reductions of the transition system in Fig. 6.2, one invalid (a) and one valid (b). In (a), among others, the trace $x^1 \cdot x^2 \cdot x^0$ is lost.

It follows from Proposition 6.8 that the reduction of a dependency complete transition system is essentially lossless: if R reduces S and S is complete, then the reachable part of S can be reconstructed from R up to isomorphism. In particular, it immediately follows that deadlock states are preserved by reduction.

Proposition 6.10. *If R and S are dependency consistent transition systems such that S is dependency complete and R reduces S, then for any reachable deadlock state $q \in Q_S$ (i.e., such that $\forall a \in Act : q \not\vdash a$) it holds that $q \in Q_R$.*

Proof. Assume $\iota_S \xrightarrow{w} q$; then by definition of reduction, $\iota_R \xrightarrow{v} q' \in Q_R$ for some v such that $w \precsim v$, meaning $w \cdot w' \simeq v$ for some w'. Due to $\forall a \in Act : q \not\vdash a$ it



(a) Invalid reduction. **(b)** Valid reduction.

Figure 6.4: Two reductions of the transition system in Fig. 6.2. The fat nodes and arrows are the states and transitions of the reduced system.

follows that $w' = \varepsilon$. By the determinism of the transition system modulo $\simeq$ it then follows that $q' = q$. □

6.3.2 Entity-based System Specifications

In the previous section, we have introduced an abstract notion of actions and dependencies among actions. Here, we choose a specific setting in which those notions are given a concrete interpretation.

In contrast to the usual approach in which concurrent systems are often modelled by specifying individual processes which act on shared and local variables, we assume a countable universe *Ent* of so-called *entities*. Entities (or groups of them) can be thought of as global variables on which every action can perform specific operations, such as, e.g., reading, deleting, or creating. Such actions will then be said to *manipulate Ent*. Individual entities will be denoted $e, e_1, e', \ldots$

Definition 6.11. *An action a is said to* manipulate *Ent if there are associated finite and disjoint sets*

- $R_a \subseteq Ent$*, the set of entities* read *by a;*
- $N_a \subseteq Ent$*, the set of entities* forbidden *by a;*
- $D_a \subseteq Ent$*, the set of entities* deleted *by a;*
- $C_a \subseteq Ent$*, the set of entities* created *by a.*

A special class of actions are those actions of which the sets of deleted and created entities are empty (recall the conditional graph transformation rules

from Chapter 5, Section 5.3.3). Executing such actions does not actually change states but only check whether states satisfy the condition they specify.

This characterization of entity manipulating actions, or entity-based actions, is chosen because of its nice match with the graph transformation framework, where the graph elements of transformation rules can be similarly partitioned. The set of *Ent*-manipulating actions is denoted $Act[Ent]$. Based on the quadruple of sets associated to actions, as from Def. 6.11, we can formally define the stimulation and disabling dependency relations.

Definition 6.12. *Let a and b be Ent-manipulating actions. Then,*

$$a \rhd b \quad :\Leftrightarrow \quad C_a \cap (R_b \cup D_b) \neq \emptyset \ \vee \ D_a \cap (C_b \cup N_b) \neq \emptyset \tag{6.5}$$

$$a \blacktriangleleft b \quad :\Leftrightarrow \quad D_a \cap (R_b \cup D_b) \neq \emptyset \ \vee \ C_a \cap (C_b \cup N_b) \neq \emptyset \tag{6.6}$$

In words, an action a stimulates an action b if a creates some entity e (i.e., $e \in C_a$) that is read by b (i.e., $e \in R_b$) or deleted by b (i.e., $e \in D_b$), and thus $C_a \cap (R_b \cup D_b) \neq \emptyset$, or if a deletes an entity e' (i.e., $e' \in D_a$) that will be created by b (i.e., $e' \in C_b$) or that is forbidden to exists for b to be enabled (i.e., $e' \in N_b$), and thus $D_a \cap (C_b \cup N_b) \neq \emptyset$. Analogously, an action a disables another action b if a deletes some entity e (i.e., $e \in D_a$) that is read by b (i.e., $e \in R_b$) or deleted by b (i.e., $e \in D_b$), and thus $D_a \cap (R_b \cup D_b) \neq \emptyset$, or if a creates an entity e' (i.e., $e' \in C_a$) that is also created by b (i.e., $e' \in C_b$) or forbidden by b (i.e., $e' \in N_b$), and thus $C_a \cap (C_b \cup N_b) \neq \emptyset$.

Since *Ent* and *Act* may both be infinite, we have to impose some restrictions to make sure that our models are effectively computable. For this purpose we make the following important assumption.

Assumption 6.13. *For every finite set $E \subseteq Ent$, the set of enabled actions is finite and computable.*

A transition system S is called *Ent*-based if $A \subseteq Act[Ent]$ and for every $q \in Q$ there is an associated finite set $E_q \subseteq Ent$, such that $E_q = E_{q'}$ implies $q = q'$.

Definition 6.14 (entity-based transition system)**.** *A transition system S is called Ent-based if Act consists of Ent-manipulating actions, and for all $q \in Q$:*

- *there is a finite set $E_q \subseteq Ent$, such that $E_q = E_{q'}$ implies $q = q'$;*
- *for all $a \in Act[Ent]$, $q \vdash a$ iff $(R_a \cup D_a) \subseteq E_q$ and $(N_a \cup C_a) \cap E_q = \emptyset$;*
- *for all $a \in enabled(q)$, $q \uparrow a$ is determined by $E_{q \uparrow a} = (E_q \setminus D_a) \cup C_a$.*

It can be shown that these three conditions on the associated events, together with the assumption that enabling is computable, actually imply feasibility, determinism and finite out-degrees. The following proposition states that this setup guarantees some further nice structural properties with respect to the dependency relations between actions.

Proposition 6.15. *Every Ent-based transition system is dependency complete and consistent, and has only feasible words as traces.*

The transition system of Fig. 6.2 can be obtained by using entities $e_{x>0}$, $e_{x>1}$, $e_{y>0}$ and $e_{y>1}$, setting $E_\iota = \emptyset$ and defining the actions according to the following table:

a	R_a	N_a	D_a	C_a
x^1				$e_{x>0}$
x^2	$e_{x>0}$	$e_{x>0}$		$e_{x>1}$
x^0		$e_{x>1}$	$e_{x>0}, e_{x>1}$	

a	R_a	N_a	D_a	C_a
y^1				$e_{y>0}$
y^2	$e_{y>0}$	$e_{y>0}$		$e_{y>1}$
y^0		$e_{y>1}$	$e_{y>0}, e_{y>1}$	

(6.7)

Models whose behaviour can be captured by entity-based transition systems are: Turing machines (the entities are symbols at positions of the tape), Petri nets (the entities are tokens), term and graph rewrite systems (the entities are sub-terms and graph elements, respectively). Computability of enabling is guaranteed by the *rule-based* nature of these models: all of them proceed by attempting to instantiate a finite set of rules on the given finite set of entities, and this always results in a finite, computable set of rule applications, which constitute the locally enabled actions.

6.4 Missed Actions and Probe Sets

In Section 6.2.2 we mentioned the difference between static and dynamic partial order reduction techniques: dynamic techniques first under-approximate the subset of transitions to be explored from every state and in later stages extend those subsets if necessary, based on some specific analysis. We select such a subset based on so-called *probe sets*; those subsets are extended when identifying *missed actions*. In this section we first introduce these concepts and show how they guarantee the correctness of our reduction.

The actual algorithm is introduced in two phases. The first version of the algorithm produces a reduced state space, but does not guarantee that actions that are enabled will eventually be explored. That is, it does not guarantee *fairness*. This is taken care of in the second version.

6.4.1 Missed Actions

Missed actions are actions that could have become enabled along an explored execution path if the actions in the path had been executed in a different order. To formalize this, we define the (weak) *difference* between words, which is the word that has to be concatenated to one to get the other (modulo independence).

Definition 6.16 (difference)**.** *Let v and w be words. The* difference *of w and v, denoted $w - v$, is the word u such that $v{\cdot}u \simeq w$.*

Clearly, $w - v$ exists if and only if $v \precsim w$; in fact, as a consequence of Proposition 6.5 part 3, it is then uniquely defined up to $\simeq$. Another central notion in this work is the *prime cause* within w of a given action a, denoted $\downarrow_a w$. Intuitively, the prime cause of a in w is the smallest weak prefix of w that includes all actions that influence a, directly or indirectly. Formally, this can be defined as follows.

Definition 6.17 (prime cause)**.** *Let w be a word and a be an action. The* prime cause of a in w*, denoted $\downarrow_a w$, is the word such that the following two conditions hold:*

1. $(w - \downarrow_a w) \not\rightarrow a$,
2. $\forall v' \precsim w : (w - v') \not\rightarrow a$ *implies* $\downarrow_a w \precsim v'$.

Unlike the difference of two words, the prime cause $\downarrow_a w$ is always defined; the definition itself ensures that it is unique up to $\simeq$. A representative of $\downarrow_a w$ can in fact easily be constructed from w by removing all actions, starting from the tail and working towards the front, that do not influence either a or any of the actions *not* removed. For instance, in Fig. 6.2 we have $x^1{\cdot}y^1{\cdot}y^2 - y^1 = x^1{\cdot}y^2$ whereas $x^1{\cdot}y^1{\cdot}y^2 - y^2$ is undefined; furthermore, $\downarrow_{y^2} x^1{\cdot}y^1 = y^1$.

Missed actions are derived from the combination of a state and a path explored from that state. We therefore first introduce a separate concept for such combinations, namely that of a *vector*.

Definition 6.18 (vector)**.** *A vector (q, w) of a transition system S consists of a state $q \in Q$ and a word w such that $q \vdash w$.*

Vectors are used especially to characterize their *target* states, in such a way that not only the target state itself is uniquely identified (because of the determinism of the transition system) but also the causal history leading up to that state; different causal histories potentially result in different missed actions. Missed actions can now be defined as follows.

Definition 6.19 (missed action)**.** *Let (q, w) be a vector. We say that an action a is* missed along (q, w) *if $q \not\vdash w{\cdot}a$ but $q \vdash v{\cdot}a$ for some $v \precsim w$. We usually say that the missed action is $\downarrow_a v{\cdot}a$ rather than just a; i.e., we include the prime cause. A missed action a is said to be* fresh *in (q, w) if $w = w'{\cdot}b$ and a is not a missed action in (q, w').*

The goal behind designating some missed actions as being fresh is to prevent from redoing missed action analysis in successive states as much as possible. Thus we only perform the analysis for fresh missed actions, i.e., actions that are missed in the current state and not yet in any previous state. The set of fresh missed actions along (q, w) is denoted $fma(q, w)$. It is not difficult to see that $v{\cdot}a \in fma(q, w)$ implies $w = w'{\cdot}b$ such that $b \rhd a$ or $b \blacktriangleleft a$.

A typical example of a missed action is $(y^1{\cdot}y^2) \in fma(\iota, x^1{\cdot}x^2{\cdot}y^1)$ in Fig. 6.2: here $\iota \not\vdash x^1{\cdot}x^2{\cdot}y^1{\cdot}y^2$ but $\iota \vdash y^1{\cdot}y^2$ with $y^1 \precsim x^1{\cdot}x^2{\cdot}y^1$. Note that indeed $y^1 \rhd y^2$.

Because of the need for missed action analysis, we need to keep track of the causal histories while traversing the transition system; hence we store vectors rather than just their target states. However, this may have a negative impact on the reduction algorithm:

- As formulated in Def. 6.19, the missed action analysis is very expensive: it involves investigating all weak prefixes of the current vector, and this defeats the (time-)gains made by the partial order reduction in the first place. In the next section we show an efficient way of identifying missed action based on a so-called *over-approximation* for the case the actions work on entities.

- In principle, after traversing a transition, the action is appended to the causal history, which means that the word w in the vector (q, w) grows unboundedly. This not only increases the cost of the missed action analysis even more, but also makes it impossible to apply the algorithm to programs that loop. We will show below under which circumstances we may *discharge* part of the causal history.

- One major disadvantage of this approach is that states may be reachable through different, non-equivalent vectors. Reaching some state visited before with a new vector might result in identifying additional missed actions. Although this can have a major influence on the performance of the algorithm, the obtained reduction in the number of visited states might still be significant.

6.4.2 Probe Sets

The most important parameter of the partial order reduction is the selection of a (proper) subset of enabled actions to be explored. For this purpose, we define so-called *probe sets*, based on the disabling among the actions enabled at a certain state (given as the target state $q \uparrow w$ of a vector (q, w)). Furthermore, with every action in a probe set, we associate a part of the causal history that can be discharged when exploring that action. Thus, our probe sets are actually partial functions.

Definition 6.20 (probe set)**.** *For a given vector (q, w), a* probe set *is a partial function p: $enabled(q \uparrow w) \rightharpoonup Act^*$ mapping actions enabled in $q \uparrow w$ onto words, such that the following conditions hold:*

1. *for all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $b \blacktriangleleft a$ implies $b \in dom(p)$;*
2. *for all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $p(a) \not\precsim \downarrow_b w$ implies $b \in dom(p)$;*
3. *for all $a \in dom(p)$, $p(a) \precsim \downarrow_a w$.*

We use $\mathcal{P}_{q,w}$ to denote the set of all probe sets for a vector (q, w). We say that an action a is *in* the probe set p if $a \in dom(p)$. The first condition states that probe sets are *left-closed* under disabling. Note that we do not require that probe sets are right-closed under disabling, i.e., we do not require that for all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $a \blacktriangleleft b$ implies $b \in dom(p)$. We have chosen to treat those cases as missed actions as will become clear in Section 6.5.1. This choice is further motivated in Section 6.7.3.

The second and third condition on probe sets govern the discharge of the causal history: the fragment that can be discharged must be contained in the prime cause of any action not in the probe set (Clause 2) and in the prime cause of a itself (Clause 3). Clause 3 can easily be understood. For all $a \in dom(p)$, it is obvious that the actions *freshly* enabled after executing a (i.e., those actions were not yet enabled before executing a) indirectly require the execution of the actions in $\downarrow_a w$. We can therefore safely discharge that part of the causal history (or any weak prefix thereof) when executing a. Clause 2 is less trivial. The point is that if $p(a) \precsim \downarrow_b w$, for all $b \in (enabled(q \uparrow w) \setminus dom(p))$, it is still safe to discharge $p(a)$ from the causal history since it only contains actions necessary for all actions currently enabled and outside the probe set. In the case $p(a) \not\precsim \downarrow_b w$, for some $b \in (enabled(q \uparrow w) \setminus dom(p))$, then there is a chance that we discharge causal history that might be required for the missed action analysis in the case we execute b from the state reached after executing a. We thus risk exclusion of system traces, and therefore those actions b must also be in the probe set.

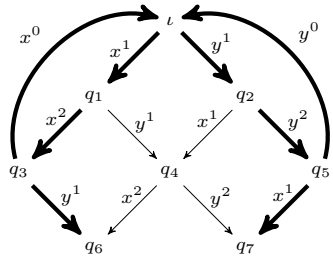
Algorithm 4 Probe set based partial order reduction, first version.

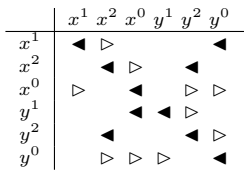
```
1:  let Q ← ∅;                                          // set of states visited
2:  let V_P ← ∅;                                        // set of vectors processed
3:  let V ← {(ι_S, ε)};                                 // set of vectors selected
4:  while V \ V_P ≠ ∅ do
5:      choose (q, w) ∈ V \ V_P;
6:      let V_P ← V_P ∪ {(q, w)};
7:      let Q ← Q ∪ {q ↑ w};
8:      for all v·m ∈ fma(q, w) do
9:          let Q ← Q ∪ {q ↑ v};
10:         let V ← V ∪ {(q ↑ v·m, ε)};
11:     end for
12:     choose p ∈ P_{q,w};
13:     let V ← V ∪ {(q ↑ p(a), w·a − p(a)) | a ∈ dom(p)};
14: end while
```

Algorithm 4 gives a first version of the reduction algorithm. Suppose we apply the algorithm on a transition system $S = (Q, \rightarrow, \iota)$. In the course of the algorithm, three sets are book kept, namely the set Q of visited states, the set V of vectors selected for examination, and the set V_P of vectors that have been processed. Initially, Q and V_P are the empty set (line 1 and line 2) and V only contains the initial vector (ι, ε). The algorithm continues until there are no more vectors to be examined (condition in line 4). If there are still unexamined vectors, an arbitrary one is selected (line 5), say (q, w), which is then added to V_P (line 6). The state $q \uparrow w$ reached by this vector is added to Q (line 7). Next, we perform the missed action analysis for this state (lines 8–11). For every missed action $v{\cdot}m$, the state $q \uparrow v$ is added to Q and the vector $(q \uparrow v{\cdot}m, \varepsilon)$ is added to V (line 10). Finally, for the vector (q, w) we select an arbitrary probe set p (line 12). For all actions selected by p, we create the corresponding vector that must be examined (line 13).

As stated before, we often characterize a reduced transition system only through its states. In Algorithm 4 this is reflected in the following way. When constructing new vectors to be examined, we remove some part of the history that lead to the current state. The state component of the new vector is typically not included in the set of states of the reduced transition system. This means that when identifying a missed action, say $v{\cdot}m$, for such a vector, say (q, w), we cannot construct a path from some state already in Q to q. Such "holes" can only be repaired if the algorithm keeps track of additional information about

how to reach q. This information should be provided at the time (q, w) is added to V.





iteration	$V \setminus V_P$	$(q, v) \in V \setminus V_P$	$fma(q, v)$	ΔQ	$p \in \mathcal{P}_{q,v}$
1	(ι, ε)			ι	(x^1, ε)
2	(ι, x^1)	(ι, x^1)		q_1	(x^2, ε)
3	$(\iota, x^1{\cdot}x^2)$	$(\iota, x^1{\cdot}x^2)$		q_3	$(x^0, x^1{\cdot}x^2), (y^1, \varepsilon)$
4	$(q_3, x^0), (\iota, x^1{\cdot}x^2{\cdot}y^1)$	(q_3, x^0)			
5	$(\iota, x^1{\cdot}x^2{\cdot}y^1)$	$(\iota, x^1{\cdot}x^2{\cdot}y^1)$	$y^1{\cdot}y^2$	q_6, q_2	
6	(q_5, ε)	(q_5, ε)		q_5	$(x^1, \varepsilon), (y^0, \varepsilon)$
7	$(q_5, x^1), (q_5, y^0)$	(q_5, x^1)		q_7	
8	(q_5, y^0)	(q_5, y^0)			

Figure 6.5: Step-by-step execution of Algorithm 4 on the example of Fig. 6.2

In Fig. 6.5 we apply this algorithm to the example system of Fig. 6.2, obtaining the reduced system in Fig. 6.4(b). The first column shows the value of $V \setminus V_P$ at the beginning of the loop; the second column represents the choice of continuation; the third is the resulting set of fresh missed actions; the fourth column gives the increase in the result set Q; and the final column shows the choice of probe set. The table of Fig. 6.5 should be read as follows. At the beginning of the first iteration (the first row), V only contains the vector (ι, ε) and $V_P = \emptyset$. In the first iteration, ι is added to the set of visited states. For this state we select the probe set $p \in \mathcal{P}_{\iota,\varepsilon}$ such that $p = \{(x^1, \varepsilon)\}$, i.e., $dom(p) = \{x^1\}$ with $p(x^1) = \varepsilon$. This is a correct probe set since $y^1 \blacktriangleleft x^1$, and both $p(x^1) \not\precsim \downarrow_{y^1} x^1$. As a result, the pair (ι, x^1) is added to V. In the second iteration, we have no choice but to select (ι, x^1) from V. We reach state q_1, which is then added to Q. At this point, we select the probe set $p \in \mathcal{P}_{\iota,x^1}$ such that $p = \{(x^2, \varepsilon)\}$. One can easily verify that this is indeed a correct probe set. After the second iteration, $V \setminus V_P = \{\iota, x^1{\cdot}x^2\}$. In the third iteration, the selected probe set $p \in \mathcal{P}_{\iota,x^1\cdot x^2}$ contains all actions enabled in state q_3. If

$x^0 \in dom(p)$, then $y^1 \blacktriangleleft x^0$ implies that $y^1 \in dom(p)$. If, for instance, we would have selected the probe set $p' = \{(y^1, \varepsilon)\} \in \mathcal{P}_{\iota, x^1 \cdot x^2}$, the action x^0 would have been detected as a missed action in state q_6, due to $y^1 \blacktriangleleft x^0$. After the third iteration, $V \setminus V_P = \{(q_3, x^0), (\iota, x^1 \cdot x^2 \cdot y^1)\}$. In the fifth iteration we reach state q_6 in which we identify $y^1 \cdot y^2$ as a missed action. As a result, the pair (q_5, ε) is added to V. Since q_6 is a deadlock state, we have $\mathcal{P}_{\iota, x^1 \cdot x^2 \cdot y^1} = \emptyset$.

In the above example, we have selected probe sets manually, such as to obtain maximal reduction. In Section 6.5.3, we will give some guidelines on how probe sets could be selected in general.

6.4.3 Correctness

In order to have a correct reduction of a transition system, we must select sufficiently many probe sets and take care of the missed actions. This can be defined by a so-called *probing* for the transition system under consideration.

Definition 6.21 (probing)**.** *Let S be a transition system. A* probing *for S is a K-indexed set $P = \{p_{q,w}\}_{(q,w) \in K}$ where*

1. *K is a set of vectors of S such that $(\iota, \varepsilon) \in K$;*
2. *for all $(q, w) \in K$, $p_{q,w}$ is a probe set such that $(q \uparrow p_{q,w}(a), w \cdot a - p_{q,w}(a)) \in K$ for all $a \in dom(p_{q,w})$;*
3. *for all $(q, w) \in K$ and all $v \cdot a \in fma(q, w)$, there is a word $u \precsim w - v$ such that $(q \uparrow v \cdot a, u) \in K$ and $u \not\rightarrow a$.*

We write $(q, w) \uparrow p(a)$ for the vector $(q \uparrow p_{q,w}(a), w \cdot a - p_{q,w}(a))$ in Clause 2.

P is called fair *if for all $(q, w) \in K$ there is a function $n_{q,w}$: $enabled(q \uparrow w) \to \mathbb{N}$, assigning a natural number to all actions enabled in $q \uparrow w$, such that for all $a \in enabled(q \uparrow w)$, either $a \in dom(p_{q,w})$, or $n_{(q,w) \uparrow p(b)}(a) < n_{q,w}(a)$ for some $b \in dom(p)$.*

In the above definition, the set K represents the vectors for which the probing, say P, contains a probe set. That is, if the vector $(q, w) \in K$ is probed, then there exists a $p_{q,w} \in (\mathcal{P}_{q,w} \cap P)$. Obviously, the vector (ι, ε), representing the initial state, should be in K, as required by Clause 1.

Clause 2 guarantees that, if $(q, w) \in K$ is probed and $p_{q,w}$ is the probe set selected for (q, w), then the vector representing the state reached by exploring any action $a \in dom(p_{q,w})$ is also contained in K and is therefore also probed. Let us take a closer look at the vector that is required to be in K for every action $a \in dom(p_{q,w})$. The fact that $a \in dom(p_{q,w})$ means that from the state

$q{\uparrow}w$, the action a is executed, which in turn implies that the state $q'' = q{\uparrow}w{\cdot}a$ is visited, i.e., $q'' \in Q_S$. The selection of the probe set in the state $q{\uparrow}w{\cdot}a$ will, however, not be based on the actual path by which the state is reached, i.e., $w{\cdot}a$, but only on some (weak) suffix thereof. That is to say, Clause 2 specifies the deletion of causal history; this influences the probe set selection of the reached state. This is depicted in Fig. 6.6. Note that although the vector $(q, w){\uparrow}p(a)$ is included in K, this does not necessarily mean that the corresponding path and states along that path are actually included in the transition system; the actual purpose of the vector $(q, w){\uparrow}p(a)$ is to properly select a probe set in q''.

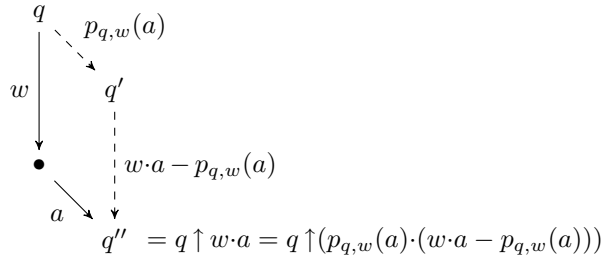


Figure 6.6: If $a \in dom(p_{q,w})$ then $(q, w) \uparrow p(a) \in K$.

Clause 3 takes care of the missed actions. Note that for missed actions, the paths representing them should actually be explored, i.e., included in the reduced transition system. A probing being fair ensures that every enabled action will eventually be included in some probe set.

The following is the core result of this chapter, on which the correctness of the algorithm depends. It states that every fair probing gives rise to a correct reduction.

Theorem 6.22. *If P is a fair probing of a transition system S, then the transition system R characterized by $Q_R = \{q \uparrow w \mid (q, w) \in dom(P)\}$ reduces S.*

In order to prove the above theorem, we need the following auxiliary properties for which the proofs are included in Appendix E.

Lemma 6.23. *Let u, v, and w be words and a be some action. Then,*

1. *If $u \precsim v$, then $u{\cdot}w - v = w - (v - u)$.*
2. *If $v \precsim {\downarrow_a}w$ then ${\downarrow_a}w \simeq v{\cdot}{\downarrow_a}(w - v)$.*

3. If $v - w \not\rightsquigarrow a$ *then* $\downarrow_a v \simeq \downarrow_a w$.

The first item states that if u is a weak prefix of v, i.e., $\exists v' : u{\cdot}v' \simeq v$, then removing v from $u{\cdot}w$ for some word w is effectively the same as removing only the actions of this v' from w. The second property claims that the prime cause of a in w is equivalent to v concatenated with the prime cause of a in $w - v$, given that v is a weak prefix of the prime cause of a in w. Part 3 of Lemma 6.23 states that if v and w only differ in actions that do not influence a, then the prime causes of a in v and in w are equivalent.

Lemma 6.24. *Let* P *be a fair probing. For all* $(q, w) \in dom(P)$ *and* $a \in enabled(q \uparrow w)$*, there is a vector* $(q', w') \in dom(P)$ *such that* $q \xrightarrow{v} q'$ *for some* $v \precsim \downarrow_a w$*, and* $q \uparrow w \xrightarrow{u \cdot a} q' \uparrow w'$ *for some* $u \not\rightsquigarrow a$.

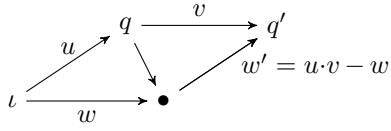
The above result claims when an action a is enabled by a vector (q, w), then either a is executed from $q{\uparrow}w$ or from some state, say $\bar{q}$, that can be reached from $q{\uparrow}w$ by a path u that contains only actions independent with a, i.e., $u \not\rightsquigarrow a$. We do not require that $\bar{q}$ is reached by actually executing u from $q{\uparrow}w$; it is sufficient to reach $\bar{q}$ from q by a path that is equivalent to $w{\cdot}u$. Lemma 6.24 can be proved by induction on $n_{q,w}(a)$. The full proof of Lemma 6.24 is included in Appendix E.

We now have all the ingredients required to prove Theorem 6.22.

Proof of Theorem 6.22. For each trace w of S we define the *extensions* of w, X_w, as the set of K-vector end states reachable from $\iota \uparrow w$, together with the sequence of actions leading there.

$$X_w \quad = \quad \{(u{\cdot}v - w, q \uparrow v) \mid (q, v) \in K, \iota \xrightarrow{u} q, u \precsim w \precsim u{\cdot}v\} \ . \tag{6.8}$$

Hence, X_w consists of those pairs of traces w' and states q' such that q' is the target of some vector $(q, v) \in K$, and w' leads up to q' from $\iota \uparrow w$; i.e., $\iota \xrightarrow{w \cdot w'} q'$. This can be schematically depicted as follows:



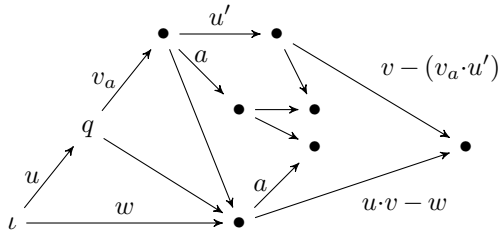
The requirement in (6.8) that $w \precsim u{\cdot}v$ then implies that the behaviour of w is included in $u{\cdot}v$, which in turn is explored in the reduced transition system.

For every reachable $q \in Q_S$, consider w_q such that $\iota \xrightarrow{w_q} q$. For every $(w', q') \in X_{w_q}$ it follows that $q' \in Q_R$ and $q \xrightarrow{w'} q'$. Thus, in order to show that Q_R induces a reduction of S in the sense of Def. 6.9, it is sufficient to prove that all X_w are non-empty. We prove this by induction on $k_w = \min\{|w - u| \mid (q, v) \in K, \iota \xrightarrow{u} q, u \precsim w\}$.

Base case. If $k_w = 0$ then $|w - u| = 0$ for some $(q, v) \in K$, $\iota \xrightarrow{u} q$ and $u \precsim w$. It follows that $w \simeq u \precsim u{\cdot}v$ and hence $(v, q \uparrow v) \in X_w$.

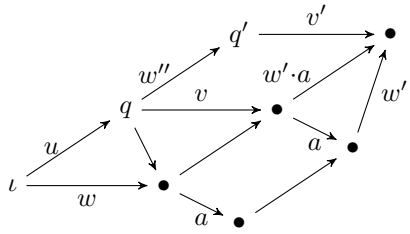
Induction hypothesis. Assume X_w is non-empty whenever $k_w \leq n$; we will prove $X_{w\cdot a}$ (for arbitrary a such that $\iota \vdash w{\cdot}a$) is also non-empty. Regard $(u{\cdot}v - w, q \uparrow v) \in X_w$; hence $\iota \xrightarrow{u} q$, $(q, v) \in K$ and $u \precsim w \precsim u{\cdot}v$. We recognize the following cases:

- $u{\cdot}v - w \rightsquigarrow a$. Then $w{\cdot}a \not\precsim u{\cdot}v{\cdot}a$ and hence $(\downarrow_a(w - u), a)$ is missed along (q, v). Let us denote $v_a = \downarrow_a(w - u)$. Clause Def. 6.21.3 then states there is a u' such that $(q \uparrow (v_a{\cdot}a), u') \in K$, $a \not\rightsquigarrow u'$ and $v_a{\cdot}u' \precsim v$. The situation can be depicted schematically as follows:



  Clearly $u{\cdot}v_a{\cdot}a \precsim w{\cdot}a$ and $|w{\cdot}a - u{\cdot}v_a{\cdot}a| < |w - u|$. It follows from the induction hypothesis that $X_{w\cdot a}$ is non-empty.

- $u{\cdot}v - w \not\rightsquigarrow a$. Then $\iota \uparrow w \vdash a$ implies $q \uparrow v = \iota \uparrow u{\cdot}v = \iota \uparrow w{\cdot}(u{\cdot}v - w) \vdash a$ due to dependency consistency (see (6.4)); hence, $a \in \mathit{enabled}(q \uparrow v)$. Lemma 6.24 then implies there is some $(q', v') \in K$ such that $q \xrightarrow{w''} q'$ for some $w'' \precsim \downarrow_a v$ and $q \uparrow v \xrightarrow{w'{\cdot}a} q' \uparrow v'$ for some $w' \not\rightsquigarrow a$; hence $w''{\cdot}v' \simeq v{\cdot}w'{\cdot}a$. The situation can be depicted schematically as follows:



From $u{\cdot}v{-}w \not\rightarrow a$ it follows that $v-(w-u) \not\rightarrow a$, and so $\downarrow_a v \simeq \downarrow_a(w-u)$ (due to Lemma 6.23.2 and Lemma 6.23.3). But then

$$u{\cdot}w'' \precsim u{\cdot}\downarrow_a v \simeq u{\cdot}\downarrow_a(w-u) \precsim u{\cdot}(w-u) \simeq w \ .$$

Furthermore, $w{\cdot}a \precsim u{\cdot}v{\cdot}w'{\cdot}a$ due to $w \precsim u{\cdot}v$ in combination with $u{\cdot}v-w \not\rightarrow a$ and $w' \not\rightarrow a$. We may conclude $(u{\cdot}w''{\cdot}v'-w{\cdot}a, q' \uparrow v') \in X_{w{\cdot}a}$. □

If we investigate Algorithm 4, it becomes clear that this is not yet correct. The total collection of vectors and probe sets produced by the algorithm give rise to a correct probing in the sense of Def. 6.21 (where the u of Clause 3 is always set to ε), and also generates a probing; however, this probing is not fair. As a result, Algorithm 4 suffers from the so-called "ignoring problem", well-known from other partial order reduction techniques (see, e.g., [185, 91]). This refers to the fact that some enabled action is never explored. Such a case might occur for cyclic state spaces. If, in Fig. 6.7, the solid arrows represent the reduced state space, one can easily see that action b is ignored in all states q_0, q_1, and q_2. Therefore, the probing producing this reduced state space is not fair, resulting in some behaviour of the system such as, e.g., the word $b{\cdot}a_0{\cdot}a_1{\cdot}a_2{\cdot}a_0{\cdot}a_1{\cdot}a_2\cdots$, being ruled out.

6.5 The Probe Set Algorithm

In this section, we put the finishing touch on the algorithm: ensuring fairness, identifying missed actions, and constructing probe sets. For this, we take the entity-based setting from Section 6.3.2.

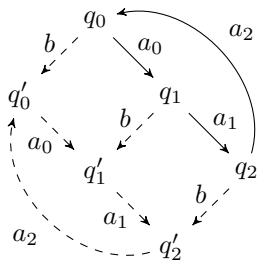


Figure 6.7: The "ignoring problem", where action b might be ignored.

6.5.1 Identifying Missed Actions

As we have discussed in Section 6.4, finding the missed actions $fma(q, v)$ by investigating all weak prefixes of v negates the benefits of the partial order reductions.

In the the entity-based setting of Section 6.3.2, however, a more efficient way of identifying missed actions can be defined on the basis of an *over-approximation*. We define the over-approximation of the target state of a vector (q, w), denoted $q{\Uparrow}w$, as the union of all entities that have appeared along that vector.

Definition 6.25 (over-approximation)**.** *For a given vector (q, w), the* over-approximation *of (q, w), denoted $q{\Uparrow}w$, is defined as follows:*

$$q{\Uparrow}w \quad := \quad E_q \cup \bigcup_{a \in A_w} C_a \ .$$

In addition, we define *weak enabledness* of actions for a set of entities by only checking for the presence of read and deleted entities.

Definition 6.26. *An action a is* weak enabled *by a set E of entities, denoted $E \Vdash a$, iff $(R_a \cup D_a) \subseteq E$.*

Based on the above definitions we can introduce the set of *potentially missed actions* for a given vector, which is a superset of the set of fresh missed actions.

Definition 6.27 (potentially missed actions)**.** *Let $(q, w{\cdot}b)$ be a vector. Then, $a \in Act$ is a* potentially missed action *if either $q \vdash w{\cdot}a$ and $b \blacktriangleleft a$, or the following conditions hold:*

1. *a is weakly but not strongly enabled: $q{\Uparrow}w \Vdash a$ and $q{\uparrow}w \nvdash a$;*

2. *a was somewhere disabled:* $\exists c \in A_w : c \blacktriangleleft a$;
3. *a is freshly enabled:* $b \triangleright a$.

We will use $pma(q, v)$ to denote the set of potentially missed actions in the vector (q, v). It is not difficult to see that $pma(q, v) \supseteq fma(q, v)$ for arbitrary vectors (q, v). However, even for a given $a \in pma(q, v)$ it is not trivial to establish whether it is really missed, since this still involves checking if there exists some $v' \precsim v$ with $q \uparrow v' \vdash a$, and we have little prior information about v'. In particular, it might be that v' is smaller than the prime cause $\downarrow_a v$. For instance, if $E_q = \{1\}$, $C_b = \{2\}$, $D_c = \{1, 2\}$ and $R_a = \{1, 2\}$ then $q \nvdash v{\cdot}a$ with $v = b{\cdot}c{\cdot}b$, and $\downarrow_a v = v$; nevertheless, there is a prefix $v' \precsim v$ such that $q \vdash v'{\cdot}a$, viz. $v' = b$.

In some cases, however, the question whether an action is really missed is much easier to answer; namely, if the prime cause $\downarrow_a v$ is the only possible candidate for such a v'. The prime cause can be computed efficiently by traversing backwards over v and removing all actions not (transitively) influencing a.

For ensuring that for a vector (q, w), we only have to investigate the prime causes of all potentially missed actions, we introduce the notion of *reversing actions*. Informally, two actions are reversing if one (partially) undoes the results of the other. In the entity-based setting this can be formalized as follows.

Definition 6.28 (reversing actions)**.** *Two entity-based actions a, b are* reversing *if $C_a \cap D_b \neq \emptyset$ or $D_a \cap C_b \neq \emptyset$. A word w is said to be* reversing free *if no two actions $a, b \in A_w$ are reversing.*

For a word w and an action a, the set of actions in w that are reversing with respect to a is denoted $rev_a(w)$, i.e., $rev_a(w) = \{b \in A_w \mid a, b \text{ are reversing}\}$. Dually, reversing freedom means that no action (partially) undoes the effect of another. For instance, in the example above b and c are reversing due to $C_b \cap D_c = \{1\}$, so v is not reversing free. The following result now states that for reversing free vectors, we can efficiently determine the fresh missed actions. The proof is included in Appendix E.

Proposition 6.29. *Let (q, v) be a vector with v reversing free.*

1. *For any action a, $q \vdash v'{\cdot}a$ with $v' \precsim v$ implies $\downarrow_a v \precsim v'$.*
2. *$fma(q, v) = \{a \in pma(q, v) \mid q \vdash \downarrow_a v{\cdot}a\}$.*

6.5.2 Ensuring Fairness

To ensure that the probing we construct is fair, we will keep track of the "age" of the enabled actions. That is, if an action is *not* probed, its age will increase

in the next round, and probe sets are required to include at least one action whose age is maximal. This is captured by a partial function $\alpha\colon Act \rightharpoonup I\!N$. To manipulate ageing functions, we define the following operators, where A is a set of actions:

$$\begin{aligned} \alpha \oplus A &:= \{(a, \alpha(a)+1) \mid a \in dom(\alpha)\} \cup \{(a,0) \mid a \in A \setminus dom(\alpha)\} \\ \alpha \ominus A &:= \{(a, \alpha(a)) \mid a \notin A\} \ . \end{aligned}$$

Intuitively, $\alpha \oplus A$ initializes the age of the actions in A to zero, and increases all other ages; $\alpha \ominus A$ removes the actions in A from α. Furthermore, $\max(\alpha)$ denotes the set of oldest actions in α. Formally:

$$\max(\alpha) := \{a \in dom(\alpha) \mid \forall b \in dom(\alpha) : \alpha(a) \geq \alpha(b)\} \ .$$

The fairness criterion on probe sets can now be formalized by introducing a *satisfaction* relation between sets of actions and ageing functions. A set A of actions *satisfies* an ageing function α iff $\alpha = \emptyset$ or $A \cap \max(\alpha) \neq \emptyset$. For probe sets we thus have the following: a probe set p is fair with respect to an ageing function α iff $dom(p)$ satisfies α, denoted $p \models \alpha$.

6.5.3 Constructing Probe Sets

When constructing probe sets, there is a trade-off between the size of the probe set and the length of the vectors. On the one hand, we aim at minimizing the size of the probe sets; on the other hand, we also want to minimize the size of the causal history. For example, probe sets consisting of pairs $(a, \emptyset)$ only (for which the second and third condition of Def. 6.20 is fulfilled vacuously) are typically small, but then no causal history can be discharged. Another extreme case is when a probe set for (q, w) consists of pairs $(a, \downarrow_a w)$. In this case, the maximal amount of causal history is discharged that is still consistent with the third condition of Def. 6.20, but the probe set domain is likely to equal the set of enabled actions, resulting in no reduction at all.

The probe sets $p_{q,w}$ we construct will furthermore ensure that the vectors of the new continuation points are reversing free. Therefore, for every $p_{q,w}$ we additionally require that for all $a \in dom(p_{q,w})$: $rev_a(w) \subseteq A_{p(a)}$. Since $rev_a(w) \subseteq A_{\downarrow_a w}$, this does not conflict with Def. 6.20.

An interesting probe set $p_{q,w}$ could be constructed such that $p_{q,w}$ satisfies the condition on disabling actions and furthermore $p_{q,w}(a) = \downarrow_a w$ except for one action, say a', which is mapped to the empty vector, i.e., $p_{q,w}(a') = \varepsilon$. This

Algorithm 5 Probe set based partial order reduction algorithm, definitive version.

```
 1: let Q ← ∅;                                              // set of states visited
 2: let V_P ← ∅;                                            // set of vectors processed
 3: let V ← {(ι_S, ε, ∅)};                                  // set of vectors selected
 4: while V \ V_P ≠ ∅ do
 5:     choose (q, w, α) ∈ V \ V_P;
 6:     let V_P ← V_P ∪ {(q, w, α)};
 7:     let Q ← Q ∪ {q↑w};
 8:     for all v·m ∈ fma(q, w) do
 9:         let Q ← Q ∪ {q↑v};
10:         let V ← V ∪ {(q↑v·m, ε, ∅)};
11:     end for
12:     choose p ∈ P_{q,w} such that p ⊨ α, and ∀a ∈ dom(p) : rev_a(w) ⊆ A_{p(a)};
13:     let α ← α ⊕ enabled(q↑w) ⊖ dom(p);
14:     let V ← V ∪ {(q↑p(a), w·a − p(a), α) | a ∈ dom(p)};
15: end while
```

action a' then ensures that no further action needs to be included in the probe set. The selection of this action a' can be based on the length of its prime cause within w.

There is a wide range of similar heuristics that use different criteria for selecting the first action from which to construct the probe set or for extending the causal history to be removed. Depending on the nature of the transition system to be reduced, specific heuristics might result in more reduction. This is a matter of future experimentation.

6.5.4 Putting All Together

Extending Algorithm 4 with fair probe sets and properly updated ageing functions results in Algorithm 5, being the definitive version of the algorithm. We will briefly discuss the main differences. Ageing functions are included in the elements in the set V of vectors to be examined. Instead of tuples, V now contains triples (q, w, α) representing a vector (q, w) extended with an ageing function α. Before creating continuations for the actions in the probe set, the ageing function is first properly updated (line 13), i.e., the age of actions that are currently enabled and outside the probe set is increased, and actions in the probe set are removed from the ageing function. For missed actions, continuations are created with empty ageing functions (line 10). The probe sets that

are selected in this algorithm must be fair with respect to the ageing function and result in reversing free continuations (line 12).

The correctness of Algorithm 5 is proved using Theorem 6.22. The proof relies on the fact that the algorithm produces a fair probing, in the sense of Def. 6.21. This can easily be verified by inspecting the different clauses one by one.

Theorem 6.30. *For a transition system S, Algorithm 5 produces a set of states $Q \subseteq Q_S$ characterizing a reduction of S.*

For our running example of Figs. 6.2 and 6.5, there are several observations to be made.

- The probe sets we constructed in Fig. 6.5 satisfy the reversing freedom condition. Note that (in terms of (6.7)) the action x^0 reverses x^1 and x^2, and likewise, y^0 reverses y^1 and y^2.
- The run in Fig. 6.5 is *not* fair: after the first step, the age of y^1 becomes 1 and hence this should be chosen in preference to x^2. This suggests that our method of enforcing fairness is too rigid, since the ignoring problem does not actually occur in this example.

6.6 Probe Sets for Graph Production Systems

We actually aim at applying the algorithm developed in previous sections to a setting in which state spaces are generated from graph production systems. As yet there is no implementation of the proposed algorithm in the entity-based setting, let alone in the Groove Tool Set (for the graph transformation framework). Therefore, we here indicate 'manually' how the algorithm should be applied in a graph transformation framework.

Although graph productions also specify the creation, preservation, and deletion of graph elements, and the notion of forbidden entities corresponds to negative application conditions on graph elements, there are some issues to be worked out carefully. We first elaborate on these issues. Thereafter we introduce a small example graph production system on which we apply the algorithm manually, thus producing a correct reduced graph transition system.

Graph transition systems and entity-based transition systems have many commonalities. Both frameworks, for example, do not satisfy the assumptions presented in Section 6.1 concerning the pre-known bound on the number of actions and the way entire systems are constructed from individual processes. But

there are also some fundamental differences between them. A minor difference is that in the graph formalism, the basic building blocks for individual states are nodes and edges, instead of entities. The existence of edges depends on the existence of their source and target nodes, whereas for entities there are no such constraints on their existence. Furthermore, for entity-based transition systems, actions are defined in terms of sets of entities that are preserved, deleted, created, and forbidden. In the graph transformation context, however, individual computation steps are represented by rule applications being pairs (p, m) consisting of a transformation rule p and a matching m of that rule to some graph G.

For entity-based transition systems we had to assume that the set of actions enabled in some state is finite and computable. In the graph transformation context these assumptions are satisfied naturally. Since graphs that are generated in a finite number of steps are of finite size, the number of applications of a single transformation rule to such graphs is also finite. Thus, given a graph G and a transformation rule p, one can compute the set of all rule applications of p to G. The fact that graph production systems consist of a finite set of transformation rules then implies the bound on the number of rule applications for a single graph.

6.6.1 Graph Elements as Entities

For applying the probe set partial order reduction algorithm to graph production systems, there are at least two alternatives. A first approach is to instantiate the framework of abstract enabling and disabling relations in the context of graph transformations. Alternatively, we can define a translation from rule applications to entity-based actions that encode the effect of the rule application in terms of entities. Here, we elaborate on the second approach. Then, graph elements are conceptually thought of as entities. That is, every graph G that is generated in a finite amount of steps is encoded by a finite set E_G, such that $E_G \subseteq \mathit{Ent}$. At first, we only consider graph transformation rules without negative application conditions.

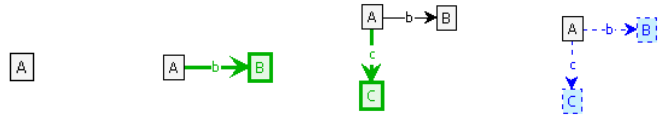
To simplify the presentation of the encoding, we consider graphs as the set-wise union of the disjoint sets of their nodes and edges. We then introduce sequences of entities for every graph element created by each of the rules. Formally, if X_p denotes the set of graph elements to be created by a transformation rule p, i.e., $X_p = R \setminus p(L)$ (due to the simplification), every graph element $x \in X_p$ has an associated sequence $c_x \in \mathit{Ent}^*$ of *candidate* entities. The entity-based encoding of a rule application (p, m) can then be characterized by the

matching and the actual choice of the candidate for every graph element to be created. This choice cannot be arbitrary. In fact, the encoding must choose the first (or the "smallest") candidate in the sequence that is still available, i.e., not yet in the graph. This means that all smaller candidates may not be available, i.e., they are required to be in the graph. Some notation: if $c_x = e_1, e_2, \ldots$ is the sequence of candidates for some graph element x, then $c_x\lfloor_i$ denotes the ith element of c_x, e.g., here $c_x\lfloor_2 = e_2$.

For a rule application (p, m), the corresponding entity-based action a then is a pair $a = (m, \theta\colon X_p \to \mathbb{N})$, where θ is the *choice function* mapping every element to be created to the index of the chosen entity in the corresponding sequence of candidates, such that:

- $R_a = m(L \cap dom(p)) \cup \bigcup_{x \in X_p} \{c_x\lfloor_i \mid i < \theta(x)\}$;
- $D_a = m(L \setminus dom(p))$;
- $C_a = \{c_x\lfloor_{\theta(x)} \mid x \in X_p\}$.

The encoding can be explained more intuitively by the following example. Suppose we have the graph production system as depicted in Fig. 6.8, consisting a the start graph (depicted in Fig. 6.8(a)) and two transformation rules. The encoding of the start graph contains one entity representing the single A-labelled node.



(a) Start graph. **(b)** Rule create-B. **(c)** Rule create-C. **(d)** Rule del-B-C.

Figure 6.8: Example graph production system.

In Fig. 6.9 we have depicted an excerpt of the state space of the graph production system from Fig. 6.8. All graph elements are accompanied by a number representing their identity, which in turn can be interpreted as the corresponding entity. Here, action a actually encodes the only application of the rule create-B to the start graph. Likewise, the action b encodes an application of the rule create-C. In Table 6.1 all actions of this example are listed, with the name of the rule of which they encode an application, and the actual associated

sets of entities. Apparently, for the action a, the choice function θ_a mapped the B-labelled node to entity 2 and the b-labelled edge to entity 3.

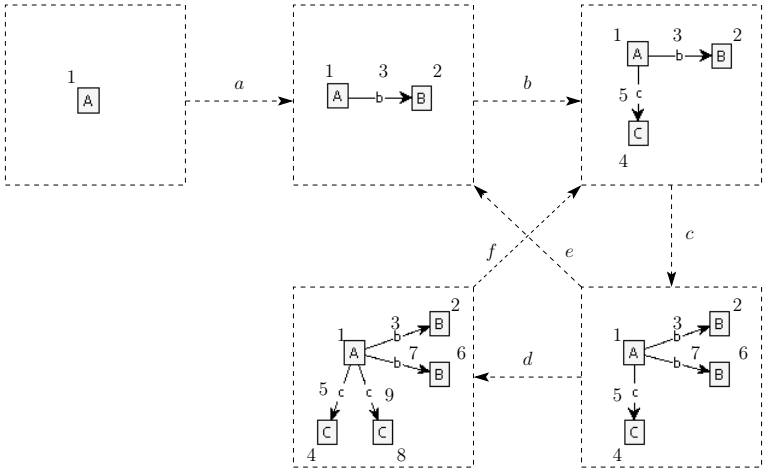


Figure 6.9: Excerpt from the state space generated by the graph production system from Fig. 6.8.

Action	**Rule**	R	C	D	N
a	create-B	$\{1\}$	$\{2,3\}$	$\emptyset$	$\emptyset$
b	create-C	$\{1,2,3\}$	$\{4,5\}$	$\emptyset$	$\emptyset$
c	create-B	$\{1\}$	$\{6,7\}$	$\emptyset$	$\emptyset$
d	create-C	$\{1,6,7\}$	$\{8,9\}$	$\emptyset$	$\emptyset$
e	del-B-C	$\{1\}$	$\emptyset$	$\{4,5,6,7\}$	$\emptyset$
f	del-B-C	$\{1\}$	$\emptyset$	$\{6,7,8,9\}$	$\emptyset$

Table 6.1: Entity-based encoding of the actions from Fig. 6.9.

Negative Application Conditions and Simple Graphs

Allowing the use of negative application conditions, introduces some further issues. For example, when a rule forbids the existence of an A-labelled node, then the corresponding entity-based action should basically forbid the existence of all

possible entities that may represent such a node, which may be an infinite number. Therefore, the original definition of entity-based actions (recall Def. 6.11) should allow the set of forbidden entities to be infinite.

Still, there are some special cases of negative applications that fit in the current framework. One example are so-called *edge-embargoes*, i.e., NACs that forbid the existence of edges between two existing nodes. This is due to the fact that in our simple graph formalism (recall Section 2.1), there can at most be one edge with a specific label between any two nodes.

One of the somewhat unnatural features of the simple graph formalism is related to the creation of edges between existing nodes. If a rule application specifies the creation of an edge that already exists in the graph, the new edge will be identified with the existing one. Basically, such a rule should be interpreted as two different rules of which only one is applicable at a time. The first rule specifies the creation of the edge, requiring that it does not yet exist in the graph (using an edge-embargo); the second rule requires the existence of the edge and preserves it. In general, this problem can be worked around by first specifying all the separate cases for such rules, although this results in an exponential blow-up in the number of created edges. Needless to say, freshly created edges of which the source or target node is also created do not have to be considered in this respect.

6.6.2 Example: Concurrent Append

In the following, we will show how the algorithm generates a reduced state space for an example graph production system. For this, we model a system in which two processes try to append associated values to the end of some list in a concurrent fashion. The initial configuration and the rules of this example are depicted in Fig. 6.10 and Fig. 6.11, respectively. The full graph transition system is depicted in Fig. 6.12.

Initially, both processes have a this-pointer to the first Entry of the list. From the initial configuration, both processes can move their this-pointer to the next Entry. This is specified by the next-rule depicted in Fig. 6.11(a). If one process reaches the last Entry of the list (that Entry carries a last-flag), it can create a fresh Entry to which it attaches its associated value, as specified by the append-rule depicted in Fig. 6.11(b). The return-rule, depicted in Fig. 6.11(c), performs some cleanup actions. Note that in this example, the next-rule must be split into two rules as described above, due to the creation of the this-edge between two existing nodes. One of those two rules, however, will actually never be applicable since it requires the existence of two outgoing this-edges from a

single Append-node while there can at most be one.

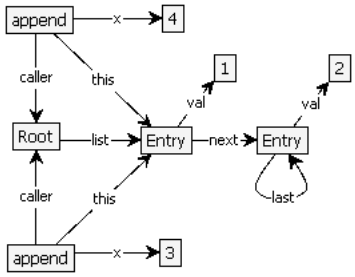
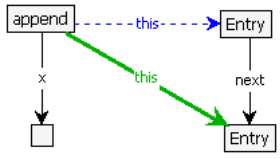
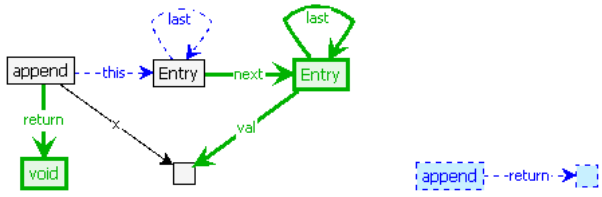


Figure 6.10: Start graph of the concurrent append graph production system.



(a) next.



(b) append. **(c)** return.

Figure 6.11: Graph productions of the concurrent append example.

From the initial configuration, both processes can propagate their this-pointer forward, independent of each other. This can be observed from the diamond formed by the states $s0$, $s1$, $s2$, $s4$, and their intermediate next-transitions (in Fig. 6.12). In Fig. 6.13, we have depicted the states $s0$, $s1$, and $s2$ together with their encoding in entities, depicted as blue numbers.

It is easy to see that both matchings of the next-rule must have some graph elements in common, e.g., both Entrys of the list. In terms of entities, if the first

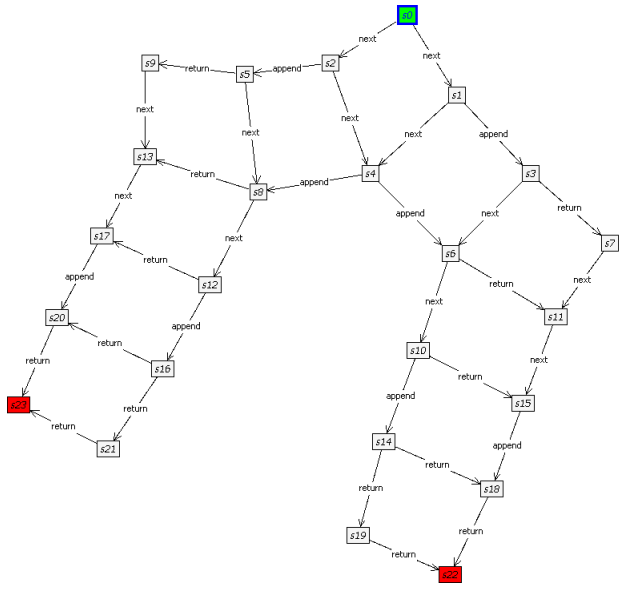


Figure 6.12: Full state space of the concurrent append example.

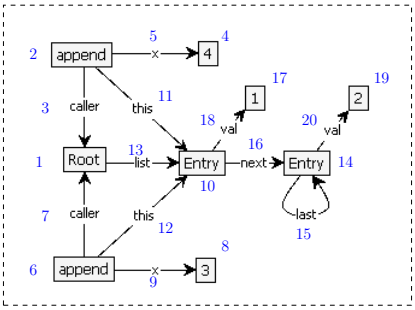
next-application (resulting in state $s1$) is encoded as action a and the second next-application (resulting in state $s2$) is encoded as action b, we have:

$$R_a = \{2, 4, 5, 6, 10, 11, 14, 15, 16\},\ D_a = \{11\},\ C_a = \{21\},\ \text{and}\ N_a = \emptyset\ ,$$

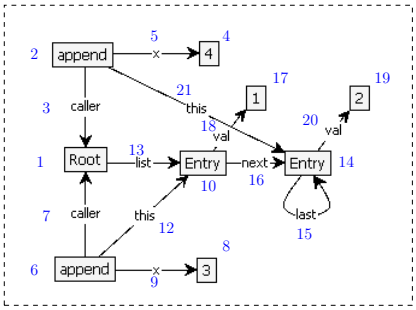
$$R_b = \{6, 7, 8, 9, 10, 12, 14, 15, 16\},\ D_b = \{12\},\ C_b = \{22\},\ \text{and}\ N_b = \emptyset\ .$$

Since a and b delete disjoint sets of graph elements, and the next-rule does not have NACs, they do not disable each other. In terms of entities, the corresponding actions share some entities they read, but their associated sets of deleted entities are disjoint. Obviously, the sets of forbidden entities of both actions are empty, since the rule they originate from does not include any NACs. Apparently, the entities 21 and 22 are the first (and only) available entities to represent the this-pointers between the nodes represented by entities 2 and 14 (for action a), and 6 and 14 (for action b), respectively.

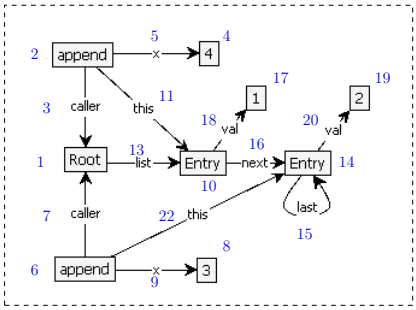
When both processes have moved their this-pointer forward, thereby bringing the system in state $s4$, the append-rule has two applications, one for each process. From Fig. 6.12 it appears that both applications disable each other, as can



(a) State $s0$.



(b) State $s1$.



(c) State $s2$.

Figure 6.13: States $s0$, $s1$, and $s2$, including their entity-encoding.

be observed from the fact that in both states $s6$ and $s8$, the other append-application does not apply anymore. This follows immediately from the fact that both append-applications delete the last-flag of the last Entry of the list. Again, in terms of entities, the corresponding actions both delete entity 15. If one process appends it value to the list, the other process has to move its this-pointer forward once again, after which it can then append its associated value to the list. Note that this last append-application is based on a different matching then the previous append-application and therefore encoded as a different entity-based action, e.g., the append-transitions between the states $s4$ and $s6$ and between the states $s10$ and $s14$ are based on different matchings.

By applying the probe set algorithm to this example, we can save exploring almost half the number of states and more than half the number of transition. Fig. 6.14 depicts one of the possible reduced state spaces of this example. For

this reduction, we highlight the following. First of all, in the initial state, there are three possible probe sets, namely one containing either of the enabled rule applications, and one containing both. In this case, a probe set is selected in which only one rule application is included. Due to the fairness condition, the probe set in state $s2$ must then contain the rule application that was omitted in the probe set of the initial state. A similar situation occurs in state $s12$, since in state $s8$ we excluded the application of the return-rule from the probe set. Therefore, in state $s12$ we have no choice but to select a probe set including that rule application. Fortunately, this rule application gives rise to a singleton probe set, due to its independence with the append-application.

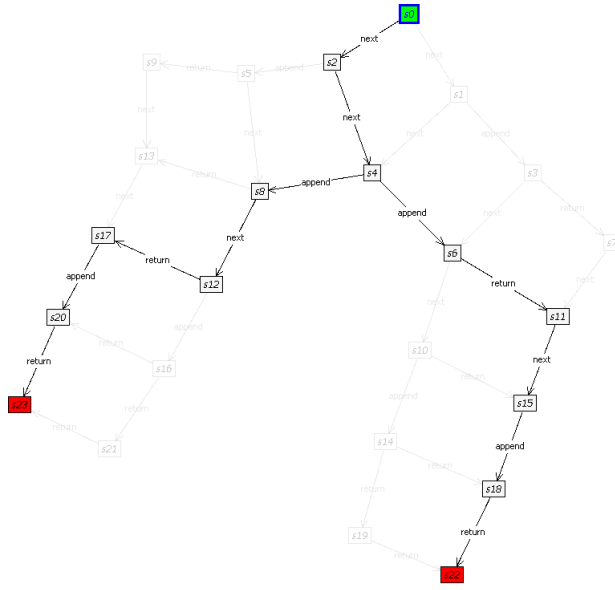


Figure 6.14: Reduced state space of the concurrent append example.

Remark 6.31. *The perceptive reader might have noticed that the above example seems to be inconsistent from a theoretical point of view with the description of how rule applications are encoded as entity-based actions. Concretely, both applications of the next-rule in the initial state (see Fig. 6.13) should have selected the same entity for the new this-pointer to be created, namely entity 21. This would cause the two next-applications to disable each other, although in the*

example they are independent. From a practical point of view, there are two reasons why the example still holds. Firstly, when dealing with simple graphs, edges have no identity. That is to say, edges can uniquely be identified through their source, target, and label. Therefore, the label of the edge to be created and the entities representing the source and target node of that edge determine entity to be selected. Secondly, in an implementation, rule applications are computed one by one. The encoding of one rule application then effects the availability of entities for other applications of the same rule (to the same state). This guarantees that independent rule applications result in corresponding entity-based actions that are also independent.

6.7 Conclusion

6.7.1 Summary

We have proposed a new algorithm for dynamic partial order reduction with the following features:

- it can reduce systems that have no non-trivial persistent sets (and so traditional methods do not have an effect);
- it is based on abstract enabling and disabling relations, rather than on concurrent processes. This makes it suitable, e.g., for graph production systems;
- it uses a universe of actions that does not need to be finite or completely known from the beginning; rather, by adopting an entity-based model, enabled and missed actions can be computed on-the-fly. This makes it suitable for dynamic systems, such as (object-oriented) software;
- it can deal with cyclic state spaces.

We have proved the algorithm correct (in a rather strong sense) and shown it on a small running example. Finally, we have indicated how the different concepts should be instantiated in the graph transformation framework for the algorithm to be applied in that setting. However, an implementation, for both the entity-based and the graph transformation setting, is as yet missing. It is planned to implement the algorithm in the Groove Tool Set.

6.7.2 Related Work

Traditional partial order reduction (see e.g., [91, 185]) is based on statically determined symmetric dependency relations, e.g., for constructing *persistent sets* (see Section 6.2). More recently, dynamic partial order reduction techniques have been developed that compute dependency relations on-the-fly. In [85], for example, partial order reduction is achieved by computing persistent sets dynamically. The main difference with the static approach is that it maintains an additional ordering relation (called the "happens-before" relation) on the transitions of the path under consideration. This ordering relation is then used to identify the *backtracking points*, i.e., the paths of which no representative is yet contained in the reduced state space. This technique performs a stateless search, which is the key problem of applying it to cyclic state spaces. Furthermore, since the algorithm ensures that, eventually, the subsets of transitions explored from every state are persistent, for (sub-)state spaces with no non-trivial persistent sets, no reduction will be obtained.

In [95], Gueta et al. introduce a *Cartesian* partial order reduction algorithm which is based on reducing the number of context switches. This approach has been shown also to work in the presence of cycles. Like the persistent set approach, this approach is based on processes or threads performing read and/or write operations on local or shared variables. The setting we propose is more general in the sense that actions are able to create or delete entities that can be used as communication objects. Therefore, our algorithm is better suited for systems in which resources are dynamically created or destroyed without an a priori bound.

6.7.3 Discussion

Left-Closing versus Right-Closing Probe Sets. In the current version, probe sets are constructed as to satisfy the conditions stated in Def. 6.20. The first conditions states that every probe set should be left-closed under disabling. One might expect that, similarly, a probe set should also be *right-closed* under disabling, i.e., for all $a \in dom(p)$ and $b \in enabledq \uparrow w$, $a \blacktriangleleft b$ implies $b \in dom(p)$. In many cases, however, right-closing the probe set could make it unnecessary large. Instead we make sure that those actions will be marked as missed actions. Alternatively, to reduce the missed action analysis we could have defined probe sets constructively, first requiring the conditions as from Def. 6.20 and then right-closing it once, again requiring the conditions and right-closing it once more, repeating these steps until the result of this procedure reaches a fix point.

We believe that our way of including the traces initially missed by not right-closing probe sets under disabling, but by identifying them as missed actions is less complex and therefore very likely to be less expensive. As long as there is no implementation with which we can perform experiments on these issues, we cannot prove this believe.

Subobject Transformation Systems. In Section 6.6 we have shown one way to instantiate the abstract framework of enabling and disabling relations in the context of graph transformations. The basic idea was to treat graph elements as entities and to encode rule applications as entity-based actions.

Another instantiation could be based on the theory of *sub-object transformation systems* [43]. We then rely on the more traditional notions of dependency relations between rule applications, namely sequential and parallel independence. Subobject transformation systems (STSs, for short) allow direct analysis of all possible symmetric and asymmetric dependency relations between graph productions without requiring an explicit match. An STS consists of direct derivations and a special graph T (called the *type graph*) of which all derived graphs are a sub-graph, specified in terms of typing morphisms. In particular, for all graph elements created by any of the derivations, the type graph contains a distinguished element, thereby enabling the construction of all dependency relations, without the need for explicit matchings.

To use this theory in the context of dynamic partial order reduction, the basic idea would then be to translate every derivation for which the missed action analysis must be performed, to an STS. The over-approximation of the derivation is then represented by the *type graph* of that STS. After deducing all dependency relations, we can then determine which potentially missed action is actually missed.

7 Conclusions

7.1 Summary

Applying graphs and graph transformations in the various phases of the software engineering process is an emergent field of research. The software engineering process typically consists of phases such as *requirement engineering*, *system design* from various points of perspective (e.g., structure and behaviour), *system implementation*, various levels of system *testing* and *verification*, and finally the system's *deployment* and *maintenance* [154].In this dissertation we have focuses on the phases of software specification and verification. In this dissertation we have applied the graph transformation technique in two of these phases, namely for the specification (or implementation) and verification of systems. In particular we have focussed on *object-oriented* systems and shown that the graph transformation framework provided natural ways to deal with their highly dynamic nature.

In Chapters 4 through 6 we indicated how graph transformation can be helpful in various phases and how (variations of) often used techniques translate to the graph transformation framework. In Chapter 2 we provided some background information on different algebraic approaches to graph transformation and a list of some related tools in that field. In Chapter 3 we introduced a uniform framework for handling so-called *attributed graphs*, a prerequisite for targeting the object-oriented paradigm.

In Chapter 4 we have developed a first step towards software model checking based on the graph transformation framework. We have defined an artificial object-oriented programming language called Taal. The concrete syntax and

static semantics of Taal have been defined using the traditional approach. That is to say, the concrete syntax is defined as an EBNF grammar and the parsing and static analysis processes are performed by a compiler. Typically [3], a compiler produces low-level machine code (that can directly be executed on the machine), as is the case for example for C [116], or some higher-level form of byte code (that is then interpreted by a platform-specific virtual machine), for example in the case of Java [93]. The Taal compiler, in contrast, produces a *graph model* of the abstract syntax of the compiled Taal program. We have developed two graph production systems that define the control flow and operational semantics of Taal. By applying those graph production systems to the abstract syntax graph of some arbitrary Taal program, the execution of that Taal program is simulated. This is done using the graph transformation tool we have developed, namely Groove [157]. That is, every rule application simulates the effect of a (set of) computational step(s) typically performed at machine level or by the virtual machine. The graph transition system resulting from applying taal-sem to some Program Graph then includes all possible behaviours of the original program.

The next step towards software model checking using graph transformations has been to investigate to what extent existing model checking algorithms can be applied to systems specified as graph production systems. In Chapter 5 we proposed an algorithm for model checking arbitrary graph production systems for which the required properties are expressed as formulae in the linear time temporal logic *LTL*. Since, in general, termination of graph productions systems is undecidable, we combined an existing *on-the-fly* LTL model checking algorithm with ideas from *bounded* model checking. This algorithm verifies the state space in an iterative fashion using so-called *boundary conditions*. Proper boundary conditions ensure that successive boundaries give rise to ever-larger, though finite sub-state spaces. Each finite sub-state space is then model checked on-the-fly, i.e., the state space generation process and the model checking process are performed in an interleaved fashion. We have implemented the algorithm in the Groove Tool Set and performed some experiments (1) to gain some insight in the performance of these algorithms when applied in the context of graph transformations and (2) to identify the limitations of our approach.

In Chapter 6 we have addressed one of the main drawbacks of the model checking approach, namely the *state explosion problem*. Traditional partial order reduction algorithms such as, e.g., the persistent/stubborn set approach [91, 185], are based on assumptions that do not hold in the graph transformation framework. Therefore, we have developed a new dynamic partial order reduction technique. For every state, a subset of enabled actions (the so-called *probe set*)

will be probed, i.e., explored. If later on it turns out that some probe set causes some system execution to be ruled out, this is taken care of by ensuring that the algorithm will also examine that execution. We have introduced the probe set algorithm in the context of *entity-based transition systems* and showed how graph transition systems can be encoded as such. We have proven that the algorithm produces a *trace automaton* of the original transition system which means that every execution path of the system is represented in the reduced state space, either explicitly or as a so-called (weak) *prefix* of some other system execution. As yet, we have not implemented the algorithm. It is planned for the near future to incorporate the probe set algorithm in the GROOVE Tool Set.

For graphs to be of practical use, especially in an object-oriented setting, there is a need to specify data in graphs. This can be achieved by including nodes that stand for data values such as, e.g., the integers and Boolean values. Such graphs are often called *attributed graphs* since ordinary nodes (and in some approaches also ordinary edges) can have attributes attached. Transformations of attributed graphs must then be capable of changing the regular graph structure, but also of changing the values of attributes. Such transformations are therefore called attributed graph transformations. Several approaches have been developed for specifying and transforming attributed graphs. In Chapter 3, we have developed a novel approach by introducing a *uniform* and transparent framework for the specification of attributed graphs and their transformation. Our framework makes the technical background of the transformation more transparent and reduces the implementation efforts required for extending the GROOVE Tool Set to support the use of attributes. A technical advantage of our approach is that algebra abstractions can be specified in terms of graph morphisms without the need to restrict to graph morphisms that correspond to actual algebra homomorphisms. More specifically, abstract algebra graphs are allowed to be non-deterministic. We have shown that every concrete transformation has a representative at the abstract level. In fact, sets of concrete transformations are mapped to single abstract transformations, thereby potentially reducing the size of the transition system. In the context of model checking concrete and abstract graph transition systems, this type of reduction ensures that LTL properties are reflected by such abstractions.

7.2 Discussion and Evaluation

One could question why we applied our approach to the artificial object-oriented language TAAL and not to some existing language such as, e.g., JAVA. Our

motivation is partly based on our preference not having to include tricky mechanisms like, for example, exception handling. More importantly, by defining a fresh language one can investigate the efforts required for extending the semantics definition of the language when introducing new language features. Stated differently, by starting from scratch, one can more easily investigate the *modularity* of the approach. For Taal, we introduced parallelism, i.e., the ForkStat language construct, in a later stage, and observed that this extension only required to add ForkStat and ForkedOperImpl specific rules, without the need to change the existing rules. Obviously, it would be naive to conclude that our approach is highly modular on the basis of this one extension.

In this work we have chosen to focus on model checking graph production system against properties expressed in the propositional temporal logic LTL. Therefore, we can only express properties in terms of atomic propositions that hold in individual states. Graph transition systems, however, provide much more information. For example, every *graph transition* consists, among others, of a rule name and a graph morphism; the former indicates of which transformation rule the graph transition represents an application, the latter includes exact information about which graph elements of the source state are mapped to graph elements of the target state. In Section 7.4.3 we discuss this point in further detail.

Another point of discussion could be the choice to extend the graph transformation tool Groove with model checking functionality instead of using existing model checking tools as a back-end. Our motivation is twofold. First, one of our interests is to investigate to what extent existing model checking techniques apply to the graph transformation framework. Second, as mentioned in the previous section graph transition systems provide a lot of information about the behaviour of the system including information about the evolution of individual graph elements along system executions. When translating graph transition systems to models that can be interpreted by existing model checking tools, this type of information will very likely get lost. By extending the Groove Tool Set with model checking functionality, we create opportunities to investigate the full potential of the graph transformation framework with respect to system verification.

Despite the advantages of our uniform attributed graph transformation framework, there is much space for improvement, in particular when considering the current concrete syntax for specifying attributed graph transformations. In the case the update of an attribute value requires nested algebraic operations, we prefer to specify this as a usual algebraic expression instead of the corresponding graph structure. The same holds for visualizing attributed graph transforma-

tion rules. If nested algebraic operations are included, it can be difficult to quickly recognize the parts of the transformation rule specifying the regular graph changes and the attribute changes. One straightforward way to solve the problems mentioned above is to introduce a clear separation between the concrete and abstract syntax of graph transformation rules, and using a different way to visualize the attribute changes.

In the context of model checking graph production systems, there are some topics we have not addressed in this work. An important one is what to do with counter-examples produced by the verification procedure. On the concrete level, counter-examples should give the system engineer some insight in the incorrectness of the system. In the case of *LTL* model checking, a counter-example is an actual execution of the system that does not satisfy some given property. Providing tool support to step through the counter-example might be helpful in understanding faulty system execution. When model checking is performed on some abstraction of the actual system, a counter-example might be a so-called *false negative*, caused by a too coarse abstraction. Such counter-examples might be left to the system engineer to be interpreted. In this case, the original abstraction must then be manually refined. There are also approaches in which false negatives are used as input to refine the original abstraction in such a way that they will not reappear in later stages of the verification procedure. Such approaches are often referred to as *counter-example guided abstraction-refinement* (or CEGAR, for short). König and Kozioura [119] have applied this technique in a context where graph transformation systems are abstracted to so-called *Petri graphs*. Their way of refining the abstraction involves the colouring of nodes and disallowing an abstraction to merge nodes if their corresponding nodes in the Petri graph have the same colour. This approach has been implemented and experimented with in the tool Augur [120].

7.3 Limitations of the Graph Transformation Framework

There are some serious issues that require further research for the graph transformation framework to benefit its full potential in practice. First of all, one should be aware of the *complexity* of the graph matching problem. Another major issues is the poor *scalability* of the approach. Finally, the *visualization* (e.g., layouting) of graphs can be problematic.

7.3.1 Graph Matching Complexity

A first limitation of the graph transformation framework discussed here is the complexity of the graph matching problem. For a graph transformation rule to be applied to some host graph, we need to identify a subgraph of the host graph such that there exists a total graph morphism from the left-hand-side of the rule to that subgraph. This problem is often called the *subgraph matching problem*, which is known to be *NP-complete*. Under specific circumstances, however, sub-graph isomorphisms might be identified very efficiently. For instance Dodds and Plump [55] have shown that when certain conditions are met, graph transformations can be performed in *constant time*. Those conditions typically restrict the structure of graphs to be matched. Dodds and Plump, for example, restrict to so-called *rooted graphs* of which the nodes, in addition, have a bounded outdegree. If such graphs are also deterministic, complexity results can be improved even further. When graphs represent states of object-oriented programs, these types of restrictions are very likely to be satisfied.

Within graph transformation tools, computing rule applications has been implemented using different strategies. For instance, in Agg [182], rule applications are computed by encoding the problem as a *constraint satisfaction problem* [168]. Another approach is to construct so-called *model-specific search plans* (see, e.g., [193, 88, 105]) which fix the order in which graph matching operations are performed for the different graph elements. Obviously, such approaches aim at finding the optimal such an ordering, thereby improving the time-performance of graph transformation tools with respect to computing rule applications. This approach has also been implemented in the Groove Tool Set [157].

7.3.2 Scalability

One important issue in scalability is the ability to *structure* graph production systems, possibly in a *modular* way. One can imagine that for many (large) graph production systems, the set of transformation rules can be partitioned into subsets of rules, each subset specifying the behaviour of some specific "subsystem". One approach to structure graph production systems is based on so-called *graph transformation units*, originally proposed by Kreowksi and Kuske [122, 125] (and extended in subsequent papers). A graph transformation unit (or simply transformation unit) U consists, among other things, of a set $\mathcal{R}_U$ of transformation rules local to U and a set $\mathcal{U}_U$ of *imported* graph transformation units. Using this approach, a transformation unit can specify the behaviour of

some "atomic" (sub-)system (in the case $\mathcal{U}_U = \emptyset$) or of some (sub-)system which is partially composed of other sub-systems (in the case $\mathcal{U}_U \neq \emptyset$).

On the level of individual graphs, a similar problem arises when graphs become large and complex (especially for *nonplanar* graphs): identifying interesting parts of such graphs may become a hard and time-consuming operation. The structuring of individual graphs and the transformation of such *structured* or *hierarchical graphs* is a different topic of research that has been studied in a series of papers: see, e.g., [153, 80, 179] for the definition and usage of structured graphs, and [183, 29, 57, 145] for approaches to transform such graphs.

7.3.3 Visualizing Graphs

Scalability is also an issue when visualizing graphs, although for relative small graphs their visualization is often already non-trivial. There exists a large body of research on graph drawing algorithms and conventions; [14] can serve as a good starting point for the interested reader. In [14], Di Battista et al. focus on layouting *static* graphs, i.e., how a single graph can best be visualized. In the context of graph transformations, where graphs change during the transformation process, the layout of one graph can be based on the layout of the graph from which it originates. That is, layouting each of the individual graphs can be done in an incremental (or evolutional) manner. In [108], Jucknath-John et al. extend an existing graph layout algorithm with the concept of *node aging*, thereby ensuring that positions of older nodes become more stable. They also discuss how *layout patterns* can be used as guidance to layout local graph structures.

When dealing with hierarchical graphs (see Section 7.3.2) visualization of individual graphs might be guided by their internal hierarchy. Tools could then provide means to navigate through the hierarchy, by collapsing or expanding hierarchical sub-graphs in an on-demand fashion.

7.4 Future Work

In the following, we will give an outline for further research on the usage of the graph transformation framework in the software engineering process.

7.4.1 Implementing the Probe Set Algorithm

In Chapter 6 we have proposed a new dynamic partial order reduction algorithm and proved its correctness with respect to certain preservation criteria. At the time of writing, no implementation of the algorithm exists. Therefore, we also do not have any insight in the actual performance of the algorithm on systems with differently shaped state spaces. It is planned for the near future to extend the Groove Tool Set with this algorithm and perform experiments of the actual reduction it produces for various types of graph production systems.

To do:

- implement the probe set based dynamic partial order reduction in the Groove Tool Set;
- develop and implement heuristics for non-trivial probe set construction;
- perform experiments on various graph production systems.

7.4.2 Graph-Based Language Engineering

In Chapter 4 we have shown how graph transformations can be used to define the dynamic semantics of an artificial object-oriented programming language called Taal. There we have focussed on specifying the dynamic semantics (here, the control flow and operational semantics) of Taal in terms of graph transformations. In cases where also the concrete syntax of the language is graphical, we believe that using graph transformation for the remaining phases can also be helpful.

Language engineers often make a distinction between *general-purpose languages* and *domain-specific languages* (although the boundary is sometimes vague). Whereas the former can be used to implement solutions involving many different contexts, obviously, the latter are meant to be in very specific domains such as, e.g., the field of *business process modelling*. Especially in the latter case, many such languages are a lot like each other and often share a common core. In order to get a good insight in their actual differences, a graphical description of their semantics (provided that those semantics are defined in the same formalism) will very likely be more helpful than a textual description using natural language.

To do:

- develop a generic graph-based framework to support the overall software engineering process. This framework should allow to specify software ar-

tifacts in graph-based formalisms and define their (static and dynamic) semantics in terms of graph transformations.

7.4.3 Extensions to Modal and Quantified Logics

When verifying object-oriented programs, typical properties one might be interested in involve the *evolution* of individual objects *along* executions of the system. In the graph transformation framework, transitions are based on graph morphisms mapping graph elements of the source graph to graph elements of the target graph. Those morphisms thus provide exactly the required information to determine which graph elements are created, preserved, or deleted. In Chapter 5 we addressed that our choice for model checking properties expressed as *LTL* formulae puts a strong restriction on the kind of properties we can specify. In particular, *LTL* does not include means to express how certain states could be reached in terms of system actions (or rule applications in the context of graph transformation). Extending the approach to support the expression of such kinds of properties requires to consider more expressive logics such as *modal* and *quantified* logics.

Some example modal logics are the Hennessy-Milner Logic [102] and the *modal μ-calculus* [121]. These logics include modal operators like $\langle a \rangle \phi$ and its dual $[a]\phi$. These operators allow to express that after performing an action a (or in the graph transformation framework: by applying the rule named a) some state is reached in which ϕ holds (for $\langle a \rangle$) or all reached states reached by performing an action a satisfy ϕ.

In the literature some relevant quantified logics have been studied. Baldan et al. [8] proposed a more general variant of the monadic second-order logic $\mu\mathcal{L}2$ [10]. They introduced an approach to verify graph transition systems based on finite approximations of *unfoldings* of such systems. Although their logic includes operators to track individual graph elements along executions, it does not provide primitives to reason about *allocation* and *deallocation* of graph elements. An approach in which such primitives are included in the logic has been proposed by Distefano et al. [54, 53]. Their logic $\mathcal{A}\ell\ell$TL (for Allocational Temporal Logic) is an extension of LTL in which one can existentially quantify over variables that are mapped to individual graph elements. By allowing such mappings to be partial, variables that are not mapped represent deallocated graph elements. In this approach, however, individual elements in a state are treated in a uniform way, or, stated differently, state data is *unstructured*. Rensink has extended this approach by enriching states with *abstract algebras*, thereby providing means to structure state data [159]. He has shown that combining such

finite *algebra automata* with formulae in *QCTL* (for Quantified *CTL*, in which the evolution of individual state variables can be expressed) can be transformed to an equivalent propositional *CTL* formula over an ordinary Kripke structure.

To do:

- (further) develop (graph-based) evolution logics and investigate to what extent existing model checking algorithms transfer to the graph transformation framework.

7.4.4 Graph-Based State Space Reduction Techniques

In this dissertation we have proposed some techniques to combat the state explosion problem when model checking graph production systems. Recently, alternative approaches have been developed. We will shortly mention a few of them. In the future we could investigate how our approach to model checking graph production systems could benefit from such approaches.

Abstract Graph Transformation. In [163], Rensink and Distefano propose a framework for *abstract graph transformation*. Instead of performing the actual transformation in the usual way, i.e., on concrete graphs, they propose to construct an abstract graph transition system by performing transformation in a three-step procedure. Transformation rules are then applied to so-called *shapes* being canonical representations of the actual concrete graphs. The first step of their transformation process consists of *materializing* a matching. That is, they identify a sub-shape where the rule applies and construct a graph shape in which the sub-shape involved in the matching is concretized. The second step consist of *transforming* the concretized structure. The third step is then to *normalize* the obtained result as to construct the target shape. Rensink and Distefano have illustrated how abstract graph transformations potentially reduce infinite concrete graph transition systems to finite abstract graph transition systems.

In [16], Bauer et al. have proposed a modal-logic based graph abstraction that could be used in the framework of abstract graph transformation just mentioned. Their abstraction maps concrete graphs to abstract graphs by identifying nodes with equivalent *local* structures, therefore called a *neighbourhood abstraction*. Their abstraction is parameterized with the radius of the neighbourhood that determines the equivalence relation between nodes; this allows for automatic abstraction refinement. In addition, they introduced a modal logic for which all properties are both preserved *and* reflected by their abstraction.

As mentioned in Section 7.1, König and Kozioura [120] have proposed a different type of abstraction framework for the verification of graph transformation systems, namely by approximating graph transformation systems by Petri nets via an *unfolding* construction.

Transactional Graph Transformation. In the context of graph transformations, there is often a need for structuring the rules of a graph production system or specifying the order in which they may be applied. In *flat* graph production systems, the order in which rules must be applicable is often directed by temporary graph elements that are then processed (e.g., propagated or deleted) by successive rule applications. Baldan et al. [6, 7] have introduced the notion of *graph transactions* in which they distinguish between *stable* and *unstable* graphs; the latter being graphs containing such temporary graph elements. A sequence of rule applications transforming one stable graph into another stable graph via unstable graphs only, is then considered a graph transaction. In fact, a graph transaction (in [7] called an *abstract transaction*) represents all possible orderings of the individual transformation steps of such a sequence that all result in the same (or isomorphic) graph. In many cases we are only interested in whether all reachable stable graphs satisfy specific properties. Therefore it would be sufficient to construct a graph transition system in which unstable graphs are hidden and only stable graphs and graph transactions are considered as first-class entities. For many cases this would result in a useful abstraction and a significant reduction of the state space when compared to the state space that also includes all unstable graphs and their intermediate transformation steps.

Nested Quantification in Graph Transformations. An approach that is closely related to transactional graph transformations is to allow *nested quantification* to an arbitrary depth in graph transformation rules (recall Section 4.10.3). Whereas usually graph transformation rules with negative application conditions allow a single level of nested quantification, the expressive power of single graph transformation rules increases when rules can be constructed using nested levels of existential and universal quantification on an alternating basis. This approach provides further control on the structure of the state space since an application of such a single rule actually represents a sequence of sub-rule applications, without those sub-rule applications being interrupted by (dependent or independent) applications of some other (or the same) rule.

To do:

- develop and implement additional graph abstractions;

- develop further methods to intuitively specify and visualize transactional graph transformations and nested quantified graph transformation rules;
- investigate to what extent transactional graph transformations and nested quantification in graph transformations can effectively be used in practice.

7.4.5 Directed Model Checking

A recent trend in model checking is to search for the state space for counter-examples directed on the basis of heuristics. This approach is therefore known as *directed model checking*; see, e.g., [59, 60]. This technique combines classic model checking techniques (aiming at finding *erroneous* states or system executions) with heuristic based search techniques, which have a long tradition in the area of artificial intelligence, in particular the sub-field of *action planning* (where the aim is to identify *goal* scenarios).

Some ideas have been posed on combining directed model checking with graph transformation techniques; see, e.g., [61]. Further research is needed to investigate the benefits of this combination. One direction would be to investigate whether heuristics can be specified in terms of a desired graph structure, or *goal graph*. That would require to develop methods to determine some sort of *distance* between an arbitrary graph and the goal graph. Alternatively, heuristics could be based on some *level* to which an arbitrary graph is *equivalent* to such a goal graph. The level of equivalence could, for instance, be defined in terms of the maximal number of graph elements that can successfully be matched to the goal graph, assuming that the order in which the different graph elements are matched is fixed. In the context of graph transformation, such an equivalence level could then determine which branch of the state space first to explore to reach the goal state in (hopefully) a minimal number of steps.

To do:

- develop heuristics to determine the distance between any graph and a goal graph;
- implement directed state space exploration strategies in graph transformation tools such as, e.g., Groove.

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Journal on Software and System Modeling*, 5(3):261–288, 2006.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, first edition, 1986.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[5] D. Balasubramanian, A. Narayanan, C. vanBuskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. In Zündorf and Varró [200].

[6] P. Baldan, A. Corradini, F. L. Dotti, L. Foss, F. Gadducci, and L. Ribeiro. Towards a notion of transaction in graph rewriting. In Roberto Bruni [165], pages 39–50.

[7] P. Baldan, A. Corradini, L. Foss, and F. Gadducci. Graph transactions as processes. In Corradini et al. [41], pages 199–214.

[8] P. Baldan, A. Corradini, and F. Gaducci. Specifying and verifying UML Activity Diagrams via graph transformation. In C. Priami and P. Quaglia, editors, *Proceedings of the IST/FET International Workshop on Global Computing (GC 2004)*, volume 3267 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2004.

[9] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K. G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 381–395. Springer, 2001.

[10] P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In Corradini et al. [40], pages 14–30.

[11] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 1–3, 2002.

[12] A. Balogh, A. Németh, A. Schmidt, I. Rath, D. Vágó, D. Varró, and A. Pataricza. The VIATRA2 model transformation framework. 2005. Presented at ECMDA 2005 – Tools Track.

[13] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[14] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[15] M. Bauderon. A uniform approach to graph rewriting: The pullback approach. In M. Nagl, editor, *Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science (WG'95)*, volume 1017 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 1995.

[16] J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink. A modal-logic based graph abstraction. In R. Heckel and G. Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*. Springer, 2008.

[17] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[18] M. R. Berthold, I. Fischer, and M. Koch. Attributed graph transformation with partial attribution. In H. Ehrig and G. Taentzer, editors, *Proceedings Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, pages 171–178. Technical University of Berlin, 2000.

[19] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5–6):505–525, 2007.

[20] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[21] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In R. Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[22] I. Boneva, F. Hermann, H. Kastenberg, and A. Rensink. Simulating multigraph transformations using simple graphs. In Ehrig and Giese [73].

[23] L. Brim, I. Černá, and M. Nečesal. Randomization helps in LTL model checking. In *Proceedings of the Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modelling and Verification*, volume 2165 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2001.

[24] K. B. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for toople: A statically-typed object-oriented programming lnaguage. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 603–626. Springer, 1994.

[25] K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25:225–290, 2003.

[26] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[27] J. R. Büchi. Weak second order logic and finite automata. *Z. Math. Logic, Grundlag. Math.*, 5:66–62, 1960.

[28] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[29] G. Busatto, G. Engels, K. Mehner, and A. Wagner. A framework for adding packages to graph transformation systems. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2000.

[30] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *Transactions on Software Engineering*, 30(6):388–402, 2004.

[31] Y. Choueka. Theories of automata on omega-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[32] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.

[33] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[34] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction-refinement. In Emerson and Sistla [77], pages 154–169.

[35] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[36] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[37] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[38] E. M. Clarke, S. Jha, R. Enders, and T. Filkhorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1–2):77–104, 1996.

[39] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating Java code to graph transformation systems. In Ehrig et al. [68], pages 383–398.

[40] A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Proceedings of the 1st International Conference on Graph Transformations (ICGT'02)*, volume 2505 of *Lecture Notes in Computer Science*. Springer, 2002.

[41] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors. *Proceedings of the 3rd International Conference on Graph Transformations (ICGT'06)*, volume 4178 of *Lecture Notes in Computer Science*. Springer, 2006.

[42] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-pushout rewriting. In Corradini et al. [41], pages 30–45.

[43] A. Corradini, F. Hermann, and P. Sobociński. Subobject transformation systems. *Applied Categorical Structures*, 6(3):389–419, 2007.

[44] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3–4):241–265, 1996.

[45] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In Rozenberg [167], pages 163–246.

[46] C. Courcoubetis, editor. *Proc. of the 5th Conf. on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*. Springer, 1993.

[47] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2–3):275–288, 1992.

[48] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 269–282, 1979.

[49] F. S. de Boer and C. Pierik. How to cook a complete Hoare logic for your pet OO language. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proceedings of the Second International Symposium on Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *Lecture Notes in Computer Science*, pages 111–133. Springer, 2003.

[50] J. de Lara and H. Vangheluwe. Using AToM^3 as a Meta-CASE environment. In *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS 2002)*, pages 642–649, 2002.

[51] C. Demartini, R. Iosif, and R. Sisto. dspin: A dynamic extension of spin. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops (SPIN'99)*, volume 1680 of *Lecture Notes in Computer Science*, pages 261 – 276. Springer, 1999.

[52] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific Publishing Co., Inc., 1995.

[53] D. Distefano. *On Model Checking the Dynamics of Object-Based Software*. PhD thesis, University of Twente, 2003.

[54] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer, 2002.

[55] M. Dodds and D. Plump. Graph transformation in constant time. In Corradini et al. [41], pages 367–382.

[56] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, 1982.

[57] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64(2):249–283, 2002.

[58] D. A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.

[59] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Software Tools for Technology Transfer*, 5(2–3):247–267, 2004.

[60] S. Edelkamp, S. Leue, and W. Visser, editors. *Directed Model Checking*, volume 06172 of *Dagstuhl Seminar Proceedings*, 2006.

[61] S. Edelkamp and A. Rensink. Graph transformation and AI planning. In *Proceedings of Knowledge Engineering Competition (ICKEPS)*, 2007.

[62] H. Ehrig. Attributed graphs and typing: Relationship between different representations. Technical report, Technische Universität Berlin, 2003.

[63] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approached to Software Engineering (FASE'05)*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[64] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Formal integration of inheritance with typed attributed graph transformation for efficient vl definition and model manipulation. In *Proceedings of the 2005 Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 71–78. IEEE, 2005.

[65] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. *Fundamenta Informaticae*, 74(1):31–61, 2006.

[66] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[67] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume Volume 2: Applications, Languages and Tools. World Scientific, 1999.

[68] H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *Proceedings of the 2nd International Conference on Graph Transformations (ICGT'04)*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[69] 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 Rozenberg [167], pages 247–312.

[70] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[71] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[72] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Proceedings of the 14th Annual Symposium on Foundations of Computer Science (FOCS 1973)*, pages 167–180. IEEE Computer Society, 1973.

[73] K. Ehrig and H. Giese, editors. *Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 6 of *Electronic Communications of the EASST*, 2007.

[74] E. A. Emerson. *Branching Time Temporal Logics and the Design of Correct Concurrent Programs*. PhD thesis, Harvard University, 1981.

[75] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[76] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1–2):105–131, 1996.

[77] E. A. Emerson and A. P. Sistla, editors. *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[78] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modelling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceeding of the Third International Conference on the Unified Modeling Language (UML'00)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[79] G. Engels, A. Kleppe, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. From UML Activities to TAAL – toward behaviour-preserving model transformations. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA 2008)*. Springer, 2008.

[80] G. Engels and A. Schürr. Encapsulated hierachical graphs, graph types, and meta types. In A. Corradini and U. Montanari, editors, *Proceedings of*

the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA'95), volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[81] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of UML Activities using dynamic meta modeling. In M. M. Bonsangue and E. B. Johnsen, editors, *Proceedings of the 9th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007.

[82] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2000.

[83] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thomsen Publishing Inc., 2nd edition, 1997.

[84] M. Fitting. On quantified modal logic. *Fundamenta Informaticae*, 39(1):5–121, 1999.

[85] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 110–121. ACM Press, 2005.

[86] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[87] P. Gastin and D. Oddoux. Fast ltl to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[88] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast spo-based graph rewriting tool. In Corradini et al. [41], pages 383–397.

[89] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verfication of linear temporal logic. In *Proceedings of the 13th Symposium on Protocol Specification, Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[90] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the 2nd Workshop on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1991.

[91] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[92] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[93] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[94] R. Grosu and S. A. Smolka. Monte Carlo model checking. [97], pages 271–286.

[95] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In D. Bosnacki and S. Edelkamp, editors, *Proc. of the 14th Int. Workshop on Software Model Checking (SPIN 2007)*, volume 5495 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[97] N. Halbwachs and L. D. Zuck, editors. *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*. Springer, 2005.

[98] J.-H. Hausmann. *Dynamic Meta-Modelling*. PhD thesis, University of Paderborn, 2006.

[99] R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 9(1):83–110, 2001.

[100] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In Corradini et al. [40], pages 161–176.

[101] R. Heckel and A. Wagner. Ensuring consistency of conditional graph rewriting – a constructive approach. In A. Corradini and U. Montanari, editors, *Proceedings of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 118–126. Elsevier, 1995.

[102] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming (ICALP'80)*, volume 85 of *Lecture Notes in Computer Science*. Springer, 1980.

[103] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[104] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[105] Á. Horváth, G. Varró, and D. Varró. Generic search plans for matching advanced graph patterns. In Ehrig and Giese [73].

[106] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[107] R. Janicki and M. Koutny. Structure of concurrency. *Theoretical Computer Science*, 112:5–52, 1993.

[108] S. Jucknath-John, D. Graf, and G. Taentzer. Evolutionary layout of graph transformation sequences. In Zündorf and Varró [200].

[109] W. Kahl. A fibred approach to rewriting – how the duality between adding and deleting cooperates with the difference between matching and rewriting. Technical Report Technical Report 9702, Fakultät für Informatik, Universität der Bundeswehr München, 1997.

[110] H. Kastenberg. Towards attributed graphs in GROOVE (work in progress). In R. Heckel, B. König, and A. Rensink, editors, *Proceedings of the Workshop on Graph Transformation for Verification and Concurrency (GT-VC'05)*, number TR-CTIT-05-34 in CTIT Technical Report, pages 91–98, 2005.

[111] H. Kastenberg, A. 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*

(FMOODS'06), volume 4037 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2006.

[112] H. Kastenberg, A. Kleppe, and A. Rensink. Engineering object-oriented semantics using graph transformations. Technical Report 06-12, University of Twente, 2006.

[113] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. In A. Valmari, editor, *Proceedings of the 13th International Workshop on Model Checking Software (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006.

[114] H. Kastenberg and A. Rensink. Dynamic partial order reduction using probe sets. In F. van Breugel and M. Chechik, editors, *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR 2008)*, volume X of *Lecture Notes in Computer Science*. Springer, 2008. To appear.

[115] H. Kastenberg and A. Rensink. On-the-fly bounded model checking of rule-based systems, 2008. Submitted.

[116] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, second edition edition, 1988.

[117] A. Kleppe and A. Rensink. On a graph-based semantics for UML Class and Object diagrams. In *Proceedings of the 7th International Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT'08)*, Electronic Communications of the EASST, 2008. To appear.

[118] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained; The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[119] B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, volume 3920 of *Lecture Notes in Computer Science*, pages 197–211. Springer, 2006.

[120] B. König and V. Kozioura. Augur 2 – a new version of a tool for the analysis of graph transformation systems. In Roberto Bruni [165], pages 201–210.

[121] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[122] H. J. Kreowski and S. Kuske. Graph transformation units and modules. In Ehrig et al. [67], pages 607–638.

[123] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[124] J.-H. Kuperus. Nested quantification in graph transformation rules. Master's thesis, University of Twente, 2007.

[125] S. Kuske. *Transformation Units – A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.

[126] S. Kuske. A formal semantics of UML State Machines based on structured graph transformations. In M. Gogolla and C. Kobryn, editors, *Proceedings of the 4th International Conference on the Unified Modeling Language (UML 2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2001.

[127] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML Class, Object and State Diagrams based on graph transformations. In M. Butler, L. Petre, and K. Sere, editors, *Proceedings of the Third International Conference on Integrated Formal Methods (IFM'02)*, volume 1939 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2000.

[128] S. Lack and P. Sobociński. Adhesive categories. In I. Walukiewicz, editor, *Proceedings of the 7th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'04)*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2004.

[129] S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *RAIRO - Theoretical Informatics and Applications*, 39(3):522–546, 2005.

[130] L. Lambers. A new version of GTXL: An exchange format for graph transformation systems. In T. Mens, A. Schürr, and G. Taentzer, editors, *Proceedings of the International Workshop on Graph Based Tools (GRaBaTs'04)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 51–63. Elsevier, 2004.

[131] L. Lamport. "sometime" is sometimes "not never" – on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–185. ACM, 1980.

[132] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, 1992.

[133] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL 1985)*, pages 97–107. ACM, 1985.

[134] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[135] M. Löwe, M. Korf, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley & Sons Ltd., 1993.

[136] A. W. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[137] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[138] S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled, Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[139] Microsoft Corporation. The .net framework, 2007.

[140] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[141] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[142] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2003)*, pages 742–745. ACM Press, 2003.

[143] Object Management Group (OMG). MDA Guide, June 2003. Version 1.0.1.

[144] Object Management Group (OMG). OMG Unified Modelling Language specification, November 2007. Version 2.1.2.

[145] W. Palacz. Algebraic hierarchical graph transformation. *Journal of Computer and System Sciences*, 68:497–520, 2004.

[146] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings of the 5th GI-Conference Karlsruhe on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[147] T. Parr. ANTLR: Another tool for language recognition, 2007. Available from www.antlr.org.

[148] D. Peled. All from one, one for all: on model checking using representatives. In Courcoubetis [46], pages 409–423.

[149] J. L. Pfaltz, M. Nagl, and B. Böhlen, editors. *Proceedings of the Second International Workshop on the Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science*. Springer, 2004.

[150] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[151] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[152] D. Plump and S. Steinert. Towards graph programs for graph algorithms. In Ehrig et al. [68], pages 128–143.

[153] T. W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proceedings of the International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 389–400. Springer, 1979.

[154] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 6th edition, 2005.

[155] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[156] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[157] A. Rensink. The GROOVE Simulator: A tool for state space generation. In Pfaltz et al. [149], pages 479–485.

[158] A. Rensink. Isomorphism checking in GROOVE. In Zündorf and Varró [200].

[159] A. Rensink. Model checking quantified computation tree logic. In C. Baier and H. Hermanns, editors, *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR 2006)*, volume 4137 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2006.

[160] A. Rensink. Nested quantification in graph transformation rules. In Corradini et al. [41], pages 1–13.

[161] A. Rensink. Isomorphism checking in GROOVE. In *Graph-Based Tools (GraBaTs'06)*, volume 1 of *Electronic Communications of the EASST*, 2007.

[162] A. Rensink. Explicit state model checking for graph grammars. In P. Degano, R. D. Nicola, and J. Meseguer, editors, *Festschrift Ugo Montanari*, Lecture Notes in Computer Science. Springer, 2008. To be published.

[163] A. Rensink and D. Distefano. Abstract graph transformation. In S. Mukhopadhyay, A. Roychoudhury, and Z. Yang, editors, *Proceedings of the Third International Workshop on Software Verification and Validation (SVV 2005)*, volume 157 of *Electronic Notes in Theoretical Computer Science*, pages 39–59. Elsevier, 2006.

[164] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In Ehrig et al. [68], pages 226–241.

[165] D. V. Roberto Bruni, editor. *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, volume 211 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2008.

[166] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, 2004.

[167] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[168] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 6th International Workshop on Theory and Applications of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2000.

[169] Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In P. S. J. Whittle and G. Booch, editors, *Proceedings of the 6th International Conference on the Unified Modeling Language (UML 2003)*, volume 2863 of *Lecture Notes in Computer Science*, pages 92–95. Springer, 2003.

[170] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [67], pages 1487–500.

[171] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In Halbwachs and Zuck [97], pages 174–190.

[172] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

[173] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen. *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993.

[174] R. Smelik. Specification and construction of control flow semantics: A generic approach using graph transformations. Master's thesis, University of Twente, 2006.

[175] R. Smelik, A. Rensink, and H. Kastenberg. Specification and construction of control flow semantics. In *Proceedings of the IEEE Symposium on Visual Language and Human-Centric Computing (VL/HCC 2006)*, pages 65–72. IEEE Computer Society, 2006.

[176] N. Sombekke. Graph-based semantics of the .Net Intermediate Language. Master's thesis, University of Twente, 2007.

[177] F. Somenzi and R. Bloem. Efficient Büchi automata from ltl formulae. In Emerson and Sistla [77], pages 248–263.

[178] J. Swaine. Heathrow terminal 5 still losing 1,000 bags a day. *The Daily Telegraph*, July 10th, 2008.

[179] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, Technische Universität Berlin, 1996.

[180] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In Pfaltz et al. [149], pages 446–453.

[181] G. Taentzer and G. T. Carughi. A graph-based approach to transform xml documents. In L. Baresi and R. Heckel, editors, *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*. Springer, 2006.

[182] G. Taentzer, C. Ermel, and M. Rudolf. The AGG approach: Language and tool environment. In Ehrig et al. [67], pages 163–246.

[183] G. Taentzer, M. Koch, I. Fischer, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. In Ehrig et al. [70], pages 269–340.

[184] H. Tauriainen. *Automata and Linear Temporal Logic: Translation with Transition-based Acceptance*. PhD thesis, Helsinki University of Technology, 2006.

[185] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1991.

[186] A. Valmari. On-the-fly verification with stubborn sets. In Courcoubetis [46], pages 397–408.

[187] M. Y. Vardi. Sometimes and not never re-revisited: On branching versus linear time. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 1998)*, volume 1466 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1998.

[188] M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference*

on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.

[189] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–344, 1986.

[190] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[191] D. Varró. A formal semantics of UML Statecharts by model transition systems. In Corradini et al. [40], pages 378–392.

[192] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and System Modeling*, 3(2):85–113, 2004.

[193] G. Varró, D. Varró, and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer, editors, *Proceedings of the International Workshop on Graphs and Model Transformation (GraMoT'05)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 191–205. Elsevier, 2006.

[194] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[195] VMTS. Vmts website. http://vmts.aut.bme.hu/, 2008.

[196] J. Warmer and A. G. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

[197] P. Wegner. Dimensions of object-based language design. In *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1987)*, pages 168–182, 1987.

[198] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[199] A. Winter. Gxl: Graph exchange language. In *Dagstuhl Seminar Interoperability of Reengineering Tools*, 2001.

[200] A. Zündorf and D. Varró, editors. *Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs'06)*, volume 1 of *Electronic Communications of the EASST*, 2006.

A Basic Category Theory

A.1 Categories, Functors, and Natural Transformations

In this appendix we introduce some of the basic concepts of category theory that are used in the work described in this dissertation. It is therefore far from complete. For further details on category theory, the intereseted reader is referred to, e.g., [13].

Categories are mathematical structures in which there is a distinction between the *objects* of the category and the relations between the objects, so called *arrows* or *morphisms*. Categories will be denoted $\mathbf{C}, \mathbf{D}, \ldots$ For a category $\mathbf{C}$, its objects and arrows will be denoted $Obj_{\mathbf{C}}$ and $Arr_{\mathbf{C}}$, respectively. If, for some category $\mathbf{C}$, f is an arrow from an object $C \in Obj_{\mathbf{C}}$ to an object $D \in Obj_{\mathbf{C}}$, this will be denoted $f\colon C \to D$. Categories contain one special morphism for every object C: the *identity* morphism, denoted id_C, being an arrow from the object to itself. For every arrow $f\colon C \to D$ it must hold that $f \circ id_C = id_D \circ f = f$. For a category $\mathbf{C}$, the set of all identity morphisms is denoted ID_C. Furthermore, in any category the composition of those arrows is again an arrow in that category. The composition of two arrows f and g is often denoted with $f \circ g$. Arrow composition must be *associative*, i.e., $(f \circ g) \circ h = f \circ (g \circ h)$, for all arrows f, g, h.

Usually, a number of special types of arrows (or morphisms) are distinguished. Here we recall *isomorphims* and *monomorphisms*.

Definition A.1 (isomorphism)**.** *An arrow $f\colon C \to D$ is an* isomorphism *if there exists another arrow $g\colon D \to C$ such that $g \circ f = id_C$ and $f \circ g = id_D$.*

Definition A.2 (monomorphism)**.** *An arrow $f\colon B \to C$ is a* monomorphism *(or* mono*, for short) if for all arrows $g, h\colon A \to B$, $f \circ g = f \circ h$ implies $g = h$.*

We write $f\colon A \xrightarrow{\sim} B$ [$f\colon A \hookrightarrow B$] to denoted that f is an isomorphism [monomorphism].

Example A.3. *In mathematics, the most basic category is* **Set***, having all sets as objects and all functions as morphisms. A more restricted category is the one having sets as objects and set-inclusions as morphisms.*

Category theorists and mathematicians (and sometimes even computer scientists) are continually searching for (equivalence) relations between categories of their interest. The basic building blocks in defining such relations are *functors*. Intuitively, a functor between two categories, say $\mathbf{C}$ and $\mathbf{D}$, maps every object in $Obj_{\mathbf{C}}$ to an object in $Obj_{\mathbf{D}}$, and every arrow in $Arr_{\mathbf{C}}$ to an arrow in $Arr_{\mathbf{D}}$, such that specific conditions are satisfied as defined below.

Definition A.4 (functor)**.** *Let $\mathbf{C}$ and $\mathbf{D}$ be two categories. A* functor *F from $\mathbf{C}$ to $\mathbf{D}$, denoted $F\colon \mathbf{C} \to \mathbf{D}$, is a mapping that*

- *assigns to each object $C \in Obj_{\mathbf{C}}$ an object $F(C) \in Obj_{\mathbf{D}}$,*
- *assigns to each $\mathbf{C}$-arrow $f\colon C \to D$ an $\mathbf{D}$-arrow $F(f)\colon F(C) \to F(D)$,*

such that the following properties are satisfied:

- *$F(id_C) = id_{F(C)}$, for all objects $C \in Obj_{\mathbf{C}}$;*
- *$F(g \circ_{\mathbf{C}} f) = F(g) \circ_{\mathbf{D}} F(f)$, for all arrows $f\colon C \to D$ and $g\colon D \to E \in Arr_{\mathbf{C}}$.*

The two conditions state that the identity morphisms and morphism composition, respectively, must be preserved. Whenever given two categories $\mathbf{C}$ and $\mathbf{D}$, and two functors $F\colon \mathbf{C} \to \mathbf{D}$ and $G\colon \mathbf{D} \to \mathbf{C}$, it is interesting to investigate the relation between F and G. In special cases, it may hold that F and G are each others *inverse*, denoted $F = G^{-1}$ and $G = F^{-1}$, which means that each undoes the changes introduced by the other. F and G are then said to establish an *isomorphism*. Other kinds of functors are, for example, *free* or *forgetful* functors, which introduce or forget, respectively, specific structures in the objects of the categories under consideration.

For comparing functors, natural transformations are introduced.

Definition A.5 (natural transformation)**.** *Given two categories $\mathbf{C}$ and $\mathbf{D}$ and functors $F, G\colon \mathbf{C} \to \mathbf{D}$, a* natural transformation *$\alpha : F \Rightarrow G$ is a family of morphisms $\alpha = (\alpha_A)_{A \in Obj_{\mathbf{C}}}$ with $\alpha_A\colon F(A) \to G(A) \in Arr_{\mathbf{D}}$, such that for all morphisms $f\colon A \to B \in Arr_{\mathbf{C}}$, it holds that $\alpha_B \circ F(f) = G(f) \circ \alpha_A$ (see Fig. A.1).*

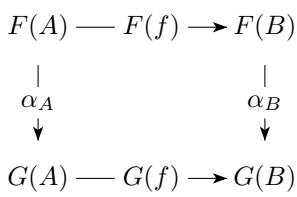


Figure A.1: Requirement for a natural transformation α.

Natural transformations will be used to show that two categories are *equivalent*, as defined next.

Definition A.6 (equivalent categories)**.** *Two categories* $\mathbf{C}$ *and* $\mathbf{D}$ *are called* equivalent*, written* $\mathbf{C} \cong \mathbf{D}$*, if there are functors* $F\colon \mathbf{C} \to \mathbf{D}$ *and* $G\colon \mathbf{D} \to \mathbf{C}$ *and natural transformations* $\alpha : G \circ F \Rightarrow ID_{\mathbf{C}}$ *and* $\beta : F \circ G \Rightarrow ID_{\mathbf{D}}$ *that are component-wise isomorphisms, i.e.,* $\alpha_A : G(F(A)) \xrightarrow{\sim} A$ *and* $\beta_B : F(G(B)) \xrightarrow{\sim} B$ *are isomorphisms for all* $A \in Obj_{\mathbf{C}}$ *and* $B \in Obj_{\mathbf{D}}$*, respectively.*

One of the interesting things about equivalent categories is the following.

Theorem A.7. *[13] Whenever two categories* $\mathbf{C}$ *and* $\mathbf{D}$ *are equivalent, all (co-)limits in* $\mathbf{C}$ *are mapped to (co-)limits in* $\mathbf{D}$*, and vice versa.*

A.2 Pushouts and Pullbacks

The theory on graph transformation heavily relies on the categorical concepts of *pushout*, *pushout complements*, and *pullbacks*. In the following we define these concepts for arbitrary categories.

Definition A.8 (pushout)**.** *A* pushout diagram *for a span* f, g *(see Fig. A.2) is a commuting square of the form (i) below, such that for every commuting square of the form (ii) there is a unique arrow* h *such that the diagram (iii) commutes. The cospan* f', g' *is called the* pushout *of* f, g*; sometimes the term pushout is used (somewhat imprecisely) for the shared target object* G*.*

If the commuting square in Fig. A.3 is a pushout, then the span $G \xrightarrow{g} G_2 \xrightarrow{f'} G_{12}$ *is the* pushout complement *of the span* $G \xrightarrow{f} G_1 \xrightarrow{g'} G_{12}$ *(and vice versa).*

Definition A.9 (pullback)**.** *A* pullback diagram *for a cospan* f, g *(see Fig. A.4)is a commuting square of the form (i) below, such that for every commuting square of the form (ii) there is a unique arrow* h *such that the diagram (iii) commutes.*

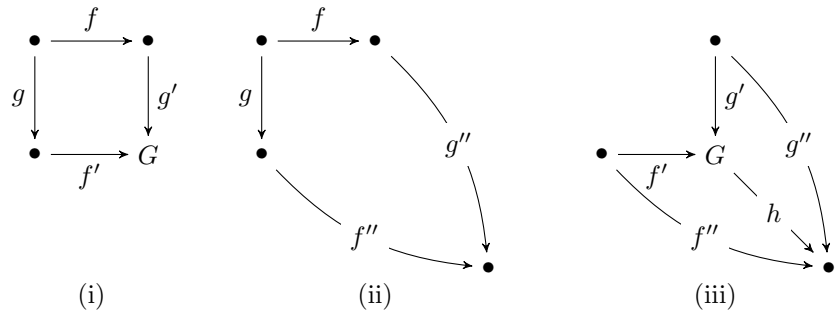


Figure A.2: Pushouts.

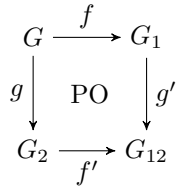


Figure A.3: Pushout and pushout complement.

The span f', g' is called the pullback *of f, g; sometimes the term pullback is used (somewhat imprecisely) for the shared source object G.*

By a *pushout along a monomorphism*, we mean a pushout, as in Fig. A.2(i), in which either f or g is a monomorphism.

For arbitrary categories, pushouts and pullbacks compose and decompose, as stated in the following proposition.

Proposition A.10. *See Fig. A.5.*

1. *Pushouts compose and decompose; that is, if* (1) *is a pushout diagram, then* (2) *is a pushout diagram if and only if* $(1+2)$ *is a pushout diagram;*
2. *Pullback compose and decompose; that is, if* (2) *is a pullback diagram, then* (1) *is a pullback diagram if and only if* $(1+2)$ *is a pullback diagram.*

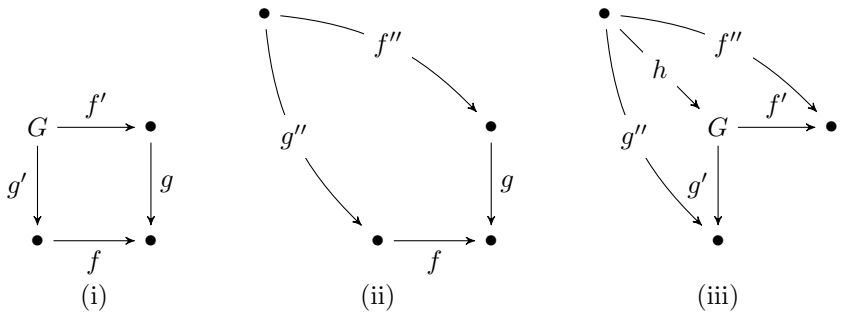


Figure A.4: Pullbacks.

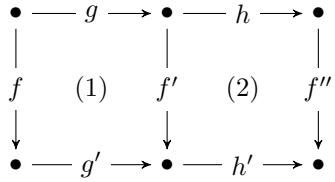


Figure A.5: Composing and decomposing pushouts and pullback.

A.3 Adhesive Categories

Some of the proofs of Chapter 3 rely on results from *adhesive categories* [128, 129]. Adhesive categories build on the notion of *Van Kampen squares.*

Definition A.11 (Van Kampen square)**.** *A* Van Kampen (VK) square *(a) is a pushout which satisfies the following condition: given a commutative cube (b) in which (a) forms the bottom and the back faces are pullbacks, the front faces are pullbacks if and only if the top face is a pushout.*

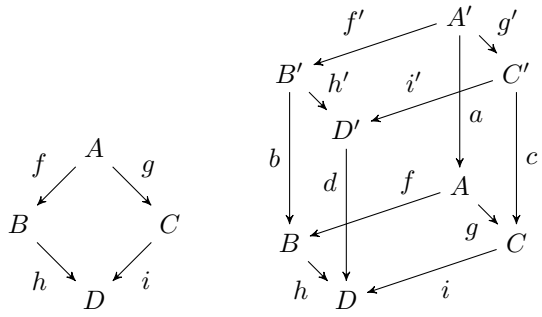
Adhesive categories can now be defined as follows.

Definition A.12 (adhesive category)**.** *A category* **C** *is said to be* adhesive *if*

(i) **C** *has pushouts along monomorphisms;*

(ii) **C** *has pullbacks;*

(iii) pushouts along monomorphisms are VK-squares.



The conditions on adhesive categories essentially ensure that such categories are "set-like"; that is, the pushout is "union-like" and the pullback is "intersection-like". Examples of adhesive categories are **Set**, **Graph**, and $\mathbf{AlgGraph}^{+}(SIG)$.

The following are consequences of adhesiveness.

Proposition A.13 ([128, Lemma 12])**.** *Monomorphisms are stable under pushout.*

Proposition A.14 ([128, Lemma 13])**.** *Pushouts along monomorphisms are also pullbacks.*

Proposition A.15 ([128, Lemma 15])**.** *In an adhesive category, pushout complements of monos (if they exist) are unique up to isomorphism.*

B Proofs of Chapter 3

B.1 Proofs of Section 3.2

Lemma 3.8. *If SIG is a graph structure signature, then the categories* $\mathbf{Alg}(SIG)$ *and* $\mathbf{AlgGraph}(SIG)$ *are equivalent.*

This can be proved by functors $\mathcal{F}_{\mathsf{graph}}, \mathcal{U}_{\mathsf{graph}}$ defined as follows.

- $\mathcal{F}_{\mathsf{graph}}$ maps every algebra A in $\mathbf{Alg}(SIG)$ to the graph G_A defined by

$$\begin{array}{rcl} N_{G_A} & = & \bigcup_{s \in S} A_s \\ E_{G_A} & = & \{(d, o, op_o(d)) \mid o \in OP, d \in A_{\sigma(o)}\} \\ \mathsf{src}_{G_A} & = & \{(e, d) \mid e = (d, o, d')\} \\ \mathsf{tgt}_{G_A} & = & \{(e, d') \mid e = (d, o, d')\} \\ \mathsf{lab}_{G_A} & = & \{(e, o) \mid e = (d, o, d')\} \ . \end{array}$$

  $\mathcal{F}_{\mathsf{graph}}$ maps every algebra morphism $(h_s)_{s \in S}$ to a graph morphism (h^N, h^E) defined by

$$\begin{array}{rcl} h^N : d & \mapsto & h_s(d) \quad \text{if } d \in A_s \\ h^E : (d, o, d') & \mapsto & (h^N(d), o, h^N(d')) \ . \end{array}$$

- $\mathcal{U}_{\mathsf{graph}}$ maps every graph G in $\mathbf{AlgGraph}(SIG)$, with typing (t_N, t_E), to

the algebra A_G defined by

$$\begin{aligned} A_s &= \{n \in N_G \mid t_N(n) = s\} \quad \text{for all } s \in S \\ op_o &= \{(\mathsf{src}(e), \mathsf{tgt}(e)) \mid e \in E_G, t_E(e) = o\} \quad \text{for all } o \in OP \ . \end{aligned}$$

$\mathcal{U}_{\mathsf{graph}}$ maps every graph morphism $(h_N, h_E)\colon G \to H$ in $\mathbf{AlgGraph}(SIG)$ to an algebra morphism $(h_s s)_{s \in S}$ defined by

$$h_s : d \quad \mapsto \quad h_N(d) \quad \text{if } t_N(d) = s.$$

Lemma 3.28. We first have to show that $\mathcal{F}_{\mathsf{graph}}$ and $\mathcal{U}_{\mathsf{graph}}$ are indeed functors.

- The images of $\mathcal{F}_{\mathsf{graph}}$ are graphs, resp. graph morphisms by construction. Identities and morphism composition are obviously preserved due to the fact that morphisms on both sides are built up from functions over sets, and identity morphisms and morphism composition are defined component-wise on the functions.

  To show that G_A is in fact in $\mathbf{AlgGraph}(SIG)$, we need the typing morphism required in Def. 3.6. This is defined by:

  $$\begin{aligned} t_A^N : n &\mapsto s \quad \text{if } n \in A_s \\ t_A^E : e &\mapsto \mathsf{lab}(e) \end{aligned}$$

  To see that the edge existence criterion for algebra graphs is satisfied, note that for all $n \in N_{G_A}$ and all $o \in OP$ with $\sigma(o) = t_A^N(n)$, $op_o(n) \in N_{G_A}$ and $(n, o, op_o(n)) \in E_{G_A}$. To see that the uniqueness criterion for deterministic algebra graphs is satisfied, assume $e_1, e_2 \in E_{G_A}$ such that $\mathsf{src}(e_1) = \mathsf{src}(e_2) = n$ and $\mathsf{lab}(e_1) = \mathsf{lab}(e_2) = o$. By construction of E_{G_A}, it follows that $\mathsf{tgt}(e_1) = \mathsf{tgt}(e_2) = op_o(n)$ and hence $e_1 = e_2$.

- To see that the images of $\mathcal{U}_{\mathsf{graph}}$ are algebras, we need to show that for all $o \in OP$, the constructed op_o are functions from $\sigma(o)$ to $\tau(o)$. First of all, the correctness of the typing follows from the typing morphism t_G: if $t_G^E(e) = o$ then $t_G^N(\mathsf{src}(e)) = \sigma(o)$ and $t_G^N(\mathsf{tgt}(e)) = \tau(o)$ by the fact that t_G is a graph morphism. The fact that op_o is a function follows from the edge existence criteria of deterministic algebra graphs: for every $o \in OP$ and $d \in A_{\sigma(o)}$, there is precisely one edge in G such that $\mathsf{src}(e) = d$ and $\mathsf{lab}(e) = o$; hence $op_o(d)$ is well-defined.

  The fact that morphisms, identities and morphism composition are carried over from $\mathbf{AlgGraph}(SIG)$ to $\mathbf{Alg}(SIG)$ follows from the same argument

above: morphisms are constructed component-wise from the underlying functions, and this construction behaves the same in both categories.

To prove the equivalence of $\mathbf{Alg}(SIG)$ and $\mathbf{AlgGraph}(SIG)$, we show that $\mathcal{F}_{\mathsf{graph}}$ and $\mathcal{U}_{\mathsf{graph}}$ are "inverse up to isomorphism".

- $\mathcal{U}_{\mathsf{graph}}(\mathcal{F}_{\mathsf{graph}}(A)) = A$ for all algebras A in $\mathbf{Alg}(SIG)$. This is immediate from the definition of the functors.

- $\mathcal{F}_{\mathsf{graph}}(\mathcal{U}_{\mathsf{graph}}(G)) \cong G$ for all graphs G in $\mathbf{AlgGraph}(SIG)$. Let us denote $H = \mathcal{F}_{\mathsf{graph}}(\mathcal{U}_{\mathsf{graph}}(G))$. The only reason why $G = H$ fails is the fact that edge identities are discarded by $\mathcal{U}_{\mathsf{graph}}$ and cannot be reconstructed precisely. The isomorphism $\psi\colon G \to H$ is therefore given by:

$$\begin{array}{rcl} \psi^N : n & \mapsto & n \\ \psi^E : e & \mapsto & (\mathsf{src}(e), \mathsf{lab}(e), \mathsf{tgt}(e)) \ . \end{array}$$

The fact that this is an isomorphism is due to the uniqueness of the edges of G. □

B.2 Proofs of Section 3.3.1

Lemma B.1. *Let SIG and SIG' be two signatures and $h\colon SIG \to SIG'$ be a signature morphism. Then, $\mathcal{F}(h)\colon \mathcal{F}(SIG) \to \mathcal{F}(SIG')$ (see Fig. B.1).*

Proof. Assume the following notational convention:

$$\begin{array}{rcl} SIG & = & (S, OP) \\ SIG' & = & (S', OP') \\ \mathcal{F}(SIG) & = & \langle (FS, FOP), \prec \rangle \\ \mathcal{F}(SIG') & = & \langle (FS', FOP'), \prec' \rangle \ . \end{array}$$

We have to show that for all operations $f\colon u \to d \in FOP$ it holds that

$$\begin{array}{rcl} \sigma(\mathcal{F}(h)_{OP}(f)) & = & \mathcal{F}(h)_S(\sigma(f)) \\ \tau(\mathcal{F}(h)_{OP}(f)) & = & \mathcal{F}(h)_S(\tau(f)) \ , \end{array}$$

i.e., $\mathcal{F}(h)$ commutes with both σ and τ. For this we distinguish between the following two cases for f, namely

- $f = \overline{o}$, thus according to Def. 3.14: $u = \overline{\sigma(o)}$ and $d = \tau(o)$,
- $f = p_{u,k}$, thus according to Def. 3.14: $u = \overline{\sigma(o)}$ and $d = s_k$,

with $o\colon s_1 \cdots s_k \cdots s_n \to s \in OP$.

We show the correct typing for the first case, i.e., $f = \overline{o}$, as follows. The commutativity of $\mathcal{F}(h)$ and σ can be shown as follows:

$$
\begin{array}{lll}
 & \sigma(\mathcal{F}(h)_{OP}(\overline{o})) = \mathcal{F}(h)_S(u) & \\
\Leftrightarrow & \{ \text{ lhs: first bullet from Def. 3.17 } \} & \\
 & \sigma(\overline{h_{OP}(o)}) = \mathcal{F}(h)_S(u) & \\
\Leftrightarrow & \{ \text{ lhs: first bullet from Def. 3.17 stating } \sigma(\overline{o'}) = \overline{\sigma(o')} \ \} & \\
 & \overline{\sigma(h_{OP}(o))} = \mathcal{F}(h)_S(u) & \\
\Leftrightarrow & \{ \text{ rhs: case assumption } u = \overline{\sigma(o)} \ \} & \\
 & \overline{\sigma(h_{OP}(o))} = \mathcal{F}(h)_S(\overline{\sigma(o)}) & \\
\Leftrightarrow & \{ \text{ rhs: case assumption } o\colon s_1 \cdots s_n \to s \ \} & \\
 & \overline{\sigma(h_{OP}(o))} = \mathcal{F}(h)_S(\overline{s_1 \cdots s_n}) & \\
\Leftrightarrow & \{ \text{ rhs: second bullet from Def. 3.17 } \} & \\
 & \overline{\sigma(h_{OP}(o))} = \overline{\sigma(h_{OP}(o))} & \\
\Leftrightarrow & \texttt{true} \ . &
\end{array}
$$

The commutativity of $\mathcal{F}(h)$ and τ can be shown as follows:

$$
\begin{array}{lll}
 & \tau(\mathcal{F}(h)_{OP}(\overline{o})) = \mathcal{F}(h)_S(\tau(o)) & \\
\Leftrightarrow & \{ \text{ rhs: combining } \tau(o) \in D \text{ (Def. 3.10) and first case for } \mathcal{F}(h)_S \text{ in Def. 3.17 } \} & \\
 & \tau(\mathcal{F}(h)_{OP}(\overline{o})) = h_S(\tau(o)) & \\
\Leftrightarrow & \{ \text{ lhs: first bullet Def. 3.17 } \} & \\
 & \tau(\overline{h_{OP}(o)}) = h_S(\tau(o)) & \\
\Leftrightarrow & \{ \text{ lhs: first bullet Def. 3.17 stating } \tau(\overline{o'}) = \tau(o') \ \} & \\
 & \tau(h_{OP}(o)) = h_S(\tau(o)) & \\
\Leftrightarrow & \{ \text{ lhs: } h \text{ is a signature morphism } \} & \\
 & h_S(\tau(o)) = h_S(\tau(o)) &
\end{array}
$$

$\Leftrightarrow$ `true` .

We show the correct typing for the second case, i.e., $f = p_{u,k}$, as follows. The commutativity of $\mathcal{F}(h)$ and σ is analogous to the first case. Using the notation of Def. 3.17, let $p'_{u',k} = \mathcal{F}(h)_{OP}(p_{u,k})$. The commutativity of $\mathcal{F}(h)$ and τ can be shown as follows:

$$
\begin{array}{lll}
 & \tau(\mathcal{F}(h)_{OP}(p_{u,k})) = \mathcal{F}(h)_S(s_k) & \\
\Leftrightarrow & \{ \text{ rhs: first case for } \mathcal{F}(h)_S \text{ in Def. 3.17 } \} & \\
 & \tau(\mathcal{F}(h)_{OP}(p_{u,k})) = h_S(s_k) & \\
\Leftrightarrow & \{ \text{ lhs: second bullet from Def. 3.17 stating } \tau(p'_{u',k}) = h_S(\tau(p_{u,k})) \} & \\
 & h_S(\tau(p_{u,k})) = h_S(s_k) & \\
\Leftrightarrow & \{ \text{ lhs: case assumption } o\colon s_1 \cdots s_k \cdots s_n \to s \} & \\
 & h_S(s_k) = h_S(s_k) & \\
\Leftrightarrow & \texttt{true} \ . &
\end{array}
$$

And thus for all $f\colon u \to d \in FOP$ it holds that $\mathcal{F}(h)_{OP}(f)\colon \mathcal{F}(h)_S(u) \to \mathcal{F}(h)_S(d) \in FOP'$. □

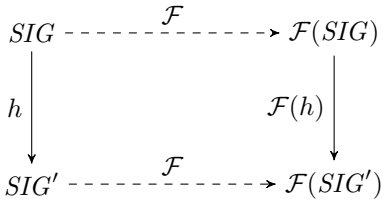


Figure B.1: Flattening the signature morphism $h\colon SIG \to SIG'$.

Lemma 3.18. *$\mathcal{F}$ is a functor from the category* **Sig** *to the category* **USig**.

Proof. For an arbitrary signature $SIG \in Obj_{\mathbf{Sig}}$ and any two arrows $f\colon SIG \to SIG'$ and $g\colon SIG' \to SIG''$ we have to show that

$$\mathcal{F}(id_{SIG}) = id_{\mathcal{F}(SIG)} \tag{B.1}$$
$$\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f) \ . \tag{B.2}$$

For proving (B.1) let $USIG = \mathcal{F}(SIG)$. With $id_{SIG} = (id_S, id_{OP})$ we then get:

- for all $d \in D_{USIG}$ we have: $\mathcal{F}(id_{SIG})_S(d) = id_{SIG,S}(d) = d = id_{\mathcal{F}(SIG),S}(d)$,
- for all $u \in U_{USIG}$, we have $\tau(p_{u,k}) \in D_{USIG}$ for $k = 1, \ldots, |\Pi_u|$, and therefore

$$
\begin{array}{ll}
 & \mathcal{F}(id_{SIG})_S(u) \\
\Leftrightarrow & \{ \text{ by Def. 3.10 } \} \\
 & \mathcal{F}(id_{SIG})_S(\overline{\tau(p_{u,1}) \cdots \tau(p_{u,n})}) \\
\Leftrightarrow & \{ \text{ first case of } \mathcal{F}(h)_S \text{ in Def. 3.17 } \} \\
 & \overline{id_{SIG,S}(\tau(p_{u,1})) \cdots id_{SIG,S}(\tau(p_{u,n}))} \\
\Leftrightarrow & \{ \text{ by Def. 3.14 } \} \\
 & \overline{id_{\mathcal{F}(SIG),S}(\tau(p_{u,1})) \cdots id_{\mathcal{F}(SIG),S}(\tau(p_{u,n}))} \\
\Leftrightarrow & \{ \text{ by definition of } id_{\mathcal{F}(SIG),S} \text{ on data sorts } \} \\
 & \overline{\tau(p_{u,1}) \cdots \tau(p_{u,n})} \\
\Leftrightarrow & \{ \text{ by definition of } id_{\mathcal{F}(SIG),S} \text{ on product sorts } \} \\
 & id_{\mathcal{F}(SIG),S}(\overline{\tau(p_{u,1}) \cdots \tau(p_{u,n})}) \\
\Leftrightarrow & \{ \text{ by Def. 3.10 } \} \\
 & id_{\mathcal{F}(SIG),S}(u) \quad .
\end{array}
$$

- for all $f\colon u \to d \in (F \cup \Pi)_{USIG}$ we have to show that $\mathcal{F}(id_{SIG})_{OP}(f)$ has the same typing as $id_{\mathcal{F}(SIG),OP}(f)$. We distinguish the following cases for f:
  - $f = \overline{o}$, thus according to Def. 3.14: $u = \overline{\sigma(o)}$ and $d = \tau(o)$,
  - $f = p_{u,k}$, thus according to Def. 3.14: $u = \overline{\sigma(o)}$ and $d = s_k$,

  with $o\colon s_1 \cdots s_k \cdots s_n \to s \in OP$. For $f = \overline{o}$ the required commutativity for the parameter sort has been shown in the previous bullet. For the target sort the proof in the first bullet applies due to $\tau(\overline{o}) \in D$ (see Def. 3.10) and $\tau(\overline{o}) = \tau(o)$ (see Def. 3.14). For the case $f = p_{u,k}\colon \overline{\sigma(o)} \to s_k$, where $k = 1, \ldots, |\Pi_u|$, the commutativity of the parameter and target sorts can be shown analogously.

For proving (B.2) let $USIG'' = \mathcal{F}(SIG'') = \langle (D'' \cup U'', F'' \cup \Pi''), \prec \rangle$:

- for all $d'' \in D''$ we have:

$$\begin{aligned}
\mathcal{F}(g \circ f)_S(d'') &= \mathcal{F}(g_S \circ f_S)(d'') \\
&= (g_S \circ f_S)(d'') \\
&= g_S(f_S(d'')) \\
&= \mathcal{F}(g_S)(\mathcal{F}(f_S)(d'')) \\
&= \mathcal{F}(g)_S(\mathcal{F}(f)_S(d'')) \\
&= (\mathcal{F}(g)_S \circ \mathcal{F}(f)_S)(d'') \ .
\end{aligned}$$

- for all $u'' \in U''$ we have:

$$\begin{aligned}
\mathcal{F}(g \circ f)_S(u'') &= \mathcal{F}(g \circ f)_S(\overline{\tau(p_{u'',1}) \cdots \tau(p_{u'',n})}) \\
&= \overline{\mathcal{F}(g \circ f)_S(\tau p_{u'',1}) \cdots \mathcal{F}(g \circ f)_S(\tau(p_{u'',n}))} \\
&= \overline{(g \circ f)_S(\tau p_{u'',1}) \cdots (g \circ f)_S(\tau(p_{u'',n}))} \\
&= \overline{(g_S \circ f_S)(\tau p_{u'',1}) \cdots (g_S \circ f_S)(\tau(p_{u'',n}))} \\
&= \overline{g_S(f_S(\tau p_{u'',1})) \cdots g_S(f_S(\tau(p_{u'',n})))} \\
&= \overline{\mathcal{F}(g_S)(\mathcal{F}(f_S)(\tau p_{u'',1})) \cdots \mathcal{F}(g_S)(\mathcal{F}(f_S)(\tau(p_{u'',n})))} \\
&= \overline{\mathcal{F}(g)_S(\mathcal{F}(f)_S(\tau p_{u'',1})) \cdots \mathcal{F}(g)_S(\mathcal{F}(f)_S(\tau(p_{u'',n})))} \\
&= \overline{(\mathcal{F}(g)_S \circ \mathcal{F}(f)_S)(\tau p_{u'',1}) \cdots (\mathcal{F}(g)_S \circ \mathcal{F}(f)_S)(\tau(p_{u'',n}))} \\
&= (\mathcal{F}(g)_S \circ \mathcal{F}(f)_S)(\overline{\tau(p_{u'',1}) \cdots \tau(p_{u'',n})}) \\
&= (\mathcal{F}(g)_S \circ \mathcal{F}(f)_S)(u'') \ .
\end{aligned}$$

- for all $f'' \colon u'' \to d'' \in (F'' \cup \Pi'')$ we have to show that the typing of $\mathcal{F}(g \circ f)_{OP}(f'')$ equals the typing of $(\mathcal{F}(g)_{OP} \circ \mathcal{F}(f)_{OP})(f'')$. We make the following case distinction for f'':
  - $f'' = \overline{o''}$, thus according to Def. 3.14: $u'' = \overline{\sigma(o'')}$ and $d'' = \tau(o'')$,
  - $f'' = p''_{u'',k}$, thus according to Def. 3.14: $u'' = \overline{\sigma(o'')}$ and $d'' = s''_k$,

  with $o'' = g_{OP}(o') = (g \circ f)_{OP}(o)$ and $o \colon s_1 \cdots s_k \cdots s_n \to s \in OP$. For $f'' = \overline{o''}$ the required commutativity for the parameter sort has been shown in the previous bullet. For the target sort the proof in the second previous bullet applies due to $\tau(\overline{o''}) \in D''$ (see Def. 3.10) and $\tau(\overline{o''}) = \tau(o'')$ (see Def. 3.14). For the case $f'' = p''_{u'',k}$ the commutativity of the parameter and target sorts can be shown analogously. □

B.3 Proofs of Section 3.3.2

Lemma B.2. *Let SIG and SIG' be two uniform signatures and $h\colon SIG \to SIG'$ be a uniform signature morphism. Then, $\mathcal{U}(h)\colon \mathcal{U}(SIG) \to \mathcal{U}(SIG')$ (see Fig. B.2).*

Proof. Assume the following notational convention:

$$\begin{aligned} SIG &= \langle (D \cup U, F \cup \Pi), \prec \rangle \\ SIG' &= \langle (D' \cup U', F' \cup \Pi'), \prec' \rangle \\ \mathcal{U}(SIG) &= (D, OP) \\ \mathcal{U}(SIG') &= (D', OP') \ . \end{aligned}$$

Using here, locally, the notation "$f^*(x_1 \cdots x_n)$)" for "$f(x_1) \cdots f(x_n)$", we have to show that for all $o_f\colon s_1 \cdots s_n \to s \in OP$ it holds that

$$\begin{aligned} \sigma(\mathcal{U}(h)_{OP}(o_f)) &= \mathcal{U}(h)^*_S(\sigma(o_f)) \\ \tau(\mathcal{U}(h)_{OP}(o_f)) &= \mathcal{U}(h)_S(\tau(o_f)) \ , \end{aligned}$$

i.e., $\mathcal{U}(h)$ commutes with both σ and τ. Let $o_f\colon s_1 \cdots s_k \cdots s_n \to s$ with $f\colon u \to d \in F$, i.e., $\tau(p_{u,k}) = s_k$. The commutativity of $\mathcal{U}(h)$ and σ can then be shown as follows:

$$\begin{array}{lll} & \sigma(\mathcal{U}(h)_{OP}(o_f)) = \mathcal{U}(h)^*_S(\sigma(o_f)) & \\ \Leftrightarrow & \{ \text{ rhs: by definition of } o_f \text{ (see Def. 3.19) } \} & \\ & \sigma(\mathcal{U}(h)_{OP}(o_f)) = \mathcal{U}(h)^*_S(\tau(p_{u,1}) \cdots \tau(p_{u,n})) & \\ \Leftrightarrow & \{ \text{ rhs: combining } \tau(p_{u,k}) \in D \text{ (see Def. 3.19) and first bullet of Def. 3.20 } \} & \\ & \sigma(\mathcal{U}(h)_{OP}(o_f)) = h_S(\tau(p_{u,1})) \cdots h_S(\tau(p_{u,n})) & \\ \Leftrightarrow & \{ \text{ lhs: second bullet of Def. 3.20 } \} & \\ & h_S(\tau(p_{u,1})) \cdots h_S(\tau(p_{u,n})) = h_S(\tau(p_{u,1})) \cdots h_S(\tau(p_{u,n})) & \\ \Leftrightarrow & \texttt{true} \ . & \end{array}$$

The commutativity of $\mathcal{U}(h)$ and τ can be shown as follows:

$$\begin{array}{lll} & \tau(\mathcal{U}(h)_{OP}(o_f)) = \mathcal{U}(h)_S(\tau(o_f)) & \\ \Leftrightarrow & \{ \text{ rhs: by definition of } o_f \text{ (see Def. 3.19) } \} & \end{array}$$

$$
\begin{array}{lll}
 & \tau(\mathcal{U}(h)_{OP}(o_f)) & = \quad \mathcal{U}(h)_S(\tau(f)) \\
\Leftrightarrow & \{ \text{ rhs: combining } \tau(f) \in D \text{ (see Def. 3.19) and first bullet of Def. 3.19 } \} & \\
 & \tau(\mathcal{U}(h)_{OP}(o_f)) & = \quad h_S(\tau(f)) \\
\Leftrightarrow & \{ \text{ lhs: second bullet of Def. 3.20 } \} & \\
 & h_S(\tau(f)) & = \quad h_S(\tau(f)) \\
\Leftrightarrow & & \texttt{true} \ .
\end{array}
$$

We thus may conclude that for all $o_f: s_1 \cdots s_n \to s \in OP$ it holds that $\mathcal{U}(h)_{OP}(o_f): \mathcal{U}(h)_S(s_1) \cdots \mathcal{U}(h)_S(s_n) \to \mathcal{U}(h)_S(s) \in OP'$. □

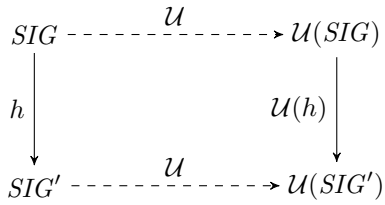


Figure B.2: Unflattening the uniform signature morphism $h: SIG \to SIG'$.

Lemma 3.21. *$\mathcal{U}$ is a functor from the category* **USig** *to the category* **Sig**.

Proof. For an arbitrary uniform signature $USIG \in Obj_{\mathbf{USig}}$ and any two arrows $f: USIG \to USIG'$ and $g: USIG' \to USIG''$ we have to show that

$$\mathcal{U}(id_{USIG}) = id_{\mathcal{U}(USIG)} \tag{B.3}$$

$$\mathcal{U}(g \circ f) = \mathcal{U}(g) \circ \mathcal{U}(f) \ . \tag{B.4}$$

For proving (B.3) let $SIG = \mathcal{U}(USIG)$:

- for all $s \in S_{SIG}$ we have: $\mathcal{U}(id_{USIG})_S(s) = id_{USIG,S}(s) = s = id_{\mathcal{U}(USIG),S}(s)$,
- for all $o_f \in OP_{SIG}$ with $f: u \to d \in F_{USIG}$ we have to show that the typing of $\mathcal{U}(id_{USIG})_{OP}(o_f)$ equals the typing of $id_{\mathcal{U}(USIG),OP}(o_f)$. We show the required commutativity for the parameter sort as follows (using again the *-notation, i.e., "$f^*(x_1 \cdots x_n)$" denoting "$f(x_1) \cdots f(x_n)$"):

$$\mathcal{U}(id_{USIG})^*_S(\sigma(o)) = \mathcal{U}(id_{USIG})_S(\tau(p_{u,1})) \cdots \mathcal{U}(id_{USIG})_S(\tau(p_{u,n}))$$

$$\begin{aligned} &= id_{USIG,S}(\tau(p_{u,1})) \cdots id_{USIG,S}(\tau(p_{u,n})) \\ &= id_{\mathcal{U}(USIG),S}(\tau(p_{u,1})) \cdots id_{\mathcal{U}(USIG),S}(\tau(p_{u,n})) \\ &= id^*_{\mathcal{U}(USIG),S}(\sigma(o)) \ . \end{aligned}$$

We show the required commutativity for the target sort of o_f as follows:

$$\begin{aligned} \mathcal{U}(id_{USIG})_S(\tau(o_f)) &= id_{USIG,S}(\tau(o_f)) \\ &= id_{\mathcal{U}(USIG),S}(\tau(o_f)) \ . \end{aligned}$$

For proving (B.4) let $USIG'' = \mathcal{U}(SIG'')$:

- for all $s'' \in S''$ we have:

$$\begin{aligned} \mathcal{U}(g' \circ f')_S(s'') &= \mathcal{U}(g'_S \circ f'_S)(s'') \\ &= (g'_S \circ f'_S)(s'') \\ &= g'_S(f'_S(s'')) \\ &= \mathcal{U}(g'_S)(\mathcal{U}(f'_S)(s'')) \\ &= \mathcal{U}(g')_S(\mathcal{U}(f')_S(s'')) \\ &= (\mathcal{U}(g')_S \circ \mathcal{U}(f')_S)(s'') \ . \end{aligned}$$

- for all $o'' \colon s''_1 \cdots s''_n \to s'' \in OP''$ we show the required commutativity for the parameter sorts as follows (using again, locally, the *-notation):

$$\begin{aligned} \mathcal{U}(g' \circ f')^*_S(\sigma(o'')) &= \mathcal{U}(g' \circ f')^*_S(s''_1 \cdots s''_n) \\ &= \mathcal{U}(g' \circ f')_S(s''_1) \cdots \mathcal{U}(g' \circ f')_S(s''_n) \\ &= \mathcal{U}(g'_S \circ f'_S)(s''_1) \cdots \mathcal{U}(g'_S \circ f'_S)(s''_n) \\ &= (g'_S \circ f'_S)(s''_1) \cdots (g'_S \circ f'_S)(s''_n) \\ &= g'_S(f'_S(s''_1)) \cdots g'_S(f'_S(s''_n)) \\ &= \mathcal{U}(g'_S)(\mathcal{U}(f'_S)(s''_1)) \cdots \mathcal{U}(g'_S)(\mathcal{U}(f'_S)(s''_n)) \\ &= (\mathcal{U}(g'_S) \circ \mathcal{U}(f'_S))(s''_1) \cdots (\mathcal{U}(g'_S) \circ \mathcal{U}(f'_S))(s''_n) \\ &= (\mathcal{U}(g'_S) \circ \mathcal{U}(f'_S))^*(s''_1 \cdots s''_n) \\ &= (\mathcal{U}(g'_S) \circ \mathcal{U}(f'_S))^*(\sigma(o'')) \ . \end{aligned}$$

For the target sort we show the required commutativity as follows:

$$\mathcal{U}(g' \circ f')_S(s'') = \mathcal{U}(g'_S \circ f'_S)(s'')$$

$$
\begin{aligned}
&= (g'_S \circ f'_S)(s'') \\
&= g'_S(f'_S(s'')) \\
&= \mathcal{U}(g'_S)(\mathcal{U}(f'_S)(s'')) \\
&= (\mathcal{U}(g'_S) \circ \mathcal{U}(f'_S))(s'') \ .
\end{aligned}
$$

□

B.4 Proofs of Section 3.3.3

Lemma B.3. *$(\mathcal{U} \circ \mathcal{F})(SIG) = SIG$, for arbitrary signatures SIG.*

Proof. Let $SIG = (S, OP)$ be an arbitrary signature, $USIG' = \mathcal{F}(SIG) = \langle (D' \cup U', F' \cup \Pi'), \prec' \rangle$, and $SIG' = \mathcal{U}(SIG') = \mathcal{U}(\mathcal{F}(SIG)) = (S', OP')$. We then have to prove that $SIG' = SIG$. We will first prove that $S \subseteq S'$, $S' \subseteq S$, $OP \subseteq OP'$, and $OP' \subseteq OP$. Thereafter, we will show that for all operation symbols, their typing is preserved correctly.

$S \subseteq S'$: Assume $s \in S$. Obviously, then also $s \in S'$ and therefore $S \subseteq S'$.

$S' \subseteq S$: Assume $s \in S'$. This must mean, by construction of S', that $s \in D'$ which, on its turn, must mean that, by construction of D', that $s \in S$. And thus $S' \subseteq S$.

$OP \subseteq OP'$: Assume $o \in OP$. By construction of F we then have $oper(o) \in F$, which, by definition of $oper$, means that $o \in F$. By construction of OP', it then holds that $oper^{-1}(o) = o \in OP'$ and thus $OP \subseteq OP'$.

$OP' \subseteq OP$: Assume $o \in OP'$. This must mean, by construction of OP', that $oper^{-1}(o) \in F$. By definition of $oper$ we then have $o \in F$. By construction of F, it then holds that $oper^{-1}(o) = o \in OP$ and thus $OP' \subseteq OP$.

Proving the correct typing preservation will be done for the parameter sorts and the target sorts separately. For every operation symbol $o\colon s_1 \cdots s_n \rightarrow s \in OP$ the parameter sort of the corresponding operation symbol in OP' can be shown to be equal to the parameter sort of the original o. Let $f = oper(o)$ with $\sigma(f) = u = \overline{\sigma(o)}$, and $o' = oper(f)$. We then have:

$$
\begin{aligned}
& \sigma(o') \\
= \ & \{ \text{ by Def. 3.19 } \} \\
& \tau(p_{u,1}) \cdots \tau(p_{u,n})
\end{aligned}
$$

$$
\begin{array}{ll}
= & \{ \text{ by Def. 3.14 } \} \\
& s_1 \cdots s_n \\
= & \{ \text{ by case assumption } o\colon s_1 \cdots s_n \to s \ \} \\
& \sigma(o) \ .
\end{array}
$$

For the target sort of o' we have the following:

$$
\begin{array}{ll}
& \tau(o') \\
= & \{ \text{ by Def. 3.19 } \} \\
& \tau(f) \\
= & \{ \text{ by Def. 3.14 } \} \\
& \tau(o) \ .
\end{array}
$$

Finally, we may conclude that $SIG' = SIG$. □

Lemma B.4. *Let USIG be a uniform signature. For the signature* $USIG' = \mathcal{F}(\mathcal{U}(USIG))$ *there exists an isomorphism* $g_{USIG}\colon USIG \to USIG'$.

Proof. Let $USIG = \langle (D \cup U, F \cup \Pi), \prec \rangle$, $SIG = \mathcal{F}(USIG) = (S, OP)$, and $USIG' = \mathcal{U}(SIG) = \langle (D' \cup U', F' \cup \Pi'), \prec' \rangle$. The isomorphism $g_{USIG}\colon USIG \to USIG'$ can be constructed as follows. For g_S[1] we have the following:

- for all $d \in D$ we have: $g_S(d) = d$, since these data sorts are preserved by both $\mathcal{F}$ and $\mathcal{U}$,
- for all $u \in U$ we have: $g_S(u) = \overline{\tau(p_{u,1}) \cdots \tau(p_{u,n})}$, where $n = |\Pi_u|$.

For g_{OP} we have the following:

- for all $f \in F$ we have: $g_{OP}(f) = \overline{\mathcal{U}_{OP}(f)}$. Note that $\tau(g_{OP}(f)) = g_S(\tau(f))$ since $\tau(f) \in D$ and $\sigma(g_{OP}(f)) = g_S(\sigma(f))$.
- for all $p_{u,k} \in \Pi$ we have: $g_{OP}(p_{u,k}) = p'_{g_S(u),k}$ with $\sigma(p'_{g_S(u),k}) = g_S(u)$ and $\tau(p'_{g_S(u),k}) = g_S(\tau(p_{u,k}))$.

[1] The notation $g_{USIG,S}$ would be more appropriate, but we use g_S as a shorthand notation when the uniform signature is clear from the context; the same remarks holds for g_{OP}.

The partial ordering on Π' is defined by $\mathcal{F}$ and can be verified to satisfy the following condition:

$$\forall p_1, p_2 \in \Pi : p_1 \prec p_2 \iff g_{OP}(p_1) \prec' g_{OP}(p_2) \ .$$

□

B.5 Proofs of Section 3.4.1

Lemma B.5. *Let $SIG = (S, OP)$ be an arbitrary signature. Then, for all SIG-algebra homomorphisms $h\colon A \to A'$ it holds that $\mathcal{F}^{\mathcal{A}}(h)\colon \mathcal{F}^{\mathcal{A}}(A) \to \mathcal{F}^{\mathcal{A}}(A')$ (see Fig. B.3).*

Proof. Assume the following notational conventions:

$$\begin{aligned} A &= (S_A, OP_A) \\ A' &= (S_{A'}, OP_{A'}) \\ \mathcal{F}^{\mathcal{A}}(A) &= (FS_A, FOP_A) \\ \mathcal{F}^{\mathcal{A}}(A') &= (FS_{A'}, FOP_{A'}) \ . \end{aligned}$$

For an arbitrary operation symbol $f\colon u \to d \in FOP_A$ we have to show that for all $a \in A_u$ it holds that

$$(\mathcal{F}^{\mathcal{A}}(h)_d \circ op_{\mathcal{F}^{\mathcal{A}}(A),f})(a) \quad = \quad (op_{\mathcal{F}^{\mathcal{A}}(A'),f} \circ \mathcal{F}^{\mathcal{A}}(h)_u)(a) \ .$$

For this to be shown, we make the following case distinction for $op_{\mathcal{F}^{\mathcal{A}}(A),f}$:

- $op_{\mathcal{F}^{\mathcal{A}}(A),f} = op_{\overline{o}}$, thus according to Def. 3.14 $u = \overline{\sigma(o)}$ and $d = \tau(o)$,
- $op_{\mathcal{F}^{\mathcal{A}}(A),f} = \pi_{u,k}$, thus according to Def. 3.14 $u = \overline{\sigma(o)}$ and $d = s_k$,

with $o\colon s_1 \cdots s_k \cdots s_n \to s \in OP$.

We show the required commutativity for the first case as follows. Let $a \in A_u$, $a_i = \pi_{u,i}(a)$ for $1 \leq i \leq n$, $op_{\overline{o}} = op_{\mathcal{F}^{\mathcal{A}}(A),\overline{o}}$, and $op'_{\overline{o}} = op_{\mathcal{F}^{\mathcal{A}}(A'),\overline{o}}$. We then have

$$\begin{aligned} & (\mathcal{F}^{\mathcal{A}}(h)_d \circ op_{\overline{o}})(a) \\ = \quad & \{ \text{ applying first bullet of Def. 3.30 since } \tau(\overline{o}) \in D \ \} \\ & (h_d \circ op_{\overline{o}})(a) \end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ by definition of the } \circ\text{-operator } \} \\
&\quad\; h_d(op_{\overline{o}}(a)) \\
&= \quad \{ \text{ by definition of } op_{\overline{o}} \text{ and letting } a_i = \pi_{u,i}(a) \ \} \\
&\quad\; h_d(op_o(a_1, \ldots, a_n)) \\
&= \quad \{ \text{ since } h \text{ is an algebra homomorphism } \} \\
&\quad\; op'_o(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \\
&= \quad \{ \text{ by definition of } op_{\overline{o}} \text{ in Def. 3.29 } \} \\
&\quad\; op'_{\overline{o}}(((h_{s_1} \circ \pi_{u,1}) \otimes \cdots \otimes (h_{s_n} \circ \pi_{u,n}))(a)) \\
&= \quad \{ \text{ applying second bullet of Def. 3.30 } \} \\
&\quad\; (op'_{\overline{o}} \circ \mathcal{F}^{\mathcal{A}}(h)_u)(a) \quad .
\end{aligned}
$$

For the second case, let $\pi_{u,k} = \pi_{u,k}^{\mathcal{F}^{\mathcal{A}}(A)}$ and $\pi'_{u,k} = \pi_{u,k}^{\mathcal{F}^{\mathcal{A}}(A')}$. We then have

$$
\begin{aligned}
&\quad\; (\mathcal{F}^{\mathcal{A}}(h)_{s_k} \circ \pi_{u,k})(a) \\
&= \quad \{ \text{ applying first bullet of Def. 3.30 since } \tau(p_{u,k}) \in D \ \} \\
&\quad\; (h_{s_k} \circ \pi_{u,k})(a) \\
&= \quad \{ \text{ by definition of function composition } \} \\
&\quad\; h_{s_k}(\pi_{u,k}(a)) \\
&= \quad \{ \text{ combining second bullet of Def. 3.30 and } h \text{ being a homomorphism } \} \\
&\quad\; \pi'_{u,k}(\langle (h_{s_1} \circ \pi_{u,1})(a), \ldots, (h_{s_n} \circ \pi_{u,n})(a) \rangle) \\
&= \quad \{ \text{ by definition of the } \otimes\text{-operator } \} \\
&\quad\; \pi'_{u,k}(\langle ((h_{s_1} \circ \pi_{u,1}) \otimes \cdots \otimes (h_{s_n} \circ \pi_{u,n}))(a) \rangle) \\
&= \quad \{ \text{ second bullet of Def. 3.30 } \} \\
&\quad\; \pi'_{u,k}(\mathcal{F}^{\mathcal{A}}(h)_u(a)) \\
&= \quad \{ \text{ by definition of function composition } \} \\
&\quad\; (\pi'_{u',k} \circ \mathcal{F}^{\mathcal{A}}(h)_u)(a) \quad .
\end{aligned}
$$

All together, we may conclude that all operations for the operation symbols in FOP' commute with $\mathcal{F}^{\mathcal{A}}$. □

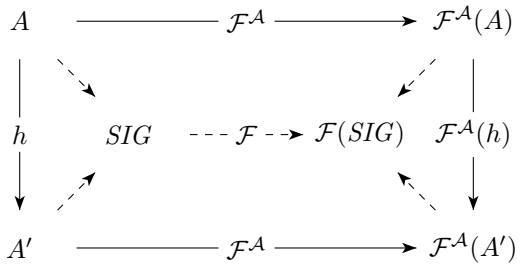


Figure B.3: Flattening of SIG-algebra homomorphism $h\colon A \to A'$.

Lemma B.6. *$\mathcal{F}^{\mathcal{A}}$ is a functor from the category* $\mathbf{Alg}(SIG)$ *to the category* $\mathbf{Alg}(USIG)$.

Proof. Let $SIG = (S, OP)$ be a signature and $\mathcal{F}(SIG) = \langle (D \cup U, F \cup \Pi), \prec \rangle$. For an arbitrary SIG-algebra B and any two SIG-algebra homomorphisms $g\colon B \to B'$ and $h\colon B' \to B''$ we have to show that

$$\mathcal{F}^{\mathcal{A}}(id_B) = id_{\mathcal{F}^{\mathcal{A}}(B)} \tag{B.5}$$

$$\mathcal{F}^{\mathcal{A}}(h \circ g) = \mathcal{F}^{\mathcal{A}}(h) \circ \mathcal{F}^{\mathcal{A}}(g)\ . \tag{B.6}$$

For proving (B.5) we have $id_B = (id_{B,s})_{s \in S}$ and therefore:

- for all $d \in D$ and all $a \in A_{B,d}$ we have: $\mathcal{F}^{\mathcal{A}}(id_{B,d})(a) = id_{B,d}(a) = a = id_{\mathcal{F}^{\mathcal{A}}(B),d}$,
- for all $u \in U$ and all $a \in A_u$ let $\pi_{u,k}(a) = a_k$ $(\in A_{\tau(p_{u,k})})$ for $k = 1, \ldots, |\Pi_u|$ and therefore:

$$\begin{array}{ll} & \mathcal{F}^{\mathcal{A}}(id_{A,u})(a) \\ = & \{ \text{ due to the functionality condition } \} \\ & \mathcal{F}^{\mathcal{A}}(id_{A,u})(\langle \pi_{u,1}(a), \ldots, \pi_{u,n}(a) \rangle) \\ = & \{ \text{ second bullet of Def. 3.30 } \} \\ & \langle id_{A,s_1}(\pi_{u,1}(a)), \ldots, id_{A,s_n}(\pi_{u,n}(a)) \rangle \\ = & \{ \text{ by definition of } id_{A,s_k} \text{ for } 1 \leq k \leq n \ \} \\ & \langle \pi_{u,1}(a), \ldots, \pi_{u,n}(a) \rangle \\ = & \{ \text{ by definition of } id_{\mathcal{F}^{\mathcal{A}}(A),u} \ \} \\ & id_{\mathcal{F}^{\mathcal{A}}(A),u}(\langle \pi_{u,1}(a), \ldots, \pi_{u,n}(a) \rangle) \end{array}$$

Requirement (B.6) can be proven through the following cases:

- for all $d \in D_B$ we have:

$$\begin{array}{lll} \mathcal{F}^{\mathcal{A}}(h \circ g)_d & = & \mathcal{F}^{\mathcal{A}}(h_d \circ g_d) \\ & = & h_d \circ g_d \\ & = & \mathcal{F}^{\mathcal{A}}(h_d) \circ \mathcal{F}^{\mathcal{A}}(g_d) \\ & = & \mathcal{F}^{\mathcal{A}}(h)_d \circ \mathcal{F}^{\mathcal{A}}(g)_d\ . \end{array}$$

- for all $u \in U_B$ we have:

$$
\begin{aligned}
& \mathcal{F}^{\mathcal{A}}(h \circ g)_u \\
= \;& (((h \circ g)_{s_1} \circ \pi_{u,1}) \otimes \cdots \otimes ((h \circ g)_{s_n} \circ \pi_{u,n})) \\
= \;& (((h_{s_1} \circ g_{s_1}) \circ \pi_{u,1}) \otimes \cdots \otimes ((h_{s_n} \circ g_{s_n}) \circ \pi_{u,n})) \\
= \;& (((\mathcal{F}^{\mathcal{A}}(h_{s_1}) \circ \mathcal{F}^{\mathcal{A}}(g_{s_1})) \circ \pi_{u,1}) \otimes \cdots \otimes ((\mathcal{F}^{\mathcal{A}}(h_{s_n}) \circ \mathcal{F}^{\mathcal{A}}(g_{s_n})) \circ \pi_{u,n})) \\
= \;& (((\mathcal{F}^{\mathcal{A}}(h)_{s_1} \circ \mathcal{F}^{\mathcal{A}}(g)_{s_1}) \circ \pi_{u,1}) \otimes \cdots \otimes ((\mathcal{F}^{\mathcal{A}}(h)_{s_n} \circ \mathcal{F}^{\mathcal{A}}(g)_{s_n}) \circ \pi_{u,n})) \\
= \;& (((\mathcal{F}^{\mathcal{A}}(h)_{s_1} \circ (\mathcal{F}^{\mathcal{A}}(g)_{s_1} \circ \pi_{u,1}) \otimes \cdots \otimes ((\mathcal{F}^{\mathcal{A}}(h)_{s_n} \circ (\mathcal{F}^{\mathcal{A}}(g)_{s_n} \circ \pi_{u,n})) \\
= \;& \mathcal{F}^{\mathcal{A}}(h)_u \circ ((\mathcal{F}^{\mathcal{A}}(g)_{s_1} \circ \pi_{u,1}) \otimes \cdots \otimes (\mathcal{F}^{\mathcal{A}}(g)_{s_n} \circ \pi_{u,n})) \\
= \;& \mathcal{F}^{\mathcal{A}}(h)_u \circ \mathcal{F}^{\mathcal{A}}(g)_u \qquad \square
\end{aligned}
$$

B.6 Proofs of Section 3.4.2

Lemma B.7. *Let* $USIG = \langle (D \cup U, F \cup \Pi), \prec \rangle$ *be an arbitrary uniform signature and* $\mathcal{U}(USIG) = (S, OP)$ *be the corresponding unflattened signature. Furthermore, let* $g\colon B \to B'$ *be a uniform USIG-algebra homomorphism. Then,* $\mathcal{U}^{\mathcal{A}}(g)\colon \mathcal{U}^{\mathcal{A}}(B) \to \mathcal{U}^{\mathcal{A}}(B')$ *(see Fig. B.4).*

Proof. Assume the following notational convention:

$$
\begin{aligned}
B &= (D_B \cup U_B, F_B \cup OP_B) \\
B' &= (D_{B'} \cup U_{B'}, F_{B'} \cup OP_{B'}) \\
\mathcal{U}^{\mathcal{A}}(B) &= (D_B, OP_{\mathcal{U}^{\mathcal{A}}(B)}) \\
\mathcal{U}^{\mathcal{A}}(B') &= (D_{B'}, OP_{\mathcal{U}^{\mathcal{A}}(B')}) \ .
\end{aligned}
$$

We have to show that for all operation symbols $f\colon u \to d \in F$ such that $o_f\colon s_1 \cdots s_k \cdots s_n \to s \in OP$, given $\tau(p_{u,k}) = s_k$, it holds that

$$
\mathcal{U}^{\mathcal{A}}(g)_s(op_{o_f}(a_1, \ldots, a_n)) = op'_{o_f}(\mathcal{U}^{\mathcal{A}}(g)_{s_1}(a_1), \ldots, \mathcal{U}^{\mathcal{A}}(g)_{s_n}(a_n)) \ .
$$

For all $u \in U$ and all $a \in A_{B,u}$ with $\pi_{u,k}(a) = a_k$ $(\in A_{\tau(p_{u,k})})$, the above equality can be proven in the following way:

$$
\begin{aligned}
\mathcal{U}^{\mathcal{A}}(g)_s(op_{o_f}(a_1, \ldots, a_n)) &= g_s(op_{o_f}(a_1, \ldots, a_n)) \\
&= g_s(op_f(a))
\end{aligned}
$$

$$\begin{aligned} &= op'_f(g_u(a)) \\ &= op'_{o_f}(g_{s_1}(a_1), \ldots, g_{s_n}(a_n)) \\ &= op'_{o_f}(\mathcal{U}(g_{s_1})(a_1), \ldots, \mathcal{U}(g_{s_n})(a_n)) \\ &= op'_{o_f}(\mathcal{U}(g)_{s_1}(a_1), \ldots, \mathcal{U}(g)_{s_n}(a_n)) \ . \end{aligned}$$

For this equality to be true, we rely on the totality condition on B. That is, we need $A_{B,u}$ to contain an element for every combination of the elements from the projection carrier sets. □

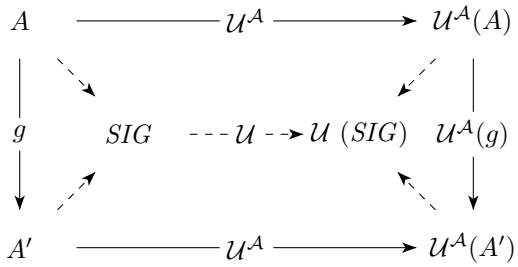


Figure B.4: Unflattening of *SIG*-algebra homomorphism $g\colon A \to A'$.

Lemma B.8. *$\mathcal{U}^{\mathcal{A}}$ is a functor from the category* **UAlg**(*USIG*) *to the category* **Alg**(*SIG*).

Proof. Let $USIG = \langle (D \cup U, F \cup \Pi), \prec \rangle$ and $\mathcal{U}(USIG) = (D, OP)$ be the corresponding unflattened signature. Then, for an arbitrary uniform *USIG*-algebra B and any two uniform *USIG*-algebra homomorphisms $g\colon B \to B'$ and $h\colon B' \to B''$ we have to show that

$$\mathcal{U}^{\mathcal{A}}(id_B) = id_{\mathcal{U}^{\mathcal{A}}(B)} \tag{B.7}$$

$$\mathcal{U}^{\mathcal{A}}(h \circ g) = \mathcal{U}^{\mathcal{A}}(h) \circ \mathcal{U}^{\mathcal{A}}(g) \ . \tag{B.8}$$

For proving (B.7) we have $id_B = (id_{B,s})_{s \in (D \cup U)}$ and therefore for all $d \in D$ and all $a \in A_{B,d}$ we have: $\mathcal{U}^{\mathcal{A}}(id_{B,d})(a) = id_{B,d}(a) = a = id_{\mathcal{U}^{\mathcal{A}}(B),d}$.

Requirement (B.8) can be shown correct as follows. For all $d \in D$ it holds that:

$$\begin{aligned} \mathcal{U}(h \circ g)_d &= (h_d \circ g_d) \\ &= \mathcal{U}(h_d) \circ \mathcal{U}(g_d) \end{aligned}$$

$$= \quad \mathcal{U}(h)_d \circ \mathcal{U}(g)_d \ .$$

□

B.7 Proofs of Section 3.4.3

Lemma B.9. *Let SIG be an arbitrary signature. Then,* $(\mathcal{U}^{\mathcal{A}} \circ \mathcal{F}^{\mathcal{A}})(B) = B$, *for any SIG-algebra* B.

Proof. Let $SIG = (S, OP)$ and $\mathcal{F}(SIG) = \langle (D \cup U, F \cup \Pi), \prec \rangle$. Furthermore, let $B = (S_B, OP_B)$ be an arbitrary SIG-algebra, $UB = \mathcal{F}^{\mathcal{A}}(B)$ be the corresponding uniform $\mathcal{F}(SIG)$-algebra, and $B' = \mathcal{U}^{\mathcal{A}}(UB)$. From the results in the previous section we can conclude that B' is again a SIG-algebra. For proving that $B = B'$ we will first show that B and B' have equal carrier sets. Thereafter, we will show that the operations in B' have exactly the same semantics as their counterparts in B.

For all $s \in S$ the fact that $A_{B,s} = A_{B',s}$ holds trivially, since both $\mathcal{F}^{\mathcal{A}}$ and $\mathcal{U}^{\mathcal{A}}$ leave those carrier sets untouched. For all operations $o\colon s_1 \cdots s_n \to s \in OP$ we show that $op_{B,o}(a_1, \ldots, a_n) = op_{B',o}(a_1, \ldots, a_n)$ as follows. Let $a_i \in A_{B,s_i}$ for $i = 1, \ldots, n$, and $a \in A_{UB,u}$ with $u = \overline{\sigma(o)}$ and $a_i = \pi_{u,i}(a)$. Such a exist due to the functionality and totality condition from Def. 3.25. We then have:

$$\begin{aligned} op_{B,o}(a_1, \ldots, a_n) &= op_{UB,\overline{o}}(a) \\ &= op_{B',o}(a_1, \ldots, a_n) \ . \end{aligned}$$

□

Lemma B.10. *Let USIG be an arbitrary uniform signature. Then, for any uniform USIG-algebra* B, *there exists an isomorphism* $g^{\mathcal{A}}_B\colon B \to \mathcal{F}^{\mathcal{A}}(\mathcal{U}^{\mathcal{A}}(B))$.

Proof. Let $USIG = \langle (D \cup U, F \cup \Pi), \prec \rangle$, $\mathcal{F}(\mathcal{U}(USIG)) = \langle (D' \cup U', F' \cup \Pi'), \prec' \rangle$, and let $g\colon USIG \to \mathcal{F}(\mathcal{U}(USIG))$ be the isomorphic uniform signature morphism of which we have proven the existence in Lemma 3.24. The isomorphism $g^{\mathcal{A}}_B\colon B \to \mathcal{F}^{\mathcal{A}}(\mathcal{U}^{\mathcal{A}}(B))$ is a family $g^{\mathcal{A}}_B = (g^{\mathcal{A}}_{B,s})_{s \in (D \cup U)}$, with

- for all $s \in D$ and all $a \in A_{B,s}$ we have: $g^{\mathcal{A}}_{B,s}(a) = a$ since those carrier sets are preserved by both $\mathcal{U}^{\mathcal{A}}$ and $\mathcal{F}^{\mathcal{A}}$;
- for all $s \in U$ and all $a \in A_{B,s}$ we have: $g^{\mathcal{A}}_{B,s}(a) = \langle a_1, \ldots, a_n \rangle$ where $a_i = \pi_{s,i}(a)$, for $1 \leq i \leq n$.

Note that for $g_B^{\mathcal{A}}$ to be an isomorphism, the functionality and the totality condition (from Def. 3.25) need to be satisfied. On the one hand, if the functionality condition is not satisfied their might exist a $u \in U$ and an $f \in F$ such that A_u contains two distinct elements, say a and a', such that $\pi_{u,k}(a) = \pi_{u,k}(a')$, for all $k = 1, \ldots, |\Pi_u|$ and $op_f(a) \neq op_f(a')$. This would mean that, in the algebra $g^{\mathcal{A}}(B)$, the operation $op_{g_{OP}(f)}$ is not functional on $\langle a_1, \ldots, a_k \rangle$, and therefore, $g^{\mathcal{A}}(B)$ is not an algebra. On the other hand, if the totality condition is not satisfied, the product carrier set $A_{g_S(u)}$ might contain more elements than the carrier set A_u it originates from, by which $g^{\mathcal{A}}$ cannot be an isomorphism.

Proving the correct preservation of the semantics of all operation requires to show that for all operation symbols $f\colon u \to d \in (F \cup \Pi)$ and all $a \in A_u$ it holds that

$$(g_{B,d}^{\mathcal{A}} \circ op_f)(a) \quad = \quad (op_{g_{OP}(f)} \circ g_{B,u}^{\mathcal{A}})(a) \ . \tag{B.9}$$

For showing the correct preservation of the semantics of the operations $f\colon u \to d \in (F \cup \Pi)$, the cases $f \in F$ and $f \in \Pi$ should be considered separately. When $f \in F$ we show the satisfaction of condition (B.9) for all $a \in A_{B,u}$ as follows. Let $a_i = \pi_{u,i}(a)$, for $1 \leq i \leq n$.

$$\begin{aligned}
& (g_{B,d}^{\mathcal{A}} \circ op_f)(a) \\
= \quad & \{ \text{ by definition of function composition } \} \\
& g_{B,d}^{\mathcal{A}}(op_f(a)) \\
= \quad & \{ \text{ by Def. 3.32 } \} \\
& op_{\mathcal{U}_{OP}(f)}(g_{s_1}^{\mathcal{A}}(a_1), \ldots, g_{s_n}^{\mathcal{A}}(a_n)) \\
= \quad & \{ \text{ by definition of } g_{B,s}^{\mathcal{A}} \text{ for sorts } s \in D \ \} \\
& op_{\mathcal{U}_{OP}(f)}(a_1, \ldots, a_n) \\
= \quad & \{ \text{ by definition Def. 3.29 } \} \\
& op_{\overline{\mathcal{U}_{OP}(f)}}(\langle a_1, \ldots, a_n \rangle) \\
= \quad & \{ \text{ by definition of } g_{OP} \ \} \\
& op_{g_{OP}(f)}(\langle a_1, \ldots, a_n \rangle) \\
= \quad & \{ \text{ by definition of } g_{B,u}^{\mathcal{A}} \ \} \\
& op_{g_{OP}(f)}(g_{B,u}^{\mathcal{A}}(a)) \\
= \quad & \{ \text{ by definition of function composition } \}
\end{aligned}$$

$$(op_{g_{OP}(f)} \circ g^{\mathcal{A}}_{B,u})(a)) \ .$$

For the case $f = p_{u,k} \in \Pi$ we show the satisfaction of condition (B.9) for all $a \in A_{B,u}$ as follows. Let $a_k = \pi_{u,k}(a) \in A_{\tau(f)}$ and $a'_k \in A_{\tau(f)}$, for $1 \leq k \leq |\Pi_u|$.

$$\begin{array}{ll} & (g^{\mathcal{A}}_{B,d} \circ op_f)(a) \\ = & \{ \text{ by definition of function composition } \} \\ & g^{\mathcal{A}}_{B,d}(op_f(a)) \\ = & \{ \text{ by letting } a_k = \pi_{u,k}(a) \ \} \\ & g^{\mathcal{A}}_{B,d}(a_k) \\ = & \{ \text{ by definition of } g^{\mathcal{A}}_{B,s} \text{ for sorts } s \in D \ \} \\ & a_k \\ = & \{ \text{ by definition of product carrier sets and projection operations } \} \\ & g_{OP}(f)(\langle a'_1, \ldots, a'_{k-1}, a_k, a'_{k+1}, \ldots, a_n \rangle) \\ = & \{ \text{ by selecting the tuple in which } a_i = \pi_{u,i}(a) \ \} \\ & g_{OP}(f)(\langle a_1, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_n \rangle) \\ = & \{ \text{ by definition of } g^{\mathcal{A}}_{B,s} \text{ for sorts } s \in U \ \} \\ & g_{OP}(f)(g^{\mathcal{A}}_{B,u}(a)) \ . \end{array}$$

Note that in the step in which we select the tuple $\langle a_1, \ldots, a_n \rangle$ we rely on the functionality condition in Def. 3.29 which ensures that there exists exactly one $a \in A_{B,u}$ for which $a_i = \pi u, k(a)$ for $1 \leq i \leq n$. Finally, we may conclude that $g^{\mathcal{A}}_B$ is an isomorphism from B to $\mathcal{F}^{\mathcal{A}}(\mathcal{U}^{\mathcal{A}}(B))$. □

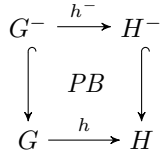
B.8 Proofs of Section 3.5

Theorem 3.49. AttrGraph$(\mathcal{E}_{\mathcal{F}(SIG)})$ *and* **UAttrGraph**$(\mathcal{F}(SIG))$ *are equivalent categories.*

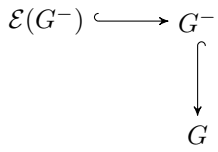
For this to be proven we need the following result.

Proposition B.11. *Let* **G** *be a full sub-category of* **Graph**.

1. *Every arrow* $h\colon G \to H$ *in* **REmb**(**G**) *corresponds uniquely to a pullback diagram in* **Graph** *of the following form:*

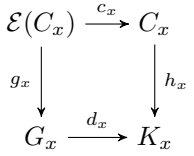


2. Let $\mathcal{E}\colon \mathbf{G} \to \mathbf{Graph}$ be an embedding functor. For every G in $\mathbf{REmb}(\mathcal{E})$, the following diagram has a pushout complement:



Proof sketch of Clause 2. The pushout complement H can be constructed by removing all nodes in $N_{G^-} \setminus N_{\mathcal{E}(G^-)}$ from G, together with their incident edges. (Note that those incident edges are not incident to any node in $N_G \setminus N_{G^-}$, due to the fact that G is glued over $\mathcal{E}(G^-)$; this is why no information is lost and G can be reconstructed from H and G^-.) The corresponding arrows are induced by the embeddings $\mathcal{E}(G^-) \subseteq H$ and $H \subseteq G$. □

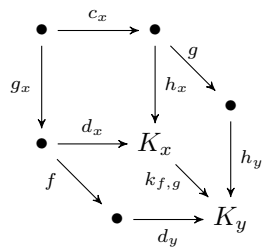
Proof of Theorem 3.49. In this proof, we abbreviate $\mathcal{E}_{\mathcal{F}(SIG)}$ to $\mathcal{E}$. For $x = (G_x, C_x)$ in $\mathbf{AttrGraph}(\mathcal{E})$ (implying $C_x \in \mathbf{AlgGraph}(\mathcal{F}(SIG))$), let $g_x\colon \mathcal{E}(C_x) \to G_x$ denote the embedding of the discrete graph $\mathcal{E}(C_x)$ into G_x, and let $c_x\colon \mathcal{E}(C_x) \to C_x$ be the embedding of $\mathcal{E}(C_x)$ into C. The embeddings g_x and c_x set up a span in **Graph**, from which we construct a pushout diagram as follows:



Now define $\mathcal{F}\colon \mathbf{AttrGraph}(\mathcal{E}) \to \mathbf{UAttrGraph}(\mathcal{F}(SIG))$ by mapping every object x to K_x, and every arrow $(f, g)\colon x \to y$ to the unique arrow $k_{f,g}$ from K_x to

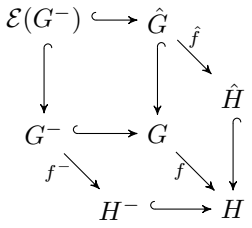
K_y such that the following diagram commutes:

(i)



In the other direction, define $\mathcal{U}\colon \mathbf{UAttrGraph}(\mathcal{F}(SIG)) \to \mathbf{AttrGraph}(()\mathcal{E})$ by mapping every object G of $\mathbf{UAttrGraph}(\mathcal{F}(SIG))$ to $(\hat{G}, G^-)$, where $\hat{G}$ is the pushout complement of the diagram $\mathcal{E}(G^-) \hookrightarrow G^- \hookrightarrow G$ (whose existence we stated in Proposition B.11.2). Every arrow $f\colon G \to H$ is mapped to the pair of arrows $(\hat{f}, f^-)$, where $\hat{f}\colon \hat{G} \to \hat{H}$ is the pullback of f along the morphism from $\hat{H}$ to H that is part of the pushout diagram for H.

(ii)



The reason why $\hat{f}$, constructed as the pullback of f over $\hat{H} \hookrightarrow H$, indeed starts in $\hat{G}$ is due to the fact that **Graph** is adhesive, in combination with Proposition A.15: if we complete diagram (ii) to a cube by also adding $\mathcal{E}(H^-)$ and $\mathcal{E}(f^-)$, we get a Van Kampen square of which all side faces are pullbacks; hence (due to adhesiveness) the top face is a pushout, and since (also due to adhesiveness) the pushout complement is unique it must be the case that $\mathsf{src}(\hat{f}) = \hat{G}$.

For the equivalence result, it remains to be proved that $\mathcal{F}$ and $\mathcal{U}$ are inverse modulo isomorphism. For the objects, this follows from the fact that $\mathcal{F}$ essentially constructs the pushout and $\mathcal{U}$ the pushout complement of a square in which the other two corners are fixed. For the arrows, note that also the diagram (i) can be completed to a Van Kampen cube; then the $\mathcal{F}$- and $\mathcal{U}$-constructions

construct the "front faces" by composition, respectively decomposition of the front down arrow ($k_{f,g}$ in diagram (i) and h in diagram (ii)); since, again, all other parts of the cube are fixed (modulo isomorphism), these constructions are indeed inverse to one another. □

TAAL Artifacts

This appendix includes the EBNF grammar that defines the concrete syntax of TAAL (Section 4.3.4).

C.1 TAAL EBNF Grammar

C.1.1 Non-terminals

Listing C.1 depicts the part of the TAAL EBNF grammar involving its non-terminals.

```
ParsedProgram ::=
    <PROGRAM_START> <STRING>
    <CURLY_OPEN> ParsedExpression <CURLY_CLOSE>
    ( ParsedTypeDecl )*
    <PROGRAM_END>

ParsedTypeDecl ::=
    <TYPE_START> <STRING> [ <EXTENDS> ParsedTypeRef ]
    ( ParsedVarDecl <SEMICOLON> | ParsedOperDecl )*
    <TYPE_END>

ParsedVarDecl ::=
    <STRING> <COLON> ParsedTypeRef [ <ASSIGN> ParsedExpression ]

ParsedOperDecl ::=
    <STRING>
    <BRACKET_OPEN> [ ParsedVarDecl ] ( <COMMA> ParsedVarDecl )* <
        BRACKET_CLOSE>
    [ <COLON> ParsedTypeRef ]
    [ <LOCALS> ( ParsedVarDecl <SEMICOLON> )* ]
    [ <CURLY_OPEN> ( ParsedStatement )* <CURLY_CLOSE> ]
```

```
ParsedBlockStat ::=
    <CURLY_OPEN> ( ParsedStatement )* <CURLY_CLOSE>

ParsedStatement ::=
    ParsedExpression [ <ASSIGN> ParsedExpression ] <SEMICOLON>
    | ParsedReturnStat <SEMICOLON>
    | ParsedConditionalStat
    | ParsedWhileStat
    | ParsedBlockStat

ParsedExpression ::=
    ( ParsedLitExp | ParsedCreateExp | ParsedPropCallExp )
    ( <DOT> ParsedPropCallExp )*

ParsedConditionalStat ::=
    <IF> condition = ParsedExpression
    <THEN> thenParParsedStatement
    [ <ELSE> elseParParsedStatement ]
    <ENDIF>

ParsedWhileStat ::=
    <WHILE> ParsedExpression <DO>
    ( ParsedStatement )*
    <ENDWHILE>

ParsedReturnStat ::=
    <RETURN> ParsedExpression

ParsedCreateExp ::=
    <NEW> ParsedTypeRef <BRACKET_OPEN> <BRACKET_CLOSE>

ParsedTypeRef ::=
    <STRING>

ParsedPropCallExp ::=
    <STRING>
    <BRACKET_OPEN>
    [ ParsedExpression ] ( <COMMA> actual = ParsedExpression )*
    <BRACKET_CLOSE>
    | <STRING>

ParsedLitExp ::=
    <STRINGLITERAL>
    | <NUMBERLITERAL>
    | <TRUE>
    | <FALSE>
    | <NULLLITERAL>
```

Listing C.1: Non-terminals in TAAL.

C.1.2 Terminals

Listing C.2 depicts the part of the TAAL EBNF grammar involving its terminals.

```
PROGRAM_START   ::= "program"
PROGRAM_END     ::= "endprogram"
TYPE_START      ::= "class"
TYPE_END        ::= "endclass"
EXTENDS         ::= "extends"
NEW             ::= "new"
ACTION          ::= "action"
LOCALS          ::= "locals"
IF              ::= "if"
THEN            ::= "then"
ELSE            ::= "else"
ENDIF           ::= "endif"
TRUE            ::= "true" | "TRUE"
FALSE           ::= "false" | "FALSE"
BRACKET_OPEN    ::= "("
BRACKET_CLOSE   ::= ")"
CURLY_OPEN      ::= "{"
CURLY_CLOSE     ::= "}"
COLON           ::= ":"
SEMICOLON       ::= ";"
COMMA           ::= ","
DOT             ::= "."
WHILE           ::= "while"
DO              ::= "do"
ENDWHILE        ::= "endwhile"
RETURN          ::= "return"
ASSIGN          ::= ":="
NULLLITERAL     ::= "null"
STRING          ::=
                    ["a"-"z", "A"-"Z", "_"]
                    ( ["a"-"z", "A"-"Z", "0"-"9", "_" ] )*
NUMBERLITERAL   ::=
                    ["0"-"9"] (["0"-"9"])*
                    ( "." ["0"-"9"] (["0"-"9"])* )?
                    ( ("e" | "E") ( "+" | "-" )?
                    ["0"-"9"] (["0"-"9"])* )?
STRINGLITERAL   ::=
                    // from Java 1.1 grammar
                    "\'"
                    (
                      (~["\'","\\","\n","\r"])
                        |
                        ("\\"
                          ( ["n","t","b","r","f","\\","\'","\""]
                            | ["0"-"7"] ( ["0"-"7"] (["0"-"7"])?
                          )?
                        )
                      )
                    )*
                    "\'"
```

Listing C.2: Terminals in Taal.

Graph Production System of the Leader Election Protocol

In this appendix we show our adapted implementation and erroneous implementation of the leader election protocol by Dolev et al. [56] in terms of graph production systems. First, in Section D.1, we recapitulate the original algorithm and the basic ideas. In Section D.2, we then discuss our way of specifying the different aspects in this protocol (Section D.2.1) and we show and explain the rules that constitute the behaviour of the protocol (Section D.2.2).

D.1 The Leader Election Protocol

Dolev et al. [56] have developed an protocol for the leader election problem in a unidirectional ring. The protocol assumes that processes communicate synchronously, thereby implying that the order of communication is preserved. Processes can be *active* and *passive*; passive processes merely act as communication relays, passing on all messages they receive. Next to its *id*, each process stores two values: *max* and *left*, both are integers. Initially, for each process p, we have $max(p) = id(p)$; the number $left(p)$ is used to store the number of the active process to p's left.

Processes can send two types of messages:

- in odd rounds, every active process p sends a message $\langle 1, i \rangle$,
- in even rounds, every active process p sends a message $\langle 2, i \rangle$,

where i has the value of either $max(p)$ or $left(p)$. The behaviour of the protocol can be specified by only focussing on active processes, since passive one only pass on the messages they receive. The behaviour of active processes is then specified as shown in Algorithm 6.

Algorithm 6 Behaviour of active processes p.

```
 1: procedure RUN
 2:     sendMessage(⟨1, max(p)⟩);
 3:     while true do
 4:         message ⟨i, j⟩ := receiveMessage();
 5:         if i == 1 then
 6:             if i ≠ max(p) then
 7:                 sendMessage(⟨2, i⟩);
 8:                 left(p) := i;
 9:             else
10:                 halt;              // the process p' with id(p') = max(p) is the leader
11:             end if
12:         end if
13:         if i == 2 then
14:             if left(p) > j and left(p) > max(p) then
15:                 max(p) := left(p);
16:                 sendMessage(⟨1, max(p)⟩);
17:             else
18:                 become passive;
19:             end if
20:         end if
21:     end while
22: end procedure
```

D.2 A Graph Transformation Implementation

In this section we describe our graph transformation implementation of the leader election protocol described in Section D.1.

D.2.1 Start Graph

An example start graph for our graph transformation implementation of the leader election protocol is depicted in Fig. D.1. We model processes as Process-labelled nodes. Processes can be *active* or *passive*, which is indicated by

additional passive and active labels on the Process-nodes. The identifiers each Process has to pick are at first collected by a single node labelled Numbers through number-labelled edges. The numbers *left* and *max* are encoded by value nodes pointed to by equally labelled edges. Initially, the *left*-value of each Process is set to -1.

The fact that all Processes communicate synchronously is encoded by a Scheduler-node. Initially, the Scheduler has an outgoing init-edge to every Process; Processes with an incoming init-edge have not yet send their initial message along the ring. In the remaining rounds, the Scheduler will have a go-edge to Processes that have not yet send their message in the current round.

Although it does not influence the correctness of the protocol, the graph also contains a counter Round, which keeps track of the current round of the protocol. In states in which the protocol reaches a final configuration, i.e., when one Process has been elected as the leader, the value of this counter tells us how many rounds were needed for the election.

We have slightly adapted the protocol as proposed by Dolev et al. The difference is that we have included some rules that change the initial ring-configuration. Changing the ring-configuration means that either extend the ring with an additional Process or delete one Process from the ring. To ensure that the ring-configuration does not change during an election, we have included a Clock-node in the graph. It is only allowed to change the ring-configuration as long as the Clock has not yet started ticking. A ticking Clock has a self-edge labelled ticking.

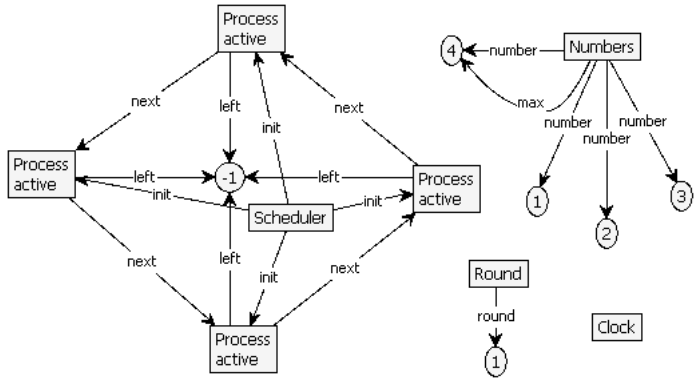
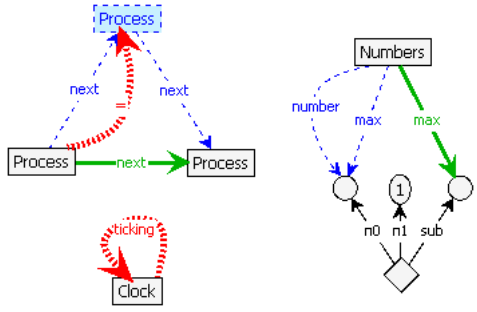


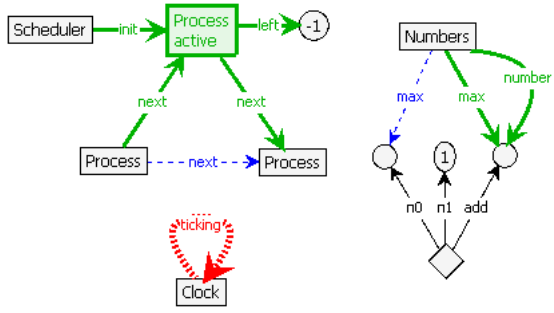
Figure D.1: Start graph representing the configuration of a ring with 4 processes.

D.2.2 Rules

The rules depicted in Fig. D.2 specify how the configuration of the ring can be changed; applications of the new-process-rule [del-process-rule] increase [decrease] the size of the ring.



(a) Rule del-process.



(b) Rule new-process.

Figure D.2: Transformation rules to change the ring configuration.

The rule pick-number, shown in Fig. D.3, specifies the behaviour of a Process picking its unique identifier from the set of identifiers still available. As soon as one Process has selected its identity, the Clock starts ticking. This rule also implements the requirement that for every Process p, its value $max(p)$ is initially equal to its identity.

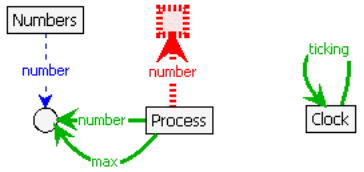


Figure D.3: Rule pick-number.

Sending the first message is specified by the init-rule (Fig. D.4). Every Process p sends a message $\langle 1, max(p)\rangle$ to its successor Process.

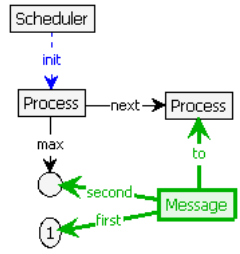


Figure D.4: Rules init.

Passive Processes only pass on the messages they receive. This is specified by the passive-rule, depicted in Fig. D.5.

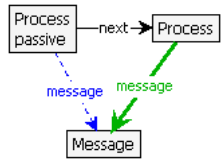
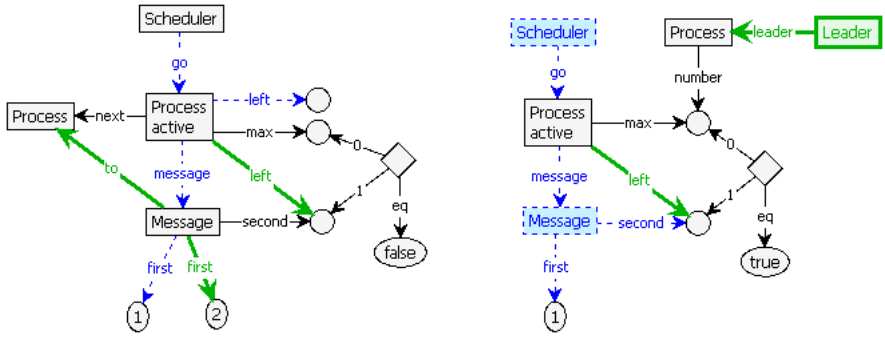


Figure D.5: Rule passive.

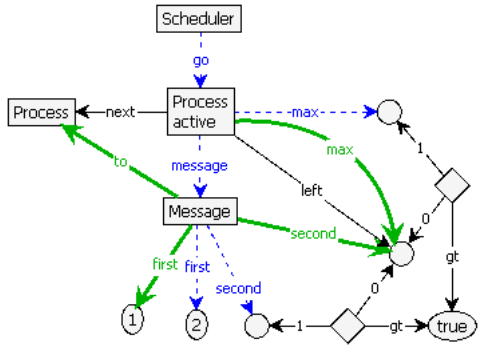
The rules shown in Fig. D.6 specify the behaviour of active Processes when a message $\langle 1, j \rangle$ has arrived. The go-1-1-rule (Fig. D.6(a)) corresponds to the case dealt with in lines 7–8 of Algorithm 6. Halting of a process due to the fact that a leader has been elected (line 10) is specified by the go-1-2-rule (Fig. D.6(b)).



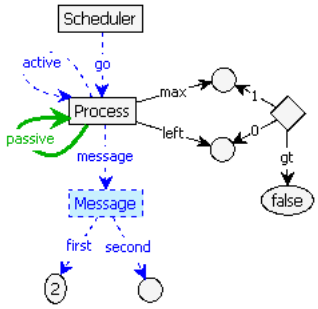
(a) Rule go-1-1 **(b)** Rule go-1-2

Figure D.6: Sending messages in the first sub-round.

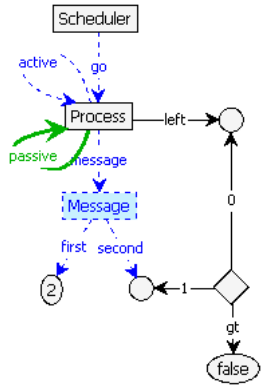
Messages of the form $\langle 2, j \rangle$ are processed by the rules shown in Fig. D.7. The go-2-1-rule (Fig. D.7(a)) corresponds to lines 15–16; the rules go-2-2a (Fig. D.7(b)) and go-2-2b (Fig. D.7(c)) specifies the two situations in which a Process becomes passive (line 18).



(a) Rule go-2-1



(b) Rule go-2-2a



(c) Rule go-2-2b

Figure D.7: Sending messages in the second sub-round.

At the end of every odd and even round, the messages are physically passed on. This is specified by the pass-message-rule.

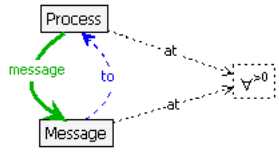


Figure D.8: Rule pass-message.

If a leader has been elected the leader-rule (Fig. D.9) is applicable.

Figure D.9: Rule leader.

The next round is initiated by creating a fresh go-edge from the Scheduler to every individual Process, as specified by the next-phase-rule (Fig. D.10).

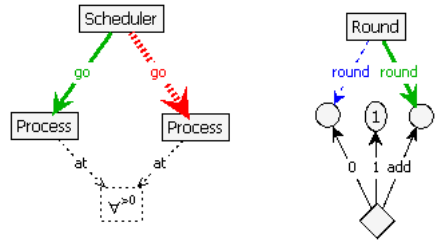


Figure D.10: Rule next-phase.

D.2.3 Priorities

The rules in our leader election protocol graph production system have different priorities to ensure that the rules correctly implement the protocol. In Table D.1, we have listed the priorities of each of the rules.

Priority	Rules
10	del-process, new-process, and pick-number
5	init and passive
2	go-1-1, go-1-2, go-2-1, go-2-2a, and go-2-2b
1	pass-message
0	leader and next-phase

Table D.1: Rule priorities for the rules specifying the leader election protocol.

D.3 Artificial Error

In one of the experiments we verify an implementation of the leader election protocol that contains an artificial error. Fig. D.11 depicts the rule that causes the error. The effect of this rule is that the Scheduler-node is deleted from the graph. As a result, all init or go-edges will be deleted as well, thereby bringing the system in a deadlock state without a leader being elected.

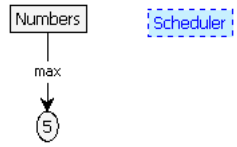


Figure D.11: Rule destroy.

E Proofs of Chapter 6

E.1 Proofs of Section 6.3

Proposition 6.5. *Let v, w, w_1, and w_2 be words. Then,*

1. *If v is feasible and $v \simeq w$, then w is feasible;*
2. *$\simeq$ is symmetric over the set of feasible words;*
3. *$v{\cdot}w_1 \simeq v{\cdot}w_2$ if and only if $w_1 \simeq w_2$.*

Proof.

1. Proved by induction on the generating clause of $\simeq$. Assume $v = v_1{\cdot}a{\cdot}b{\cdot}v_2$ is feasible and $a \not\rightsquigarrow b$; we prove that then $w = v_1{\cdot}b{\cdot}a{\cdot}v_2$ is feasible.

   Assume that $c{\cdot}\bar{w}{\cdot}d$ is a subword of w and $d \rightsquigarrow c$. If it is a sub-word of either v_1 or v_2, we are trivially done. Otherwise, we have to recognize the following cases.

   - v_1 ends on $c{\cdot}\bar{w}$ and $d = b$. Then $c{\cdot}\bar{w}{\cdot}a{\cdot}d$ is a sub-word of v, hence there is an influence chain through $\bar{w}{\cdot}a$. From $a \not\rightsquigarrow b$ we may conclude that the forward chain of influence from c to d in v remains in w, which satisfies the proof obligation.
   - $\bar{w} = \bar{w}'{\cdot}b$ such that v_1 ends on $c{\cdot}\bar{w}'$ and $d = a$. Then $c{\cdot}\bar{w}'{\cdot}d$ is a sub-word of v, hence there is a chain of influences through $\bar{w}$. But then this same chain also goes through $\bar{w}$, hence the proof obligation is satisfied.

- $\bar{w} = \bar{w}' \cdot b \cdot a \cdot \bar{w}''$ such that v_1 ends on $c \cdot \bar{w}'$ and v_2 starts with $\bar{w}'' \cdot d$. Then $c \cdot \bar{w}' \cdot a \cdot b \cdot \bar{w}'' \cdot d$ is a sub-word of v, hence there is a chain of influences through $\bar{w}' \cdot a \cdot b \cdot \bar{w}''$. From $a \not\rightsquigarrow b$ we know that a and b cannot both be in any forward chain of influence, hence this chain remains in $\bar{w}$, thereby satisfying the proof obligation.
- $\bar{w} = a \cdot \bar{w}'$ such that $c = b$ and v_2 starts with $\bar{w}' \cdot d$. Then $c \cdot \bar{w}' \cdot d$ is a sub-word of v, hence there is a chain of influences through $\bar{w}$. But then this chain also exists in $\bar{w}$, satisfying the proof obligation.
- $c = a$ and v_2 starts with $\bar{w} \cdot d$. Then $c \cdot b \cdot \bar{w} \cdot d$ is a sub-word of v, hence there is a chain of influences through $b \cdot \bar{w}$. Since $a \not\rightsquigarrow b$, b cannot be part of this chain; hence this chain also goes through $\bar{w}$, satisfying the proof obligation.

2. It suffices to show that the generating rule of $\simeq$ is symmetric. Indeed, if $v \cdot a \cdot b \cdot w \simeq v \cdot b \cdot a \cdot w$ due to the fact that $a \not\rightsquigarrow b$, then due to the feasibility of the word also b does not influence a and hence $v \cdot b \cdot a \cdot w \simeq v \cdot a \cdot b \cdot w$.

3. The proof relies on the fact that any derivation of $v \cdot w_1 \simeq v \cdot w_2$ can be transformed into a derivation of $w_1 \simeq w_2$, by omitting all steps of the proof that swap actions originally from v. □

Proposition 6.15. *Every Ent-based transition system is dependency complete and consistent, and has only feasible words as traces.*

Proof. For proving that every *Ent*-based transition system is dependency complete and consistent we have to show that all properties of Def. 6.7 are satisfied. Let q be a state and a, b be actions.

- For requirement (6.1) we have the following. Given that $q \vdash a$ and $a \rhd b$ we have to recognize the following cases:
  - $C_a \cap (R_b \cup D_b) \neq \emptyset$: in this case, action b reads or deletes entities created by a. Obviously, this occurrence of b cannot be executed from q, i.e., $q \nvdash b$.
  - $D_a \cap (C_b \cup N_b) \neq \emptyset$: in this case, action b creates or forbids entities deleted by a. Again, this occurrence of b can therefore not be executed from q, i.e., $q \nvdash b$.
- For requirement (6.2) we have the following. Given that $q \vdash a$ and $a \blacktriangleleft b$ we have to recognize the following cases:

  - $D_a \cap (R_b \cup D_b) \neq \emptyset$: in this case, action b reads or deletes entities deleted by a. Obviously, this occurrence of b cannot be executed after from $q \uparrow a$, i.e., $q \nvdash a{\cdot}b$.
  - $C_a \cap (C_b \cup N_b) \neq \emptyset$: in this case, action b creates or forbids entities created by a. Also here, this occurrence of b can therefore not be executed from $q \uparrow a$, i.e., $q \nvdash a{\cdot}b$.

- For requirement (6.3) we have the following. Given that $q \vdash a$ and $a \not\triangleright b$ we have to recognize the following cases:
  - $C_a \cap (R_b \cup D_b) = \emptyset$: in this case, action b does not read or deleted entities just created by a. Therefore, this occurrence of b can also be executed from q, i.e., $q \vdash b$.
  - $D_a \cap (C_b \cup N_b) = \emptyset$: in this case, action b does not create or forbid entities deleted by a. This means that b can as well be executed from q, i.e., $q \vdash b$.

- For requirement (6.4) we have the following. Given that $q \vdash a$, $q \vdash b$, and $a \not\blacktriangleleft b$ we have to recognize the following cases:
  - $D_a \cap (R_b \cup D_b) = \emptyset$: in this case, actions a and b are both enabled in q and b does not require the existence of entities deleted by a. Thus, after executing a from q, action b is still enabled, i.e., $q \vdash a{\cdot}b$.
  - $C_a \cap (C_b \cup N_b) = \emptyset$: in this case, again actions a and b are both enabled in q and b does not require the non-existence of elements created by a. Therefore, after executing a from q, action b is still enabled, i.e., $q \vdash a{\cdot}b$.

For proving that all traces of *Ent*-based transition systems are feasible words we have to show that every trace fulfills the requirement from Def. 6.3. Let w be a word, and $a{\cdot}v{\cdot}b$ be a sub-word of w with $b \rightsquigarrow a$. This means that either $b \triangleright a$ or $a \blacktriangleleft b$. In the latter case, it holds that $D_a \cap (R_b \cup D_b) \neq \emptyset$ or $C_a \cap (C_b \cup N_b) \neq \emptyset$. In either case, action b is not enabled directly after executing a. For b to be enabled after the sub-word $a{\cdot}v$, the sub-word v must contain some or multiple actions that undo the effect of a which disabled b. Those actions then form the chain of forward influence. In the case $b \triangleright a$ we have that $C_b \cap (R_a \cup D_a) \neq \emptyset$ or $D_b \cap (C_a \cup N_a) \neq \emptyset$. Similarly, some or multiple actions in v must ensure that all preconditions for b that are not yet fulfilled directly after executing b are satisfied. Those actions are then part of the chain of forward influence.

□

E.2 Proofs of Section 6.4.3

Lemma 6.23. *Let u, v, and w be words and a be some action. Then,*

1. *If $u \precsim v$, then $u{\cdot}w - v = w - (v - u)$.*
2. *If $v \precsim {\downarrow_a}w$ then ${\downarrow_a}w \simeq v{\cdot}{\downarrow_a}(w - v)$.*
3. *If $v - w \not\rightarrow a$ then ${\downarrow_a}v \simeq {\downarrow_a}w$.*

For proving the second property of Lemma 6.23 we need the following auxiliary results.

Lemma E.1. *If $a{\cdot}v \simeq u{\cdot}w$ with $a \in A_u$ then $u \simeq a{\cdot}u'$ for some word u'.*

For Lemma E.1 to be proved we introduce *permutation functions*. Given a feasible word w a permutation function $f_w{:}\, \mathbb{N} \to \mathbb{N}$ for w is a mapping from indices (or positions) in w to new indices representing the new position of the action in w at index i, such that the obtained word, say $\bar{w}$, is equivalent to w, i.e., $w \simeq \bar{w}$. Permutation functions satisfy the following property.

Lemma E.2. *Let w be a feasible word and f be a permutation function for w. We then have for all indices $x, y < |w|$:*

$$x < y \;\wedge\; f(x) > f(y) \quad\Longrightarrow\quad w(x) \not\rightarrow w(y) \;\wedge\; w(y) \not\rightarrow w(x)$$

where $w(x)$ denotes the action in w at index x.

Proof. Assume f transforms the word w into the equivalent word $\bar{w}$. From the definition of the equivalence $\simeq$ (recall Def. 6.4) we know that there exists a sequence of words $w_1, w_2, \ldots, w_n$ such that

$$w_1 \simeq w_2 \simeq \cdots \simeq w_n \;\;,$$

with $w = w_1$ and $\bar{w} = w_n$ and all w_i feasible due to Proposition 6.5 part 1. Every two successive words w_i and w_{i+1} only differ in the ordering of two successive and independent actions, i.e., $w_i = w'{\cdot}a{\cdot}b{\cdot}w'' \simeq w'{\cdot}b{\cdot}a{\cdot}w'' = w_{i+1}$, for some words w' and w'', actions a and b, with $a \not\rightarrow b$ and $b \not\rightarrow a$. For such a sequence of equivalent words we can construct a sequence of permutation functions $f_1, f_2, \ldots, f_n$ such that f_i transforms w into w_i and $f = f_n$, for $1 \leq i \leq n$.

We prove the property by induction in the index of the permutation function in the sequence $f_1, f_2, \ldots, f_n$.

Base case: in this case we have $i = 1$ and since f_1 transforms w in to itself, the property is satisfied vacuously (there are no indices x and y such that $x < y$ and $f_1(x) > f_1(y)$).

Induction step: assume the property holds for f_i, for all $i < n$. Suppose $w_i = w' \cdot a \cdot b \cdot w''$ and $w_{i+1} = w' \cdot b \cdot a \cdot w''$ with $a \not\rightsquigarrow b$ and $|w'| = k$. Due to feasibility of w_{i+1}, it follows that $b \not\rightsquigarrow a$. The index j, $f_{i+1}(j)$ is then defined as follows:

$$f_{i+1}(j) = \begin{cases} f_i(j) & \text{if} \quad f_i(j) \leq k \text{ or } f_i(j) > k+2 \\ f_i(j)+1 & \text{if} \quad f_i(j) = k+1 \\ f_i(j)-1 & \text{if} \quad f_i(j) = k+2 \ . \end{cases}$$

Now take two indices x, y such that $x < y$ and $f_{i+1}(x) > f_{i+1}(y)$. To prove: $w_i(x) \not\rightsquigarrow w_i(y)$ and $w_i(y) \not\rightsquigarrow w_i(x)$. For this to be proved we distinguish the following cases:

- $f_i(x) \leq k$: in this case we have $f_{i+1}(x) = f_i(x)$. From $f_{i+1}(x) > f_{i+1}(y)$ we have $f_{i+1}(y) < f_{i+1}(x) \leq k$ which implies $f_{i+1}(y) = f_i(y)$. Thus, $f_i(x) > f_i(y)$ for which we can apply the induction hypothesis.
- $f_i(x) = k+1$: in this case we have $f_{i+1}(x) = k+2$. We distinguish two sub-cases:
  - $f_{i+1}(y) = k+1$: this implies $w_i(x) = a$ and $w_i(y) = b$ of which we know (due to feasibility) $w_i(x) \not\rightsquigarrow w_i(y)$ and $w_i(y) \not\rightsquigarrow w_i(x)$.
  - $f_{i+1}(y) \leq k$: in this sub-case we have $f_i(y) = f_{i+1}(y) \leq k < k+1 = f_i(x)$. Now, we can again apply the induction hypothesis.
- $f_i(x) = k+2$: in this case we have $f_{i+1}(x) = k+1$. Additionally we have $f_{i+1}(y) \leq k$. Therefore, $f_i(y) = f_{i+1}(y) \leq k \leq k+2 = f_i(x)$ for which we can apply the induction hypothesis.
- $f_i(x) > k+2$: in this final case we have $f_{i+1}(x) = f_i(x) > k+2$. Furthermore, $f_{i+1}(y) \leq k+2$ and therefore $f_i(y) \leq k+2 < f_i(x)$ in which case we can also apply the induction hypothesis. □

Based on Lemma E.2 we can prove Lemma E.1.

Proof of Lemma E.1. From $a \cdot v \simeq b \cdot w$ and $a \neq b$ we know that there exist words v', v'' such that $v = a \cdot v' \cdot b \cdot v''$ with $b \notin A_{v'}$. Since $a \cdot v' \cdot b \cdot v'' \simeq b \cdot w$, there exists a permutation function f that transforms $a \cdot v' \cdot b \cdot v''$ in to $b \cdot w$. Suppose $|a \cdot v' \cdot b| = i$, i.e., b is at index i of the word $a \cdot v' \cdot b \cdot v''$. We then have $f(i) = 1$. Obviously, for all indices j with $j < i$ it holds that $f(j) > f(i)$. And therefore we may

conclude that for all actions $c \in A_{a \cdot v'}$ we have that $c \not\rightsquigarrow b$ and $b \not\rightsquigarrow c$. From this we may conclude that $a \cdot v \simeq b \cdot a \cdot v' \cdot v''$. □

We now can prove Lemma 6.23 as follows.

Proof of Lemma 6.23 part 1. The property can be proved by induction on the length of u.

Base case: If $|u| = 0$, we have $u = \varepsilon$. Obviously, $\varepsilon \cdot w - v = w - v = w - (v - \varepsilon)$.

Induction step: Assume the property holds whenever $|u| \leq n$, for some $n < |v|$. Let $\bar{v} = v - u$ (i.e., $v \simeq u \cdot \bar{v}$) and $\bar{w} = u \cdot w - v$ (i.e., $u \cdot w \simeq v \cdot \bar{w}$). Without loss of generality, assume $u' \simeq u \cdot a$ for some action $a \notin A_u$ which implies that $w \simeq a \cdot w'$ for some word w'. That is to say, u' contains one action from w, thereby shortening w to w'. Now, let $v \simeq u \cdot a \cdot \bar{v}'$, for some word $\bar{v}'$. The property can then be proved as follows:

$$\begin{aligned} u' \cdot w' - v &= u \cdot a \cdot w' - v && \text{(E.1)} \\ &= a \cdot w' - (v - u) && \text{(E.2)} \\ &= a \cdot w' - (u \cdot a \cdot \bar{v}' - u) \\ &= a \cdot w' - (a \cdot \bar{v}') \\ &= w' - \bar{v}' \\ &= w' - (u \cdot a \cdot \bar{v}' - u \cdot a) \\ &= w' - (v - u') \ . \end{aligned}$$

Note that the step from the right-hand-side of (E.1) to (E.2) applies the induction hypothesis.

□

Lemma E.3. *Let u, v, w be words. If $w - v$ is defined, then $u \cdot w - u \cdot v \simeq v - w$.*

Proof. This follows immediately from Lemma 6.23 part 1. □

Lemma E.4. *Let w be a word and $a, \bar{a}$ be actions. Then $\bar{a} \in A_{\downarrow_a(\bar{a} \cdot w)}$ implies $\bar{a} \cdot \downarrow_a w \simeq \downarrow_a(\bar{a} \cdot w)$.*

Proof. The proof consist of showing that for $\bar{a} \downarrow_a w$ both requirements of Def. 6.17 are fulfilled, thereby implying that $\bar{a} \downarrow_a w \precsim \downarrow_a(\bar{a} \cdot w)$, and since $|\bar{a} \downarrow_a w| = |\downarrow_a(\bar{a} \cdot w)|$ we may conclude $\bar{a} \downarrow_a w \simeq \downarrow_a(\bar{a} \cdot w)$. When proving both requirements, let $\bar{w} = \bar{a} \cdot w$.

The first requirement of Def. 6.17 requires to prove that $(\bar{w} - \bar{a}{\cdot}{\downarrow_a}w) \not\rightsquigarrow a$. This follows immediately from the fact that $(\bar{w} - \bar{a}{\cdot}{\downarrow_a}w) \simeq (\bar{a}{\cdot}w - \bar{a}{\cdot}{\downarrow_a}w) \simeq (w - {\downarrow_a}w)$ (which holds due to Lemma E.3).

The second requirement of Def. 6.17 can be proved as follows. Let v be an arbitrary word such that $v \precsim \bar{a}{\cdot}w$. From $v \precsim \bar{a}{\cdot}w$ we know $\exists u : \bar{a}{\cdot}w \simeq v{\cdot}u$. We now have to prove that $u \not\rightsquigarrow a$ implies $\bar{a}{\cdot}{\downarrow_a}w \precsim v$. Combining $\bar{a}{\cdot}w \simeq v{\cdot}u$ with Lemma E.1 we obtain $\exists v' : \bar{a}{\cdot}w \simeq \bar{a}{\cdot}v'{\cdot}u$. Then $(\bar{w} - v) \simeq (\bar{a}{\cdot}w - \bar{a}{\cdot}v') \simeq (w - v')$. Now, $u \simeq (w - v') \not\rightsquigarrow a$ implies ${\downarrow_a}w \precsim v'$ and therefore we have $\bar{a}{\cdot}{\downarrow_a}w \precsim \bar{a}{\cdot}v' \simeq v$. □

Proof of Lemma 6.23 part 2. This property can be proved by induction on the length of v.

Base case: If $|v| = 0$ then $v = \varepsilon$, and trivially $\varepsilon \precsim w$. Also $\varepsilon{\cdot}{\downarrow_a}w - \varepsilon \simeq {\downarrow_a}w$.

Induction step: Assume the property holds whenever $|v| \leq n$, for some $n < |w|$. Let $\bar{v} = {\downarrow_a}w - v$ (i.e., ${\downarrow_a}w \simeq v{\cdot}\bar{v}$). The fact that $v \precsim {\downarrow_a}w$ means $\exists w' : {\downarrow_a}w \simeq v{\cdot}w'$, and ${\downarrow_a}w \precsim w$ means $\exists w'' : w \simeq {\downarrow_a}w{\cdot}w''$. We thus have $w \simeq v{\cdot}w'{\cdot}w''$. Now, let $v' = v{\cdot}\bar{a} \precsim {\downarrow_a}w$ with $\bar{a} \notin v$ and ${\downarrow_a}w - v' = \bar{v}'$.

$$
\begin{aligned}
{\downarrow_a}w &= v{\cdot}{\downarrow_a}(w - v) && \text{(E.3)}\\
&= v{\cdot}{\downarrow_a}(v{\cdot}w'{\cdot}w'' - v)\\
&= v{\cdot}{\downarrow_a}(w'{\cdot}w'')\\
&= v{\cdot}{\downarrow_a}(\bar{a}{\cdot}\bar{w}'{\cdot}w'') && \text{(E.4)}\\
&= v{\cdot}\bar{a}{\cdot}{\downarrow_a}(\bar{w}'{\cdot}w'') && \text{(E.5)}\\
&= v{\cdot}\bar{a}{\cdot}{\downarrow_a}(v{\cdot}\bar{a}{\cdot}\bar{w}'{\cdot}w'' - v{\cdot}\bar{a})\\
&= v{\cdot}\bar{a}{\cdot}{\downarrow_a}(w - v{\cdot}\bar{a}) \ .
\end{aligned}
$$

The step from (E.4) to (E.5) is correct due to Lemma E.4.

□

Proof of Lemma 6.23 part 3. If $v - w$ is defined we have that $w \precsim v$. We prove the property by induction on the length of w.

Base case: If $|w| = 0$ (i.e., $w = \varepsilon$), $v - w = v - \varepsilon = v$. But then $v - w \not\rightsquigarrow a$ means that $v \not\rightsquigarrow a$, which implies ${\downarrow_a}v = {\downarrow_a}w = \varepsilon$.

Induction step: Assume the property holds whenever $|w| = n$, for some $n \in I\!N$. Now let w' be a word such that, without loss of generality, $w' \simeq w{\cdot}\bar{a}$

(i.e., $|w'| = |w| + 1$) and $v - w'$ being defined (which implies $w' \precsim v$). We can now distinguish the following two cases:

- $\downarrow_a w' \not\simeq \downarrow_a w$: in this case the action $\bar{a}$ must also be part of v, otherwise $v - w'$ is not defined. The case assumption implies that $\bar{a} \in A_{v-w}$. However, this causes a contradiction with the assumption that $v - w \not\rightsquigarrow a$. Apparently, $\downarrow_a w' \simeq \downarrow_a w$.
- $\downarrow_a w' \simeq \downarrow_a w$: in this case clearly $\bar{a}$ does not influence a. We thus have $\downarrow_a w' \simeq \downarrow_a w$ for which the induction hypothesis then provides the proof obligation that $\downarrow_a w' \simeq \downarrow_a v$. □

Lemma 6.24. *Let P be a fair probing. For all $(q, w) \in dom(P)$ and $a \in enabled(q \uparrow w)$, there is a vector $(q', w') \in dom(P)$ such that $q \xrightarrow{v} q'$ for some $v \precsim \downarrow_a w$, and $q \uparrow w \xrightarrow{u \cdot a} q' \uparrow w'$ for some $u \not\rightsquigarrow a$.*

Proof. By induction on $n_{q,w}(a)$. For brevity we write $p = p_{q,w}$ for the moment.

Base case. If $n_{q,w}(a) = 0$, it must be the case that $a \in dom(p)$, hence $u = \varepsilon$, $v = p(a)$ and $(q', w') = (q, w) \uparrow p(a)$ satisfy the requirements.

Induction step. Assume the property holds whenever $n_{q,w}(a) \leq m$ for a certain m, and assume $n_{q,w}(a) = m + 1$. If $a \in dom(p)$ then we are done immediately. Otherwise let $b \in dom(p)$ be the action such that $n_{(q,w) \uparrow p(b)}(a) < n_{q,w}(a)$, guaranteed by fairness. By Clause 2 of Def. 6.20, it follows that $p(b) \precsim \downarrow_a w$ (otherwise $a \in dom(p)$). Let $q'' = q \uparrow p(b)$ and $w'' = (w - p(b)) \cdot b$, then by the fact that P is a family of probe sets it follows that $(q'', w'') = (q, w) \uparrow p(b) \in K$.

By the induction hypothesis applied to (q'', w''), we have that there is a vector $(q', w') \in K$ such that $q'' \xrightarrow{v} q'$ for some $v \precsim \downarrow_a w''$, and $q'' \uparrow w'' \xrightarrow{u \cdot a} q' \uparrow w'$ for some $u \not\rightsquigarrow a$.

Taking both steps together, we see that $q \xrightarrow{p(b) \cdot v} q'$ and $q \uparrow w \xrightarrow{b \cdot u \cdot a} q' \uparrow w'$. Moreover, Lemma 6.23.1 implies that $p(b) \cdot v \precsim \downarrow_a w$ due to $p(b) \precsim \downarrow_a w$ and $v \precsim \downarrow_a (w - p(b)) \cdot b = \downarrow_a (w - p(b))$ (the latter equality due to $b \not\rightsquigarrow a$); and finally, $b \cdot u \not\rightsquigarrow a$ due to $b \not\rightsquigarrow a$ and $u \not\rightsquigarrow a$. This establishes the proof obligation. □

□

E.3 Proofs of Section 6.5.1

Proposition 6.29. *Let (q, v) be a vector with v reversing free.*

1. *For any action a, $q \vdash v'{\cdot}a$ with $v' \precsim v$ implies $\downarrow_a v \precsim v'$.*
2. *$fma(q, v) = \{\downarrow_a v{\cdot}a \mid a \in pma(q, v) \wedge q \vdash \downarrow_a v{\cdot}a\}$.*

Proof.

1. This property can be proved by induction on the length of v'.

   Base case: If $|v'| = 0$, i.e., $v' = \varepsilon$, $q \vdash a$ implies $\downarrow_a v = \varepsilon$. Indeed, $\downarrow_a v = \varepsilon \precsim \varepsilon = v'$.

   Induction step: Suppose the property holds whenever $|v'| \leq n$ for some $n < |v|$. Let $v'' = v'{\cdot}b$ for some action $b \in A_v$. To prove: $q \vdash v'{\cdot}b{\cdot}a$ with $v'{\cdot}b \precsim v$ implies $\downarrow_a v \precsim v'{\cdot}b$. We can now distinguish two cases:

   - $b \in A_{\downarrow_a v}$: this case causes a contradiction. If $b \in A_{\downarrow_a v}$, this means that b is necessary for a to be enabled. The fact that v is reversing free means that v (and therefore also v') does not contain other actions that stimulate a like b does. This contradicts with the assumption that $q \vdash v{\cdot}a$.
   - $b \notin A_{\downarrow_a v}$: in this case we have that b does not influence the enabledness of a and thus $q \vdash v'{\cdot}a$. Given $v'{\cdot}b \precsim v$ which clearly implies $v' \precsim v$. The induction hypothesis then provides the fact that $\downarrow_a v \precsim v'$. This clearly implies $\downarrow_a v \precsim v'{\cdot}b$.

2. Given (q, v) a vector (i.e., $q \vdash v$) with v reversing free, $a \in pma(q, v)$, and $q \vdash \downarrow_a v{\cdot}a$. Let $v = v'{\cdot}b$, for some action $b \in A_v$. We now distinguish two cases, according to Def. 6.27:

   - the case in which $q \vdash v'{\cdot}a$ and $b \blacktriangleleft a$: if in addition $b \in A_{\downarrow_a v}$ this would contradict with v reversing free and $q \vdash \downarrow_a v{\cdot}a$ since v (and therefore also $\downarrow_a v$) does not contain any action that undoes the reason why b disables a. Thus, it must hold that $b \notin A_{\downarrow_a v}$ which implies that $\downarrow_a v' = \downarrow_a v$. Since $q \vdash v'{\cdot}a$ this means that a is enabled in (q, v') and thus a is indeed a *fresh* missed action in (q, v).
   - the case in which the following conditions hold:

     (a) $q \Uparrow v \vdash a$ and $q \uparrow v \nvdash a$;
     (b) $\exists c \in A_v : c \blacktriangleleft a$;

(c) $b \rhd a$.

The fact that a is a missed action follows directly from the fact that $q \nvdash v{\cdot}a$ and $q \vdash \downarrow_a v{\cdot}a$. The action a also being freshly missed follows from combining $b \rhd a$ with v being reversing free, since that means that there does not exists a word $v'' \precsim v'$ such that $q \vdash v''{\cdot}a$ which implies that a is not missed along (q, v').

□

Samenvatting

Het (niet-)correct functioneren van velerlei software systemen heeft een grote invloed op hoe wij ons dagelijks leven kwalificeren. Zowel software bedrijven als academische onderzoeksgroepen in de informatica besteden veel inspanning aan het toepassen en ontwikkelen van technieken ter verbetering van de correctheid van software systemen. In dit proefschrift richten we ons op het gebruiken en ontwikkelen van graaf gebaseerde technieken voor het specificeren en verifiëren van software systemen in het algemeen, en object georiënteerde system in het bijzonder. We werken twee manieren om de correctheid (en daarmee de kwaliteit) van dergelijke te verbeteren in detail uit. Aan de ene kant onderzoeken we het potentieel van het gebruik van de graaf transformatie techniek voor het formeel specificeren van de dynamische semantiek van (object georiënteerde) programmeertalen. Aan de andere kant ontwikkelen we technieken voor het verifiëren van systemen waarvan het gedrag is gespecificeerd als graaf productie systemen. De meeste technieken die in dit werk zijn ontwikkeld, zijn geïmplementeerd in de GROOVE Tool Set.

Vaak worden systeemtoestanden geïdentificeerd door middel van de waarden die zijn toegekend aan toestand-variabelen (vaak zijn dat primitieve data typen zoals integers en Booleans). In het object georiënteerde paradigma kunnen objecten (dat zijn instanties van klassen) worden gezien als toestand-variablen waarvan de interne toestand afhankt van de waarden van de attributen (ook wel velden). We beginnen met het introduceren van een uniform raamwerk voor het specificeren and transformeren van geattribueerde grafen. In dit raamwerk worden geattribueerde grafen gespecificeerd louter in termen van graaf structuren en graaf morfismen. Een van de belangrijkste voordelen van zo een aanpak is dat het de inspanning voor het implementeren van een graaf transformator

voor geattribueerde grafen reduceerd in vergelijking met bestaande aanpakken. Verder belichten we dat ons uniforme raamwerk een natuurlijke manier aanbied ter abstractie over de ondersteunde data domeinen, zonder ons te hoeven beperken tot algebra homomorfismen.

Nadat een systeem is ontworpen, moet het geïmplementeerd worden in een programmeertaal die zich het beste leent voor het type systeem dat moet worden gemaakt. Voorbeelden van populaire programmeertalen zijn Java, C en C♯. De semantiek van dergelijke programmeertalen is vaak specificeerd in natuurlijke taal. Helaas zijn dergelijke specificaties vaak moeilijk te begrijpen. Erger nog, ze bieden vaak ruimte voor meerdere interpretaties. Dat wil zeggen, ze zijn ambigu. In dit werk laten we zien hoe het graaf transformatie raamwerk formele en intuitive middelen biedt voor het specificeren van de operationele semantiek van programmeertalen. Daarvoor introduceren we een kunstmatige, object georiënteerde programmeertaal genaamd Taal, en definiëren we de control flow en executie semantiek in termen van graaf transformatie regels. Daarnaast bieden we ook de benodigde tool ondersteuning om het gedrag van eigenlijke Taal-programma's te simuleren.

Nadat een systeem is gespecificeerd als een graaf productie systeem, moet het gedrag op correctheid worden geverifieerd. Daarvoor introduceren we een algoritme dat een bekend on-the-fly model checking algoritme combineert met ideeën van bounded model checking. We richten ons op het verifiëren van het temporele gedrag van dergelijke systemen. Dat betekend dat de eigenschappen die worden geverifieerd worden uitgedrukt als formules in een (lineaire) temporele logica, zoals bijvoorbeeld *LTL*; de namen van de graaf transformatie regels vormen de verzameling van atomaire proposities.

We hebben de Groove Tool Set uitgebreid met verificatie functionaliteit in plaats van het uitvoeren van de model checking procedure met behulp van bestaande model checking tools. De belangrijkste motivatie hiervoor is de mogelijkheid om te onderzoeken hoe we het potentieel van het graaf transformatie raamwerk optimaal kunnen benutten, in het bijzonder gericht op het aanpakken van het toestand-explosie probleem gebruik makend van partiële order reductie technieken. Helaas zijn veel van dergelijke technieken gebaseerd op veronderstellingen die niet gelden in onze opzet, bijvoorbeeld met betrekking tot het aantal acties (of operaties) die kunnen worden uitgevoerd. Daarom hebben we een dynamisch partiële order reductie algoritme ontwikkeld gebaseerd op het selecteren van zogenaamde probe sets. Het algoritme is gebaseerd op asymmetrische relaties tussen acties als die elkaar stimuleren of onmogelijk maken. We hebben het algoritme ontwikkeld voor entiteit-gebaseerde systemen waarin toestanden uniek worden gekarakteriseerd als entiteit-verzamelingen, en acties

entiteiten kunnen lezen, verwijderen, creëren en hun afwezigheid kunnen eisen. Deze opzet is gekozen omdat het goed overeenkomt met het graaf transformatie raamwerk. Bovendien beschrijven we hoe grafen kunnen worden gecodeerd als entiteit-verzamelingen en hoe toepassingen van graaf transformatie regels vertaald kunnen worden in overeenkomende entiteit-gebaseerde acties. We laten zien dat ons algoritme een correct gereduceerde toestandsruimte produceerd waarbij alle executiepaden van het systeem bewaard blijven. Op dit moment is er geen implementatie van ons algoritme en daarom hebben we geen statistieken over hoeveel reductie er kan worden behaald voor verschillende typen systemen.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

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 Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*.

Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, 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 Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calam. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27