Abstract Graph Transformation Theory and Practice

Eduardo Zambon

Abstract Graph Transformation Theory and Practice

Eduardo Zambon

Graduation committee:

Chairman:	Prof. dr. ir. Anton J. Moutaanh
Promoter:	Prof. dr. ir. Arend Rensink

Members:

Prof. dr. ir. Mehmet Akşit	University of Twente
Prof. dr. Frank de Boer	Centrum Wiskunde & Informatica
Prof. dr. Dino Distefano	Queen Mary University of London
Prof. dr. ir. Joost-Pieter Katoen	University of Twente/RWTH
Prof. dr. Barbara König	Universität Duisburg-Essen
Prof. dr. dr. h.c. Reinhard Wilhelm	Universität des Saarlandes

CTIT

CTIT Ph.D. Thesis Series No. 12-239

Centre for Telematics and Information Technology

University of Twente

P.O. Box 217 – 7500AE Enschede, NL

IPA Dissertation Series No. 2013-03

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

Netherlands Organisation for Scientific Research

The work presented in this thesis was supported by the GRAIL project (Graphs for Abstract Interpretation of Languages), funded by NWO grant 612.000.632.

ISSN 1381-3617

ISBN 978-90-365-3477-2

<http://dx.doi.org/10.3990/1.9789036534772>

Printed by: Wöhrmann Print Service

Copyright © 2013 Eduardo Zambon

ABSTRACT GRAPH TRANSFORMATION THEORY AND PRACTICE

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, January 24th, 2013 at 16.45 hrs.

by

Eduardo Zambon

born on 2 September 1979
in Vila Velha-ES, Brazil

This dissertation is approved by:

Prof. dr. ir. Arend Rensink (promotor)

To Mom (in memoriam)

Acknowledgements

First and foremost I would like to thank my daily supervisor and promotor, *Arend Rensink*. I was extremely fortunate to have the opportunity to work with someone that is well-versed not only in the theoretical foundations of formal methods but also in tool development and programming. This, in turn, is reflected on the contents and structure of this thesis, thus showing his influence in the composition of this work. In fact, it is safe to say that this thesis would have never come to existence without his invaluable help. At times when my confidence was shaken by discouraging results or setbacks, Arend always kept an optimistic view and urged me to keep moving forward. For all that I would like to express my most sincere gratitude: working with him was pleasure and an honour.

Second, I would like to thank all my colleagues at the FMT group, for providing such a nice working environment. In particular, I am grateful to: *Jaco*, for making me feel welcome to his group from the very first day; *Joke*, for her incredible efficiency in disentangling even the most complicated situations; *Mark*, for always helping me with my expat problems, especially when I got stuck due to the language barrier; *Amir*, for providing me with a place to live when I suddenly became homeless; and *Maarten*, for always finding the time to discuss my work.

I also want to thank the members of my graduation committee for accepting to take part in the evaluation of this thesis and for their suggestions for improvement.

I am glad to say that on these 4.5 years in the Netherlands I was able to form a great group of friends, which were always around to celebrate the good times and offer support during the bad ones. I will avoid making a list of people, since it would be rather long. In the end there is no need to name names; they know who they are and I am sure we will still be in touch for many years to come.

Last but certainly not least, I would like to thank my friends and family in Brazil for always providing the necessary support, despite the distance. There were quite some bumps in the road until here but their encouragement certainly helped me along the journey.

Abstract

The *verification* of systems with respect to a desired set of behavioural properties is a crucial step in increasing our confidence that these systems will correctly function under all circumstances. Although it would be desirable to verify all (computer) systems that we use in our daily life, the sheer complexity of the verification tasks often limit their application to *critical* systems. A system is considered critical when its incorrect behaviour can cause severe damage, such as loss of lives or the destruction of valuable equipment.

One well-established verification method is model checking. This method takes as input a system specification (the *model*) and a desired property, and *checks* if the model satisfies the property. A usual approach within this technique is *explicit state* model checking, which tries to exhaustively enumerate all configurations (states) that the system may exhibit. This enumerative procedure is called the *exploration* of the model *state space*.

State space exploration can be used in many different settings but it cannot be directly applied to systems that have infinite state spaces. Classes of such systems include any model where an unbounded number of entities can be created. This is the case, for example, in a model representing a memory heap, where data structures can be dynamically allocated.

To handle infinite-state models, some form of *abstraction* is necessary to ensure that the verification will terminate. An obvious requirement for such abstractions is that it should still be possible to use the abstract state space representation to check if the system conforms to the property of interest.

In this thesis, we use *graph transformation* as our modelling formalism for system specification. Graph transformation is a Turing-powerful, declarative rule-based formalism, with a mature theoretical foundation and a thriving tool environment. Our work focus on model checking of graph transformation systems, more specifically on the exploration of their state spaces, and in particular for infinite-state graph transformation systems. We present two abstraction techniques that yield a finite over-approximation of the behaviour of such infinite-state graph transformation systems, thus enabling verification on the abstract level.

The first technique, called *neighbourhood abstraction* is discussed under both a theoretical and a practical focus; the former concerning the formal definition and correctness of the abstraction method, and the latter discussing its implementation in GROOVE, our graph transformation tool set. Experimental results are also given in order to assess the performance of the developed tools. For the second abstraction technique, called *pattern abstraction*, we present its complete theoretical foundation while providing some indications on how its practical implementation can be realised.

Contents

1	Introduction	1
1.1	System analysis/verification	1
1.2	Graph transformation	2
1.3	The big picture	3
1.4	Graph abstractions	4
1.5	Extracting graphs models from programs	5
1.6	Thesis structure and contributions	7
I	Preliminaries, related work, and Java graphs	9
2	Preliminaries	11
2.1	Approaches to graph transformation	11
2.2	The GROOVE tool set	14
2.3	Graph transformation and program semantics	19
2.4	An informal introduction to graph abstractions	21
3	Related work	27
3.1	Java, graphs and model checking	27
3.2	Guided search	28
3.3	Abstraction methods	29
4	From Java to graphs	45
4.1	Introduction	45
4.2	Preliminaries	46
4.3	Approach	47
4.4	Java syntax type graph	50
4.5	Control flow grammar	54
4.6	Conclusion	56
II	Graph abstraction	59
5	Essential concepts	61
5.1	Graphs	61
5.2	Simple graphs	62
5.3	Approximated counting	67

5.4	Transition systems, simulations and model checking	70
5.5	Test cases for experiments	70
6	Neighbourhood abstraction – Theory	75
6.1	Introduction	75
6.2	Neighbourhood shapes	76
6.3	Neighbourhood shape transformation	82
6.4	Conclusion	86
7	Neighbourhood abstraction – Practice	87
7.1	Introduction	87
7.2	Construction of a NSTS	87
7.3	Operation materialise	89
7.4	Initial results and discussion	96
7.5	Subsumption for state space reduction	98
7.6	Experiments and results	101
7.7	Conclusion	105
8	Pattern abstraction	107
8.1	Introduction	107
8.2	Pattern graphs	107
8.3	Pattern shapes	121
8.4	Quasi-shapes	126
8.5	Pattern shape transformation	139
8.6	Conclusion	141
9	Conclusion	143
9.1	Thesis summary and discussion of contributions	143
9.2	Future work	146
	Bibliography	146

Introduction

Modern society is dependent on a multitude of different systems, most of which are computer-based. A common characteristic of all computer systems is the need for a controlling program, which specifies the system functionality. However, computer programming is an extremely complex task and therefore prone to errors. Thus, given the sheer amount of programs being developed daily, it is expected that some of these software behave incorrectly. Depending on how critical a computer system is, its incorrect operation may range from being a mere nuisance (*e.g.*, a game crashing in a mobile phone) to being life-threatening (*e.g.*, a plane crashing due to faulty controls).

In very broad terms, this thesis is concerned with the verification of software systems. The term software is used very loosely here, in order to encompass not only the standard definition of a program written in well-known programming languages, but also artefacts (system models) that have a properly defined execution semantics. Given a collection of properties for which a system is expected to be compliant, system verification then amounts to checking that the system indeed conforms to the properties specified. This is of particular importance for critical systems that must perform correctly under all circumstances.

1.1 System analysis/verification

The verification of software systems has already been an old concern for some computer scientists. However, given the ever-growing use of software in our society, and the consequent problems and damages due to programming errors, over the last years this concern became more wide-spread, and hence attracted more attention from researchers. As a natural consequence of this fact, several different methods and techniques have been proposed. We can roughly divide software verification techniques in two types: *deductive* or *enumerative*.

Deductive methods date back to Turing [Tur89], being mostly based on the principles defined by Floyd [Flo67] and Hoare [Hoa69], and later refined by Dijkstra [Dij76]. These methods rely on an axiomatic definition of the semantics of the programming language elements and on inference rules that allow one to reason about the desired correctness properties of a program in a compositional way. Among the tools developed under this approach we can cite the KeY System [BHS07], Why/Krakatoa [FM07], jStar [DP08] and ESC/Java [FLL⁺02] for the verification of Java programs, and the Spec# tool [BLS04] for analysing code written in a super-set of the C# language.

Enumerative methods try to (partially or exhaustively) explore the possible states of a program. A program state corresponds to a snapshot of the program dynamic structures in memory, *e.g.*, heap, stack, threads, locks and program counters. The transitions between states are given by the execution semantics of the language in which the program is written. In fact, an exhaustive exploration mechanism can be seen as a non-deterministic machine that generates (all) possible execution paths of the input program and produces a transition system representing the program state space. A well-known enumerative technique is model checking [CGP99, BK08], where the desired correctness properties of the program are checked to hold over a transition system. Among currently available software model checkers we can cite Java PathFinder (JPF) [VHBP00] and Bogor [DHHR05] for Java, and MoonWalker [dBNR09] for C# programs.

In this thesis we focus on one particular kind of enumerative method: model checking of graph transformation systems.

1.2 Graph transformation

Graph transformation has been advocated as a flexible formalism, suitable for modelling systems with dynamic configurations or states. This flexibility is achieved by the fact that the underlying data structure, that of graphs, is capable of capturing a broad variety of systems. Some areas where graph transformation is being applied include the visual modelling of systems, the formal specification of model transformations, and the definition of graph languages, to name a few.

Essentially, whenever a system consists of entities with relations between them, this can be naturally captured by a graph in which nodes stand for the entities and edges for the relations. If, in addition, a main characteristic of such a system is that entities are created or deleted and the relations between them can change, then the transformation of graphs comes into play.

The core concept of graph transformation is the rule-based modification of graphs, where each application of a rule leads to a graph transformation step. A transformation rule is basically formed by two graphs, a left-hand side (LHS) and a right-hand side (RHS). The *application* of a rule to a *host graph* amounts to finding an occurrence of the LHS in the host graph and replacing this occurrence by the RHS. This application produces a new graph, which is the result of the transformation.

In our work, we use graph transformations to model the dynamic behaviour of systems. A set of graph transformation rules, referred to as a *graph production system* (GPS), can be seen as a declarative specification of how the analysed system behaves, and how its state changes. We use the term *state space* to denote the set of all possible system configurations. In our setting, the initial state of a system is represented by an initial host graph that is associated with a GPS. This combination of a rule set plus an initial graph is called a *graph grammar*.

There are several distinct approaches to graph transformation, some of which are presented and discussed at an informal level in Section 2.1. In Chapter 5,

Figure 1.1: Overview of the verification cycle proposed.

we define the basic notion of graph transformation and graph grammars that we use, and we formalise their semantics in terms of labelled transition systems.

1.3 The big picture

Figure 1.1 provides a picture of a verification cycle for graph transformation systems, as envisioned by Rensink [Zam10, ZR11]. The required input is a graph grammar, modelling the system to be analysed. This grammar is given to an exploration engine, that tries to apply the grammar rules in all possible ways to the initial host graph, and also to all graphs resulting from a transformation step. This exploration produces a graph transition system (GTS), where states correspond to graphs and transitions to rule applications.

The usual concrete exploration method traverses the state space using a search strategy, for example, depth-first search. During this search we may profit from our representation of states as graphs and perform an on-the-fly symmetry reduction to shrink the size of the generated GTS. Since isomorphic graphs have the same “behaviour”, they are considered equivalent; the symmetry reduction collapses isomorphic states into a single representative, thus reducing the GTS size (see Section 5.2.3).

However, there are important classes of systems that have extremely large or even infinite state spaces and therefore cannot be (fully) explored using a concrete exploration technique. Interesting examples of such infinite-state systems are graph grammars modelling the manipulations of data structures in a memory heap, or communication networks of arbitrary (unbounded) size.

To handle such classes of grammars, we employ *graph abstraction* methods. Similarly to symmetry reduction in a concrete setting, an abstract exploration collapses equivalent concrete states under an abstract representative, with the behaviour of the abstract state encompassing all possible behaviour of the collapsed concrete states. The definition and practical evaluation of interesting graph abstractions are the main topics investigated in this thesis (see Section 1.4

for an additional discussion).

In addition or apart to on-the-fly reduction techniques such as isomorphism collapsing, a GTS can also be contracted by off-line methods, such as bi-simulation minimisation. These methods are outside the scope of this thesis and are only mentioned here for completeness.

After producing a GTS, we can perform model checking against a given set of correctness properties that the system is expected to exhibit. This check produces either a verdict that the system is indeed correct, or a counter-example, represented as a sequence of rule applications, that produces an error. This counter-example is then traced back to the grammar, so that the user can inspect the error.

A graph abstraction ensures that the GTS obtained by the exploration is finite but the cost for this finiteness is precision. In general, the abstract GTS is an over-approximation of the concrete system, and spurious extraneous behaviour may be introduced by the abstraction. This extraneous behaviour may lead to spurious sequences (traces) of rule applications that have no concrete counterpart. This problem can be addressed with an abstraction refinement loop: if a spurious trace is obtained, the abstraction is refined to exclude the trace from the abstract GTS.

As an exploration/model checking engine we use GROOVE, our graph transformation tool set, which is presented in Section 2.2.

1.4 Graph abstractions

As stated before, a common approach for dealing with infinite-state systems is *abstraction*. Being a rather general term, it can be used to indicate many distinct methods used in system analysis (*e.g.*, symbolic model checking). In particular, given our setting, we are interested in *state abstractions*, where “similar” concrete states are collapsed under an abstract representative. This notion of abstraction is the basis of well-known techniques such as abstract interpretation [CC77] and shape analysis [SRW02].

State “similarity” is the point where many abstractions differ; in order to define what “similar” means one has to look at the underlying framework one uses to represent systems. In our case, the framework corresponds to graph transformation and graph grammars, and therefore we work with *graph abstractions*. Such abstractions also occur in several distinct flavours, which depend, among other factors, on the specific graph representation being used. In this thesis we present two of such graph abstractions: *neighbourhood abstraction* (Chapters 6 and 7) and *pattern abstraction* (Chapter 8). Borrowing from the terminology introduced in shape analysis, our abstract graphs are called *shapes*.

Neighbourhood abstraction defines graph similarity according to a *neighbourhood radius*, a parameter of the abstraction. Nodes of a graph are grouped with respect to their neighbourhood similarity, which considers the node “kind” (*i.e.*, its labels) and its associated incoming and outgoing edges. Similar nodes are then collapsed to produce a *neighbourhood shape*, which serves as a representative in the abstract domain for a possibly infinite set of concrete graphs.

To obtain a finite state space, rule applications are lifted to neighbourhood shapes, which form a finite universe. Returning to the verification cycle given in Figure 1.1, in this case the abstract exploration engine operates on neighbourhood shapes, and its resulting artefact is a *shape transition system*, which is an over-approximation of the concrete (possibly infinite) GTS of the given grammar.

The theory of neighbourhood abstraction was developed by Rensink and others [BBKR08, BKR⁺12]. A main contribution of this thesis is the practical implementation and integration of neighbourhood abstraction into the GROOVE tool set (see Chapter 7). This implementation was then experimentally evaluated for performance. The shortcomings identified during the implementation and experimental evaluation of neighbourhood abstraction motivated the development of the new theory of pattern abstraction.

Pattern abstraction is based on the concept of a *pattern*: a graph configuration of interest that should be preserved by the abstraction. Patterns are grouped into a *pattern graph*, which is a hierarchical structure that captures the incremental composition of smaller patterns into larger ones.

Pattern graphs are abstracted into *pattern shapes*. Graph similarity in this setting is taken as *pattern equivalence*, which groups nodes of a pattern graph with isomorphic patterns. In the same vein as neighbourhood abstraction, similar pattern graph nodes are collapsed to obtain a pattern shape. As before, pattern shapes form a finite universe, and are transformed by rule applications. Thus, in this case, abstract exploration again produces a shape transition system, although with states formed by pattern shapes. This transition system also serves as an over-approximation of the original concrete GTS and therefore can also be used for verification.

The theory of pattern abstraction constitutes a main contribution of this thesis. An informal presentation of neighbourhood and pattern abstraction is given in Section 2.4. Both methods are properly formalised and discussed in Part II.

1.5 Extracting graphs models from programs

A distinct line of research also investigated in this thesis is the translation of source code to a graph-based representation. One recent key change in current verification techniques is the development of approaches that can analyse the correctness of software written in commonly used imperative programming languages. Previously, those methods were limited to the so-called “modelling languages” (*e.g.*, Promela, LOTOS, CSP, etc), which have a clean and simple definition and are thus more amenable to formal treatment.

Although graph grammars are a quite expressive (Turing complete) formalism, from the point of view of a programmer accustomed with imperative languages, graph transformation is usually seen as a modelling language. Therefore, to be able to analyse imperative programs using the verification cycle in Figure 1.1, we need a way to bridge the gap between the program source code and the system model (graph grammar) representation expected as input.

Figure 1.2: Graph model extraction from program code.

Figure 1.2 shows our proposed translation from program code to a graph grammar. The input is the program source code, written in some programming language, *e.g.*, Java. The code is analysed by a compiler that produces as output an abstract syntax graph (ASG). This ASG is essentially the usual abstract syntax tree produced by a language parser enriched with type and variable bindings.

The ASG, together with definitions of the language control flow semantics, is the input of a flow construction mechanism, which builds a flow graph for the given ASG. This flow graph represents how the execution point of the program should flow through the ASG, according to the rules of the programming language in use. This flow construction can be achieved by several means, for example by writing a dedicated tool for the task. In our case, we again use graph transformations, by employing a GPS that captures the control flow semantics of Java.

The flow graph resulting from the flow construction is connected to the ASG, resulting in a program graph, which can be “executed” by another GPS that describes the Java execution semantics. A program graph together with this execution GPS forms the system model that can then be taken as the input for the previously discussed verification cycle for graph grammars.

It should be noted that the only variant elements of the process illustrated in Figure 1.2 are the input code and its associated graphs. The transformation constructs and the language semantics defined by the two GPS are fixed for each programming language of choice. Thus, although the development of these transformations and artefacts may be a laborious task due to the complexity of a fully fledged language such as Java, this development effort has to be made only once, and the resulting components can be re-used for all different input programs.

In Chapter 4 we discuss two of the elements in Figure 1.2. The development of a graph compiler from Java source code to GROOVE graphs is presented in detail. We also give a description of the flow construction method used and the associated control flow GPS that was elaborated. The contributions of Chapter 4 are not directly related to Part II but there is an indirect connection, as described above. Apart from producing the input required for the verification

cycle of Figure 1.1, the graph model extraction of Figure 1.2 may also be interesting in its own right. Examples of other possible usages of this translation include any technique for program analysis, for instance, compiler optimisation of intermediate code, and static analysis.

1.6 Thesis structure and contributions

This thesis is composed of nine chapters, which are grouped in two parts. This introduction aside, the following is a summary of the contents and contributions of each chapter.

Chapter 2 introduces and discusses several concepts necessary for understanding the remaining chapters. We present some of the approaches for graph transformation defined in the literature, and we explain which one we follow. Furthermore, we give an extensive presentation of the GROOVE functionalities, that are later extended in Chapter 7. In addition, we provide examples and explanations of the neighbourhood and pattern abstractions that are defined in Part II.

Chapter 3 gives an overview of related work, in particular on graph abstractions and similar methods.

Chapter 4 presents two components of the model extraction procedure shown in Figure 1.2. We detail how software written in imperative programming languages (in particular, Java) can be translated to graphs. We present the most important points of this translation, with focus on the implementation of the graph compiler that converts Java code to GROOVE graphs. Furthermore, we discuss the rationale in modelling a graph grammar that captures the control flow semantics of Java. This work was previously published in [RZ09a, RZ09b].

Chapter 5 contains the common concepts upon which Chapters 6 to 8 are built. In this chapter, we define formally the graphs we use, and we introduce an interleaving semantics for graph grammars. Practical aspects of state space exploration are also discussed.

Chapter 6 summarises some key aspects of the theory of neighbourhood abstraction, while giving it a formal treatment that is simpler than the ones previously published [BBKR08, BRKB07]. This new presentation of the theory can be seen as an additional contribution of the thesis.

Chapter 7 describes the practical aspects of the implementation of neighbourhood abstraction in GROOVE. We detail two versions of the implementation, which are experimentally evaluated. This work was previously published in [RZ10, ZR12]. The tool resulting from this implementation effort and the experimental analysis constitute a main contribution of this work.

Chapter 8 presents the new theory of pattern abstraction, originally published in [RZ12a]. While the presentation in this chapter is kept at the theoretical level, several definitions and operations are constructive, thus hinting on how they can be implemented in practice. The contents of this chapter form a major contribution of this thesis.

Chapter 9 concludes the thesis with an analysis of the obtained results and

an indication for future work.

Part I of the thesis is composed of Chapters 2 to 4, whereas Part II groups Chapters 5 to 8. These two parts have distinct flavours: in Part I we keep the presentation at an informal level, where in Part II the theory is properly formalised. The intention with this distinction is to provide different possibilities for reading the thesis. For example, a reader already knowledgeable in graph abstractions may want to skip Part I and go straight to Part II. On the other hand, a different reader interested only in an overview may profit more from the contents of Part I, in particular Chapter 2.

Part I

Preliminaries, related work, and Java graphs

Preliminaries

In this chapter we give a more detailed introduction to concepts that are needed for understanding the rest of this thesis. We begin with a brief presentation of the existing approaches to graph transformation, followed by a detailed explanation of the GROOVE functionalities. We then continue with a simple example that illustrates how the semantics of a programming language can be captured by graph transformation rules. This example is used to demonstrate a simple application of the abstract interpretation framework, which paves the way for the discussion on graph abstractions. We conclude the chapter with some examples of neighbourhood and pattern abstractions that serve as an informal introduction to the methods described in Part II.

2.1 Approaches to graph transformation

Graph transformation (or *graph rewriting*) [Roz97] is a rule-based transformation technique with a solid theoretical foundation and growing tool support. *Graph transformation systems* (or *graph grammars*) are a very flexible framework that can be used to represent not only the static state of a system but its dynamic evolution as well. The core concept of graph transformation is the modification of graphs according to transformation (or production) rules.

A rule specifies both the conditions under which it can be applied and the changes to be performed to a *host graph*. In its basic form, a *transformation rule* $r = \langle L, R \rangle$ is composed of two graphs, a left-hand side (LHS) L and a right-hand side (RHS). *Applying* rule r amounts to finding a *match* of L in the source host graph and replacing this occurrence of L by R , thus producing a new target graph, the transformation result.

The key points for specifying the result of a rule application are how to delete L from the host graph and how to connect R with the context. The different proposals for these points led to the development of several different graph transformation approaches, some of which are listed below. (See [Roz97] for a detailed presentation.)

- *Node label replacement*. In this approach, the LHS L is composed of a single node, to be replaced by an arbitrary graph R . The connection of R with the context is determined by an embedding relation dependent on node labels.
- *Hyper-edge replacement*. In this approach, the LHS L consists of a labelled hyper-edge that is replaced by an arbitrary hyper-graph R with designated attachment nodes corresponding to the connection nodes of L . Although

Figure 2.1: Example of a DPO transformation [EEPT06].

similar with the previous approach, the use of hyper-graphs avoids the need for an additional embedding relation.

- *Algebraic approach.* This method is based on pushout constructions (from category theory), used to model the gluing of graphs. There are two main variants of this method: the *double pushout* (DPO) approach and the *simple pushout* (SPO) approach, discussed in more detail in the following. Another algebraic approach more recently introduced is the *sesqui pushout* [CHHK06], which is a mixture of the DPO and SPO approaches.

In the following we use the concepts of *graphs* which are formed by nodes and edges, and of *graph morphisms* which relate a source and target graph by mapping elements of the source graph to elements of the target graph. These concepts are properly defined in Chapter 5.

2.1.1 Double pushout (DPO) approach

In the DPO approach a transformation rule is formed by three graphs $\langle L, K, R \rangle$, where L and R are the LHS and RHS graphs of the rule and K is their intersection, called the *interface* graph. The LHS L indicates the preconditions for the rule application, while the RHS R describes the post-conditions. Interface K specifies a graph part that has to exist for the rule to be applicable but that is not changed by the transformation.

In the general case, L , K and R are related by two total graph morphisms, and the rule is represented by a *span*, a categorical construction that corresponds to a diagram of the form $r = (L \leftarrow K \rightarrow R)$. On a simplified setting, we can consider the rule graphs to be composed of sets of elements (nodes and edges) and thus the morphisms of the rule span correspond to sub-set inclusions

(*embeddings*). In this case, $L \setminus K$ describes the graph part that has to be deleted by the rule and $R \setminus K$ indicates the part to be created.

For a rule r to be *applicable* to a host graph G , a *match* m of L into G has to exist, where m must be structure-preserving, *i.e.*, m is a morphism from L to G . The *application* of r to G according to match m comprises two steps. First all nodes and edges matched by $L \setminus K$ are removed from G . The remaining structure $D = (G \setminus m(L)) \cup m(K)$ still has to be a legal graph, which imposes additional restrictions on m , called *gluing conditions*, described below. In the second step of rule application, graph D is glued together with $R \setminus K$ to obtain the derived graph H .

The additional gluing conditions imposed on an application are the following.

- *Dangling edge condition*: if a node v is deleted by the application then all edges incident to v must also be marked to be deleted by the rule. This ensures that no edges in D are left dangling, *i.e.*, without a source or target node.
- *Identification condition*: for every graph element $x \in G$ marked to be deleted by the rule application, $m^{-1}(x)$ must specify an unique pre-image in L . This avoids inconsistencies when constructing D , the so-called *delete-preserve* and *delete-delete* conflicts.

If these gluing conditions fail for a match m then m is invalid and the rule cannot be applied.

An example of a rule application is given in Figure 2.1. Graphs are shown inside boxes, with nodes depicted as small black circles and edges drawn as directed arrows between source and target nodes. The number next to each node indicates the node identity. Edge identities are not shown but can be inferred from the incident nodes of the edge. The arrows connecting boxes are graph morphisms (in this case, embeddings); the identities of graph elements are abstracted away through these morphisms. The two commuting squares identified with PO1 and PO2 correspond to pushouts in a categorical setting, hence the naming of the approach.

2.1.2 Single pushout (SPO) approach and comparison with DPO

It is possible to modify the rule span $r = (L \leftarrow K \rightarrow R)$ used in the DPO approach in such a way that L and R are directly related by a *partial* graph morphism $p: L \rightarrowtail R$ with $\text{dom}_p = K$. Similarly, span $G \leftarrow D \rightarrow H$ can be considered a partial graph morphism $s: G \rightarrowtail H$ with $\text{dom}_s = D$. This then leads to a single pushout diagram where the match is a total morphism and the other morphisms are partial. The gluing conditions of the DPO method are no longer necessary in the SPO approach; instead, the method defines that dangling edges are implicitly removed and deletion has precedence over preservation in delete-preserve conflicts. An example of a SPO rule application is given in Figure 2.2.

An interesting consequence of the DPO approach is that rule applications do not have “side-effects”, *i.e.*, a rule only deletes graph elements that were explicitly specified to be removed. Furthermore, if the morphisms of the pushout squares are one-to-one functions (*i.e.*, injective), the application of a rule r can be “reversed” by a complementary rule $\bar{r}$, which has L and R interchanged.

Figure 2.2: Example of a SPO transformation [EEPT06].

In contrast with the DPO method, the SPO approach does not require the entire context of a rule match for the application, meaning that the existence of a match immediately implies the existence of a rule application. In the DPO approach this is not the case since matches may not satisfy the gluing conditions.

Throughout the thesis we will restrict ourselves to SPO constructions since this is the approach followed by GROOVE, our graph transformation tool.

2.2 The GROOVE tool set

GROOVE [Ren03a, GdMR⁺12] is a general purpose graph transformation tool set that uses simple labelled graphs. The core functionality of GROOVE is to recursively apply all rules from a predefined set (the *graph production system* – GPS) to a given *start graph*, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs.

The main component of the GROOVE tool set is the Simulator, a graphical tool that integrates (among others) the functionalities of rule and host graph editing, and of interactive or automatic state space exploration.

2.2.1 Host graphs and rules

In GROOVE, the host graphs, *i.e.*, the graphs to be transformed, are simple graphs with labelled nodes and edges. In simple graphs, edges do not have an identity, and therefore parallel edges (*i.e.*, multiple edges with the same label, and source and target nodes) are not allowed. In the graphical representation, nodes are depicted as rectangles and edges as binary arrows between two nodes. Node labels can be either node types or flags, respectively displayed in **bold** and *italic* inside their corresponding node. Nodes can have at most one type label

Figure 2.3: Example GROOVE rule and legend.

and zero or more flags. Flags are used to model a Boolean condition, which is true for a node if the flag is there and false if it is absent.

A GROOVE transformation rule consists of the following:

- a pattern that must be present in the host graph in order for the rule to be applicable;
- subpatterns that must be absent in the host graph in order for the rule to be applicable;
- elements (nodes and edges) to be deleted from the graph; and
- elements (nodes and edges) to be added to the graph.

All these elements are combined into a single graph; colours and line strokes are used to visually distinguish them. Alternatively, one may think in terms of *application conditions* and *modifications*: of the former, we distinguish positive (which must be present in order to apply a rule) and negative (which must be absent in order to apply a rule) ones, whereas of the latter, we distinguish deletion and creation of elements. Figure 2.3 shows a small example illustrating most of these concepts:

- The black (continuous thin) *reader* elements, in this case two nodes labelled **A** and **C**, must be present and are preserved – in fact, they form a positive application condition.
- The red (dashed fat) *embargo* elements, in this case a **parent**-labelled edge with a **P**-labelled target node, must be absent in the graph – in fact, each connected subgraph of embargo elements forms a separate negative application condition.
- The blue (dashed thin) *eraser* elements, in this case a **child**-labelled edge from the **A**-node to the **C**-node, must be present and are deleted.
- The green (continuous fat) *creator* elements, in this case a **parent**-labelled edge with a **P**-labelled target node, are created.

When a node type or flag is used in a non-reader element but the node itself is not modified, the node type or flag is prefixed with a character to indicate its role. The characters used are +, −, and !, for creator, eraser, and embargo elements, respectively.

The overall effect of the rule in Figure 2.3 is to search for **A**- and **C**-nodes connected by a **child**-edge but without a **parent**-edge to a **P**-node, and to modify this by removing the **child**-edge and adding a **parent**-edge to a fresh **P**-node. For instance, the rule can be applied to the graph in Figure 2.4(a) in two ways, one of which results in the graph on the centre (b). The other application removes the other **child**-edge from the same **A**-node, as shown in Figure 2.4(c).

Figure 2.4: Example application of the rule in Figure 2.3.

2.2.2 State space exploration

The most distinguishing feature of GROOVE, when compared with other graph transformation tools, is the fact that it does not just carry out a single sequence of transformations from a given start state, but can explore and store the entire state space of reachable graphs. This provides a rich source of information for further analysis. In fact, GROOVE offers a choice for the exploration strategy to be used:

- Depth-first (DFS) full exploration, also with on-the-fly linear temporal logic model checking.
- Breadth-first (BFS) full exploration. In some grammars, this enables finding shortest paths to certain graphs.
- Linear, random linear, and conditional exploration. This allows simulation without covering all states, for instance if the state space is too large.

We illustrate this GROOVE functionality with a solution for the leader election case study proposed by König [Kön09] at the GraBaTs 2009 tool contest.

A simple distributed leader election protocol [CR79] works as follows. There exists a set of processes arranged in a ring, *i.e.*, every process has a unique predecessor and a unique successor. Furthermore, each process has a unique identity, a natural number. The leader to be elected is the process with the smallest identity, however this number is not known at the start of the protocol.

At the beginning, every process generates a message (*Mid*) with its own identity and sends it to its successor. A received message with content *Mid* is treated as follows by a process with identity *Pid*:

1. if $Mid < Pid$, the process forwards the message to its successor;
2. if $Mid = Pid$, the process declares itself the leader; and
3. if $Mid > Pid$, the process discards the message.

In total the graph grammar modelling the protocol is composed of five transformation rules, presented, in GROOVE notation, in Figure 2.5(b-f). Figure 2.5(a) shows a start graph representing a network ring with three processes (modelled as nodes). There is an extra auxiliary node **ids** containing identities ranging from 1 to 3. This node is used by rule **pk-id** (Figure 2.5(b)) to generate all possible permutations of processes with different identities in the network.

Message creation is handled by rule **c-msg**, given in Figure 2.5(c). Each application of this rule produces a new message with the proper process identity. To avoid the creation of duplicated messages, the rule flags the process as **active**;

Figure 2.5: The complete GROOVE grammar modelling the leader election protocol: (a) Example start state with a ring of three processes. (b) Rule that distributes identities among processes. (c) Rule for creating a message with the process identity. (d) Rule modelling the propagation of a message. (e) Rule that elects a leader process. (f) Rule for discarding a message.

the same rule also checks for the absence of such flag (embargo **! active**). Figure 2.5(d) shows rule **prop**, which implements item 1 of the protocol described above ($M.id < P.id$). Similarly, items 2 and 3 are respectively captured by rules **elect** and **d-msg**, displayed in Figure 2.5(e-f).

The state space of the protocol is obtained by applying the rules of the grammar to the start state of Figure 2.5(a) and to all its subsequent states. The state space is shown in Figure 2.6, represented as a *graph transition system* (GTS). States are displayed as rectangles and have an unique graph associated with them. The names of the rules applied on the transformations from one state to the other are written as labels of the transitions between states. The top state (*s1*) is the start state and the bottom states (*s66* and *s70*) are final states, *i.e.*, no other state can be reached from these states by applying any rule.

The state space of Figure 2.6 can be divided in two parts. In the upper section (the states above *s11* and *s12*) only rule **pk-id** is applied to create the ring. The lower section is where the rules related to the protocol itself are

Figure 2.6: The GTS of the leader election protocol.

applied. An interesting point about the upper part of the state space is that it creates all permutations of a network with size n . Given n processes, there are $n!$ different permutations of nodes which determine the different number of orderings in which rule `pk-id` can be applied on the nodes. However, we know that due to the symmetry of a ring, only $(n - 1)!$ different rings exist with the same set of nodes. Not detecting the identical states in this case would lead to a state space that is almost n times bigger. GROOVE automatically finds

Figure 2.7: Extra rules used for verification.

the identical states (isomorphic graphs) and avoids duplicating already existing states. This can be seen in Figure 2.6, all six ($3!$) different feasible orders of applications of `pk-id` are shown in states $s5$ to $s10$, which are reduced to two ($2!$) states: $s11$ and $s12$. The protocol is only checked on these two generated rings.

2.2.3 State space analysis

After a GTS is generated, there are different options to analyse it. For instance, given its graphical representation, one can visually check the GTS in order to ascertain that the behaviour of the model is as expected. However, in most cases the state space of a graph grammar is quite large, preventing its visualisation. Another possible analysis method is to use a querying language for inspecting a GTS. In this case, the GROOVE Simulator provides an embedded Prolog interpreter, a recent extension described in [GZR⁺11]. A third analysis option is to use the GROOVE model checker, which allows us to verify properties specified in CTL (computation tree logic) or LTL (linear temporal logic) formulæ.

Returning to the leader election example and its corresponding GTS in Figure 2.6, we add two rules to the grammar to assist us with the model checking part. Figure 2.7 shows these new *inspection* rules. As done with the other grammar rules, at each state we try to match the inspection rules. If a match can be found then the property described by the rule graph holds at the state, which is then annotated with the matched rule name. (Thus, such rule names serve as atomic propositions of the temporal logic.)

Rule **e-leader** (Figure 2.7(a)) checks for the existence of a leader and ensures that it indeed has the lowest possible identity. (This particular rule assumes that the minimum identity is 1.) Rule **m-leaders**, shown in Figure 2.7(b), checks if there are two distinct leaders in the same state. The negation of the latter specifies a safety property that should hold on all states of the GTS, and that can be verified with model checking; the property holds if there is no counterexample to the formula $AG(!m-leaders)$ (expressed in CTL). Rule **e-leader** can be used to check for a liveness property of the system, expressing that all paths in the GTS eventually lead to the choice of a leader. This property holds in the GTS if we have no counterexample for the CTL formula $AF(e-leader)$. For the GTS in Figure 2.6, both these properties can be verified.

2.3 Graph transformation and program semantics

Graph transformation rules can, for instance, be used to simulate the execution semantics of a programming language. We illustrate this with a simple example.

Listing 2.8: A Java example of a circular buffer with three cells.

<pre> class Cell { Object val; Cell next; } class Buffer { Cell first, last; Buffer() { first = new Cell(); first.next = new Cell(); first.next.next = new Cell(); last = first.next.next; last.next = first; } </pre>	<pre> void put(Object arg) { if (last.next.val == null) { last = last.next; last.val = arg; } } void drop() { if (first.val != null) { first.val = null; first = first.next; } } </pre>
---	---

Figure 2.9: Graph representation of the state of a Buffer object after its constructor call. Fields `val` are implicitly initialised with a `null` reference.

Consider the snippet of Java code shown in Listing 2.8, which implements a circular buffer with three Cells. Each Cell has two fields: `next`, a reference to its adjacent position, and `val`, which can hold a reference to an Object. The state of a Buffer object in memory, immediately after the execution of its constructor, can be easily captured by a graph, depicted in Figure 2.9. In this representation, nodes stand for instances of classes and edges represent the references (object fields). The Buffer class has two methods, `put` and `drop`.

Method `put` inserts the given argument Object after the current `last` element, provided that the Buffer is not full. Figure 2.10(a) shows the graph transformation rule that models the execution of the `put` method. The LHS defines the nodes and edges of the host graph involved in the transformation and provides the conditions for the rule application, *i.e.*, `last.next.val == null`. The argument of the `put` method is assumed to be an arbitrary non-null Object, which in Figure 2.10(a) is identified by an edge labelled `arg`, incoming from a node in the frame of the method call. The RHS of rule `put` establishes the effect of the rule application, *i.e.*, the `next` pointer is moved and the argument is stored.

Figure 2.10: Two graph transformation rules whose effect mimic the semantics of their homonymous methods. The rules are in the single-graph GROOVE notation.

Method `drop` discards the `first` element of a non-empty `Buffer`. The corresponding transformation rule is given in Figure 2.10(b). The special edge labelled by `!=` prevents the associated node to be matched with the `null` node.

The exploration of all possible applications of the `put` and `drop` rules over the initial host graph shown in Figure 2.9 yields a GTS that captures all possible states of a `Buffer` object. If we consider that the objects stored in the buffer are distinguished by an unique identifier drawn from a finite set D , with cardinality $|D| = c$, the number n of states of the GTS in this example can be estimated *a priori*, with $(c - 1)^3 \leq n \leq c^3$. Assuming that D represents the set of Java integers (which cannot be null), we have that $c = 2^{32}$, and thus the value of n , even for such a small example, is already flabbergasting. In order to control such blow-up we need to abstract away irrelevant information.

2.4 An informal introduction to graph abstractions

The leader election example presented in Section 2.2.2 serves not only to illustrate the current capabilities of the GROOVE tool set but also to point out a limitation. While the desired safety and liveness properties of the protocol can be verified for the ring configurations with three processes, this result cannot be immediately transferred to rings with four or more processes. To do so, one can use larger start graphs, modelling a ring with $k > 3$ nodes, and generate the corresponding GTS at each increment of k . This approach, however, is greatly hindered by the combinatorial explosion that arises when trying to enumerate all possible ring configurations. In practice, the largest case that can be analysed by the current GROOVE version is $k = 7$, giving rise to a state space of almost 5 million states.

The situations described in the paragraph above and at the end of the previous section demonstrate the infamous and dreaded *state space explosion problem*, which is inherent to explicit state model checking. In order to fight this combinatorial explosion, several techniques are proposed to prune the search space, *e.g.*, partial-order reduction, symmetry reduction, bi-simulation minimisation, etc. However, while these optimisations can improve a tool performance by several orders of magnitude, they are still not sufficient for checking a system independently of its size. For example, ideally one would like to verify the correctness of a protocol such as the one in the leader election example for all

Figure 2.11: Transition system of the Buffer GPS, after data abstraction.

possible values of k .

Furthermore, there are important classes of systems that have infinite state spaces and therefore cannot be (fully) explored using traditional explicit state model checking techniques. The general idea behind the concept of abstraction is that we can throw away information from the concrete model that is not relevant for verifying the property of interest.

2.4.1 Abstract Interpretation

Continuing with the discussion of the previous section, the kind of information that can be considered irrelevant is dependent on the properties that one wishes to verify. For the circular buffer example, we might want to check if indeed no element is inserted if the buffer is full. To verify such property, it is not necessary to keep track of the concrete values stored in the buffer; it suffices to know that the values are *non-null*. This simplifies the GTS to only 4 states, shown in Figure 2.11, and makes the verification of the property a trivial task.

The abstraction just described can be placed within the theory of abstract interpretation, developed by Cousot and Cousot [CC77]. An abstraction from a set of concrete values C to an element of an abstract set A is given by an *abstraction function* $\alpha : 2^C \rightarrow A$, and conversely by a *concretisation function* $\gamma : A \rightarrow 2^C$. The elements of C and A are required to be ordered in a lattice and α and γ must be monotonic with respect to this ordering. A very simple abstraction based on our buffer example may take $C = D \cup \{\text{null}\}$, with an ordering $(\text{null} < D)$; and $A = \{\perp, \text{null}, \text{non-null}, \top\}$, with an ordering $(\perp < \{\text{null}, \text{non-null}\} < \top)$. Let $B \subseteq 2^C$ and $a \in A$, the abstraction and concretisation functions are defined as

$$\alpha(B) = \begin{cases} \perp & \text{if } B = \emptyset \\ \text{null} & \text{if } B = \{\text{null}\} \\ \text{non-null} & \text{if } B \subseteq D \\ \top & \text{otherwise} \end{cases}$$

$$\gamma(a) = \begin{cases} \emptyset & \text{if } a = \perp \\ \{\text{null}\} & \text{if } a = \text{null} \\ D & \text{if } a = \text{non-null} \\ C & \text{otherwise} \end{cases}$$

Value $\top$ is the most coarse approximation and represents every subset of C . On the other hand, $\perp$ is an inconsistent approximation and thus γ maps it to the empty set.

The key point of an abstract interpretation is that, with respect to the correctness properties that one wants to verify, the abstraction is an over-approximation of the concrete system. Thus, if a property holds on the abstract domain, it is guaranteed to hold on the concrete domain. This is the reason why we can check the correctness of the program of Listing 2.8 using the abstract GTS of Figure 2.11. However, it might be the case, due to loss of precision in the abstraction, that a property does not hold on the abstract domain but actually holds in the concrete domain, a so-called *false positive* error report.

2.4.2 Graph Abstractions

The example of the previous section is an interesting case of abstraction from the data domain of a program, which can be used to shrink the program state space to a reasonable size. However, such abstractions fail to cope with structures of unbounded size, *e.g.*, a linked list. Regardless of data abstraction, the transition system of such a structure is infinite and thus unsuitable to enumerative verification methods without some previous manipulation. What we need is a method to deal with the graph structure, *i.e.*, a method for *structural abstraction*.

Our graph abstractions are based on the concepts of shape analysis [SRW98, SRW02] and abstract interpretation. A *graph shape* is an abstraction that captures the underlying structure of a set of concrete graphs, acting as their representative in the abstracted domain. The basis of this technique falls within the same idea of abstract interpretation presented in the previous section, except that now the abstraction function maps a set of concrete graphs to a corresponding graph shape. Roughly speaking, the abstraction is essentially a morphism from a graph to a shape that satisfies some extra conditions. A graph that can be abstracted to a shape is called a *concretisation* of that shape.

Multiplicities

A single node or edge of a shape may represent several nodes or edges of some concrete graph. The abstractions defined in this thesis carry over counting information, that allow us to determine the number of summarised graph elements in a shape. To ensure that the universe of shapes is finite, counting has to be approximate, up to some parameterised bound of precision. Counting up to some bound is done using *multiplicities*.

A multiplicity represents a set of natural numbers. We write multiplicities as bold numbers, super-scripted with a $+$ sign if the multiplicity stands for “many”. Thus, for example, **1** and **3** stand for precisely one and three, respectively, and **2⁺** stands for two or more.

Neighbourhood abstraction

One possible type of graph abstraction is *neighbourhood abstraction*, which is based on neighbourhood similarity: two nodes are considered indistinguishable if they have the same incoming and outgoing edges, and the opposite ends of

Figure 2.12: (a) A neighbourhood shape representing an infinite set of bipartite graphs. (b), (c) Two concretisations of the shape in (a).

Figure 2.13: (a) A neighbourhood shape representing cycles of three nodes. (b), (c), (d) The three possible concretisations of the shape in (a).

those edges are also comparable. Graphs are abstracted by folding all indistinguishable nodes into one, where incident edges are also combined.

A *neighbourhood shape* is a graph together with a *node multiplicity* function that indicates, for each node of the shape, how many concrete nodes it summarises. In addition, shape nodes are partitioned into groups. Edges with the same source node and ending into nodes of the same group (or, respectively, edges with the same target node and starting in nodes of the same group) are not distinguished, and their concrete numbers are indicated by the *edge multiplicity* functions of the shape.

Figures 2.12 and 2.13 present two neighbourhood shapes and some of their concretisations. Node multiplicities are indicated at a top corner of the node box and dotted rectangles indicate the partitioning of nodes into groups. Edges have an associated multiplicity to each end point, called *outgoing edge multiplicity* when associated with the edge source and *incoming edge multiplicity* when associated with the target. Multiplicities can be shared by several edges; this is indicated in figures by a grey arc covering the related edges. All edges related in one of their end points have their other end in the same group of nodes.

The neighbourhood shape in Figure 2.12(a) represents an infinite set of bipartite graphs with two or more **A**-labelled nodes and three or more (2^+ plus 1) **B**-labelled nodes connected by *c*-labelled edges. Every **A**-node has at least two outgoing *c*-edges and all **B**-nodes except one have exactly one incoming edge; the remaining **B**-node has at least two incoming edges. Two possible concretisations

for this shape are shown in Figure 2.12(b,c).

Figure 2.13(a) shows a neighbourhood shape representing the set of concrete graphs with three **A**-nodes forming cycles of n -edges. There are three of such concrete graphs, shown in Figure 2.13(b,c,d).

Pattern abstraction

A second abstraction method developed in our work is *pattern abstraction*, where patterns are simple graphs describing the structures of interest that should be preserved in the abstract domain. Patterns are collected into *pattern graphs*, layered graphs that capture the hierarchical composition of smaller patterns into larger ones. Pattern graphs are then abstracted into *pattern shapes* by collapsing equivalent patterns.

In a pattern graph, nodes are “labelled” by patterns, *i.e.*, each node of a pattern graph has an associated simple graph. Patterns are related by morphisms, which are used as labels for the edges of the pattern graph. The key point of this construction is that pattern graphs and simple graphs are equivalent: a simple graph can be *lifted* into a pattern graph representation and, conversely, a pattern graph can be *flattened* back into its corresponding simple graph. Due to this correspondence with simple graphs, the universe of pattern graphs can be infinite, which, as with neighbourhood abstraction, requires a structural abstraction, in this case into pattern shapes.

A pattern shape is a pattern graph enriched with node and edge multiplicity functions. In contrast with a neighbourhood shape, multiplicities are associated with an edge of the pattern shape, instead of the edge end points. Figure 2.14 shows a pattern shape and its corresponding pattern graph concretisation. Nodes are drawn with dashed lines and the associated patterns are shown inside the node. Edges are depicted as arrows labelled with their corresponding morphism, with the exception that identity morphisms are not shown to avoid clutter. For pattern shapes, multiplicities are placed near their associated graph elements.

The pattern shape in Figure 2.14(a) represents sequences of three **A**-nodes connected by n -edges. Due to the multiplicities constraints, the shape admits a single concrete pattern graph (modulo isomorphism), shown in Figure 2.14(b). The flattening of this pattern graph yields the simple graph depicted in Figure 2.14(c), which corresponds exactly to the graph of Figure 2.13(d), modulo isomorphism. Thus, this example illustrates a case where pattern abstraction is more precise (*i.e.*, a shape admits fewer concretisations) than neighbourhood abstraction.

Abstract state space exploration

The abstraction of graphs into shapes (both neighbourhood and pattern shapes) serves to partition the concrete state space into similar states, where the concrete partition is represented in the abstract domain by a corresponding shape. However, because the concrete state space cannot be directly explored (since it is infinite or prohibitively large), the partition of similar concrete states cannot

Figure 2.14: (a) A pattern shape representing a cycle of three **A**-nodes. (b) The unique pattern graph concretisation of the shape in (a). (c) The simple graph obtained by flattening (b).

be directly computed. Thus, we need to operate only on the abstract level, which requires the application of transformation rules on shapes.

Shape transformation is defined separately for neighbourhood and pattern shapes, due to their distinct representation. The common requisite for shape transformations is that they must capture all possible graph transformations for all graphs in the set of concretisations of a shape. This condition is necessary to ensure that the abstraction simulates the original system. Given the definitions for shape transformations, we can perform an abstract state space exploration, which follows the same principle of its concrete version: recursively applying all rules of the grammar to all generated shapes. The result of this procedure is an (over-approximating and finite) shape transition system that can then be used for model checking, as usual.

All the concepts introduced in this section are properly formalised in Part II. The theory of neighbourhood abstraction is presented in Chapter 6 and practical aspects of its implementation are given in Chapter 7. The theory of pattern abstraction is given in Chapter 8.

Related work

The amount of research dedicated to system analysis and verification is enormous and, therefore, so is the amount of related work published. Thus, composing a comprehensive literature survey of such a large research field is a challenging and time consuming task. For this reason, in this chapter we limit our discussion to the investigations that we consider most similar to the work presented throughout the other chapters of this thesis.

We begin this chapter with a brief discussion on work focused on the verification of Java software, and on similar translations of Java code to graphs. We then proceed to mention some work on *guided search*, which is a setting where abstractions are used to provide heuristics for state space exploration. Finally, the larger part of this chapter is devoted to the discussion of graph-based abstraction techniques, and we conclude with an overview comparing their characteristics.

3.1 Java, graphs and model checking

Essentially, every compiler uses an internal representation of compiled programs that is similar to the abstract syntax graph discussed in Chapter 4. Our intention in that chapter is to make this representation explicit and to model it as a typed graph. Here, we mention other approaches for the analysis of Java code and their choice for program representation.

Java programs as graphs. First of all, the OMG meta-model for Java [Gro04] has a clear correspondence with the type declaration part of our Java type graph (see Figures 4.3 and 4.4). However, this meta-model “stops” before the executable level: it does not go below method declarations. Furthermore, it appears that the standardisation effort is so big that this definition suffers from a maintenance problem: the OMG meta-model is for Java 1.3, and no newer version seems to be forthcoming. A project with just the same aim, documented in [DMSS01], seems to have suffered the same fate.

The idea of generating type and instance graphs by using an existing Java grammar (rather than an existing compiler) was explored in [AP04] and demonstrated on a subset of Java. The motivation there was to enable model transformation rather than verification. Unfortunately, that work was never published.

Closest in spirit to our work is [CDFR04], which defines a semantics for Java on the basis of graph transformations. A large part of that paper is devoted to the definition of a type graph and the corresponding instance graphs; however,

this is entirely a manual effort, in that no attempt was made to use an existing grammar. No doubt partially as a consequence of this, the fragment of Java covered is relatively small.

Our proposed idea of translating Java source code to abstract syntax graphs was developed independently by Striewe, Balz and Goedicke [SBG10]. Their work differs mainly on the choice of the target tool: while we focused in creating input graphs for GROOVE, they chose to use AGG [Tae04], another graph transformation tool.

Model checking Java programs. The automatic extraction of a finite-state model from Java code for the purpose of model checking was addressed by Corbett *et al.* [CDH⁺00], with the Bandera tool set. They propose to use data abstraction as one of the techniques for building tractable models for verification. An interesting component of Bandera is the Slicer, which removes variables and structures from the code that are not relevant for checking the property of interest. In some sense, this can also be seen as a form of abstraction, although one that operates over the source code instead of the program state.

Anand, Păsăreanu and Visser [APV06] propose the use of structural abstraction and shape analysis in the context of symbolic model checking. In their setting, a program state is captured by a symbolic heap configuration, which is represented using a graph-based formalism. There are similarities between their method for symbolic state subsumption and matching, and our proposal to use shapes as graph abstractions. A drawback of their approach is the need for code instrumentation, which in our case we believe will not be required. An interesting aspect of the work by these authors is that the use of symbolic execution allows for the modular verification of compilation units, *e.g.*, libraries. The proposed symbolic execution technique was implemented as an extension of the Java PathFinder model checker [APV07].

Symmetry reduction by graph isomorphism. The use of graph isomorphism for symmetry reduction was investigated by Turner *et al.* [TLSB07], and Spermann and Leuschel [SL08], in the context of the ProB model checker. In their work, the internal model checker representation of a state was translated to a graph, which was then given to an external isomorphism checking tool: Nauty [McK81]. They report empirical results to show the effectiveness on the state space reduction. A large part of their work was devoted to the translation of states into a graph representation, a problem that we do not face in our setting, since our states are already graph-based. Other researchers address the problem of symmetry reduction directly over the internal state representation of their model checker of choice. This is the case of Lerda and Visser [LV01] for Java PathFinder, and Robby *et al.* [RDHI03] for Bogor.

3.2 Guided search

Guided search is a term used to denote several methods that can be employed for *partial* state space exploration. In contrast to *exhaustive* exploration, partial

traversal usually boils down to finding the shortest path between a given start and goal state. There is a relation of opposition between solution quality (the path length) and search effort (states traversed) and many heuristics can be used to guide the search. Several of such heuristics involve some form of abstraction.

In [HMZM96], Holte *et al.* tackle the area of problem solving in artificial intelligence. The authors define a so-called “explicit graph notation”, where the state space is represented by a labelled transition system (LTS), and they proceed to define abstraction algorithms that can be used to speed-up the search. One of such algorithms, called STAR, works by building state classes that are connected up to a certain abstraction radius. Despite having many similar concepts with our work on neighbourhood abstraction, the abstractions used by Holte are not structural abstractions; they operate on the LTS level and not on the state representation. Furthermore, the concrete state spaces considered in [HMZM96] are always finite.

In [EJLL06], Edelkamp *et al.* consider the problem of partial analysis/exploration of the state space of graph transformation systems. As in the work of Holte *et al.*, this amounts to a guided search over the concrete state space where abstraction can be used as an heuristic. Properties of interest for the analysis usually encompass existential checks for graph structures; *e.g.*, is a graph with a certain node and edge configuration reachable from the start state? Any abstraction that preserves reachability of the goal state in the abstract state space can be used to define an heuristic for the guided search in the concrete level. Since the graph abstractions presented in this thesis preserve reachability, they could in principle be used as the abstraction mechanism for an heuristic search. However, performance may be an issue, since computing the transitions of an abstract state is usually a rather expensive operation.

3.3 Abstraction methods

Abstraction is an essential ingredient in nearly all methods for system analysis and verification, and as such there is a vast body of work describing the use of abstractions in different domains. In this section, we discuss the related work that was studied during the development of this thesis.

3.3.1 Shape analysis

Shape analysis is the seminal work upon which our abstractions are based. While serving as inspiration to many researchers, shape analysis has spawned several branches of investigation in different domains. Here we focus our discussion on the two papers that present the shape analysis method in its entirety, and serve as the base references of the area.

Solving shape analysis problems in languages with destructive updating [SRW98]. This journal paper presents a technique for the static analysis of programs that perform destructive updates on memory heaps. Since the number of objects that can be created in the heap is unbounded, an abstraction is used to produce finite graphs that over-approximate the possible configurations

of heap structures, or, in other words, the possible “shapes” of the heap. The method can be used, for instance, to analyse structural invariants, *e.g.*, to check that the structure of an input list or tree is preserved by a program.

The shape analysis algorithm of the paper is presented and proven correct within the framework of abstract interpretation [CC77]. The algorithm is based on an abstraction of the memory, called *static shape graph* (SSG). A SSG is a finite directed graph that over-approximates the actual memory configurations that can arise during program execution. The shape analysis algorithm is an iterative procedure that computes a set of SSGs at every program point. In order to ensure that every SSG is of bounded size, multiple memory cells can be represented by a single *summary node* of a SSG.

Concrete acyclic structures may be abstracted to cyclic SSGs. Thus, in order to distinguish between cyclic and acyclic concrete structures, every SSG has an additional component represented as Boolean node predicates. Predicated *is-shared* (denoted $is^\#$), for example, indicates if a node of the SSG is pointed to by more than one memory reference. (In our setting, this sharedness corresponds to an incoming edge multiplicity larger than one – see Section 2.4.2 and Chapter 6.)

The paper defines a simple imperative programming language with single assignments and pointer traversal. The concrete semantics of such language is given in terms of *deterministic shape graphs* (DSG) and program statements are considered DSG-transformers. This allows an analogy with graph transformation, in the sense that both have a sort of “graph-based” representation, and that the transformation of a shape by a program statement can be seen as a sort of application of a graph transformation rule.

The abstract semantics of the language is defined in terms of sets of SSGs, with corresponding abstract transformations constructed for SSGs. One key point of these transformations is the need to *materialise* (*i.e.*, to extract) some nodes out of summary nodes. (Our abstractions perform an equivalent operation, and thus we once more borrow the same terminology.) Predicate $is^\#$ plays an important role during materialisation: it can be used to discard certain invalid configurations and thus it helps in limiting the number of possible SSGs at a certain program point.

Parametric shape analysis via 3-valued logic [SRW02]. Continuing the work from [SRW98], this paper expands and generalises the original shape analysis algorithm to create a parametric framework that can be instantiated in different ways to create distinct algorithms, providing varying degrees of efficiency and precision. The main novelty of this work is the use of 3-valued logical structures to represent heap configurations that can occur during program execution. Parametrisation is achieved by modifying the predicates used in the 3-valued logic; for instance, when working with lists, or trees, certain structural predicates can be added to the execution in order to improve the algorithm precision.

In this paper 2-valued and 3-valued logical structures are used to represent concrete and abstract stores, respectively. In other words, interpretations of unary and binary predicates encode the contents of variables and pointers. First-order formulæ extended with transitive closure are used to specify struc-

tural properties such as sharing, cyclicity, etc. Other predicates are also used to specify how the heap is affected by the execution of the statements in the programming language. One of such predicates, for example, is the *is-shared* predicate discussed before; others, more elaborate predicates, can now also be formulated: *e.g.*, an unary predicate expressing that a certain memory location (node) is reachable from a certain variable.

The representation of shapes as logical structures bears a strong resemblance to the logical approach to graph transformation developed by Courcelle [Cou90]. Actually, this similarity is further strengthened by the fact that the authors define a concrete visual syntax for the logical structure, which allows them to continue to draw shapes with the usual graphical notation (node as entities, edges as arrows). The key difference here lies in the program semantics, which is defined in terms of sets of logical structures. The transformation of such structures is carried out by logical predicate updates. These updates use the same formulæ for both 3-valued and 2-valued logic; the use of Kleene’s logical operations then relate the concrete (2-valued) world with the abstract (3-valued) world. Kleene’s logic has a third truth value that means “unknown”, which is used to indicate imprecise information about summary nodes.

An essential ingredient of the parametric framework proposed in the paper is the use of *instrumentation predicates*. These predicates are not explicitly part of the language semantics but are paramount for an efficient execution of the abstraction method. The challenge here is how to properly update such instrumentation predicates: these operations have to be provided by the user and usually require an specification by complex logical formulæ. Still, instrumentation predicates can only affect the algorithm precision: the method is proven correct regardless of the predicates used.

3.3.2 Petri graph unfolding

This section presents the extensive corpus of work developed by König *et al.* on Petri graph unfolding, a well-known method for the verification of infinite-state graph grammars.

A static analysis technique for graph transformation systems [BCK01]. The paper presents a technique for the verification of graph transformation systems with infinite state spaces. Graph transformation is defined for hyper-graphs with labelled hyper-edges, using an algebraic construction similar to the DPO approach but with additional conditions on the rules.

Given a graph grammar formed by a set of transformation rules and a start hyper-graph, the method constructs an *approximated unfolding*: a finite structure, called *Petri graph*, that is composed of a hyper-graph and a Petri net. The Petri graph can then be used to analyse the original system. In particular, the paper shows that every hyper-graph reachable from the start graph has a structure-preserving mapping to the (hyper-graph component of the) Petri graph. This guarantees that the Petri graph is an over-approximation of the original system for graph properties that are reflected by graph morphisms, *i.e.*,

if the property holds for the Petri graph then it holds for all reachable graphs of the transformation system.

Generally speaking, the Petri graph captures all structures that can occur in the reachable graphs of the system, and dependencies for rule applications are recorded by the Petri net transitions. Given that the causality between rule applications is greatly over-approximated in Petri graphs, usually only safety properties are considered in this setting.

Technically, the algorithm that computes the approximated unfolding works as follows. First, a Petri graph representing the start graph of the system is computed. Then, two operations (called *unfolding* and *folding*) are applied as long as possible to this Petri graph. The algorithm is shown to be terminating and confluent, meaning that the resulting Petri graph is finite and unique for a given graph grammar.

An unfolding construction has been previously proposed for graph transformation systems but it could only lead to a finite unfolding if the system had a finite state space. Since the systems considered in this paper are infinite-state, the technique was properly adapted to always yield a finite unfolding, at the cost of becoming an over-approximation.

The paper concludes with an example: checking the absence of deadlocks in a graph grammar modelling the dining philosophers problem. Philosophers are allowed to duplicate, leading to an infinite-state system. After computing the unfolding for the grammar, the final Petri graph was analysed using language automata.

Static analysis of distributed systems with mobility specified by graph grammars - A case study [BCK02]. In this paper the authors use the Petri graph unfolding technique introduced in [BCK01] to verify safety properties on an infinite-state graph grammar modelling a computer network. The grammar allows for the duplication of both network locations and processes. The property verified is that no process running in an unsafe location can reach a safe location protected by a firewall. This is the first appearance of the firewall grammar, later used by other authors (see, for instance, Section 5.5).

The unfolding of a grammar is constructed inductively by beginning from the start graph and at each step applying all rules in all possible ways, *without* deleting the LHS, and recording each occurrence of a rule and each new graph element generated in the rewriting process. The result is an acyclic branching structure describing the behaviour of the grammar. In particular, every reachable graph embeds in the unfolded Petri graph. Any marking of the Petri graph can be interpreted as a reachable graph of the grammar and the net transitions can be seen as occurrences of rewriting rules. This means that the unfolding explicitly shows the causal dependencies among rule occurrences and created graph elements.

The unfolding is usually infinite, even if the grammar is finite-state. To ensure that the Petri graph is finite, parts of the graph are folded, until a structural fix point is reached. This produces an over-approximation of the original grammar execution. The Petri net underlying the approximated unfolding can be analysed with standard Petri net techniques, for example, to establish a bound

on the number of edges with a certain label in reachable graphs.

Approximating the behaviour of graph transformation systems [BK02]. The authors continue the research on the unfolding Petri graph technique introduced in [BCK01]. In this paper the original work from [BCK01] is generalised, where a method for constructing finite approximations of the behaviour for certain types of graph transformation systems is introduced. The paper shows how to construct under- and over-approximations with arbitrary k -depth of precision. When k is increased, both approximations converge to the exact system behaviour.

Under-approximations are called k -truncations, where the unfolding of a grammar is stopped at causal depth k . For infinite-state systems, any truncation (regardless of chosen k) is an under-approximation of the system behaviour, in the sense that any computation on the truncation can be performed in the original system but not vice-versa. Still, truncations can be used to check certain liveness properties of the form “eventually A”. This analysis, however, can lead to false positives, *i.e.*, failure to verify a property may stem from the under-approximation and improving the precision k may show that in fact the property does hold.

Over-approximations are called k -coverings, represented by a Petri graph, as discussed in [BCK01]. The improvement over the original paper is that the k -covering approximation is exact up to causal depth k . Coverings can be used to check certain safety properties of the form “always A”. As in the previous case, this analysis can lead to false positives, *i.e.*, properties that do not hold in the covering may hold in the original system, since the covering is an over-approximation and may introduce spurious behaviour.

The graph transformation rules used follow the DPO approach, with the following restrictions on the rule span (these are the same restrictions from [BCK01, BCK02]): (i) the interface graph is discrete, *i.e.*, it contains no edges; (ii) no two edges in the LHS have the same label; (iii) the morphism between the LHS and the interface graph is bijective on nodes; and (iv) the LHS is connected. Condition (i) implies that rules can delete and produce but not preserve edges. Also, a rule cannot consume two edges with the same label (condition (ii)). Condition (iii) states that nodes cannot be deleted. The paper presents a way to circumvent this by using a sort of garbage collection for nodes that get disconnected.

The over-approximation algorithm is an improvement over the one given [BCK01], where now folding operations are only allowed after k steps of unfolding. This means that the resulting k -covering is exact up to causal depth k . When folding steps are performed the Petri graph will contain cycles.

The final part of the paper discuss the verification of properties on the approximations. The authors explain how their unfolding technique is an specific instance of abstract interpretation and show how $\langle \alpha, \gamma \rangle$ -simulations can be used to relate approximations and the original system. The advantage of such approach is that μ -calculus formulæ are preserved/reflected by the abstractions, and thus showing that an $\langle \alpha, \gamma \rangle$ -simulation exists is sufficient to allow the verification of temporal logic properties on the approximations. The authors then

identify fragments of the modal μ -calculus that are preserved. For a fragment with the diamond modality ($\Diamond$), a k -truncation can be used, whereas a fragment with the box modality ($\Box$) requires a k -covering.

The paper concludes with a μ -calculus formula using the box modality that expresses the correctness property of the firewall grammar. Verification of this property requires the use of a 1-covering because the analysis is inconclusive with a 0-covering.

A logic for analysing abstractions of graph transformation systems [BKK03]. The paper continues to extend the Petri graph unfolding approach that was originally given in [BCK01, BK02]. A main change from the previous papers is that here the presentation uses multi-graphs instead of hyper-graphs.

The authors introduce a monadic second-order logic ($\mathcal{L}_2$) over graphs that can express several interesting properties (*e.g.*, connectivity and cyclicity) and show how the formulæ of $\mathcal{L}_2$ can be verified on Petri graphs. In $\mathcal{L}_2$, quantification is allowed over edges but not over nodes. The logic is initially used to describe state predicates, *i.e.*, structural graph properties. Later on in the text, $\mathcal{L}_2$ is extended with μ -calculus constructs, allowing the reasoning on temporal properties.

The idea discussed in the paper is as follows. Given a graph grammar $\mathcal{G}$, its covering $\mathcal{U}(\mathcal{G})$ is computed, using the already established unfolding approach. The covering $\mathcal{U}(\mathcal{G})$ approximates $\mathcal{G}$ via a simulation consisting of pairs of graphs (sub-graphs of the Petri graph) and Petri net markings. Given a formula F on graphs, expressing a desired state property in $\mathcal{G}$, a corresponding formula $M(F)$ on the markings of $\mathcal{U}(\mathcal{G})$ is constructed such that, for any pair in the simulation, satisfaction of $M(F)$ on the marking implies satisfaction of F on the corresponding graph associated with the marking.

After presenting logic $\mathcal{L}_2$, the rest of the paper explains how this logic can be encoded into equivalent multi-set formulæ, *i.e.*, it deals with the translation from F to $M(F)$. A key point to observe is that the Petri graph is a finite structure, and since M formulæ are evaluated over the Petri graph, decidability and completeness are guaranteed. However, the encoding of $\mathcal{L}_2$ into multi-set formulæ may result in an exponential blow up in a translated formula size.

Verifying finite-state graph grammars: an unfolding-based approach [BCK04]. The authors continue the work on the Petri graph unfolding technique presented in [BCK01, BK02, BKK03]. The difference here is that the graph grammars considered for verification are finite-state, where previous papers dealt with infinite-state grammars. Taking such finiteness property into account, the authors present a partial-order on the coverings, allowing the generation of a finite unfolding that fully mimics the semantics of the original grammar.

The technique presented in the paper works for any finite-state graph transformation system, with the following restriction: graphs reachable from the start graph are considered equivalent not only up to isomorphism but also up to isolated nodes. Hence, there is no bound to the number of nodes generated, only to the nodes that are connected to some edge at any state.

The authors show that any finite-state grammar has a finite unfolding fragment that is complete, *i.e.*, that provides full information about the system regarding reachability. However, there is no constructive characterisation of the finite complete prefix, meaning that it is not feasible to implement an algorithm for the prefix computation. The authors then identify a subclass of graph grammars, called *read-persistent*, where the prefix computation is possible. An example of a class of read-persistent grammars consists of all grammars where any edge preserved by a certain rule is never consumed by any other rule.

The final part of the paper explains how the monadic second order logic $\mathcal{L}_2$ from [BKK03] can be used to express some graph properties of interest and how the formulæ can be verified on the covering.

Verifying red-black trees [BCE⁺05]. The paper shows how the techniques presented at the previously cited papers of this section can be used to verify the correctness of element insertion into red-black trees. A red-black tree is a balanced search tree that satisfies the following conditions:

- (S) The tree is sorted, *i.e.*, in-order tree traversal yields a sorted sequence of keys.
- (RL) The root and leaves are coloured black.
- (D) The number of black nodes on the path from the root is the same for all leaves.
- (R) No path from the root to a leaf contains two consecutive red nodes.

The paper models a red-black tree as a hyper-graph, with hyper-edges representing nodes of the tree. An initial graph grammar with 16 rules is given, describing a tree insertion operation. The paper then proceeds to verify that insertion preserves the properties given above. The authors assume that Conditions (S) and (RL) are already proven and that the result of a insertion is still a tree.

Condition (R) is checked using the Petri graph unfolding technique from [BCK01, BK02, BKK03]. Due to the restriction in the theory that a rule cannot have two hyper-edges with the same label in the LHS, the grammar has to be adapted before it can be used. Condition (D) is verified with another technique for typing graph rewriting systems. The method is quite general and powerful but is not automated.

AUGUR – A tool for the analysis of graph transformation systems [KK05]. The paper presents AUGUR 1, a tool for the verification of graph grammars based on the Petri graph unfolding technique discussed in this section. The authors give an overview of the tool structure and present a running example that models a network with private and public servers, where internal and external processes navigate via connections. The property of interest, namely that external processes cannot reach private servers, cannot be initially verified using the first computed unfolding. However, the tool allows abstraction refinement based on the generated counter-example, and after this refinement is performed, the property can be verified.

Counterexample-guided abstraction refinement for the analysis of graph transformation systems [KK06]. The paper continues to extend the work on Petri graph unfolding established in [BCK01, BK02, BKK03]. The new contribution is the addition of a counterexample-guided abstraction refinement to the unfolding used in the verification of graph transformation systems.

As usual, an infinite-state graph grammar is over-approximated by a finite Petri graph. Reachability of graphs is preserved by the approximation, *i.e.*, each graph reachable from the start graph in the original system can be mapped to a reachable marking of the over-approximating Petri net. The converse is not true: some markings reachable in the Petri graph do not have a counterpart in the original system. The sequence of events in the approximation that lead to such type of markings is called a spurious run.

A spurious run that violates the property that one wants to verify in the system is called a counterexample. Counterexamples are then used to refine the abstraction, by building a more precise Petri graph that can again be used to check the property. The basic idea of the refinement is to undo some folding steps, leading to a larger Petri graph. The correctness of the over-approximation is shown to be preserved by this new refinement.

The technique described in the paper was implemented in the tool AUGUR 1 and the authors also show some experimental results. They make a comparison of the verification task for two grammars, on one hand using k -coverings where the depth of precision is increased and on the other hand using counterexample-guided refinement. The second method clearly outperforms the first in the experiments conducted.

A temporal graph logic for verification of graph transformation systems [BCKLL06]. The paper extends the work in [BKK03] for verifying properties of graph transformation systems using abstractions. As in [BKK03], the theory in this paper is developed for multi-graphs instead of the usual hyper-graphs in other papers.

In the approach from [BKK03], properties are specified as formulæ of a propositional temporal logic where the atomic predicates are monadic second-order graph formulæ. In this paper, this aspect is generalised, where more expressive logics are considered, allowing the interleaving of edge quantifiers and temporal modalities. This added expressivity permits one to trace the history of graph elements over time.

Most of the paper is devoted to showing how the new graph logic can be encoded in Petri net markings formulæ and to proving that this encoding is sound and complete.

AUGUR 2 – A new version of a tool for the analysis of graph transformation systems [KK08a]. The paper presents the second incarnation of the tool AUGUR 1 [KK05], used for the verification of graph transformation systems. AUGUR 1 was focused on the task of approximated unfolding of graph transformation systems, and this focus did not allow for an easy extension of the tool with other algorithms and techniques. This led to several versions of the

tool, each with a different functionality. To overcome this problem, AUGUR 2 was planned with a more general and extensible software architecture.

The input of AUGUR 2 is a graph grammar, stored in the common GTXL format. An additional required input is a property to be verified. AUGUR 2 gives two options for specifying this property:

- As a regular expression with the set of hyper-edge labels as the alphabet, describing forbidden paths that should not occur in any reachable graph of the grammar.
- As a formula written in the first-order fragment of $\mathcal{L}_2$.

The running example of the paper is the same as in [KK05]: a network with private and public servers.

Towards the verification of attributed graph transformation systems [KK08b]. The paper presents an approach for the verification of attributed graph transformation systems. The approach is based on the Petri graph unfolding technique, which was originally developed for non-attributed systems. The paper discusses the extension of the unfolding method to attributed systems and shows how an abstraction refinement method can also be used in an attributed setting.

The authors consider attributed graph transformation systems to be labelled over an algebra. These systems are approximated by attributed Petri graphs which are like the original Petri graphs of [BCK01] but extended with colouring. After performing the original unfolding method, attributes are added to the resulting Petri graph, which can then be analysed as a coloured Petri net. This colouring is a safe operation because attributes do not affect the unfolding procedure in any way. Since the data types used for attributes are often infinite, the authors also introduce data abstraction based on the framework of abstract interpretation.

The second half of the paper discusses abstraction refinement in the attributed setting. In this new setting there are two ways to refine an over-approximation: either structural refinement or attribute abstraction refinement. One of the challenges is to determine which of the two cases applies. After analysing the generated Petri graph and searching for counterexamples there are four possibilities:

- (1) The property is verified.
- (2) A non-spurious counterexample (*i.e.*, an error) is found.
- (3a) The counterexample is spurious and the over-approximation is structurally too coarse.
- (3b) The counterexample is spurious and the attribute abstraction is too coarse.

For case (3a) the original refinement method from [KK06] can be used without modification. As for case (3b) the attribute abstraction is made more exact in a predefined way. However, in this case there is no guarantee that the spurious counterexample will be eliminated. In the tool implementation such refinement steps are done up to a certain limit, and if the limit is crossed the analysis terminates with the answer “don’t know”. The paper ends with a brief discussion over a solution for the leader election protocol. This paper marks the first

appearance of this case, which is also presented in Section 2.2.2.

A framework for the verification of infinite-state graph transformation systems [BCK08]. This journal paper summarises and extends the previous papers presented so far. In particular, all but one of the rule restrictions in [BCK01, BK02] are lifted. The restriction remaining is that nodes cannot be deleted by rules, but for modelling purposes the deletion of a node can often be simulated by leaving it isolated. After lifting the other restrictions, the theory now allows for reader edges in rules, as well as distinct edges with the same label in the LHS.

In order to be able to lift these restrictions, the authors had to resort to a more complex model of Petri net. In previous papers, the Petri net component of a Petri graph was a standard Place/Transition net. This paper, on the other hand, uses *contextual Petri nets* with an additional total ordering imposed on the places.

3.3.3 Hyper-edge replacement

In this section we present summaries for a collection of papers by Noll, Katoen, and others, where a technique is designed for the analysis of heap data structures and corresponding pointer manipulating programs. The method is based on hyper-edge replacement grammars, one of the existing graph transformation approaches mentioned in Section 2.1. This line of research led to the development of a tool called Juggernaut. Unfortunately, although the tool is referenced in some papers, no mention is made about its availability or where it can be obtained.

Abstracting complex data structures by hyper-edge replacement [RN08]. This is the first instalment of the work based on hyper-edge replacement for the analysis of heap structures. Similarly to shape analysis, the properties of interest to be verified are related to the structural consistency of the heap under program execution. For example, given a linked list as input, is the output of a certain procedure still a valid list? The challenge here is the same as with other methods discussed in this chapter: pointer programs usually have infinite state spaces and therefore some form of abstraction is required.

In this approach, states of the heap are modelled by hyper-graphs, and the authors employ hyper-edge replacement grammars to specify both the data structures and their abstractions. The main idea is to use rule applications in two directions. By using a backward application of a rule, a sub-graph of the heap can be collapsed into a single hyper-edge labelled with a non-terminal symbol, thus obtaining a heap abstraction. On the other hand, forward rule application makes concrete again certain parts of the heap that were abstracted.

This paper presents a minimal pointer manipulating programming language that is very similar to the one used in shape analysis [SRW98]. The language semantics is given in terms of admissible heap configurations (hyper-graphs satisfying additional constraints). The execution of program statements in the abstract heap thus involves the following steps:

1. Execution of the assignment, which is non-deterministic because the semantics is defined for sets of heap configurations.
2. Garbage collection to remove nodes that may become disconnected.
3. Partial concretisation of the structure to avoid inadmissible heap configurations, thus avoiding the definition of semantics for abstract heap parts. This is done with forward rule applications, which is a non-deterministic step since more than one rule might be applicable.
4. Re-abstraction of the heap by backward rule applications.

Steps 3 and 4 of the procedure above have clear counterparts in our abstractions (as presented in Chapters 6 and 8). Step 3, for instance, corresponds to a materialisation operation whereas step 4 is equivalent to a shape normalisation.

If the procedure is started with an invalid heap configuration, it may fail to terminate. Therefore, in order to guarantee finiteness of the abstract state space, one has to define a “heap compactor” that essentially collapses nodes and thus makes the abstraction no longer precise. The running example in this paper is a double-linked list structure.

The key difference between this approach and ours is that in this paper graph grammars are *used* as the abstraction mechanism. This is a direct contrast to our methods, that consider abstractions *for* infinite-state grammars.

Juggrnaut: graph grammar abstraction for unbounded heap structures [HNR10]. This paper is a direct continuation of [RN08] with more emphasis put into describing the Juggrnaut tool. This paper re-introduces the presentation of [RN08] with a different example: fully branched binary trees instead of double-linked lists. This new example is used to illustrate the application of the tool to the verification of the DSW tree traversal algorithm (used in some programming languages for garbage collection). This algorithm has already been verified by the shape analysis community using the TVLA tool. The properties checked are quite extensive, namely:

1. Pointer safety: no dereferencing of null pointers.
2. Structural invariants: if the input is a tree so is the output.
3. Completeness: every tree node is visited at least once.
4. Termination.
5. Correctness: every tree node is at the same position at the end of execution.

Properties 1 and 2 can be checked on-the-fly while the abstract state space is generated. On the other hand, items 3 and 4 require the use of LTL formulas for their verification. This in turn entails the extension of the state space exploration with a dedicated node marking procedure, that produces more concrete heap configurations but in turn blows-up the state space size (from 10,000 states to 3,000,000). Finally, property 5 requires an even more extended marking, that leads to an additional blow-up. In total, all properties can be checked in around 40 minutes, which is a good performance when compared with the one obtained with TVLA.

A local Greibach Normal Form for hyper-edge replacement grammars [JHKN11]. This paper is a continuation of the theory presented in [RN08], where the correctness conditions for a hyper-edge replacement grammar (HRG)

established in [RN08] are generalised to the new conditions given in this paper. The authors now work with grammars that are in Local Greibach Normal Form (LGNF), a normal form inspired by the Greibach Normal Form (GNF) from string grammars.

In this framework, terminal edges model variables and pointers, whereas non-terminal edges represent abstract parts of the heap. Thus, hyper-graphs are heap configurations that are partially concrete and partially abstract, where heap fragments relevant for the current program state are concrete while the remaining parts are abstracted in order to ensure a finite heap representation. Pointer statements such as assignments and object creation are performed on concrete graph parts only. Thus, if pointer assignments “move” program variables too close to abstract graph fragments, a local concretisation is carried out. To do so, the heap abstraction HRG is required to be in LGNF.

In [RN08] the restriction on HRGs is taken from the first GNF proposed for graph grammars given in the literature, which limits possible data structures that can be modelled to the ones where each object is referenced by a bounded number of objects. The new normal form presented in this paper lifts this restriction and leads to grammars with less rules. This is of interest to the abstraction method at hand because smaller grammars imply less non-determinism on partial heap concretisation, which in turn means a smaller abstract state space.

The paper restricts HRGs to a special kind, called heap abstraction grammars (HAGs), which have to be locally concretisable. This property ensures that every concretisation step is realised with a single forward rule application. The main contribution of the paper is a procedure to convert a given grammar into an equivalent one (*i.e.*, with the same language) that is in LGNF. The conversion requires four steps, which are briefly discussed in the text. When the authors compare their LGNF to the one originally given in the literature, the reduction in grammar sizes is considerable: from 135 to 36 rules, for an example grammar modelling the abstraction of a binary tree.

Juggernaut: an abstract JVM [HBJ11]. This is a continuation of the work discussed so far, but in particular this paper focus more on practical aspects of the method instead of theory, and thus is more directly connected to [HNR10]. The running example of this paper is a binary tree with linked leaves, *i.e.*, the leaves form a single-linked list.

Expanding their original Juggernaut tool, the authors now use hyper-graphs with typed nodes, where this typing is associated with declared types of an OO-language. (This paper focus on Java and Java byte-code but it can be immediately seen that other OO-languages, *e.g.*, C#, could be targeted as well.) Given this typed setting, the framework can now be used for the verification of Java byte-code. In this sense, the authors consider their tool a Java Virtual Machine (JVM), albeit an abstract one, *i.e.*, the outcome of interest for an execution is not the program output per se, but instead an analysis on whether certain data structure properties are preserved by the program.

Most of the paper is dedicated to defining an execution environment for a JVM in terms of graphs and transformations. This is very similar to the

proposal by Rensink for a software verification cycle, as discussed in [ZR11]. Albeit Java byte-code is tackled instead of Java source and the primitive data types are severely restricted (only Booleans), the complexity of the technicalities is quite high. An interesting point raised in the paper is that their method now provides not only heap abstractions but also stack abstractions, which in turn allows the tool to handle recursive functions with unbounded stack sizes.

3.3.4 Partner and star abstraction

An initial work developed by Rensink and Distefano led to the theory of *graph shapes* [Ren04, RD06]. Working independently, Bauer and Wilhem developed the similar technique of *partner abstraction* [Bau06, BW07] and showed its applicability with a case study modelling the wireless communication of cars in motorways. Further work on this case study led to the proposal by Backes and Reineke [BR10a] of the *star abstraction* method.

Partner abstraction. Partner abstraction is initially presented in Bauer thesis [Bau06] and was later published in [BW07]. The technique is geared towards the verification of dynamic communication systems that are modelled by graph grammars. Examples of dynamic communication systems include traffic control systems and *ad hoc* networks, which are usually characterised by the unbounded creation and destruction of system entities. This unboundedness is reflected in the modelled graph grammars, which are infinite-state.

The abstraction proposed is a two-phase method: in the first step, connected components are abstracted into *clusters*; in the second step, similar clusters are combined. The first step requires the identification of *partners*: nodes with the same state (*i.e.*, label) and the same communication neighbourhood (*i.e.*, other nodes in the cluster). To ensure a finite representation, equivalent partners are fused into *summary nodes* (borrowing the terminology from shape analysis). In the second abstraction step, isomorphic clusters obtained at the first step are also summarised.

The viability of the approach is shown on the basis of a large case study, involving a communication protocol for cooperating cars on a motorway. Given that the protocol features unconnected clusters of such cars (called *platoons*), the application is ideally suited for the abstraction proposed. The technique was implemented in the tool *hiralysis*, which was then used for the automatic analysis of the car platoon protocol.

Similarities among partner abstraction and the neighbourhood abstraction presented in Chapter 6 should come as no surprise: neighbourhood abstraction is a generalisation of partner abstraction, obtained by the joint work of Bauer, Rensink and others [BRKB07]. This generalisation, however, focused on the first step of the method described above: the notion of partner equivalence was extended to the definition of *neighbourhood equivalence*, which is parameterised by a radius. On the other hand, the second step for partner abstraction has no direct counterpart in neighbourhood abstraction: although grouping of similar clusters could be related to collapsing of shapes under isomorphism, the settings where these concepts are applied are distinct.

Method	Shape analysis	Petri graph	Hyper-edge replacement	Partner/star abstraction
Intended analysis	heap manipulating programs	graph grammars	heap manipulating programs	dynamic communication systems
Graph repres.	deterministic graphs	hyper-graphs	deterministic hyper-graphs	directed labelled graphs
Tool	TVLA	AUGUR	Juggernaut	hiralysis/astra
Explor.	invariant synthesis	reachability	reachability	reachability
Graph prop.	FOL with transitive closure	theory: monadic second order logic tool: FOL	markings	theory: CTL _X tool: none

Table 3.1: Summary of the abstraction techniques discussed.

Star abstraction. A more elaborate version of a graph grammar modelling the car platoon protocol was presented as one of the case studies for the Transformation Tool Contest of 2010 (TTC’10). In this case study, Backes and Reineke [BR10b] model several aspects of the communication protocol, focusing, in particular, on the merging of two car platoons. The same authors also elaborated a solution to the problem [BR10a], based on the concept of *star abstraction*.

Backes and Reineke identify certain parts of the merge procedure that give rise to complex network topologies, which cannot be analysed with partner abstraction without the aid of additional constraints provided by a human expert. The aim of star abstraction is to provide an abstraction method that can cope with these complex topologies without the need for manual intervention.

The abstraction comprises three steps. First, for each node v of a graph, the corresponding *star* of v is built. The star is obtained by removing all nodes from the graph except v and its partner nodes, and by removing all edges not incident to v . The second step is performed separately for each star and corresponds to the usual partner abstraction (first phase); the resulting structure is an *abstract star*, which has summary nodes. The third step of star abstraction is equivalent to the second phase of partner abstraction: in order to ensure that each abstract star is unique, only one isomorphic copy of each star is kept.

The authors developed a new analysis tool called *astra*, which is compatible with the input format used for *hiralysis*. The tool performs very well for the case proposed but this was the only published example of use for the tool at the time of this writing.

3.3.5 Summary of abstraction techniques

Table 3.1 summarises the graph abstraction techniques presented in this section with respect to five criteria, given in the first column of the table, and discussed in the following.

Intended analysis. This line lists the artefacts which are intended to be analysed with the method. Both shape analysis and the abstraction based on hyper-edge replacement are targeted for the verification of programs describing pointer manipulations on the memory heap. On the other hand, the Petri

graph unfolding and partner/star abstractions are designed for the analysis of infinite-state graph grammars. In particular, the graph grammars analysed by partner/star abstraction model dynamic communication systems.

Graph repres. This item shows the graph representation used by each method. Reflecting their intended use, the graphs from shape analysis and the hyper-edge replacement abstraction are *deterministic*, meaning that a node may have at most one outgoing edge with a certain label. The graphs used by partner/star abstraction are very similar to the ones used in GROOVE. The most general graph representation are the hyper-graphs defined in the Petri graph method.

Tool. This line presents the accompanying tool for each abstraction technique. At the time of this writing, Juggernaut is the only one among these tools that still seems to be under active development.

Explor. This entry shows the expected outcome of the “exploration” of the artefacts. (The term here is a bit of a misnomer since some of the abstractions are not based on state space exploration.) The goal of shape analysis is the synthesis of invariants at all program points. The other methods in the table address only the problem of reachability, meaning that they cannot produce explicit traces representing sequence of rule applications. This is a point where our abstractions provide an advantage: we represent the abstract state space as a transition system, which indicate the inter-dependencies among rules.

Graph prop. This line lists which kind of graph properties can be represented and analysed with the method. Shape analysis provides first-order logic (FOL) extended with transitive closure, which allows for the representation of reachability properties, for example, to check if a heap object is reachable from the stack via a sequence of pointers. Similar properties can be stated in the theory for the Petri graph unfolding method, which specifies a monadic second order logic for graphs. However, only the first-order fragment of this logic is implemented in the AUGUR tool. The hyper-edge replacement abstraction have no associated logic for graph properties but special heap variables called *markers* can be used to check certain heap properties. Generally speaking, these markers can represent a fragment of LTL. The partner/star abstraction theory specifies a temporal logic for describing the evolution of graphs, but this logic was not transferred to the tool.

From Java to graphs

This chapter presents two of the transformations composing the conversion of Java code to graphs that was discussed in the introduction (see Figure 1.2). The main challenge here lies in dealing with a real imperative programming language, which has a complex specification. Thus, the creation of a conversion tool that supports the entire language definition is far from a trivial task.

The contents of this chapter were originally published in [RZ09a], with an accompanying technical report [RZ09b]. The original text of these publications have been modified to be up-to-date with current GROOVE capabilities. For instance, at the original time of writing, GROOVE did not support typed graphs.

Typing is an additional GROOVE functionality that was not discussed in Section 2.2, since it is only relevant for the contents of this chapter. Another unique characteristic of this part of the thesis is that here we make use of *attributed* graphs, which are also supported in GROOVE.

4.1 Introduction

The process of converting Java programs to graph models that can be consumed by GROOVE involves several steps, which were briefly introduced and discussed in Section 1.5. In this chapter, we focus on two of these steps: the “compilation” of Java source code into *syntax graphs*, and the construction of control flow information on top of syntax graphs to obtain *program graphs*.

A detailed view of how these transformations are implemented is given in Figure 4.1. The development of the graph compiler actually produced two artefacts, the compiler itself, and a Java *type graph* model to be used in GROOVE. The graph compiler was implemented in a way that ensures that the *instance graphs* produced conform to the type graph representation.

We based our graph compiler on the Java compiler from Eclipse [Ecl], which has an open source code. This was done by extending the Eclipse compiler source with an special visitor component, that traverses the AST of the input Java program and outputs a graph representation in the GROOVE format, as illustrated in the left part of Figure 4.1. In parallel with this visitor implementation, we inspected the Eclipse compiler source and manually constructed the corresponding Java syntax type graph. These steps are described in detail from Section 4.2 to Section 4.4.

The right part of Figure 4.1 deals with the construction of control flow information on the syntax graph produced by the compiler. This construction is

Figure 4.1: Detailed view of the conversion process from Java code to graphs. The left part of the picture shows the graph compiler construction and its use, the right part illustrates the elaboration of control flow information.

specified by a graph grammar and thus we use GROOVE to perform the transformation. This flow construction grammar is discussed in Section 4.5; it was developed along the lines of previous work by Smelik, Rensink, and Kastenberg [SRK06].

In this chapter we use the concepts of a type graph and typed instance graphs in an informal level. This is done because we are interested in more practical aspects, such as tool support. A more formal presentation can be developed, for instance, using the approach outlined in [KR08].

4.2 Preliminaries

Nodes and edges of a type graph are used as node types and edge types of the instance graphs, *i.e.*, every instance graph has a morphism to a type graph, which associates an unique type to each instance graph element. Elements of an instance graph are called instance nodes and instance edges.

An instance graph is a normal GROOVE host graph, with the usual visual representation, as described in Section 2.2. A type graph, on the other hand, has the following additional structure.

- Every node and edge has an associated unique *label* (or *name*). This is also taken as the label of the respective instance nodes and edges.
- There is a binary acyclic *inheritance* relation over nodes. It is required that the node type of the source of each individual edge instance is smaller or equal, according to this inheritance relation, than the source of its edge type.
- Some edge types are marked as *composition* edges. Such edges encode a *part-of* relation: the target node instances are considered to be part of the source node instances with which they are connected. Since a node cannot be part of more than one other node, composition edge instances should

form a tree within the instance graph.

- Some composition edges are marked as *ordered*. For all such edges there exists a total ordering among the target node instances that are connected by edge instances to (and hence are part of) any given source node instance. Edge ordering is not supported in GROOVE, so we use additional integer attributes in the instance graphs to circumvent this limitation. The ordering is encoded by adding extra integer index attributes to the target nodes, which contain their sequence number within the ordered list.
- Nodes may have associated attributes, such as Booleans or integers. Edges are not attributed.
- Every edge has an associated *multiplicity*, which is a range $i..j$ of natural numbers with $i \leq j$, or $j = *$. The multiplicity indicates how many edge instances there should exist for every individual source node instance, where $*$ stands for an unbounded number.

Graphically, we use the following notations.

- Inheritance is indicated by special edges with open-ended arrows.
- Composition edge types have black diamonds at their source end.
- Ordered edge types are marked {ordered} at their target end.
- Multiplicities are written at the edge target end.
- Attributes are included in their respective node as `name : type` in type graphs, and as `name = value` in instance graphs, where `name` is the attribute name; `type` is one of the six attribute types (boolean, double, float, integer, long, or string); and `value` is the attribute value.

4.3 Approach

We divide the task of constructing syntax graphs from source code into two main steps: (i) defining the type graph that represents the syntactical elements of the chosen programming language, and (ii) develop a tool that constructs a valid syntax graph from syntactically correct code. A syntax graph is considered to be valid when it is an instance of the type graph developed in step (i). Essentially, the work to be done in (ii) boils down to writing a compiler that produces a syntax graph as its target language, instead of machine code.

To decide which was the more adequate approach for solving the given task, we enumerated a list of requirements that ideally should be fulfilled.

1. *Comprehensive and systematic.* The approach should allow us to elaborate the type graph in an organised and systematic way, such that in the end the process yields a model that covers the whole language.
2. *Automation.* Manual execution of steps is tedious and error-prone. Although we do not expect that a solution for the presented task can be fully automated, manual intervention should be kept to a minimum.
3. *Re-use of existing technology.* Since the construction of a compiler for a real programming language is a quite complex and time-consuming task, it is desirable to reuse available tools and components as much as possible.
4. *Flexibility.* The method applied for a certain programming language should be, to some extent, reproducible for similar languages.

Among the possible solutions that were considered but later discarded we cite:

- *Type graph construction from the natural-language Java specification.* Although flexible, this approach was considered too ad-hoc to be of interest.
- *Construction of a graph compiler from scratch.* This was the approach taken in previous research where a proof-of-concept case was presented for a small object-oriented programming language [KKR06]. Although feasible for simple languages, this approach is unattractive when working with a language that has complex syntax and semantics, such as Java.
- *Use of existing components.* There are some Java grammars¹ and parsers² that could be used for the compiler construction, however some steps of code analysis such as name and type resolution would still have to be done manually, a work that we deemed unnecessary. Furthermore, most of these tools do not receive much maintenance and therefore are not kept up to date with the language specification.

After discarding the approaches above, we decided to adapt an open-source Java compiler for our purposes. In doing so, the implementation effort is kept to a minimum, since we have only to modify the code generation phase of the compiler to construct the syntax graphs. Also, by analysing the compiler source code we are able to elaborate the type graph model in a very straightforward way. Thus, with this solution, the definition of the type graph and the construction of the syntax graph generator go hand in hand, which guarantees that a syntax graph generated from code is compliant with the type graph.

One possible drawback of this approach is the loss of freedom in the design of the graph representation; by using a certain compiler we are somewhat restricted by its structure. However, given the other benefits of the approach this is considered an acceptable compromise over the requirements.

4.3.1 Creating the type graph

To develop the chosen approach we used the Eclipse Java Compiler. This compiler is also written in Java, and the source code is available for use under the Eclipse Public License. The compiler source is divided into several packages, among which package `org.eclipse.jdt.internal.compiler.ast`³ is of particular interest, since it is where the classes that compose the abstract syntax tree (AST) built by the compiler are grouped. By analysing the package contents we are able to construct the type graph model, which is presented in Section 4.4.

The `ast` package contains, for example, classes like `Expr` and `Stmt`⁴ to represent expressions and statements of the Java language. In fact, every syntactical element of the language has a corresponding class in the `ast` package and those classes are grouped in a certain hierarchy. The topmost class is `ASTNode`, which defines a common super type for all elements of the AST. The `ast` package also

¹<http://wwwantlr.org/grammar/list>

²<http://code.google.com/p/javaparser/>

³Through the rest of the chapter we adhere to the following convention: elements of Java code are shown in typewriter font, while elements of the type graph or instance graphs are shown in **sans serif** font.

⁴In fact, these are abbreviations of class names in the compiler source code. We make this change here for presentation purposes, in order to save space. The full list of abbreviations used is given in Figure 4.3.

Figure 4.2: Example of type graph construction from the compiler source code.

provides a visitor interface, which defines methods to navigate over the AST nodes in a DFS-like manner.

The way the type graph is constructed from elements of the `ast` package can be better explained with an example. Figure 4.2 shows on the left the relevant code of the class that implements an `if` statement, and on the right the corresponding part of the type graph constructed from this code. We start with the class name, `IfStmt`, that gives rise to an homonymous node type in the type graph. Also, since `IfStmt` is a subclass of `Stmt` we create another node type for the super class, with an associated inheritance edge. Class fields referencing other classes of the AST become compositions (in some cases, ordered ones) in the type graph, with labels matching the field names. This is the case for the fields named `condition`, `thenStmt`, and `elseStmt` in the example of Figure 4.2. Additionally, the way the visitor pattern is implemented in the class provides some guidance over the multiplicities of the compositions just created. From the implementation of the `traverse` method we see that fields `condition` and `thenStmt` are always visited. Therefore we conclude that the **IfStmt** node type must have mandatory `condition` and `thenStmt` compositions, a fact that is illustrated by the 1 multiplicities in the type graph. On the other hand, the check for non-nullness of the `elseStmt` field indicates that it may not always exists, thus we mark the multiplicity as 0..1.

By manually analysing the classes in the `ast` package in the same way as described in the example above we can elaborate a large part of the type graph. This provides us with the comprehensive and systematic approach for the construction that we sought. However, there are some elements of the Java static semantics that have to be handled in a case-by-case basis. An example of these elements are associations that resolve name and type references, which correspond to binding edges on instance graphs. The intuition for identifying where these bindings must be created is simple: any reference must be bound to a corresponding declaration.

4.3.2 Constructing syntax graphs from code

To construct syntax graphs from Java code we changed the back-end of the Eclipse Java Compiler; by stopping the compiler after parsing and code analysis but before machine code generation we can profit from the work done until this stage. Specifically, name and type references are already resolved, simplifying the construction of the syntax graph.

We developed a syntax graph generator that implements the AST visitor interface provided by the compiler. The generator traverses the AST performing the following steps.

- For each node in the AST a corresponding node in the syntax graph is created. The class type of an AST node is obtained via reflection and stored as a type label of the syntax graph node.
- For each reference field in an AST node we create a corresponding edge in the syntax graph. This step is trivial since all bindings were already constructed.

We created a test input program for each node type in the type graph. With these test cases we can inspect the syntax graphs produced by the compiler and check for implementation errors. An example of such input program is given in Section 4.4.3, along with the syntax graph generated. The complete set of test cases can be found in the accompanying technical report [RZ09b].

4.4 Java syntax type graph

In this section we present the type graph for the executable constructs of Java. Language elements that do not have an effect on the execution of a program, such as comments and annotations, were intentionally left out. Apart from these the type graph covers the whole language specification up to version 1.6.

Given the size of the model, we present it in parts for ease of understanding. First we discuss the inheritance structure of the type graph, showing the hierarchy of node types. Some of the relations among node types are given subsequently in separate figures. For the complete list of node types we again refer the reader to the corresponding technical report [RZ09b].

4.4.1 Inheritance structure

The inheritance structure of the type graph is shown in Figure 4.3. It is composed of 75 node types, mapped directly from the compiler classes. The 13 nodes with labels in *italic* correspond to abstract classes in the compiler.

One interesting abstract node type is **Stmt**, which not only has several concrete subtypes representing a variety of language constructs, but also has four other abstract subtypes, including **Expr**. This unusual relation between expressions and statements comes from the compiler implementation. From a semantic point of view, some expressions in Java, *e.g.*, assignments, may be used as statements. In the compiler source code, the designers explain that in order to avoid the creation of wrappers around expressions that are used as statements, they decided to make **Expr** a subclass of **Stmt** and let the parser handle incorrect usage of an expression as a statement. Thus, given the approach chosen for the type

Figure 4.3: Inheritance structure of the Java type graph.

graph construction, the model is bound to inherit the same design decisions made by the crafters of the compiler.

Figure 4.4: Type declaration node type.

Figure 4.5: Field declaration node type.

4.4.2 Node types, attributes and associations

Figure 4.4 depicts the **TypeDecl** node type, which represents a class or interface declaration in Java. From the figure we see that a type declaration has a **name** attribute and is composed of zero or more method declarations and zero or more field declarations. (The latter are present only when the type declaration represents a class; interfaces cannot have fields.) The field composition is marked as **{ordered}** since fields that appear later in a class can have initialisation expressions that refer to previously declared fields. Thus, during the construction of an object, its fields must be created on the same order of their declaration in the class. Additionally, to represent inheritance, the type declaration has two compositions with type references. A class can implement zero or more interfaces (**superInterface** composition) and can extend at most one class. The multiplicity of the **superClass** composition is **0..1** because only explicitly declared inheritances are indicated, *i.e.*, a type declaration that is a direct subclass of **Object** does not have a **superClass** composition. Finally, the **memberType** composition represents the Java concept of nested classes.

A field declaration is shown in Figure 4.5. It has a **name** attribute, a mandatory composition with a type reference, and an optional **initialization** expression. It also has a Boolean attribute, to indicate if the field is declared to be **static**. There is no need to represent others field modifiers, such as **public**, **private**, etc., because these are statically checked by the compiler and do not have influence on program execution.

Figure 4.6 presents an abstract method declaration, which is an abstract node type. Although it can only exist as either a method declaration or a constructor declaration, the common attributes and compositions of both these concrete node types are summarised within the abstract type. From the figure we see that every method declaration is formed by zero or more arguments (usually called formal parameters) and zero or more statements. The **signature** attribute indicates the argument types. Also, a method declaration can point

Figure 4.6: Abstract method declaration.

Figure 4.7: Message send node type (method call).

to which exceptions it can throw, and may have a list of type parameters if the method is generic.

Figure 4.7 shows the static structure used to represent a method call. In Java it is not always possible to provide a static binding to method calls due to polymorphism, thus some bindings have to be resolved at run-time. A **MessageSend** has a mandatory receiver, which is a reference to the object that triggered the call; a **signature** attribute, that is used to bind the call to a method declaration; and a list of arguments, that constitute the real parameters to be matched with the formal parameters in the method declaration. A method call may also have a list of type arguments, if the invoked method is generic.

Figure 4.8 depicts the node type for name references. It is always possible to statically determine to which variable declaration a name reference is bound. Note that the **refersTo** edge is not a composition. This is due to *aliasing*, i.e., a variable declaration may have an arbitrary number of references.

4.4.3 Example of a syntax graph

To illustrate how some of the node types and associations presented in the previous section may appear in a syntax graph, we conclude this section with an example. Figure 4.9 shows a small piece of Java code and its corresponding syntax graph. The graph has a **TypeDecl** node, which corresponds to the node type shown in Figure 4.4. The name of the declared class is stored as an attribute of the node, which also has three outgoing edges that correspond to the field and method compositions. The **FieldDecl** nodes are automatically given an extra **index** attribute to cope with the requirement that field edges must be ordered. The syntax graph also has a **MethodDecl** node which is a sub-type of the node type given in Figure 4.6. Finally, note that name and type reference nodes have an outgoing edge that binds the reference to its corresponding declaration. We assume the existence of a “system” compilation unit, where the language

Figure 4.8: Name reference.

Figure 4.9: Example of a syntax graph built from Java code. Nodes inside the dotted box are part of a “system-wide” compilation unit.

primitive types and also the classes in `java.lang.*` are declared. Part of this “system” compilation unit is shown at the bottom of Figure 4.9 (within the dotted box), with the declaration of primitive type `int`.

4.5 Control flow grammar

The second transformation step of the translation in Figure 1.2 is control flow construction. This step is performed by a graph grammar that traverses a given ASG and annotates the syntax elements with flow information. The result of this transformation is a *program graph* (ASG + control flow), named as such because it can be “run” by a graph grammar modelling the language execution semantics.

Figure 4.10 shows the type graph for the additional flow types introduced.

Figure 4.10: Type graph for control flow elements.

(a) Field declaration flow build.

(b) Flow build of block statements.

Figure 4.11: Two example rules of the control flow building grammar.

Execution control flows along **FlowElem** nodes, following *flow* edges. We connect the control flow type graph and the Java syntax type graph by taking the top type of the syntax graph, **ASTNode**, and making it a sub-type of **FlowElem**; thus, every syntactical element is actually a flow element.

Flow elements can be marked as *entry* or *exit* points of code blocks by additional edges. Generally speaking, the execution of a block starts at its *entry* element, proceeds along *flow* edges and concludes at the block *exit* point, further indicated by an **Exit** node.

Sequential statement composition yields a simple linear flow graph; however, this simplicity quickly disappears in the presence of flow-disruptive statements. Examples of such elements are *return* and *throw* statements, which can abort the block execution at the point where they occur. These cases are indicated by the specialised node type **Abort**. In a similar vein, conditional branching must be also taken into account. For example, upon reaching an **IfStmt** the conditional expression is evaluated and execution proceeds according to the Boolean result obtained, along one of the two **Branch** nodes that has the appropriate *on* attribute.

The control flow building grammar traverses the AST in post-order, thus, when building the flow information of a certain syntactical element, all its children were already previously visited. The AST node to be processed next in the traversal is marked with a *build* flag. Figure 4.11 presents two sample rules of the grammar.

The rule in Figure 4.11(a) constructs the flow information for a field declaration. The rule assumes that the associated initialisation expression was already traversed, as indicated by the presence of an eraser entry edge. The entry point for the field declaration thus becomes the entry of its initialisation expression, and after the field is initialised, the control flows to the subsequent statement, identified by an exit edge.

Blocks statements are built in a reverse order, from last to first. This is illustrated by the rule shown in Figure 4.11(b). Statement *S* was already built (as indicated by its entry edge), and the rule marks the predecessor statement of *S* (identified by the test `index ?= S.index - 1`) as the next node to be built in the traversal.

The flow grammar is composed of 195 rules, each of which is similar to the ones given in Figure 4.11, but dealing with other node types of the AST. This large number is due in part to the several different statement types of Java. Another complicating factor is the handling of flow-disruptive statements; several rules are necessary in order to properly cope with all possible cases that can arise, for instance, when an exception is thrown. Nevertheless, after the language semantics was properly understood, the construction of the grammar was reasonably straightforward, and served as an interesting modelling exercise. Furthermore, working with such a large grammar helped us to gauge and expand the usability of GROOVE.

We conclude this section with an example. Figure 4.12 shows the control flow graph built for the ASG given in Figure 4.9, with edges from the ASG grayed-out to ease the flow graph visualisation. The flow structure has two unconnected parts: the top one corresponds to code that is executed during an object initialisation, whereas the bottom one indicates the flow of a method call. Navigation over these flow graphs goes as explained previously, with the additional condition that abort edges must be followed when no outgoing flow edge exist.

As stated in the beginning of this section, a program graph such as the one in Figure 4.12 can be executed by an appropriate grammar. The construction of such execution grammar for Java is currently ongoing, as part of a student assignment. This construction is similar to the work developed by Kastenbergh [Kas08].

4.6 Conclusion

To sum up, the contributions of this chapter are the following.

- We presented a type graph that covers all executable elements of the Java programming language.
- We showed a straightforward and systematic approach for the elaboration of the type graph model, by analysing a compiler source code. Although the method described focused on Java, we believe that it can be adapted (with varying degrees of difficulty) to other programming languages as well.
- We explained how the back-end of a compiler can be modified to automatically construct a syntax graph representation from source code.

Figure 4.12: Flow graph constructed over the ASG of Figure 4.9.

- We discussed some aspects of a GROOVE grammar that can be used to construct control flow information over syntax graphs.

Apart from its intended use as described in Section 1.5, the Java type graph presented in this chapter can be of interest to any graph-based technique that relies on a representation of programs as graphs. As one example, suppose a visual programming/modelling tool that generates Java code from a graph; this could for instance, be used in the context of graph transformation-based model transformation [ALP07] or code refactoring [BGK01]. By enforcing the instance graphs to be properly typed, the tool can generate syntactically correct code.

At this point the reader might suspect that the visualisation of an instance graph can usually become verbose and confusing, even for small fragments of input code. Although we present Java graphs as pictures in this chapter, it should be stressed that we do not necessarily promote graphs as visual representations of programs (in fact, the source code is much more convenient for this purpose). Instead, the intention is to use the generated instance graphs as data structures, that can then be programmatically manipulated by GROOVE without user intervention.

To conclude, the translation of Java programs to graphs discussed in this chapter has the following qualities:

- *Comprehensiveness.* The translation covers the entire Java language specification up to version 1.6.
- *Automation.* The translation is automated with a proper tool support.
- *Maintainability.* The connection between our graph compiler and the original Eclipse Java compiler depends on a single interface declaration. This decoupling between the original compiler code and our extension makes the latter simpler to maintain.
- *Accessibility.* The explicit graph representation of programs used here make their analysis and manipulation easier, *i.e.*, this graph-based representation is more accessible to other tools and methods.
- *Executability.* The resulting program graphs are “executable”, *i.e.*, given a graph grammar modelling their execution semantics they can be “run” in GROOVE.

The graph compiler, the Java syntax type graph definition and the flow building grammar are all available for download at the address <http://groove.cs.utwente.nl/downloads/java2groove/> .

Part II

Graph abstraction

Essential concepts

In this chapter we present common concepts and definitions that are used throughout the rest of the thesis. In particular, we begin by presenting a common *graph* definition that is later extended by other structures such as labelled graphs and shapes. We then define *simple graphs* and *simple graph transformations*, which are fundamental concepts for our notion of semantics of a graph transformation system, where the behaviour is modelled by a labelled transition system. The penultimate part of this chapter deals with *multiplicities*, values for approximated counting over natural numbers. Multiplicities are used in shapes (both neighbourhood and pattern shapes) to keep track of the amount of graph elements collapsed by an abstraction. We conclude the chapter with a discussion about property preservation among (bi-)similar transition systems.

5.1 Graphs

In its basic form, a graph is composed of nodes and directed binary edges.

Definition 5.1 (Graph). A graph is a tuple $G = \langle N_G, E_G, \text{src}_G, \text{tgt}_G \rangle$ where

- N_G is a finite set of nodes;
- E_G is a finite set of edges, disjoint from N_G ; and
- $\text{src}_G : E_G \rightarrow N_G$ and $\text{tgt}_G : E_G \rightarrow N_G$ are mappings associating each edge to its source and target nodes, respectively. $\triangleleft$

Often, we equate G with $N_G \cup E_G$ for a short-hand notation. For a node $v \in N_G$, we consider the set of edges *outgoing* from and *incoming* to v , defined as $v \triangleright_G = \{e \in E_G \mid \text{src}_G(e) = v\}$ and $v \triangleleft_G = \{e \in E_G \mid \text{tgt}_G(e) = v\}$, respectively. Graphs are related by *morphisms*, structure preserving functions over nodes and edges.

Definition 5.2 (Graph morphism). A graph morphism between graphs G, H is a function $m : (N_G \cup E_G) \rightarrow (N_H \cup E_H)$, such that

- $m(N_G) \subseteq N_H$;
- $m(E_G) \subseteq E_H$;
- $\text{src}_H \circ m = m \circ \text{src}_G$; and
- $\text{tgt}_H \circ m = m \circ \text{tgt}_G$. $\triangleleft$

In words, a graph morphism maps nodes to nodes and edges to edges, and is compatible with source and target mappings. If function m is injective (surjective, bijective) then the morphism is called injective (surjective, bijective). A

Figure 5.1: (a) Simple graph representing a single-linked list with five elements. (b),(c) Two simple graph transformation rules.

bijective morphism is also called an *isomorphism* and we write $G \simeq H$ to denote that there is an isomorphism between G and H .

Graph G is a *sub-graph* of H , written $G \subseteq H$, if $N_G \subseteq N_H$ and $E_G \subseteq E_H$. If $G \subseteq H$, we use $\text{emb}(G, H)$ to denote the *embedding* of G into H , *i.e.*, the identity over G .

Some additional notation:

- We write id_A to denote the identity function on set A .
- Given a function $f : A \rightarrow B$, we use dom_f and cod_f to denote the domain A and co-domain B of f , respectively. For a set $C \subseteq A$, we write $f|_C$ to represent the restriction of f to C , *i.e.*, $f|_C : C \rightarrow B$.
- The union of functions f and g is defined as $f \cup g : (\text{dom}_f \cup \text{dom}_g) \rightarrow (\text{cod}_f \cup \text{cod}_g)$ and the intersection as $f \cap g : (\text{dom}_f \cap \text{dom}_g) \rightarrow (\text{cod}_f \cap \text{cod}_g)$. These are defined only if for any $x \in (\text{dom}_f \cap \text{dom}_g)$, f and g agree on the value for x , *i.e.*, $f(x) = g(x)$.

The *union* and *intersection* of graphs G and H are taken point-wise, which implies that the source and target maps of G and H must be compatible for these operations to be well-defined.

5.2 Simple graphs

Let Lab be a finite set of labels, partitioned into disjoint unary and binary label sets, denoted Lab^U and Lab^B , respectively.

Definition 5.3 (Simple graph). A simple graph G is a graph extended with an edge labelling function $\text{lab}_G : E_G \rightarrow \text{Lab}$, where

- for all $e \in E_G$, if $\text{lab}_G(e) \in \text{Lab}^U$ then $\text{src}_G(e) = \text{tgt}_G(e)$; and
- for any $e_1, e_2 \in E_G$, if $\text{src}_G(e_1) = \text{src}_G(e_2)$, $\text{tgt}_G(e_1) = \text{tgt}_G(e_2)$, and $\text{lab}_G(e_1) = \text{lab}_G(e_2)$, then $e_1 = e_2$. $\triangleleft$

The second condition in the definition above prohibits parallel edges with the same label, justifying the choice of the term simple graph. The first condition limits the occurrence of unary labels to self-edges, which are used to encode node labels. We write $\langle v, l, w \rangle$ to represent an edge e with $\text{src}_G(e) = v$, $\text{lab}_G(e) = l$, and $\text{tgt}_G(e) = w$. The universe of simple graphs is denoted by SGraph .

Example 5.4. Figure 5.1(a) shows a simple graph representing a single-linked list composed of five cells and a sentinel node to mark the head and tail elements of the list. Unary labels are written inside their associated node and node

identities are displayed at the top left corner of each node (edge identities are not shown). $\triangleleft$

Simple graphs are related by *simple graph morphisms*, mappings that, in addition to structure, also respect edge labels.

Definition 5.5 (Simple graph morphism). *A simple graph morphism between simple graphs $G, H \in \mathbf{SGraph}$ is a graph morphism $m : G \rightarrow H$ such that $\text{lab}_H \circ m = \text{lab}_G$.* $\triangleleft$

We write $\mathbf{SMorph}$ to represent the universe of simple graph morphisms.

5.2.1 Simple graph transformation

Graphs are transformed by rules that describe both the conditions for their application and the changes that should be performed to the host graph.

Definition 5.6 (Simple graph transformation rule). *A simple graph transformation rule $r = \langle L, R \rangle$ consists of two simple graphs $L, R \in \mathbf{SGraph}$, called left-hand side (LHS) and right-hand side (RHS), respectively.* $\triangleleft$

The relation between L and R is established by their common elements and we distinguish the following sets:

- $N_r^{\text{del}} = N_L \setminus N_R$ and $E_r^{\text{del}} = E_L \setminus E_R$ are the elements deleted by the rule;
- $N_r^{\text{new}} = N_R \setminus N_L$ and $E_r^{\text{new}} = E_R \setminus E_L$ are the sets of elements created; and
- $N_r^{\text{use}} = N_L \cap N_R$ and $E_r^{\text{use}} = E_L \cap E_R$ are the elements that remain unchanged by the rule.

We use $\mathbf{SRule}$ to denote the universe of simple graph transformation rules and we omit the subscript when rule r is clear from the context.

Example 5.7. Figure 5.1(b,c) shows rules for removing the head element of a list (*get*) and inserting a new element at the tail of the list (*put*). $\triangleleft$

Let G be a simple graph and $r = \langle L, R \rangle$ be a simple graph transformation rule such that $N_G \cup E_G$ and $N^{\text{new}} \cup E^{\text{new}}$ are disjoint. Since we do not distinguish between isomorphic graphs, this assumption can be satisfied without loss of generality by taking an isomorphic copy of R where the elements of $N^{\text{new}} \cup E^{\text{new}}$ are fresh with respect to G . An *application* of r into G involves finding an *injective* simple graph morphism $m : L \rightarrow G$ and extending m with $m \cup \text{id}_{(N^{\text{new}} \cup E^{\text{new}})}$. Such a morphism $m : (L \cup R) \rightarrow (G \cup N^{\text{new}} \cup E^{\text{new}})$ is called a *match* of r into G . In the following we simplify the notation and write only $m : L \rightarrow G$ to denote a match, implicitly assuming that m is extended to the elements created by the rule.

Definition 5.8 (Simple graph transformation). *Given a simple graph G and a simple graph transformation rule $r = \langle L, R \rangle$, let $m : L \rightarrow G$ be a match of r into G and let $N' = m(N^{\text{del}})$ and $E' = \{e \in E_G \mid e \in m(E^{\text{del}}) \text{ or } \text{src}_G(e) \in N' \text{ or } \text{tgt}_G(e) \in N'\}$ be the sets of elements to be deleted. The transformation of G by rule r according to match m is a new simple graph H , where*

- $N_H = (N_G \setminus N') \cup N^{\text{new}};$

Figure 5.2: Example of a simple graph transformation, with application of rule `get` to the graph of Figure 5.1(a). Match m is indicated with dashed arrows.

- $E_H = (E_G \setminus E') \cup E^{\text{new}};$
- $\text{src}_H = (\text{src}_G \cup (m \circ \text{src}_R))|_{E_H};$
- $\text{tgt}_H = (\text{tgt}_G \cup (m \circ \text{tgt}_R))|_{E_H};$ and
- $\text{lab}_H = (\text{lab}_G \cup \text{lab}_R)|_{E_H}.$

◁

In the definition above, dangling edges are deleted, following the SPO approach. We write $G \xrightarrow{r, m} H$ to denote that the application of r to G under m gives rise to the transformed graph H . We also write $G \xrightarrow{r} H$ if there exists a match m such that $G \xrightarrow{r, m} H$. In this case, rule r is said to be *enabled* in G .

Example 5.9. Figure 5.2 depicts the application of rule `get` (Figure 5.1(b)) to the host graph of Figure 5.1(a), under match m . Although edge identities are not shown, note that the h -edge of R must be fresh w.r.t. E_G (due to the disjointness condition for a match). ◁

5.2.2 Simple graph transition systems

A collection of related rules forms a *graph transformation system*, and by associating an initial host graph to such system we obtain a *graph grammar*.

Definition 5.10 (Simple graph grammar). A simple graph grammar is a tuple $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$, with $\mathcal{R}$ a finite set of simple graph transformation rules and G_0 an initial simple graph. ◁

We say that a simple graph grammar *generates* a simple graph transition system (SGTS), which is our model of behaviour.

Definition 5.11 (SGTS). A simple graph transition system is a tuple $\text{SGTS} = \langle \mathcal{S}, \rightarrow, \iota \rangle$ where

- $\mathcal{S} \subseteq \text{SGraph}$ is a set of states;
- $\rightarrow \subseteq \mathcal{S} \times \text{SRule} \times \mathcal{S}$ is a set of transitions; and
- $\iota \in \mathcal{S}$ is the initial state.

A simple graph grammar $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$ generates a $\text{SGTS}_{\mathcal{G}}$ if $\mathcal{S}$ is a minimal set of simple graphs such that

- $\iota = G_0;$ and

- if $G \xrightarrow{r} H$ for some $G \in \mathcal{S}$ and $r \in \mathcal{R}$, then there exists $H' \in \mathcal{S}$ where $H \simeq H'$ and $\langle G, r, H' \rangle \in \rightarrow$. $\triangleleft$

In other words, a simple graph grammar generates a SGTS if the transitions in the SGTS correspond to all possible rule applications to all reachable states, under the condition that there is only one isomorphic representative for every state. By abuse of notation, we also write $G \xrightarrow{r} H$ to indicate that $\langle G, r, H \rangle \in \rightarrow$, thus confusing a rule application with a transition of $SGTS_{\mathcal{G}}$.

Note that the conditions on $\mathcal{S}$ in the definition above are not sufficient to uniquely define a SGTS generated by a grammar. However, any two transition systems satisfying the criteria of Definition 5.11 are isomorphic [Ren03b], and thus we are justified to speak of *the* SGTS generated by a simple graph grammar.

5.2.3 Implementing the SGTS generation

The expression “a SGTS *generated* by grammar $\mathcal{G}$ ” further conveys the idea that we can construct $SGTS_{\mathcal{G}}$ with an algorithm. Given our choice of an interleaving semantics for $\mathcal{G}$ (as established in Definition 5.11) this algorithm amounts to a *reachability* procedure, *i.e.*, a computation of states (graphs) that are reachable from start state ι by one or more transitions (rule applications). We usually refer to this reachability computation as the *state space exploration* of the grammar.

Listing 5.3 gives the pseudo-code for exploring the state space of a simple grammar $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$ and constructing its associated $SGTS_{\mathcal{G}} = \langle \mathcal{S}, \rightarrow, \iota \rangle$. The algorithm maintains a set $\mathcal{F}$ of fresh, yet to be explored states.

Listing 5.3: Algorithm for state space exploration.

```

1  let  $\mathcal{S} := \emptyset, \rightarrow := \emptyset, \iota := G_0, \mathcal{F} := \{\iota\}$ 
2  while  $\mathcal{F} \neq \emptyset$ 
3  do choose  $G \in \mathcal{F}$  // which  $G$  is selected depends on the exploration strategy
4    let  $\mathcal{F} := \mathcal{F} \setminus \{G\}$ 
5    for  $r \in \mathcal{R}, m \in \text{match}(r, G)$ 
6    do let  $H := \text{apply}(r, m, G)$ 
7      if isFresh( $H, \mathcal{S}$ ) // if  $H \notin \mathcal{S}$ 
8      then let  $\mathcal{S} := \mathcal{S} \cup \{H\}, \mathcal{F} := \mathcal{F} \cup \{H\}$ 
9    fi
10   let  $\rightarrow := \rightarrow \cup \{G \xrightarrow{r} H\}$ 
11  od
12 od
```

Line 1 of Listing 5.3 initialises the transition system with empty sets and sets the first fresh state to the start graph of $\mathcal{G}$. Line 3 deals with the policy for selecting a graph G from the set $\mathcal{F}$ of states to be explored. We consider two policies, namely breadth-first search (BFS) and depth-first search (DFS). When using BFS, $\mathcal{F}$ is implemented as a queue, whereas in DFS $\mathcal{F}$ is a stack. We use the term *exploration strategies* to refer to these search policies. While the resulting $SGTS_{\mathcal{G}}$ constructed is independent (*i.e.*, unique, up to isomorphism) of the strategy used, the choice of exploration strategy may greatly influence the algorithm performance. This point is properly analysed in Chapter 7.

Line 5 handles the matching of rules into the state being currently explored. Although in Listing 5.3 there is no explicit choice for the order on which rules are picked from $\mathcal{R}$, in practice some control mechanism is used (*e.g.*, see [Sta10], Chapter 2, for a discussion on the control programs used in GROOVE). Nevertheless, the details of a control mechanism are immaterial for the topics discussed in this thesis. Thus, throughout the remaining chapters we assume that an arbitrary control mechanism is used, and that this mechanism remains the same in all exploration algorithms presented.

Continuing with the second item in line 5, procedure `match`(r, G) computes all possible matches of rule r into graph G . Efficient implementation of graph matching is a whole research area in itself (*e.g.*, see [GJR10] for the recent developments in GROOVE). However, once more, the actual algorithm of procedure `match` is irrelevant for our abstractions, and thus will not be further discussed.

Line 6 deals with the transformation of simple graph G , where procedure `apply`(r, m, G) performs a rule application, as described in Definition 5.8. Since the definition is already constructive, the algorithm for `apply` is trivial to develop and therefore is not explicitly presented here.

Line 7 handles the duplicate state detection mechanism. Procedure `isFresh`($H, \mathcal{S}$) is responsible for checking if graph H (or an equivalent isomorphic representative) is already in the set $\mathcal{S}$ of all explored graphs. Set $\mathcal{S}$ can be quite large, so this check has to be implemented with care, since it can greatly impact performance. Given that graph isomorphism can be a rather expensive check, it cannot be performed over all elements of $\mathcal{S}$. Listing 5.4 shows the algorithm for procedure `isFresh`, as originally described in [Ren07].

Listing 5.4: Algorithm for procedure `isFresh`($H, \mathcal{S}$).

```

1 let  $C := \text{certificate}(H)$ ,  $\bar{\mathcal{S}} := \{F \in \mathcal{S} \mid C = \text{certificate}(F)\}$ 
2 for  $F \in \bar{\mathcal{S}}$ 
3 do if  $H \simeq F$  // if  $H$  and  $F$  are isomorphic
4     then return false
5 fi
6 od
7 return true // we checked all candidates but none are isomorphic to  $H$ , so  $H$  is fresh

```

This algorithm is based on *graph certificates*, which basically correspond to a hashing method tuned for simple graphs. In addition to being relatively inexpensive to compute, certificates are built in such a way that graph isomorphism implies certificate equality. The converse, however, is not true, because the certificate function may produce false positives. In the algorithm of Listing 5.4, certificates are used to filter elements of set $\mathcal{S}$, thus producing a much smaller set $\bar{\mathcal{S}} \subseteq \mathcal{S}$, composed only of graphs with the same certificate of H . We then proceed to check if there exists $F \in \bar{\mathcal{S}}$ such that $H \simeq F$. If no such F is found then H is fresh w.r.t. $\mathcal{S}$.

The freshness test just described works very well in practice and is currently implemented in GROOVE. However, when executing a state space exploration in the abstract domain, a more elaborate algorithm for procedure `isFresh` is necessary in order to improve performance. This point is analysed in Chapter 7.

5.2.4 Discussion

The algorithms presented at the previous section give a (very) high-level overview of the main GROOVE functionality. While finite-state grammars can be properly handled by the exploration algorithm in Listing 5.3, we can immediately see that it fails to terminate for grammars with an infinite state space.

Example 5.12. Consider a grammar $\mathcal{G}$ formed by rules **put** and **get** in Figure 5.2 and the following start graph

$$G_0 = \boxed{L} \xrightarrow[\text{t}]{\text{h}} \boxed{C}.$$

For any state in $SGTS_{\mathcal{G}}$, rule **put** is always enabled and successive applications of this rule keep producing longer and longer lists. The figure below shows the form of $SGTS_{\mathcal{G}}$, although it cannot be effectively computed.

Infinite-state grammars such as the one in Example 5.12 motivated the development of the abstraction methods presented in the following chapters. The idea behind these abstractions is that “similar” states are grouped together, and these groups are built in such a way that the distinction between grouped states is no longer visible. Thus, the behaviour (outgoing transitions) of the abstract state is the collection of possible behaviours of the original concrete states. (This is properly formalised in Chapters 6 and 8.)

Graph “similarity” is the point where these abstractions differ; in our case, both neighbourhood and pattern similarity are abstractions over graph nodes and edges, *i.e.*, graphs are abstracted by folding equivalent (similar) elements into one abstract representative, while keeping count of their original number up to some bound of precision. Counting up to some bound is done using *multiplicities*, defined in the following section.

5.3 Approximated counting

We use ω to denote an upper bound on the set of natural numbers, *i.e.*, $\omega \notin \mathbb{N}$ and for all $k \in \mathbb{N}, k < \omega$. We write $\mathbb{N}^\omega$ as short-hand notation for $\mathbb{N} \cup \{\omega\}$ and we consider simple arithmetic operations over $\mathbb{N}^\omega$, defined in the following.

Definition 5.13 (Operations over $\mathbb{N}^\omega$). *Given $i, j \in \mathbb{N}^\omega$, addition, subtraction, and multiplication over elements of $\mathbb{N}^\omega$ are respectively defined as*

$$i + j = \begin{cases} i + j & \text{if } i, j \in \mathbb{N}, \\ \omega & \text{otherwise.} \end{cases}$$

$$i - j = \begin{cases} i - j & \text{if } i \in \mathbb{N} \text{ and } j < i, \\ 0 & \text{if } i \in \mathbb{N} \text{ and } j \geq i, \\ \omega & \text{if } i = \omega \text{ and } j \in \mathbb{N}. \end{cases}$$

$$i * j = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ i * j & \text{if } i, j \in (\mathbb{N} \setminus \{0\}), \\ \omega & \text{otherwise.} \end{cases}$$

Subtraction is undefined when both operands equal ω . $\triangleleft$

The operation symbols in the definition above are overloaded: symbols on the left represent operations over $\mathbb{N}^\omega$ while the ones on the right are the usual operations over $\mathbb{N}$.

Definition 5.14 (Multiplicity). A multiplicity is an element of the set $\mathcal{M} = \{\langle i, j \rangle \in (\mathbb{N} \times \mathbb{N}^\omega) \mid i \leq j\}$. $\triangleleft$

Multiplicities are used to represent an interval of consecutive values taken from $\mathbb{N}^\omega$, i.e., $\langle i, j \rangle$ is a compact representation for the set $\{k \in \mathbb{N}^\omega \mid i \leq k \leq j\}$. Given a multiplicity $\langle i, j \rangle \in \mathcal{M}$,

- if $i = j$ we call the multiplicity *singleton* and we use a short-hand notation by writing only the lower-bound i in **bold**, i.e., $\mathbf{i}$; and
- if $j = \omega$, we use a short-hand notation by writing the lower-bound i in **bold**, super-scripted with $+$, i.e., $\mathbf{i}^+$.

The singleton multiplicity $\mathbf{1}$ is called *concrete*. If $j > 1$ the interval is said to be a *collector* multiplicity. Set $\mathcal{M}$ is infinite, since i and j are taken from infinite sets. To ensure finiteness, we define a bound of precision, which limits the possible values of i and j .

Definition 5.15 (Bounded multiplicity). A bounded multiplicity is an element of set $\mathcal{M}^{\mathbf{b}} \subset \mathcal{M}$, defined, for a given bound $\mathbf{b} \in \mathbb{N}$, as $\mathcal{M}^{\mathbf{b}} = \{\langle i, j \rangle \in \mathcal{M} \mid i \leq \mathbf{b} + 1, j \in \{0, \dots, \mathbf{b}, \omega\}\}$. $\triangleleft$

In words, set $\mathcal{M}^{\mathbf{b}}$ consists of all closed intervals with upper bound up to $\mathbf{b}$ and all right-open intervals with lower bound at most $\mathbf{b} + 1$. In practice, bound $\mathbf{b}$ is typically set to a low value such as 1 or 2, due to the fact that abstract state space sizes grow exponentially w.r.t. $\mathbf{b}$. Normally, we do not use non-singleton closed multiplicities (e.g., $\langle 0, 1 \rangle$); instead, singleton intervals are used for precise counting and the $\mathbf{i}^+$ multiplicities stand for “ i or more”.

The fact that $\mathcal{M}^{\mathbf{b}}$ is finite is guaranteed by the following proposition.

Proposition 5.16. $|\mathcal{M}^{\mathbf{b}}| = (\mathbf{b}^2 + 5\mathbf{b} + 6)/2$. $\triangleleft$

Proof. The restriction $i \leq j$ from Definition 5.14 and the conditions on i and j from Definition 5.15 imply that $|\mathcal{M}^{\mathbf{b}}| = 1 + \dots + (\mathbf{b} + 2) = (\mathbf{b}^2 + 5\mathbf{b} + 6)/2$. $\square$

Example 5.17. If $\mathbf{b} = 1$ then $\mathcal{M}^{\mathbf{b}} = \{\mathbf{0}, \mathbf{1}, \langle 0, 1 \rangle, \mathbf{0}^+, \mathbf{1}^+, \mathbf{2}^+\}$. $\triangleleft$

Based on the operations and relations over $\mathbb{N}^\omega$ we define operations and relations over $\mathcal{M}$.

Definition 5.18 (Operations/relations over $\mathcal{M}$). Let $\langle i, j \rangle$ and $\langle i', j' \rangle$ be elements of $\mathcal{M}$ and let $k \in \mathbb{N}$. The arithmetic operations over $\mathcal{M}$ are defined as

$$\begin{aligned} \langle i, j \rangle + \langle i', j' \rangle &= \langle i + i', j + j' \rangle \\ \langle i, j \rangle - k &= \langle i - k, j - k \rangle \\ \langle i, j \rangle * \langle i', j' \rangle &= \langle i * i', j * j' \rangle. \end{aligned}$$

Relations $\leq$ and $\sqsubseteq$ are defined as

$$\begin{aligned} \langle i, j \rangle &\leq \langle i', j' \rangle && \text{if } j \leq j' \\ \langle i, j \rangle &\sqsubseteq \langle i', j' \rangle && \text{if } i \geq i' \text{ and } j \leq j'. \end{aligned} \quad \triangleleft$$

It can be immediately seen that $\sqsubseteq$ is a refinement of $\leq$. Relation $\sqsubseteq$ is called the *subsumption* relation, since the interval represented by $\langle i, j \rangle$ is completely contained in the interval represented by $\langle i', j' \rangle$. Let $m_1, m_2 \in \mathcal{M}$ be two multiplicities and let M_1, M_2 be their respective set of values from $\mathbb{N}^\omega$. Thus, if $m_1 \sqsubseteq m_2$ then $M_1 \subseteq M_2$. Similarly, if $m_1 \leq m_2$ then $\max(M_1) \leq \max(M_2)$, where $\max(M)$ is the maximal element of set M .

In order to use bounded multiplicities we project elements of $\mathcal{M}$ into $\mathcal{M}^{\mathbf{b}}$.

Definition 5.19 (Multiplicity approximation). Given a positive bound $\mathbf{b} \in \mathbb{N}$, function $\beta^{\mathbf{b}} : \mathcal{M} \rightarrow \mathcal{M}^{\mathbf{b}}$ approximates multiplicities according to $\mathbf{b}$, i.e., for $\langle i, j \rangle \in \mathcal{M}$,

$$\beta^{\mathbf{b}}(\langle i, j \rangle) = \begin{cases} \langle i, j \rangle & \text{if } i \leq \mathbf{b} \text{ and } j \leq \mathbf{b}, \\ \langle i, \omega \rangle & \text{if } i \leq \mathbf{b} \text{ and } j > \mathbf{b}, \\ \langle \mathbf{b} + 1, \omega \rangle & \text{if } i > \mathbf{b}. \end{cases} \quad \triangleleft$$

An important point to note is that multiplicity approximation preserves subsumption, proved in the following.

Proposition 5.20. For any $\langle i, j \rangle \in \mathcal{M}$ and $\langle i', j' \rangle \in \mathcal{M}^{\mathbf{b}}$, if $\langle i, j \rangle \sqsubseteq \langle i', j' \rangle$ then $\beta^{\mathbf{b}}(\langle i, j \rangle) \sqsubseteq \langle i', j' \rangle$. $\triangleleft$

Proof. Assume $\langle i, j \rangle \sqsubseteq \langle i', j' \rangle$ (otherwise the implication is trivially satisfied). From Definition 5.19 we have three cases.

1. *Case where $i \leq \mathbf{b}$ and $j \leq \mathbf{b}$.* We have that $\beta^{\mathbf{b}}(\langle i, j \rangle) = \langle i, j \rangle \sqsubseteq \langle i', j' \rangle$, which is the assumption.
2. *Case where $i \leq \mathbf{b}$ and $j > \mathbf{b}$.* From the assumption we know that $j \leq j'$, and since $j > \mathbf{b}$, we have that $j' > \mathbf{b}$. Now, since $\langle i', j' \rangle \in \mathcal{M}^{\mathbf{b}}$, the only possible value greater than $\mathbf{b}$ for j' is ω . Thus, $\beta^{\mathbf{b}}(\langle i, j \rangle) = \langle i, \omega \rangle \sqsubseteq \langle i', \omega \rangle = \langle i', j' \rangle$, which holds since we know that $i \geq i'$ from the assumption.
3. *Case where $i > \mathbf{b}$.* From the definition of multiplicity and the case condition, we have that $j > \mathbf{b}$. Using the same reasoning of case 2, we conclude that $j' = \omega$. Thus, $\beta^{\mathbf{b}}(\langle i, j \rangle) = \langle \mathbf{b} + 1, \omega \rangle \sqsubseteq \langle i', \omega \rangle = \langle i', j' \rangle$, which holds since $\langle i', j' \rangle \in \mathcal{M}^{\mathbf{b}}$ and therefore $\mathbf{b} + 1 \geq i'$. $\square$

Let $M \subseteq \mathcal{M}$ be a set of multiplicities. We write $\sum^{\mathbf{b}} M$ to denote the bounded multiplicity sum over elements of M , as a short-hand notation for $\beta^{\mathbf{b}}(\sum M)$.

The abstractions presented in the next chapters are parameterised with two multiplicity bounds $\mathbf{n}, \mathbf{e} \in \mathbb{N}$, called *node* and *edge* bounds, respectively. These bounds can be independently chosen to define the precision for node and edge counting. From here on, unless explicitly stated, we assume that both bounds are set to one. We also assume that all multiplicities are bounded, which implies that the results of arithmetic operations on multiplicities are always implicitly projected back to $\mathcal{M}^{\mathbf{b}}$ using $\beta^{\mathbf{b}}$, to ensure the results remain bounded.

5.4 Transition systems, simulations and model checking

The concrete semantic representation we use for a graph grammar $\mathcal{G}$ is given in Definition 5.11, as the SGTS (or simply, the GTS) generated by $\mathcal{G}$. In Chapters 6 and 8, we define a corresponding abstract semantic representation of $\mathcal{G}$, for each of the abstraction methods. These abstract behavioural models are commonly referred to as *shape transition systems* (STS).

An essential requirement for any STS is that it should be finite, to allow for its proper construction by an exploration algorithm. However, an even more important requirement is that a STS generated by a grammar $\mathcal{G}$ must always be related to the GTS (that would be) generated by $\mathcal{G}$, where this connection is established by a binary relation between states of the two systems. In this case, the STS is said to be an *abstraction* of the GTS. In this way, the possibly infinite GTS is abstracted by the finite STS, and, provided that the abstraction preserves the property to be checked, the analysis of the STS suffices to decide the satisfaction of the property in the GTS.

Due to our connection with GROOVE we have a particular interest in model checking, and therefore we are interested in relations between shape and graph transition systems that respect temporal logic behaviour. Usual representatives of such relations are *bi-simulation* equivalences and *simulation* relations. Whereas bi-simulation relates states that mutually mimic all individual transitions, simulation requires that one state can mimic all step-wise behaviour of the other, but not the reverse.

Bi-similarity is a quite interesting result because it implies that satisfaction of μ -calculus formulæ (and thus also CTL*, CTL, and LTL formulæ) are preserved among the two transition systems. Ideally, this is the relation we would like to establish between a GTS and a STS, but this is usually not possible due to the inherent imprecision of the abstractions.

For the cases where a bi-simulation cannot be constructed, we establish a simulation from the STS to the GTS. In contrast to bi-simulations, simulations preserve (large) fragments of temporal logics, where these fragments are still sufficient to cover all safety properties and some liveness conditions. As shown in [BK02], a simulation preserves the universal fragment of CTL*, denoted $\forall\text{CTL}^*$. This fragment allows formulæ of the form $\forall\varphi$, where φ cannot contain negation or existential quantification. The fragment $\forall\text{CTL}^*$ includes LTL, *i.e.*, for any LTL formula there exists an equivalent $\forall\text{CTL}^*$ formula.

The explanations above justify the fact that the final results presented in Chapters 6 and 8 are theorems establishing that a STS (from either neighbourhood or pattern abstraction) simulates the SGTS generated by a grammar.

5.5 Test cases for experiments

We collected 5 graph grammars from distinct problem domains, for the experimental evaluation of the implementation of neighbourhood abstraction (given in Chapter 7). These grammars are presented in this section, separately from the conducted experiments, because they are of interest in their own right, and

Figure 5.5: Grammar modelling a single-linked list, in GROOVE notation.

Figure 5.6: Grammar modelling a circular buffer.

may serve as a small benchmark suite for graph abstraction tools. All grammar rules are given in GROOVE notation.

Grammar *linked-list*

This is a grammar representing operations on a single-linked list, such as the ones first introduced in Figure 5.1. Figure 5.5(b-c) shows rules `put` and `get` in GROOVE notation; Figure 5.5(a) is the initial host graph of Example 5.12. This grammar has no variants.

Grammar *circ-buf*

This graph grammar models a circular buffer similar to the one discussed in Section 2.3. Figure 5.6(a) shows the initial host graph. The buffer cells are represented by **C**-nodes, connected in a circle of *n*-edges. An additional sentinel **B**-node is used to indicate the current *first* and *last* buffer cells, marked by edges labelled by *f* and *l*, respectively. An *empty* cell has an *e*-edge to the **B**-node. This representation was chosen to avoid the need for negative application conditions in the grammar rules.

Figure 5.6(b) shows the rule that adds a new *object* into the buffer. When this rule is applied, the *l*-edge is advanced and a new **O**-node is created to represent the *value* stored. Figure 5.6(c) gives a dual rule that removes an object from the buffer.

Note that, although the initial host graph shown in Figure 5.6(a) represents a concrete buffer with four cells, when this graph is abstracted it gives rise to a buffer of unbounded size (assuming a node bound $n = 1$). In this case, the

Figure 5.7: Grammar for constructing Euler's walks.

two **C**-nodes not marked as first or last are collapsed, and the collapsed abstract node is given a collector multiplicity.

This grammar was first presented in [RD06], and it has two variants:

- *circ-buf-0*: cell usage is marked with special labels, without reference to stored objects.
- *circ-buf-1*: in this variant the objects stored in the circular buffer are explicitly represented, as in the rules shown in Figure 5.6.

Grammar *euler*

This is a grammar that can construct Euler's walks of arbitrary size. The idea for this grammar behaviour was adapted from the classical Königsberg bridges problem from graph theory. The difference here is that the bridges topology is not fixed, but instead it is “discovered” by rule applications.

Figure 5.7 shows the grammar. Bridges are represented by **B**-nodes, which connect two *areas*, denoted by **A**-nodes. The number of bridges connected to a certain area is indicated by two labels: **e**, if the number is *even*; and **o**, if the number is *odd*. Euler is represented by an **E**-node, which navigates to different areas by crossing bridges. The area where Euler is at the moment is marked by an **h**-edge. In the initial configuration, only a single area exists without any bridges, as shown in Figure 5.7(a).

Figure 5.7(b) depicts a rule that creates a new area and connects it with a bridge to another (already existing) area. Euler is moved to the newly area created by the rule application. The particular rule shown in Figure 5.7(b) matches with existing areas that have an even number of connecting bridges; since a new bridge is connected to the existing area, its even count is changed to odd. A newly created area has always an odd count. A similar rule (not shown here) handles existing areas with an odd count of connecting bridges.

Besides creating new areas, additional bridges may be constructed between existing areas, as shown in Figure 5.7(c). The rule depicted matches two areas with even counts, which are both changed to odd by the rule. Three other similar rules handle the other possible combinations of even and odd counts.

An interesting property that can be checked with this grammar is that all Euler's walks have either zero or two areas with an odd number of connecting bridges. This grammar has two variants:

- *euler-0*: the grammar as presented above.
- *euler-1*: a simplified variant of the grammar without explicitly representing connecting bridges.

Figure 5.8: Sample of the firewall grammar. (a) An example host graph. (b-e) Some of the grammar transformation rules.

Grammar *firewall*

We use a graph grammar modelling the behaviour of a firewall in a network, adapted from König and Kozioura [KK06]. A *firewall* has *inner* and *outer interfaces*, to which *locations* can be connected. Locations are marked with the kind of interface they are connected to. Data are represented as *packets*, which can be transferred between locations or through the firewall. Packets can either be *safe* or *unsafe*. Safe packets can be *at* any location but unsafe packets cannot exist at inner locations. Figure 5.8(a) shows an example configuration of a network with simple abbreviations used for conciseness. The network has one outer location and two inner ones, and there are five packets being transmitted.

Figure 5.8 shows four transformation rules of the grammar. Figure 5.8(b,c) depicts the rules for packet creation. A safe packet can be created at any location, whereas an unsafe packet can only be created at outer locations. Infinite behaviour stems from these two rules; since they are always enabled, an infinite number of packets can be created. Figure 5.8(d) shows a rule for packet transfer between locations. Since all locations on each side of the firewall are of the same type, there is no need to distinguish between safe and unsafe packets. A dual rule (not shown here) transfers packets on the reverse direction of a *c*-edge, thus

Figure 5.9: The complete car platooning grammar.

making the connection bi-directional. The rule in Figure 5.8(e) is the firewall filter, that only allows safe packets to reach inner locations.

This grammar has the following variants:

- *firewall-[2-10]*: the grammar as illustrated before. Network structure is fixed, while packets are collapsed by the abstraction. Instances vary on the number of locations: from 2 to 10.
- *firewall-6-F*: variant of the firewall grammar with a network of six fully connected locations.

Grammar *car-platoon*

This grammar simulates a wireless communication protocol between cars, for establishing platoons in highways. Cars can enter and leave a platoon at any time, which leads to an exponential growth on the number of possible configurations. This grammar is adapted from the one given in [Bau06], and is presented in Figure 5.9. Cars are represented by nodes, which can be in different states with respect to a platoon. Car states are indicated by node flags. A car that does not belong to a platoon is in *free agent* state (flag fa). An additional rule not in Figure 5.9, has no LHS and a RHS formed by a single fa car. Thus, an unlimited number of cars can be created by this rule, leading to an infinite concrete state space. The initial host graph of this grammar is an empty graph.

When a platoon is formed, the first car is marked as the *leader* (flag ldr), with remaining cars of the platoon as the *followers* (flag flw). The communication connections between cars of a platoon are indicated by e -edges. The grammar models a *merge* manoeuvre between two platoons. Take for example, the rule shown in Figure 5.9(a). Two platoon leaders approach each other and start a merge. One of these cars changes state to become a *front leader* (flag fl), whereas the other becomes the *rear leader* (flag rl). A merge may also occur between a leader and a free agent car or two free agents. These situations are captured by the other *init-merge* rules in Figure 5.9(b-d).

Once front and rear leaders are established, the rear leader transfer its followers one by one to the front leader. A single follower transfer is captured by the rule in Figure 5.9(e). When a rear leader does not have any more followers then it turns itself into a new follower of the front leader, which goes back to state ldr . This is captured by the rule in Figure 5.9(f), finishing the merge.

Neighbourhood abstraction – Theory

Our initial take on graph abstractions was inspired by the seminal work on shape analysis by Sagiv, Reps and Wilhelm [SRW98, SRW02] and led to theoretical results on *graph shapes* by Rensink [Ren04], and Rensink and Distefano [RD06]. A similar approach called *partner abstraction* was developed in parallel by Bauer [Bau06], and Bauer and Wilhelm [BW07], and later these two approaches were unified in the framework of *neighbourhood abstraction*. The theory of neighbourhood abstraction was initially presented in a technical report [BRKB07] and partially published in [BBKR08].

The purpose of this chapter is to introduce the main concepts and results from the theory of neighbourhood abstraction in order to pave the way for the discussion on the practical aspects of the abstraction implementation given in Chapter 7. In addition, the contribution of this chapter is a new and simpler presentation of the theory. The proofs for the theorems stated here can be found in an accompanying technical report [BKR⁺12], which is an amended version of the original report [BRKB07].

6.1 Introduction

The contents of this chapter build upon the concepts introduced in Chapter 5. On the concrete level we work with simple graphs, previously defined in Section 5.2. On the abstract level we use *neighbourhood shapes*, formally defined in the following (Section 6.2). For an informal description of, and an introduction to neighbourhood shapes, see Section 2.4.2.

All shapes discussed in this chapter are neighbourhood shapes (in contrast, for instance, to pattern shapes – presented in Chapter 8), thus when the single term *shape* is used here it refers to a *neighbourhood shape*, as expected.

Before presenting the formal definition of a neighbourhood shape some additional notation is required. Recall from Definition 5.3 that simple graphs are edge-labelled only and that self-edges with labels taken from $\mathbf{Lab}^U$ are used to simulate node labels. For a simple graph G , we extend its edge labelling function $\mathbf{lab}_G : E_G \rightarrow \mathbf{Lab}$ to nodes, with signature $\mathbf{lab}_G : N_G \rightarrow \mathbf{2}^{\mathbf{Lab}^U}$, and definition $\mathbf{lab}_G(v) = \{\mathbf{a} \in \mathbf{Lab}^U \mid \exists e \in v \triangleright_G : \mathbf{lab}_G(e) = \mathbf{a}\}$, for all $v \in N_G$.

In addition, we extend the notation for edges incoming to or outgoing from a node to sets of nodes, possibly limiting the edges to a given label. Thus, for $V \subseteq N_G$ and $\mathbf{a} \in \mathbf{Lab}^B$, the sets of $\mathbf{a}$ -labelled edges outgoing from, and incoming

to nodes in V are respectively defined as

$$\begin{aligned} V \triangleright_G^{\mathbf{a}} &= \{e \in v \triangleright_G \mid \text{lab}_G(e) = \mathbf{a}, v \in V\} \text{ and} \\ V \triangleleft_G^{\mathbf{a}} &= \{e \in v \triangleleft_G \mid \text{lab}_G(e) = \mathbf{a}, v \in V\} . \end{aligned}$$

Furthermore, for an additional set of nodes $W \subseteq N_G$, we write $V \triangleright_G^{\mathbf{a}} W$ to denote the set of $\mathbf{a}$ -labelled edges going from source nodes in V to target nodes in W , defined as

$$V \triangleright_G^{\mathbf{a}} W = V \triangleright_G^{\mathbf{a}} \cap W \triangleleft_G^{\mathbf{a}} .$$

Finally, if V and W are singleton sets, we omit the brackets in the notation; e.g., $\{v\} \triangleright_G^{\mathbf{a}}$ is written as $v \triangleright_G^{\mathbf{a}}$.

Let A be a set and $\sim \subseteq A \times A$ be an equivalence relation over A . For $x \in A$, we write $[x]_{\sim}$ to denote the equivalence class of x induced by $\sim$, i.e., $[x]_{\sim} = \{y \in A \mid y \sim x\}$ and we write $A/\sim$ to denote the set of equivalence classes in A , i.e., $A/\sim = \{[x]_{\sim} \mid x \in A\}$. Additionally, given two equivalence relations $\sim$ and $\sim'$ over A , we write $\sim \subseteq \sim'$ if for all $x, y \in A$, $x \sim y$ implies $x \sim' y$. In this case, $\sim$ is called a *refinement* of $\sim'$, since any equivalence class of $\sim$ is included into some equivalence class of $\sim'$; i.e., for all $x \in A$, $[x]_{\sim} \subseteq [x]_{\sim'}$. In particular, this implies that any equivalence class of $\sim'$ can be obtained as the union of some equivalence classes in $\sim$.

6.2 Neighbourhood shapes

In this chapter, we use *neighbourhood shapes* as the abstract representatives for sets of concrete simple graphs. A neighbourhood shape is based on an underlying graph structure, extended with a grouping relation and multiplicity mappings.

Definition 6.1 (Neighbourhood shape). A neighbourhood shape is a tuple $S = \langle G_S, \sim_S, \text{mult}_S^n, \text{mult}_S^o, \text{mult}_S^i \rangle$, where

- $G_S = \langle N_S, E_S, \text{src}_S, \text{tgt}_S, \text{lab}_S \rangle$ is a simple graph;
- $\sim_S \subseteq N_S \times N_S$ is a grouping relation over nodes of S ;
- $\text{mult}_S^n : N_S \rightarrow \mathcal{M}^n$ is a node multiplicity function;
- $\text{mult}_S^o : (N_S \times \text{Lab}^B \times N_S/\sim_S) \rightarrow \mathcal{M}^e$ is an outgoing edge multiplicity function; and
- $\text{mult}_S^i : (N_S \times \text{Lab}^B \times N_S/\sim_S) \rightarrow \mathcal{M}^e$ is an incoming edge multiplicity function.

In addition, for any $\langle v, \mathbf{a}, C \rangle \in (N_S \times \text{Lab}^B \times N_S/\sim_S)$, it is required that

- $\text{mult}_S^o(v, \mathbf{a}, C) = \mathbf{0}$ if and only if $(v \triangleright_{G_S}^{\mathbf{a}} C) = \emptyset$, and
- $\text{mult}_S^i(v, \mathbf{a}, C) = \mathbf{0}$ if and only if $(C \triangleright_{G_S}^{\mathbf{a}} v) = \emptyset$. $\triangleleft$

The definition above is “generic” in the sense that any binary relation on nodes can be used as the grouping relation $\sim$. In this chapter, however, we mainly consider shapes where relation $\sim$ is taken as the *neighbourhood equivalence* $\equiv$, defined later in this section (Definition 6.7). This justifies the denomination *neighbourhood shape* in Definition 6.1.

The node multiplicity function mult_S^n records how many concrete nodes were “folded” into a given abstract node, up to bound $\mathbf{n}$. (Multiplicities and their

operations thereof were presented in Section 5.3.) Similarly, the outgoing and incoming edge multiplicity functions record how many edges with a certain label were folded into an abstract edge, up to bound $\mathbf{e}$ and a group of $\sim$ -similar opposite nodes.

Recall from Section 5.3 that multiplicity $\mathbf{1}$ is called *concrete*, and a multiplicity $\langle i, j \rangle$ with $j > 1$ is called *collector*. These same terms are used for nodes and edges of a neighbourhood shape, depending on their associated multiplicity. Thus, for example, given a node $v \in N_S$, v is called concrete if $\text{mult}_S^n(v)$ is the concrete multiplicity, and analogously for collector multiplicities.

Since shapes have a graph structure, any simple graph can be *trivially extended* into a shape, by taking the finer grouping relation possible and associating multiplicity functions that map every graph element to multiplicity $\mathbf{1}$. Formally, given a simple graph G , the trivial extension of G into a shape S is obtained by defining,

- $G_S = G$;
- $\sim_S = \{\langle v, v \rangle \mid v \in N_S\}$;
- for all $v \in N_S$, $\text{mult}_S^n: v \mapsto \mathbf{1}$; and
- for all $\langle v, \mathbf{a}, \{w\} \rangle \in (N_S \times \text{Lab}^B \times N_S / \sim_S)$,

$$\begin{aligned} \text{mult}_S^o: \langle v, \mathbf{a}, \{w\} \rangle &\mapsto \begin{cases} \mathbf{1} & \text{if there exists } e = \langle v, \mathbf{a}, w \rangle \in E_S, \\ \mathbf{0} & \text{otherwise;} \end{cases} \\ \text{mult}_S^i: \langle v, \mathbf{a}, \{w\} \rangle &\mapsto \begin{cases} \mathbf{1} & \text{if there exists } e = \langle w, \mathbf{a}, v \rangle \in E_S, \\ \mathbf{0} & \text{otherwise.} \end{cases} \end{aligned}$$

Thus, the graph structure of S is taken as G , the grouping relation $\sim_S$ puts every node into a singleton equivalence class, and every node is marked as concrete by the node multiplicity function mult^n . The conditional definitions for the edge multiplicity functions are necessary to ensure that the restriction at the end of Definition 6.1 is fulfilled.

The trivial extension of simple graphs is useful because it allows us to consider any simple graph as a shape, and therefore most definitions need to only be written for shapes. Throughout the rest of this chapter we assume that trivial extensions of simple graphs are taken as needed. We write NShape to denote the universe of neighbourhood shapes.

Neighbourhood shapes (and also simple graphs) are related by morphisms, which can be of different kinds. The distinction among morphisms is based on the kind of binary relation used to compare multiplicities. This is formalised in the following definition.

Definition 6.2 ($\preceq$ -morphism). *Given a transitive relation $\preceq$ on multiplicities, a $\preceq$ -morphism between neighbourhood shapes $X, Y \in \text{NShape}$ is a simple graph morphism $m: G_X \rightarrow G_Y$ that satisfies the following additional conditions,*

1. *for all $v, w \in N_X$, $v \sim_X w$ implies $m(v) \sim_Y m(w)$;*
2. *for all $v' \in N_Y$,*

$$\sum_{v \in m^{-1}(v')} \text{mult}_X^n(v) \preceq \text{mult}_Y^n(v') ;$$

3. for all $e' = \langle v', \mathbf{a}, w' \rangle \in E_Y$ and all $e = \langle v, \mathbf{a}, w \rangle \in m^{-1}(e')$,

$$\sum_{C \in \mathcal{C}_1}^e \text{mult}_X^o(v, \mathbf{a}, C) \preceq \text{mult}_Y^o(v', \mathbf{a}, [w']_{\sim_Y}), \text{ and}$$

$$\sum_{C \in \mathcal{C}_2}^e \text{mult}_X^i(w, \mathbf{a}, C) \preceq \text{mult}_Y^i(w', \mathbf{a}, [v']_{\sim_Y})$$

where $\mathcal{C}_1 = (m^{-1}([w']_{\sim_Y}))/\sim_X$ and $\mathcal{C}_2 = (m^{-1}([v']_{\sim_Y}))/\sim_X$.

$A \preceq$ -morphism is denoted a neighbourhood shape morphism, and a $\sqsubseteq$ -morphism is called a subsumption morphism. $\triangleleft$

Since relation $\sqsubseteq$ is a refinement of $\preceq$ (see Section 5.3), it should be clear from the definition above that all subsumption morphisms are also neighbourhood shape morphisms. We write $m : X \rightarrow Y$ to indicate that m is a $\preceq$ -morphism from X to Y . If there exists a subsumption morphism from X to Y , we say that X is *subsumed* by Y , written as $X \sqsubseteq Y$.

An essential property of shape morphisms is that they are composable.

Proposition 6.3. *Let $X, Y, Z \in \text{NShape}$ be neighbourhood shapes, and, for a given transitive relation $\preceq$, let $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ be $\preceq$ -morphisms. Then $g \circ f : X \rightarrow Z$ is also a $\preceq$ -morphism.* $\triangleleft$

Given a simple graph $G \in \text{SGraph}$ and a neighbourhood shape $S \in \text{NShape}$, if there exists a subsumption morphism $m : G \rightarrow G_S$ from the trivial extension of G into S , then m is called an *abstraction* morphism; shape S is said to be an *abstraction* of G ; and, conversely, G is a *concretisation* of S . The (possibly infinite) set of concretisations for shape S is denoted by $\text{concr}(S)$. As a corollary of Proposition 6.3 we have that, if X is a trivial extension of a simple graph and f is an abstraction morphism, then the composition $g \circ f$ is also an abstraction morphism.

Neighbourhood shapes that subsume one another are called isomorphic.

Definition 6.4 (Shape isomorphism). *Let $X, Y \in \text{NShape}$ be two neighbourhood shapes. Shapes X and Y are isomorphic, denoted $X \simeq Y$, if there exists a simple graph isomorphism $\varphi : G_X \rightarrow G_Y$ such that φ and φ^{-1} are subsumption morphisms. In this case, φ is called a shape isomorphism.* $\triangleleft$

It follows immediately from the definition that isomorphic neighbourhood shapes have the same grouping relation and equal multiplicity values among the common elements of the bijection. The lemma below describes an important property of isomorphic shapes, namely, that if two neighbourhood shapes are isomorphic then they have the same set of concretisations. This fact follows from the definitions above and Proposition 6.3.

Lemma 6.5 (Isomorphism and concretisations). *Let $X, Y \in \text{NShape}$ be two neighbourhood shapes. If $X \simeq Y$, then $\text{concr}(X) = \text{concr}(Y)$.* $\triangleleft$

The inverse, on the other hand, is not true. Consider, for instance, two neighbourhood shapes X and Y as follows: X has a single node with multiplicity two and no edges, whereas Y has two nodes, each with multiplicity one, and also no edges. Both X and Y have an unique concretisation (up to simple graph

isomorphism) which is the simple graph with two nodes and no edges, but X and Y are clearly not isomorphic.

A similar lemma states that the subsumption relation $\sqsubseteq$ on neighbourhood shapes is compatible with the subset relation on concretisations.

Lemma 6.6 (Subsumption and concretisations). *Let $X, Y \in \text{NShape}$ be two neighbourhood shapes. If $X \sqsubseteq Y$, then $\text{concr}(X) \subseteq \text{concr}(Y)$.* $\triangleleft$

The result of Lemma 6.6 is exploited in Chapter 7 in order to reduce the computational work necessary to explore the abstract state space.

6.2.1 Canonical neighbourhood shapes

Canonical neighbourhood shapes are a special family of neighbourhood shapes that have several interesting properties, established throughout the rest of this chapter. The precision of neighbourhood shapes, as considered up to now, can already be tweaked by adjusting the node and edge bounds for multiplicities (parameters $\mathbf{n}, \mathbf{e}$ – see Section 5.3). On the other hand, the precision of *canonical* neighbourhood shapes, in addition to these multiplicity bounds, is further parameterised by an *abstraction radius* i , a positive natural number. In a canonical shape, each abstract node represents concrete graph nodes that have similar neighbourhood, up to radius i .

The abstraction into canonical shapes is always defined for simple graphs (more precisely, for their trivial extensions into neighbourhood shapes), *i.e.*, for any values of $\mathbf{n}, \mathbf{e}$ and i , and for any simple graph G , there exists a corresponding canonical shape with the appropriate precision. This, however, is not the case for arbitrary neighbourhood shapes: not all neighbourhood shapes can be embedded into a canonical shape with a given radius.

In this section, we define the abstraction into canonical shapes for simple graphs and neighbourhood shapes, describing the conditions for the existence of the latter. We begin by defining the *neighbourhood equivalence* over nodes and edges of a shape, upon which the canonical shape construction is based.

Definition 6.7 (Neighbourhood equivalence). *Let $S \in \text{NShape}$ be a neighbourhood shape. For each natural $i > 0$, the i -neighbourhood equivalence relation $\equiv_i \subseteq N_S \times N_S$ is defined recursively by (for $v, w \in N_S$)*

- $v \equiv_0 w$ if $\text{lab}_S(v) = \text{lab}_S(w)$;
- $v \equiv_{i+1} w$ if
 1. $v \equiv_i w$;
 2. $\sim_S \subseteq \equiv_i$; and
 3. for all $C \in N_S / \equiv_i$ and all binary labels $\mathbf{a} \in \text{Lab}^B$,

$$\sum_{D \in C}^{\mathbf{e}} \text{mult}_S^{\mathbf{o}}(v, \mathbf{a}, D) = \sum_{D \in C}^{\mathbf{e}} \text{mult}_S^{\mathbf{o}}(w, \mathbf{a}, D), \text{ and}$$

$$\sum_{D \in C}^{\mathbf{e}} \text{mult}_S^{\mathbf{i}}(v, \mathbf{a}, D) = \sum_{D \in C}^{\mathbf{e}} \text{mult}_S^{\mathbf{i}}(w, \mathbf{a}, D),$$

where $\mathcal{C} = \{D \in N_S / \sim_S \mid D \subseteq C\}$.

Relation $\equiv_i$ is extended to edges of S by taking (for $e, e' \in E_S$): $e \equiv_i e'$ if $\text{lab}_S(e) = \text{lab}_S(e')$, $\text{src}_S(e) \equiv_i \text{src}_S(e')$ and $\text{tgt}_S(e) \equiv_i \text{tgt}_S(e')$. $\triangleleft$

Figure 6.1: Neighbourhood equivalence relation computed up to radius 2 over the simple graph of Figure 5.1(a).

From the definition, we see that two nodes are equivalent at radius 0 if they have the same labels. Equivalence classes are then refined to larger radii, where we look for the number of edges incoming from and outgoing to nodes of other equivalence classes defined in the previous step.

The requirement $\sim_S \subseteq \equiv_i$ in condition 2 intuitively says that the grouping relation $\sim_S$ should be “finer”, in the sense of grouping less nodes than the $\equiv_i$ relation we are trying to define. Note that this requirement is necessary, as it ensures that any class $D \in N_S/\sim_S$ is a subset of some $C \in N_S/\equiv_i$. If this requirement is not fulfilled, then for any $j > i$, relations $\equiv_j$ are undefined. Furthermore, note that condition 2 is always satisfied by trivially extended graphs, since the corresponding extended shape has the most fine-grained grouping relation possible. This in turn implies that relation $\equiv_i$ is never undefined when computed over graphs, for any $i \geq 0$.

Example 6.8. Figure 6.1 shows three steps for the computation of neighbourhood equivalence relation $\equiv_i$ (for i up to 2) over the simple graph originally given in Figure 5.1(a), representing a single linked list with five elements. On the first step (Figure 6.1(a)) all **C**-nodes are grouped into one equivalence class, since for the base case only node labels are considered when computing node equivalence. At radius $i = 1$, this equivalence class is split into three new ones, with the first and last cells of the list distinguished by the **h** and **t** edges. Last iteration leads to a discrete partition. $\triangleleft$

We can now construct *canonical neighbourhood shapes* based on the neighbourhood equivalence relation computed up to radius i . The idea is that $\equiv_i$ -equivalent nodes are collapsed into a single node of the canonical shape, and relation $\equiv_{i-1}$ is used as the grouping relation, as formally defined below.

Definition 6.9 (Canonical neighbourhood shape). *Let X be a neighbourhood shape and let $i \geq 1$. If relation $\equiv_i$ over X is well-defined, then the canonical neighbourhood shape of X w.r.t. equivalence $\equiv_i$ is the neighbourhood shape Y , where $N_Y = N_X/\equiv_i$ and $E_Y = E_X/\equiv_i$, and for all $[v]_{\equiv_i} \in N_Y$, $v \in N_X$, $[e]_{\equiv_i} \in E_Y$ and $e \in E_X$:*

$$- \text{src}_Y : [e]_{\equiv_i} \mapsto [\text{src}_X(e)]_{\equiv_i} \text{ and } \text{tgt}_Y : [e]_{\equiv_i} \mapsto [\text{tgt}_X(e)]_{\equiv_i};$$

Figure 6.2: Canonical neighbourhood shape constructed over relation $\equiv_1$.

- $\text{lab}_Y : [e]_{\equiv_i} \mapsto \text{lab}_X(e)$;
- $\sim_Y = \{ \langle [v]_{\equiv_i}, [w]_{\equiv_i} \rangle \mid \langle v, w \rangle \in \equiv_{i-1} \}$;
- $\text{mult}_Y^n : [v]_{\equiv_i} \mapsto \sum_{v' \in [v]_{\equiv_i}} \text{mult}_X^n(v')$; and
- for all $\langle [v]_{\equiv_i}, \mathbf{a}, C \rangle \in (N_Y \times \text{Lab}^B \times N_Y / \sim_Y)$,

$$\text{mult}_Y^o : \langle [v]_{\equiv_i}, \mathbf{a}, C \rangle \mapsto \sum_{D \in \mathcal{C}} \text{mult}_X^o(v, \mathbf{a}, D), \text{ and}$$

$$\text{mult}_Y^i : \langle [v]_{\equiv_i}, \mathbf{a}, C \rangle \mapsto \sum_{D \in \mathcal{C}} \text{mult}_X^i(v, \mathbf{a}, D),$$
 where $\mathcal{C} = \{D \in N_X / \sim_X \mid D \subseteq C\}$.

◁

Thus, nodes and edges of canonical neighbourhood shape Y are the equivalence classes of $\equiv_i$, with the labelling of edges in Y following the edge labels in X , and the grouping relation of Y taken from the previous iteration of the neighbourhood equivalence. Multiplicities of Y are the bounded sum of the multiplicities of elements in the equivalence classes of $\equiv_i$.

Let G be a simple graph and S be a neighbourhood shape, and assume for a given radius $i > 0$ that $\equiv_i$ is well-defined for S . We write $\text{abstract}(G)$ and $\text{normalise}(S)$ to denote the canonical neighbourhood shape of G and S , respectively. Morphism $\alpha : G \rightarrow \text{abstract}(G)$ is called an *abstraction morphism* and morphism $\Omega : S \rightarrow \text{normalise}(S)$ is called a *normalisation morphism*. Note that the term abstraction morphism used for α coincides with the denotation for subsumption morphisms from graphs into (arbitrary) neighbourhood shapes. This term overloading causes no conflicts since canonical shapes are neighbourhood shapes and both α and Ω are subsumption morphisms. Actually, from Definition 6.9 we can see that the relation between multiplicities is one of equality, *i.e.*, α and Ω are in fact $=$ -morphisms, a condition that is stronger than (and so implies) subsumption.

Example 6.10. Figure 6.2 shows the canonical shape for the simple graph in Figure 5.1(a), constructed with radius $i = 1$; *i.e.*, using equivalence relations $\equiv_0$ and $\equiv_1$ from Figure 6.1(a,b). Multiplicities are not explicitly indicated, and are concrete for all edges and for all nodes but one, which is distinguished by a bold line. This node is a collector with a multiplicity of “many”, in this example, 2^+ . ◁

We use $\text{CanNShape}_i^{\mathbf{n}, \mathbf{e}}$ to denote the universe of canonical neighbourhood shapes constructed up to radius i and bounded by $\mathbf{n}$ and $\mathbf{e}$. From here on, as done with multiplicity bounds, we assume a radius $i = 1$, unless explicitly indicated otherwise. As stated before, canonical shapes have a number of interesting properties, with the first one given in the following lemma, which is essentially the reverse implication of Lemma 6.5.

Lemma 6.11 (Common concretisations implies isomorphism). *Let X and Y be two canonical neighbourhood shapes in $\text{CanNShape}_i^{\mathbf{n}, \mathbf{e}}$. If $\text{concr}(X) = \text{concr}(Y)$, then $X \simeq Y$.* $\triangleleft$

Thus, although the implication in this lemma does not hold for arbitrary neighbourhood shapes, it does hold for canonical shapes. The proof for Lemma 6.11 uses the notion of *canonical names*, where each equivalence class from a neighbourhood equivalence is uniquely identified by such a name. A canonical neighbourhood shape can then be viewed as a graph whose nodes and edges are canonical names. This same concept is used to prove the following theorem, stating that the universe of canonical shapes is finite. This is, of course, a necessary condition to establish the finiteness of the abstraction.

Theorem 6.12. *Given a neighbourhood radius i and multiplicity bounds $\mathbf{n}$ and $\mathbf{e}$, the universe $\text{CanNShape}_i^{\mathbf{n}, \mathbf{e}}$ is finite (under isomorphism).* $\triangleleft$

Although the finiteness of $\text{CanNShape}_i^{\mathbf{n}, \mathbf{e}}$ can be established, the upper bound on its size is quite large, growing super-exponentially in i . An investigation on whether these upper bounds occur in practical cases led to the experimental results presented in Chapter 7.

6.3 Neighbourhood shape transformation

In this section we define how neighbourhood shapes are transformed, while also establishing how these transformations are related to transformations of shape concretisations.

Definition 6.13 (Pre-match/match into a neighbourhood shape). *Let $S \in \text{NShape}$ be a neighbourhood shape and $r = \langle L, R \rangle$ be a simple graph transformation rule. A pre-match of r into S is a neighbourhood shape morphism $m : L \rightarrow S$. We call m a match if, in addition, m is injective, and,*

1. *for all $v \in m(N_L)$, $[v]_{\sim_S} = \{v\}$;*
2. *for all $v \in m(N_L)$, $\text{mult}_S^{\mathbf{n}}(v) = \mathbf{1}$; and*
3. *for all $e = \langle v, \mathbf{a}, w \rangle \in m(E_L)$, $\text{mult}_S^{\mathbf{e}}(v, \mathbf{a}, \{w\}) = \mathbf{1} = \text{mult}_S^{\mathbf{i}}(w, \mathbf{a}, \{v\})$.* $\triangleleft$

As stated in the next lemma, the existence of a match $m : L \rightarrow S$ guarantees the existence of a match $m' : L \rightarrow G$ for any simple graph G that is a concretisation of S . A match m into shape S has an image that can be considered a concrete “sub-graph” of the shape, that is, nodes and edges in the image of m are concrete, per conditions 2 and 3 in Definition 6.13. This “concreteness” of $m(L)$ is further reinforced by the fact that the match conditions coincide with the definition of a trivial extension of a graph into a shape. Furthermore, note that a pre-match is a neighbourhood shape morphism, *i.e.*, a $\leq$ -morphism, and the conditions in Definition 6.2 guarantee that the required number of edges can indeed exist in some concretisation G , such that an injective morphism from L into G can be constructed.

Lemma 6.14. *Let $S \in \text{NShape}$ be a neighbourhood shape, $r = \langle L, R \rangle$ be a simple graph transformation rule, and $m : L \rightarrow S$ be a match of L into S . For*

any simple graph $G \in \text{concr}(S)$ with abstraction morphism $\alpha : G \rightarrow S$, there exists a match $m' : L \rightarrow G$ such that $m = \alpha \circ m'$. $\triangleleft$

The proof of this lemma is straightforward and relies on the realisation that for any graph element $x \in L$, $\alpha^{-1}(m(x))$ is a singleton set. This follows immediately from the fact that (i) m is a match, and thus $m(L)$ is concrete; and (ii) α is an abstraction morphism, and therefore α respects multiplicities.

As done with simple graph transformations, when applying a rule r to a shape S we assume without loss of generality that r and S are disjoint. In the same vein, we also assume that a match $m : L \rightarrow S$ is implicitly extended with an identity map on the elements created by r (see Section 5.2.1). Given such m , shape S can be concretely transformed, as defined below.

Definition 6.15 (Concrete neighbourhood shape transformation). Let $X \in \text{NShape}$ be a neighbourhood shape, $r = \langle L, R \rangle \in \text{SRule}$ be a simple graph transformation rule, and $m : L \rightarrow X$ be a match of r into X . The concrete neighbourhood shape transformation of X is the shape Y , where

- G_Y is the result of the simple graph transformation $G_X \xrightarrow{r,m} G_Y$;
- $\sim_Y = \sim_X \cup \sim_R$;
- $\text{mult}_Y^n = (\text{mult}_X^n \cup \text{mult}_R^n)|_{N_Y}$;
- $\text{mult}_Y^o = (\text{mult}_X^o \cup \text{mult}_R^o)|_{(N_Y \times \text{Lab}^B \times N_Y / \sim_Y)}$; and
- $\text{mult}_Y^i = (\text{mult}_X^i \cup \text{mult}_R^i)|_{(N_Y \times \text{Lab}^B \times N_Y / \sim_Y)}$. $\triangleleft$

It can be immediately seen that the multiplicity functions of Y are well-defined; this follows from (i) the fact that the trivial extension of L and the match image $m(L)$ in X are compatible, and (ii) that $(X \cap R) \subseteq m(L)$. We write $X \xrightarrow{r,m} Y$ to denote a concrete neighbourhood shape transformation, or simply $X \xrightarrow{r} Y$ if there exists a match m such that $X \xrightarrow{r,m} Y$.

The following theorem is an essential result in establishing the relation between concrete and abstract domains, stating that any simple graph transformation is captured by some concrete neighbourhood shape transformation.

Theorem 6.16. Let $X \in \text{NShape}$ be a neighbourhood shape, $r = \langle L, R \rangle \in \text{SRule}$ be a simple graph transformation rule, and $m : L \rightarrow X$ be a match of r into X . Then,

1. for any simple graph $G \in \text{concr}(X)$ with abstraction morphism $\alpha_G : G \rightarrow X$, there exists a match $m' : L \rightarrow G$ such that $m = \alpha_G \circ m'$; and
2. if $G \xrightarrow{r,m'} H$ is a simple graph transformation, then there exists a concrete neighbourhood shape transformation $X \xrightarrow{r,m} Y$ and an abstraction morphism $\alpha_H : H \rightarrow Y$. $\triangleleft$

Statement 1 of the theorem above exactly corresponds to the one given in Lemma 6.14. The proof for statement 2 amounts to defining the mapping $\alpha_H : H \rightarrow Y$ and then showing that α_H is an abstraction morphism. Based on concrete shape transformations we define transformations for canonical shapes.

Definition 6.17 (Canonical neighbourhood shape transformation).

Given a canonical neighbourhood shape $X \in \text{CanNShape}_i^{n,e}$, a simple graph

transformation rule $r = \langle L, R \rangle \in \text{SRule}$, and a pre-match $m : L \rightarrow X$, let $X' \in \text{NShape}$ be a neighbourhood shape such that $\Omega_X : X' \rightarrow X$ is a normalisation morphism, $m' : L \rightarrow X'$ is a match of r into X' , and $m = \Omega_X \circ m'$. The canonical neighbourhood shape transformation of X is the canonical shape $Y \in \text{CanNShape}_i^{\text{n,e}}$, where $X' \xrightarrow{r, m'} Y'$ is a concrete neighbourhood shape transformation and $\Omega_Y : Y' \rightarrow Y$ is a normalisation morphism. $\triangleleft$

We write $X \Rightarrow^r Y$ to denote a canonical neighbourhood shape transformation. Neighbourhood shape X' is called a *materialisation* of X according to pre-match m . An essential property is that such a materialisation can always be constructed, as formalised in the following lemma.

Lemma 6.18. *Let X be a canonical neighbourhood shape and $r = \langle L, R \rangle$ be a simple graph transformation rule. If there exists a pre-match $m : L \rightarrow X$ of r into X , then there exists a match $m' : L \rightarrow X'$ of r into a neighbourhood shape X' , such that $\Omega_X : X' \rightarrow X$ is a normalisation morphism and $m = \Omega_X \circ m'$. $\triangleleft$*

The intuition behind the result of Lemma 6.18 is that the multiplicity conditions imposed over a pre-match m guarantee that a concrete part can be extracted from X and taken as the image of match m' . The actual construction of materialised shape X' is a rather complicated procedure, which is described in detail in Chapter 7.

The next theorem is the last result of this section, establishing the connection between canonical and concrete shape transformation.

Theorem 6.19. *Let $X \in \text{CanNShape}_i^{\text{n,e}}$ be a canonical neighbourhood shape, $r = \langle L, R \rangle \in \text{SRule}$ be a simple graph transformation rule, and $m : L \rightarrow X$ be a pre-match of r into X . Then,*

1. *there always exists a match $m' : L \rightarrow X'$ of r into a neighbourhood shape $X' \in \text{NShape}$, with $\Omega_X : X' \rightarrow X$ being a normalisation morphism and $m = \Omega_X \circ m'$; and*
2. *if $X' \xrightarrow{r, m'} Y'$ is a concrete neighbourhood shape transformation, then there exists a normalisation morphism $\Omega_Y : Y' \rightarrow Y$, where $Y \in \text{CanNShape}_i^{\text{n,e}}$ and $X \Rightarrow^r Y$ is a canonical neighbourhood shape transformation. $\triangleleft$*

The proof for the first statement of Theorem 6.19 follows trivially from Lemma 6.18. The proof obligation for statement 2 amounts to showing that there always exists a canonical shape Y that normalises Y' . From Definition 6.9 we have that Y is well-defined if the neighbourhood equivalence relation $\equiv_i$ over Y' is well-defined. Thus, the obligation boils down to showing that $\equiv_i$ is well-defined for Y' . This follows from the fact that (i) X' can be normalised into X , and thus so can be the part of Y' preserved by the concrete shape transformation; and (ii) $\equiv_i$ is always well-defined for trivially extended graphs, and therefore $\equiv_i$ is also well-defined for the newly created part of Y' (taken from the trivial extension of R).

From the combination of results presented so far we have as a corollary that any simple graph transformation is captured by some canonical neighbourhood shape transformation, as illustrated in the following picture.

The left-hand diagram in the picture corresponds to the statements in Theorem 6.16. This in turn implies the diagram on the right, where its top part comes from Theorem 6.19. Thus, since morphisms α and Ω are composable, it follows that the simple graph transformation $G \xrightarrow{r} H$ is captured by the canonical neighbourhood shape transformation $X \Rrightarrow^r Y$.

6.3.1 Neighbourhood shape transition systems

Building upon the results presented up to this point, we can now define a finite over-approximation for simple graph transition systems (SGTS – Definition 5.11). This over-approximation is a neighbourhood shape transition system (NSTS), a system where states are canonical neighbourhood shapes and transitions are canonical shape transformations. The rules that produce such transformations are taken from a simple graph grammar (Definition 5.10).

Definition 6.20 (NSTS). *A neighbourhood shape transition system is a tuple $NSTS = \langle \mathcal{S}, \Rightarrow, \iota \rangle$ where*

- $\mathcal{S} \subseteq \text{CanNShape}_i^{\mathbf{n}, \mathbf{e}}$ is a set of states;
- $\Rightarrow \subseteq \mathcal{S} \times \text{SRule} \times \mathcal{S}$ is a set of transitions; and
- $\iota \in \mathcal{S}$ is the initial state.

A simple graph grammar $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$ generates a $NSTS_{\mathcal{G}}$ if $\mathcal{S}$ is a minimal set of canonical neighbourhood shapes such that

- $\iota = \text{abstract}(G_0)$; and
- if $X \Rrightarrow^r Y$ for some $X \in \mathcal{S}$ and $r \in \mathcal{R}$, then there exists $Y' \in \mathcal{S}$ where $Y \simeq Y'$ and $\langle X, r, Y' \rangle \in \Rightarrow$. $\triangleleft$

It can be immediately seen that any NSTS is finite, since the universe $\text{CanNShape}_i^{\mathbf{n}, \mathbf{e}}$ of canonical shapes and the set of grammar rules $\mathcal{R}$ are both finite (Theorem 6.12 and Definition 5.10). The fact that a NSTS is an over-approximation of a SGTS follows from this next (and final) theorem.

Theorem 6.21. *Given a simple graph grammar $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$, let $SGTS_{\mathcal{G}} = \langle \mathcal{S}, \rightarrow, G_0 \rangle$ and $NSTS_{\mathcal{G}} = \langle \mathcal{S}', \Rightarrow, \text{abstract}(G_0) \rangle$ be the SGTS and NSTS generated by $\mathcal{G}$, respectively. Transition system $NSTS_{\mathcal{G}}$ simulates $SGTS_{\mathcal{G}}$. $\triangleleft$*

The reasoning for the proof of Theorem 6.21 is based upon Lemma 6.5; since isomorphic shapes have the same concretisations, the minimality of $\mathcal{S}'$ under shape isomorphism still preserves the simulation relation for concrete states. Furthermore, due to the results from Theorem 6.16 and Theorem 6.19, for any concrete transition in $SGTS_{\mathcal{G}}$ there exists a corresponding abstract transition in $NSTS_{\mathcal{G}}$.

6.4 Conclusion

The theory of neighbourhood abstraction unifies and generalises previous approaches to abstract graph transformation [RD06, BW07]. This particular theory establishes a general treatment of *local* abstractions, that is, abstractions based on equivalence relations, where equivalence is determined by local information on nodes. In contrast, equivalences used in shape analysis often consider global properties like reachability.

An interesting result of the theory of neighbourhood abstraction that was not covered in this chapter is the definition of a modal logic with counting that is *preserved* and *reflected* by the abstraction. While preservation is necessary to ensure the soundness of the analysis, reflection, on the other hand, is a rather unusual and strong result. Furthermore, the method lends itself well to abstraction refinement: if a property cannot be established given a certain neighbourhood radius, we can increase the radius and check again.

A question that arises during the theory presentation is whether the exponential upper bound on the number of canonical shapes can actually occur in practice. In order to answer this question it is necessary to provide a working implementation of the theory. This point is covered in the next chapter, where practical aspects are discussed and experimentally evaluated. Also, several examples for the abstraction steps discussed here are given in Chapter 7.

Neighbourhood abstraction – Practice

In this chapter, we discuss in detail the practical aspects of the implementation for the neighbourhood abstraction theory presented in Chapter 6. This text unifies and updates the work originally published in [RZ10] and [ZR12].

We begin with an overview of the algorithm for abstract state space exploration and we present some of the procedures implemented. In particular, we give an extensive explanation about shape materialisation, by far the most complex step of the method. Subsequently, we discuss improvements over the exploration algorithm that are necessary to achieve an acceptable performance level. The chapter concludes with an experimental evaluation that shows the current state space sizes that can be handled by the tool.

7.1 Introduction

In this chapter we give a high-level presentation of the design choices made during the code development. We try to focus on the major challenges encountered and on the ideas for surmounting them. Low-level details of the implementation are left out on purpose, since they are usually language dependent. Nevertheless, such minutiae should also be taken into account when considering the overall contribution of this work, since low-level optimisations usually require a substantial amount of time to be completed.

Generally speaking, the execution of neighbourhood abstraction amounts to the construction of a transition system representing the abstract state space. States of this system are canonical neighbourhood shapes and transitions correspond to canonical neighbourhood shape transformations. Since these canonical transformations are in turn defined over concrete shape transformations, during the abstraction execution we have to compute a set of *shape materialisations* such that the concrete shape transformations guarantee the coverage of all rule applications in the concrete state space. The theory ensures that this materialisation set is finite, but in practice it can still be considerably large. Performance of the abstraction mechanism is quite dependent on the efficient computation of such sets: the construction should perform only the minimal amount of work necessary to guarantee the abstraction correctness.

7.2 Construction of a NSTS

Listing 7.1 presents the main algorithm for the abstract state space exploration of a simple graph grammar $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$. This algorithm computes the

$NSTS_{\mathcal{G}} = \langle \mathcal{S}, \Rightarrow, \iota \rangle$ generated by $\mathcal{G}$ (Definition 6.20). The pseudo-code structure in Listing 7.1 is very similar to the one given in Listing 5.3 for a concrete exploration, with the difference that additional operations from the abstraction mechanism are incorporated. Similarly to the concrete version, this algorithm maintains a set $\mathcal{F}$ of fresh, yet to be explored states.

Listing 7.1: Algorithm for abstract state space exploration (NSTS generation).

```

1 let  $\mathcal{S} := \emptyset, \Rightarrow := \emptyset, \iota := \text{abstract}(G_0), \mathcal{F} := \{\iota\}$ 
2 while  $\mathcal{F} \neq \emptyset$ 
3 do choose  $X \in \mathcal{F}$     // which  $X$  is selected depends on the exploration strategy
4   let  $\mathcal{F} := \mathcal{F} \setminus \{X\}$ 
5   for  $r \in \mathcal{R}, m \in \text{prematch}(r, X), \langle m', X' \rangle \in \text{materialise}(m, X)$ 
6   do let  $Y := \text{normalise}(\text{apply}(r, m', X'))$ 
7     if  $\text{isFresh}(Y, \mathcal{S}, \mathcal{F})$     // if  $Y \notin \mathcal{S}$ 
8     then let  $\mathcal{S} := \mathcal{S} \cup \{Y\}, \mathcal{F} := \mathcal{F} \cup \{Y\}$ 
9     fi
10    let  $\Rightarrow := \Rightarrow \cup \{X \Rightarrow^r Y\}$ 
11  od
12 od
13 let  $\langle \overline{\mathcal{S}}, \overline{\Rightarrow}, \overline{\iota} \rangle := \text{reduce}(\mathcal{S}, \Rightarrow, \iota)$ 

```

The same explanations given for the concrete exploration algorithm in Section 5.2.3 also hold for the algorithm in Listing 7.1. Procedure `isFresh` in line 7 is the same as the one originally presented in Listing 5.4, with the difference that isomorphism checks and graph certificates are accordingly modified for canonical neighbourhood shapes. Shape isomorphism was defined previously in Chapter 6. (Certificate computation for shapes requires a simple extension of the certificate calculated for the graph part of a shape.) The third argument of function `isFresh` in Listing 7.1 remains unused for now. This point is properly discussed in Section 7.5, together with the `reduce` operation in line 13.

The additional phases of the abstract exploration algorithm are the following.

- `abstract` constructs the canonical neighbourhood shape of a simple graph, as discussed in Section 6.2.1.
- `prematch` computes non-injective morphisms of a rule $r = \langle L, R \rangle$ into a canonical neighbourhood shape X ; such a morphism $m : L \rightarrow X$ is a pre-match (Definition 6.13). Some of the nodes and edges in the image of $m(L)$ may be collector elements; in this case they have to be materialised.
- `materialise` creates concrete nodes and edges for the image of r in X . This is a non-deterministic step, as there may be multiple options for choosing multiplicities for the materialised elements. The result of this phase is a set of (non-canonical) shapes X' and matches $m' : L \rightarrow X'$ (Definition 6.13).
- `apply` is a concrete neighbourhood shape transformation (Definition 6.15), which can be carried out as if the shape were a normal graph because at this point, match m' is injective and maps to a concrete subgraph of X' .
- `normalise` constructs a canonical neighbourhood shape Y from the shape produced by the rule application; as with step `abstract`, this again corresponds to the method discussed Section 6.2.1.

The following sections present these phases in more detail.

7.2.1 Operations abstract, prematch, apply and normalise

Execution of operation **abstract** begins with the computation of neighbourhood equivalence relation $\equiv_i$ over the start graph G_0 of a grammar $\mathcal{G}$ (this procedure is explained, for instance, in Example 6.8). After relation $\equiv_i$ is obtained, a canonical shape is constructed, as detailed in the previous chapter and illustrated in Example 6.10.

A pre-match of the LHS of a rule $r = \langle L, R \rangle$ into a canonical shape X , as formalised in Definition 6.13, is a non-injective morphism $m : L \rightarrow X$, such that node and edge multiplicities are satisfied by the mapping. Operation **prematch** uses the normal rule matching implemented in GROOVE and then discards invalid matches that do not satisfy the conditions of a $\leq$ -morphism (Definition 6.2). A pre-match m has to be massaged into a match m' , where: (i) m' is injective; (ii) all nodes in the image of m' are concrete and belong to a singleton equivalence class; and (iii) all edges in the image of m' are concrete. This adjustment from pre-match m to match m' is done by the **materialise** operation, described in the next section.

Given a match m' into a materialised shape X' , rule application is performed as usual. Operation **apply** simply uses the normal graph transformation code in GROOVE.

After transformation, a neighbourhood shape needs to be normalised, *i.e.*, some concrete parts may be merged back into the shape. This entails the computation of relation $\equiv_i$ on shapes, similarly to what was described for operation **abstract**.

7.3 Operation materialise

The materialisation operation is the most complex step of the abstraction algorithm. The complexity arises from the need to resolve all non-determinism of the method; materialised shapes returned by **materialise** are ready to be transformed by conventional rule application.

Given a pre-match m of a rule $r = \langle L, R \rangle$ into a canonical neighbourhood shape X , **materialise** finds all shapes X' such that $\Omega : X' \rightarrow X$ is a normalisation morphism and $m' : L \rightarrow X'$ is a match (Definition 6.13). The challenge in implementing this step lies in transforming the declarative characterisation of materialised shapes just given into a constructive algorithm that produces the set of all possible materialisations, based on the given shape X and pre-match m .

The materialisation algorithm iteratively changes the original shape in order to search for valid materialisations. At each iteration, one or more new shapes X' are cloned from the current shape X , and are modified. The main idea of the procedure is that all X' obtained at any iteration step are “more concrete” than X , *i.e.*, the morphism from the rule LHS L into X' is closer to satisfying the match conditions of Definition 6.13 than the morphism of L into X . This in turn implies that X is more abstract than X' in the sense that there exists a $\sqsubseteq$ -morphism from X' into X . The key correctness requirement for each materialisation step is that the union of the set of concretisations for all X' must cover all concretisations of X , *i.e.*, $\text{concr}(X) \subseteq \bigcup \text{concr}(X')$. It is impor-

Op.	Pr.	Parameters	Creates
matNode	0	$v_c \in N_X$ collector node, from which new nodes will be materialised $N_r^m \subseteq N_L$ set of rule nodes mapped to v_c by pre-match m	singNode
matEdge	1	$e_c \in E_X$ collector edge, from which new edges will be materialised $E_r^m \subseteq E_L$ set of rule edges mapped to e_c by pre-match m	pullNode
pullNode	2	$v_c \in N_X$ collector node being pulled by e_p $e_p \in E_X$ edge pulling a new node from v_c $u \in \mathcal{M}$ multiplicity of the new node	–
singNode	3	$v_s \in N_X$ node to be singularised	–

Table 7.2: Summary of sub-operations of the materialisation phase. All operations are performed on a given neighbourhood shape X , guided by a pre-match m of the LHS L of a rule r into X .

tant to note that this correctness criterion was considered only in an intuitive level when developing this operation; we do not provide a formal proof that the implementation indeed respects this criterion.

The changes performed by this step are divided into four *materialisation operations*, with their summary given in Table 7.2. The second column of Table 7.2 lists the operation priority, with zero being the highest priority. Operation **singNode** is non-deterministic, meaning that it may produce zero or more new materialisation objects. If the execution yields zero results, then it is said that the operation failed, *i.e.*, performing the operation on the materialisation object does not produce a valid shape. Note that while a single **singNode** operation may fail, the whole materialisation process always produces at least one shape. This is a result from Lemma 6.18, and is checked in the implementation by properly placed assertions in the code.

Materialisation operations are put into a priority queue and traversed in a depth-first fashion. When the queue has no more operations scheduled to be performed in shape X' , then the materialisation of X' is complete, and the shape is added to the result set of **materialise**, together with match m' that was constructed alongside X' . Since not all operations can be determined at the start of the materialisation process, at the end of an operation additional ones can be created. The relation for the creation of new operations is given in the fourth column of Table 7.2. We proceed to explain these operations in detail.

7.3.1 Materialise node

This operation materialises (creates) one or more nodes from a collector node v_c , *i.e.*, a node with multiplicity greater than one. As shown in the third column of Table 7.2, the other parameter of **matNode** is N_r^m , the sub-set of nodes in the LHS of the rule that were mapped to v_c by pre-match m . The number of new copies of the collector node is determined by the cardinality of set N_r^m . All

Figure 7.4: Example of a `matEdge` operation.

The multiplicity for the end points of e_c are both set to one (in fact nothing changes because the values were already concrete) and the multiplicity for the remainder edges in the bundles where e_c occur is subtracted by $|E_r^m| = 1$. Since the multiplicities for all edge bundles in this example are one, the remainder multiplicity is **0**, which has the effect of removing some edges from shape S_1 . In this case, node v_2 has exactly one outgoing n -edge to v_5 (as established by the adjusted match) and therefore edge $\langle v_2, n, v_3 \rangle$ cannot exist. Similarly, node v_5 can only have one incoming n -edge and thus edges $\langle v_5, n, v_5 \rangle$ and $\langle v_3, n, v_5 \rangle$ are removed from the resulting shape S_2 . At the end of this edge materialisation example, a new `pullNode` operation is created with parameters $v_c = v_3$, $e_p = \langle v_5, n, v_3 \rangle$, and $u = 1$. The conditions for the creation of this new operation are given in the next section.

7.3.3 Pull node

The theory of neighbourhood abstraction requires all nodes in the match image to be in a singleton equivalence class according to the shape grouping relation. After performing `matNode` and `matEdge` operations, we obtain a rule match where the image is concrete but that may still have non-singular nodes. This problem is repaired by the `singNode` operation, performed later on in the materialisation process. Operation `singNode`, however, assumes that the number of nodes in the shape under materialisation is already fixed. Operation `pullNode` establishes this pre-requisite, by further extracting new nodes out of collector nodes when needed.

The reasoning for deciding when to perform this operation is as follows. If there exists a node scheduled to be singularised, then we have to check for this node v_s if the opposing edge bundles can be properly split (meaning that the edges are no longer joined in a bundle). Thus, for any edge e_p incident to v_s we have to check if the opposite node v_c is a collector node and it has an edge bundle containing e_p . In this case, node v_c has to be split; we say that a new node is being “pulled out” of the collector node by edge e_p . Multiplicity u of the new node equals the multiplicity of the side of edge e_p that is adjacent to v_s .

Figure 7.5: Example of a pullNode operation.

This operation is very similar to `matNode`, with the exception that only one new node is created, which can have an arbitrary positive multiplicity, defined by parameter u . As in `matNode`, all adjacent edges of the collector node are duplicated but, in addition, the pulling edge e_p is removed from the new shape, since this edge cannot exist in the final materialisation. Operation `pullNode` is deterministic and does not create any new operations: the newly created node will not be singularised later.

Figure 7.5 gives an example of a `pullNode` operation, performed on shape S_2 produced by the `matEdge` operation in Figure 7.4. Node v_5 is scheduled to be singularised (the operation was created by a node materialisation) and so we have to check the edges incident to v_5 . For edge $e_p = \langle v_5, n, v_3 \rangle$ we see that node $v_c = v_3$ is a collector node, and this node has a shared incoming edge bundle. (This bundle cannot exist after v_5 is made singular because the sharing can no longer be represented. More specifically, in shape S_2 we have $\text{mult}^i(v_3, n, \{v_2, v_3, v_4, v_5\}) = 1$ and since v_5 has to be removed from its current equivalence class, this means that the mapping of function mult^i has to change.) Thus, e_p is a pulling edge and v_c is being pulled. Multiplicity u is given by $u = 1 = \text{mult}^o(v_5, n, \{v_2, v_3, v_4, v_5\})$. The new shape S_3 produced by this `pullNode` execution is given on the right side of Figure 7.5. The new pulled node v_6 is concrete, since $u = 1$. This value is also subtracted from the multiplicity of v_c : node v_3 now has multiplicity 0^+ in shape S_3 .

7.3.4 Equation systems

An equation system is a device used to search for valid shape configurations during the materialisation phase, in particular by operation `singNode`. Variables in these systems hold multiplicity values, and thus are called *multiplicity variables*. Despite its name, an equation system is composed of inequalities, all of the form

$$x_1 + \dots + x_k \sqsubseteq u_n * u_e$$

where x_1 to x_k are multiplicity variables, and u_n and u_e are node and edge multiplicity constants, respectively. These inequalities restrict the overall ad-

Figure 7.6: Example of a `singNode` operation.

missibility of a shape configuration.

Solving an equation system can be done with a simple search algorithm that picks an equation and assigns values to the variables such that (i) the subsumption relation is valid and (ii) the remaining equations are also satisfied. In the current implementation, each subsumption equation is translated to two inequalities that limit the lower and upper bounds of the multiplicity variables. After this translation, a dedicated cutting-plane method is executed to solve the system. Although in general Integer Programming is a NP-hard problem, in this particular setting it can usually be solved efficiently because we can include additional search heuristics in the solver implementation.

Although a single equation system may admit several solutions, the aforementioned heuristics take into account the fact that we are not looking for all solutions. Instead, we are interested in the most general ones w.r.t. subsumption. Thus, given two solutions $\vec{x}$ and $\vec{y}$, if $\vec{x} \sqsubseteq \vec{y}$ (point-wise) then $\vec{y}$ is preferred. Each valid solution produces a corresponding shape, obtained from the multiplicity values in the variables.

7.3.5 Singularise node

This is the operation with the lowest priority, being executed after all others. When performing a `singNode` operation, we assume that the rule match is final and that the number of nodes in the shape will no longer change. What is left to decide are the outgoing and incoming multiplicities of the edge bundles that will be affected by the operation.

In order to put a node v_s in a singleton equivalence class, `singNode` creates an equation system, where the variables represent the multiplicities of edges in affected bundles. For each such edge we create a multiplicity variable. Each edge bundle gives rise to an equation in the system, with variables taken from

the edges that compose the bundle and the multiplicity constants taken from the node and edge bundle multiplicities.

Figure 7.6 shows a `singNode` execution with parameter $v_s = v_5$. The input shape S_4 is obtained from shape S_3 in Figure 7.5 where node v_2 is already singularised. That operation is trivial since there are no shared edges incident to v_2 . This `singNode` execution on v_3 creates the following equation system

$$x_1 + x_2 \sqsubseteq \mathbf{1} * \mathbf{1} = \mathbf{1} \quad (7.1)$$

$$x_3 + x_4 + x_5 \sqsubseteq \mathbf{0}^+ * \mathbf{1} = \mathbf{0}^+ \quad (7.2)$$

$$x_6 + x_7 + x_8 \sqsubseteq \mathbf{1} * \mathbf{1} = \mathbf{1} \quad (7.3)$$

$$x_3 + x_6 \sqsubseteq \mathbf{0}^+ * \mathbf{1} = \mathbf{0}^+ \quad (7.4)$$

$$x_2 + x_5 + x_8 \sqsubseteq \mathbf{1} * \mathbf{1} = \mathbf{1} \quad (7.5)$$

$$x_1 + x_4 + x_7 \sqsubseteq \mathbf{1} * \mathbf{1} = \mathbf{1} \quad (7.6)$$

where the association of multiplicity variables to edges is shown in shape S_4 of Figure 7.6. The set of equations above capture all consistency requirements w.r.t. the multiplicities of shape S_4 , and, in principle, this system could be solved straight away. However, this would produce too many solutions, since the given equation system over-specifies the set of possible materialisations. Therefore, for efficiency's sake, it is necessary to simplify these equations, as discussed in the following.

First, equations (7.2) and (7.4) do not restrict any possible values of the variables in the left-hand side because their multiplicity constants are $\mathbf{0}^+$. Thus, these two equations can be discarded since they do not provide any relevant information for solving the system.

Second, we can see that both bundles of node v_3 and the outgoing bundle of node v_6 are not affected by the singularisation of v_5 since there are no edges in these bundles connected to v_5 . In this case, the multiplicities of these edge bundles will remain unchanged, and keeping their associated equations in the system is harmful, since it leads to unnecessary branching. Hence, in this example, equation (7.3) can also be discarded.

Third, and lastly, we can group variables that do not have to be considered in isolation. These variables are identified by analysing the equivalence relation that is going to be split. In this particular example, by looking at the incoming edge bundles we can see that only edges incoming from node v_5 need to distinguished. Thus, the pairs of variables $\langle x_4, x_7 \rangle$ and $\langle x_5, x_8 \rangle$ can be merged. This analysis finally leads to the following simplified equation system

$$x_1 + x_2 \sqsubseteq \mathbf{1} \quad (7.7)$$

$$x_2 + x_{58} \sqsubseteq \mathbf{1} \quad (7.8)$$

$$x_1 + x_{47} \sqsubseteq \mathbf{1} \quad (7.9)$$

which can be solved trivially and admits two solutions.

x_1	x_2	x_{47}	x_{58}
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$

The first solution leads to shape S_5 in Figure 7.6, and the second solution

Figure 7.7: NTS of the running example, constructed with radius $i = 1$ and multiplicity bounds $\mathbf{n} = \mathbf{e} = 1$.

produces shape S_6 . These two shapes are fully materialised and correspond to the final result of operation `materialise`.

As a last note, it should be pointed that while for this running example we have started with the non-simplified set of equations (7.1-6), this was done for presentation purposes only, in order to discuss the reasoning that lead to the simplified set of equations. In practice the implementation produces the final simplified equation system straight away.

7.4 Initial results and discussion

When applying the implemented abstraction to our running example of a linked list, we obtain the abstract state space depicted in Figure 7.7. For an abstraction radius of one, the NTS has 10 states and 20 transitions. There are several interesting points of note in this transition system.

- As long as node and edge multiplicities stay within their bounds, the abstract graph transformation corresponds to the concrete one. This is seen on states s_0 , s_1 , and s_2 , where the shapes are concrete.
- An abstract state may represent an unbounded number of concrete ones. State s_3 , for example, is an abstract representative for lists with four or

Grammar	Total state count	Time (s)	Memory (MB)
linked-list	10	< 1	< 1
circ-buf-0	54	< 1	2
circ-buf-1	57	< 1	2
euler-0	878	47	57
euler-1	14	< 1	< 1
firewall-2	125	2	6
firewall-3	1,625	157	110
firewall-4	4,875	3,824	432
firewall-5			
car-platoon			

Table 7.8: Initial experimental results for different graph grammars. The strategy used is DFS with parameters $i = \mathbf{n} = \mathbf{e} = 1$. Empty lines indicate timed-out executions.

- more elements. This is illustrated by the `put` transition from s_3 to itself.
- The non-determinism of the materialisation can be seen from the two `get` transitions outgoing from states s_3 , s_5 , and s_8 . Although there is only one pre-match of the rule, when materialising this pre-match two distinct shapes are produced, as detailed in the examples of the previous section.
 - The abstract state space has spurious configurations. For example, states s_4 , s_6 , s_7 , and s_9 represent lists with elements disconnected from the head, a case that does not occur in the concrete state space. These spurious shapes arise from the fact that the neighbourhood abstraction mechanism does not keep information regarding connectivity of nodes.

After the NSTS is generated, we can proceed to check the properties of interest. For the abstract transition system in Figure 7.7, we can check, for example, that the following properties hold: (i) the head cell has no predecessors; (ii) the cells are not shared; and (iii) rule `get` can be applied infinitely often. These properties are informally described in English but can be easily translated into temporal logic formulæ and structural graph properties, as detailed, for example, in Section 2.2.2. Thus, since these properties hold in the abstract NSTS, from Theorem 6.21 we can conclude that they also hold in the infinite concrete SGTS.

Initial experimental results and evaluation

Table 7.8 gives the results for the initial experiments performed with the neighbourhood abstraction implementation. The grammars listed in the table are taken from the test cases presented in Section 5.5. For each grammar we list its state space size (column **Total state count**), and the running time and memory consumption of the tool. The experiments were performed in a machine with a Intel Xeon X5365 CPU running at 3 GHz and a total 32 GB of RAM. Blank entries indicate timed-out executions. The numbers in Table 7.8 were obtained with a DFS exploration strategy and abstraction parameters $i = \mathbf{n} = \mathbf{e} = 1$.

From the experimental results it can be seen that although the state space sizes are reasonably small, running times are quite high. This is due to the complexity of the abstract exploration algorithm, in particular of the materialisation phase. Although part of the complexity is inherent to the method, it is clear that some improvements are necessary to increase performance.

Motivating the need for an implementation refactoring

The points presented so far cover the first implementation (V1) of neighbourhood abstraction that was originally developed and corresponds to the research published in [RZ10]. This first implementation can be considered a prototype, in the sense that its major purpose was to provide a working abstraction tool in order to allow experimentation. Given the code complexity, a concern that was not addressed at this point was performance optimisation, something that is reflected in the results of Table 7.8. Further work on performance improvements led to a complete code refactoring and a second abstraction implementation (V2). It is this second version that was used in the experiments presented in Section 7.6.

A major design decision made during the development of (V2) was to fix the abstraction radius i to 1. Since some additional (not shown here) experimentation with (V1) indicated that state space sizes would quickly blow up for larger radius, this restriction made for (V2) was considered an acceptable compromise and allowed for several simplifications and code optimisations.

Upon further analysis of the NSTS of Figure 7.7 we can see that multiplicities introduce an additional combinatorial factor in the abstract state space. For instance, states s_3 , s_5 , and s_8 have exactly the same structure, differing only at the multiplicity of a single node. The work presented at the next section aims to exploit this sort of state similarity in order to reduce the effort for abstract state space generation.

7.5 Subsumption for state space reduction

On concrete state space exploration, the method of graph certificates for isomorphism checking works very well in practice (see Section 5.2.3). In principle, since shapes also have a graph structure, we can immediately reuse the same algorithm of procedure `isFresh` in Listing 5.4 for abstract exploration. (Assuming, as stated before, that the computation of certificates is properly adapted for shapes.) However, neighbourhood shapes carry additional multiplicity information that is not fully taken into account when using only isomorphism checks. In order to profit from this additional information, we developed a new state equivalency test, based on the *shape subsumption* $\sqsubseteq$ relation. In this section we proceed to explain this new method of subsumption collapsing for abstract states and give a modified version for procedure `isFresh`.

7.5.1 Shape Subsumption

Soundness of the abstraction over-approximation is in part due to the result of Lemma 6.5, stating that isomorphic shapes have the same set of concretisations. In practice, however, the experimentation in Section 7.4 has shown that shape isomorphism is too strong, *i.e.*, it distinguishes too many shapes that could actually be considered equivalent, in a sense that will be properly defined shortly. Due to this fact, it stands to reason that one might be interested to use a weaker

(but still sound) characterisation of state equivalency, in order to shrink the abstract state space size.

Following the reasoning above, a theoretical result that can be exploited is the one in Lemma 6.6, stating that if shape X is subsumed by Y ($X \sqsubseteq Y$), then the set of concretisations of X is contained in the set of concretisations of Y ($\text{concr}(X) \subseteq \text{concr}(Y)$). The insight here lies in the comparison between the concretisation sets: since Y has more concretisations than X , rule applications on Y capture more behaviour than rule applications on X . In fact, all behaviour of X is subsumed by the behaviour of Y , and therefore, in an abstract exploration we can discard X and only explore Y . More formally, if $X \sqsubseteq Y$ then for all $X \Rightarrow^r Z$ there exists $Y \Rightarrow^r Z'$ such that $Z \sqsubseteq Z'$, for any $r \in \mathcal{R}$.

There is, however, a strong deterrent that prevents the immediate usage of this theoretical result in practice, namely the fact that subsumption checks can be quite expensive to compute over a large set of states. (Generally speaking, comparing two shapes for subsumption is equivalent to the matching of sub-graphs, which is a NP-complete problem.) To circumvent this hindrance, in the implementation we work with a relation that lies midway between isomorphism and subsumption in terms of strength, and that can be efficiently computed.

This new relation is called *iso-subsumption*, written as $X \sqsubseteq Y$, if X is *iso-subsumed* by Y . Iso-subsumption is an asymmetric relation built upon graph isomorphism. Shape X is iso-subsumed by shape Y if: (i) their graph structures are isomorphic, (ii) the two shapes have the same node similarity relation (modulo renaming), and (iii) all multiplicities in X are subsumed by the multiplicities in Y . This is formally expressed in the following definition.

Definition 7.1 (Iso-subsumption). *Let X and Y be two neighbourhood shapes. Shape X is iso-subsumed by Y , denoted $X \sqsubseteq Y$, if there exists a simple graph isomorphism $\varphi : G_X \rightarrow G_Y$, and, in addition:*

1. *for all $v, w \in N_X$, $v \sim_X w$ if and only if $\varphi(v) \sim_Y \varphi(w)$;*
2. *for all $v \in N_X$,*

$$\text{mult}_X^n(v) \sqsubseteq \text{mult}_Y^n(\varphi(v)) ;$$

3. *for all $\langle v, a, w \rangle \in E_S$,*

$$\text{mult}_X^o(v, a, [w]_{\sim_X}) \sqsubseteq \text{mult}_Y^o(\varphi(v), a, [\varphi(w)]_{\sim_Y}), \text{ and}$$

$$\text{mult}_X^i(w, a, [v]_{\sim_X}) \sqsubseteq \text{mult}_Y^i(\varphi(w), a, [\varphi(v)]_{\sim_Y}) .$$

◁

It is easy to see that the conditions in the definition above correspond to the conditions of a $\sqsubseteq$ -morphism (Definition 6.2), with the sole difference that multiplicity sums do not appear in Definition 7.1. This is due to the additional requirement for the existence of a graph isomorphism, which forces the correspondence between graph elements to be a bijection.

It is also trivial to see that $X \sqsubseteq Y$ implies $X \sqsubseteq Y$ and therefore the result of Lemma 6.6 still holds for relation $\sqsubseteq$. As a simple example, take X as a shape containing only one node of multiplicity 2^+ and no edges. Then take Y also as a shape with a single node, but with multiplicity 1^+ . From Definition 7.1, we have that $X \sqsubseteq Y$. Set $\text{concr}(X)$ contains graphs with two or more nodes, whereas set $\text{concr}(Y)$ has one more element, namely the graph with just one node. Thus, $\text{concr}(X) \subseteq \text{concr}(Y)$.

7.5.2 Implementation

Listing 7.9 shows the modified algorithm for procedure `isFresh`, with iso-subsumption checks incorporated. Given a new shape Y that must be tested for freshness, we begin as in the original procedure implementation, by constructing set $\bar{\mathcal{S}}$, consisting of the shapes in $\mathcal{S}$ with the same certificate as Y . Since iso-subsumption is an asymmetric relation, we must check it in both directions (lines 3 and 5 of the algorithm). Note, however, that these two subsumption checks do not require two isomorphism checks, because we can first look for an isomorphism between G_Y and G_Z (potentially the most expensive operation) and then proceed to check both subsumptions using the same isomorphism.

Listing 7.9: Algorithm for procedure `isFresh( $Y, \mathcal{S}, \mathcal{F}$ )` with iso-subsumption.

```

1 let  $C := \text{certificate}(Y)$ ,  $\bar{\mathcal{S}} := \{Z \in \mathcal{S} \mid C = \text{certificate}(Z)\}$ 
2 for  $Z \in \bar{\mathcal{S}}$ 
3   do if  $Y \sqsubseteq Z$       // if  $Y$  is iso-subsumed by  $Z$ 
4     then return false
5   else if  $Z \sqsubseteq Y$     // if  $Z$  is iso-subsumed by  $Y$ 
6     then let  $\mathcal{F} := \mathcal{F} \setminus \{Z\}$     // remove  $Z$  from the set of states to be explored
7   fi
8 fi
9 od
10 return true

```

An interesting aspect of the new `isFresh` procedure is that it can now modify the set $\mathcal{F}$ of shapes to be explored. If there exists $Z \in \bar{\mathcal{S}}$ such that $Y \sqsubseteq Z$, then Y is not fresh and can be discarded as before (lines 3 and 4 of Listing 7.9). However, if we discover that $Z \sqsubseteq Y$, then not only do we know that Y is fresh (because $\mathcal{S}$ is kept minimal w.r.t. relation $\sqsubseteq$), but also that Z should not be explored, since all its behaviour is subsumed by Y . We then remove Z from $\mathcal{F}$ (line 6), thus trimming the search space.

This trimming by removing subsumed shapes from the set of shapes to be explored is an essential step in reducing the computational effort of the abstraction. On the other hand, this reduction has the undesirable side-effect of leaving the $NSTS_{\mathcal{G}}$ under construction incomplete: states marked as iso-subsumed are not explored and therefore have no outgoing transitions. Being incomplete renders the $NSTS_{\mathcal{G}}$ unsuitable for model-checking; the system no longer corresponds to a proper over-approximation of the concrete state space, in the sense that a simulation relation between $NSTS_{\mathcal{G}}$ and $SGTS_{\mathcal{G}}$ cannot be directly constructed. However, this problem can be easily repaired by referring to the additional information provided by relation $\sqsubseteq$, as explained in the following.

After the abstract state exploration finishes, using the resulting $NSTS_{\mathcal{G}}$ and the transitive closure of iso-subsumption (denoted $\sqsubseteq^*$), we build a reduced transition system $\overline{NSTS}_{\mathcal{G}} = \langle \bar{\mathcal{S}}, \overrightarrow{\quad}, \bar{\iota} \rangle$, where the states in $\bar{\mathcal{S}}$ are only shapes that were not marked as iso-subsumed during the exploration. Formally, for any shape $X \in \mathcal{S}$, we define a function `sub` that returns the *subsumptor* of X , i.e., $\text{sub}(X) = Y \in \mathcal{S}$ such that $X \sqsubseteq^* Y$, and $\nexists Z \in (\mathcal{S} \setminus \{Y\})$ where $Y \sqsubseteq^* Z$. Thus, `sub` acts as the identity function for non-subsumed shapes, and computes the

maximum element of relation $\sqsubseteq^*$ for the subsumed ones. System $\overline{NSTS}_G$ is then defined as

$$\begin{aligned}\bar{\mathcal{S}} &= \{X \in \mathcal{S} \mid \text{sub}(X) = X\} \\ \bar{\Rightarrow} &= \{(\langle \text{sub}(X), r, \text{sub}(Y) \rangle \mid \langle X, r, Y \rangle \in \Rightarrow)\} \\ \bar{\iota} &= \text{sub}(\iota) .\end{aligned}$$

It is simple to see that the reduced transition system is complete and simulates its concrete counterpart. The definition above corresponds to the implementation of operation **reduce**, given at line 13 of Listing 7.1.

From the discussion above, one may wonder why shapes that were marked as iso-subsumed are kept in set $\mathcal{S}$ during the exploration. The reason is that removing states from $\mathcal{S}$ could leave “dangling” transitions in the set of generated transitions $\Rightarrow$. A possible solution could be the following. For $X \in \mathcal{S}$ and Y fresh, if $X \sqsubseteq Y$ then we should take all transitions in $\Rightarrow$ with X as a source or target and replace X by Y . This on-the-fly state space collapsing under iso-subsumption is not provided by the current implementation, but the tool offers a simpler option: *reachability mode*. In this mode we are only interested in the shapes that are reachable in the abstract state space, and thus there is no need to store the transitions, *i.e.*, set $\Rightarrow$ is kept empty. In this case there is no danger of having “dangling” transitions and we can remove iso-subsumed shapes from $\mathcal{S}$, thus decreasing memory usage.

7.6 Experiments and results

The theory of neighbourhood abstraction ensures that the number of canonical neighbourhood shapes is bounded, and therefore that the abstract state space is finite (NSTS– Definition 6.20). However, as stated in the previous chapter, the theoretical upper bound of the abstract state space size is still quite large, meaning that in practice we have to optimise the state space traversal in order to implement an efficient tool. Throughout this chapter we have described the major aspects of the neighbourhood abstraction implementation in GROOVE, and in this section we present some experimental results that illustrate: (i) the upper bounds on abstract state space sizes that are reachable in practice, (ii) the performance gains that are obtained with the iso-subsumption relation presented in Section 7.5, and (iii) how different exploration strategies perform in the abstract setting.

7.6.1 Results

Table 7.10 presents all the figures on state space sizes for the grammars presented in Section 5.5. Numbers for BFS and DFS exploration strategies are grouped per grammar, to ease the comparison between the two. State space sizes are given as the number of explored states, *i.e.*, the number of neighbourhood shapes produced. State counts in Table 7.10 are broken down in the following five groups.

- **Total** is the upper bound on the number of abstract states of the grammar. This number is obtained by exploring the state space with collapsing under shape isomorphism, *i.e.*, by running the exploration algorithm of Listing 7.1

Grammar	Strat.	Total	Generated	Subsumed	Relevant	Discarded
linked-list	BFS	10	10	4	6	0
	DFS		8	2	6	1
circ-buf-0	BFS	54	31	5	26	3
	DFS		39	13	26	2
circ-buf-1	BFS	57	33	16	17	0
	DFS		30	13	17	2
euler-0	BFS	878	248	96	152	54
	DFS		213	61	152	28
euler-1	BFS	14	14	4	10	0
	DFS		13	3	10	2
firewall-2	BFS	125	98	90	8	36
	DFS		50	42	8	15
firewall-3	BFS	1,625	991	971	20	549
	DFS		232	212	20	102
firewall-4	BFS	4,875	2,356	2,326	30	1,487
	DFS		427	397	30	212
firewall-5	BFS		14,878	14,818	60	10,549
	DFS		1,201	1,141	60	654
firewall-6	BFS		25,251	25,171	80	18,373
	DFS		1,783	1,703	80	1,007
firewall-6-F	BFS		183,478	182,966	512	147,028
	DFS		5,930	5,418	512	3,003
car-platoon	DFS	Out of memory after 445,439 states and 8,484,600 transitions				

Table 7.10: State space sizes for different graph grammars ($i = \mathbf{n} = \mathbf{e} = 1$).

with the original **isFresh** procedure of Listing 5.4. (These are the same numbers reported in Table 7.8.) Empty entries in this column for the larger instances of the firewall grammar indicate that the upper bound could not be computed: these runs timed out, due to state space explosion. Note that we give only one total state count for both BFS and DFS strategies, since in this case all states are explored, and therefore the total state count is the same, regardless of the strategy used. The upper bound provided by this column gives an interesting basis of comparison when analysing the reduction obtained with iso-subsumption.

- **Generated** is the number of states produced during exploration using iso-subsumption, *i.e.*, the exploration algorithm of Listing 7.1 was run with the new **isFresh** procedure of Listing 7.9. Numbers in this column correspond to the size of set $\mathcal{S}$, *i.e.*, the number of states that were added to the set of all explored states (line 8 in Listing 7.1). When comparing the number of generated states against the **Total** column we can see the reduction given by iso-subsumption. Take, for example, the *firewall-4* line, where the number of generated states using DFS with iso-subsumption is an order of magnitude smaller than the total upper bound obtained without subsumption. The gain provided by iso-subsumption can also be seen for the larger cases of the firewall grammar: runs that timed-out without subsumption can now be finished when we turn it on.
- **Subsumed** is the number of states generated that were later identified as iso-subsumed by another state. This count combines iso-subsumed states that were already explored (and thus account for “wasted” exploration effort) and also the discarded states.
- **Relevant** is the number of states that were never marked as iso-subsumed during the exploration. This column corresponds to the **Generated** column

minus the **Subsumed** one. The closer the number of generated states gets to the relevant state count the better. A perfect exploration method would generate only the relevant abstract states (*i.e.*, the reduced system $\overline{NSTS}_g$), since they are sufficient to cover all concrete behaviour.

- **Discarded** is the number of states that were marked as iso-subsumed (see **Subsumed**) and were removed from the set of states to be explored. This number corresponds to the sum of all states removed from $\mathcal{F}$ at line 6 of Listing 7.9.

7.6.2 Evaluation

When comparing the figures in Table 7.10 for BFS and DFS exploration, it is clear that DFS gives a much better performance. The DFS generated state count is smaller than the BFS count in all but one test (*circ-buf-0*), and as we move to grammars with larger state spaces the advantage increases greatly, until reaching two orders of magnitude for the *firewall-6-F* case. The reason for this performance difference between BFS and DFS is simple. Usually, the more a shape is transformed by subsequent rule applications the more abstract it becomes, until it reaches a fix-point, *i.e.*, further rule applications yield the same shape again. These more abstract shapes capture more concrete behaviour and thus can iso-subsume other shapes in the state space. As a rule-of-thumb, more abstract shapes are more likely to be part of the set of relevant states, and since they are only discovered after a succession of rule applications, these relevant states are deeper in the state space. Therefore, DFS is more likely to reach these states first. This fact can be seen from the numbers in the **Discarded** column: BFS generates a lot of states that are later discarded. This is wasted effort: in BFS a state is produced and added to sets $\mathcal{S}$ and $\mathcal{F}$ but it is very likely that later it is going to be removed from $\mathcal{F}$ (while remaining in $\mathcal{S}$). On the other hand, since DFS already found more abstract shapes, it is more probable that the search will immediately throw a new state away, without storing it on $\mathcal{S}$, since the new state will be subsumed by some other state already in $\mathcal{S}$.

Figure 7.11 gives a visual aid for the comparison between BFS and DFS, for the firewall grammar instances with a known total state count. The figure shows a stacked bar chart with the accumulated percentages of the number of states relative to the total state space size. From this chart we see that the percentage of states generated with DFS decreases as the start graph size increases. On the other hand, the percentage of states generated with BFS remains roughly the same, at around 40% the total (this can be seen from the interval sizes of the generated BFS bars in Figure 7.11).

Other metrics that must be analysed in a performance evaluation are running time and memory consumption. Results for these measurements are given in Table 7.12, for runs with and without iso-subsumption checks. The machine specification where the experiments were performed is the same as for the initial results given in Section 7.4. Blank entries indicate timed-out executions. From the numbers in Table 7.12 we see the performance improvement given by iso-subsumption: running times for the firewall grammar are two orders of magnitude smaller when iso-subsumption is used, and memory consumption is

Figure 7.11: Accumulated percentages of the number of states relative to the total state space size, for the firewall grammar instances with a known total state count.

Grammar	Time (s)				Memory (MB)			
	Subsump.	OFF	Subsump.	ON	Subsump.	OFF	Subsump.	ON
	BFS	DFS	BFS	DFS	BFS	DFS	BFS	DFS
linked-list	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
circ-buf-0	< 1	< 1	< 1	< 1	2	2	2	2
circ-buf-1	< 1	< 1	< 1	< 1	2	2	2	2
euler-0	61	47	2	2	57	57	12	11
euler-1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
firewall-2	2	2	< 1	< 1	6	6	4	2
firewall-3	177	157	5	2	110	110	49	12
firewall-4	4,448	3,824	16	3	432	432	136	25
firewall-5			347	10			1,054	85
firewall-6			1,679	16			2,001	143
firewall-6-F			3,732	55			14,277	556

Table 7.12: Running time and memory consumption for abstract state space exploration, with and without state subsumption checks.

also reduced. When comparing the running times for BFS and DFS, we see that both strategies have a similar performance when iso-subsumption is not used but when it is turned on, DFS is far more efficient than BFS, both in execution time and memory consumption. This performance figures are directly related to the number of states generated by each strategy: DFS produces far fewer states than BFS, which translates to a large performance gain. This can be confirmed visually with the chart in Figure 7.13, where the running times for the firewall grammar are plotted against start graph sizes. Clearly, DFS has a much more tamed growth (note that the time axis is in a logarithmic scale).

After identifying the best parameters for abstract exploration (DFS exploration with iso-subsumption), we stress-tested the tool using larger instances of the firewall grammar. The results for this test are presented in Table 7.14. The

Figure 7.13: Running time (with iso-subsumption on) versus start graph size for the firewall grammar.

Grammar	Generated	Relevant	Time (s)	Memory (MB)
start-2	50	8	< 1	2
start-3	232	20	2	12
start-4	427	30	3	25
start-5	1,201	60	10	85
start-6	1,783	80	16	143
start-7	4,028	140	72	456
start-8	5,319	175	238	698
start-9	9,938	280	1,504	1,461
start-10			> 3,600	

Table 7.14: Stress test for the firewall grammar, using DFS with iso-subsumption and abstraction parameters $i = \mathbf{n} = \mathbf{e} = 1$.

second and third columns correspond respectively to the number of explored states and the number of relevant states of the reduced transition system. The largest model that can be handled within a reasonable time limit is a firewall configuration with 9 locations. Execution for 10 locations was aborted after one hour.

7.7 Conclusion

In this chapter we discussed the main points of the neighbourhood abstraction implementation and we proposed a new method for state collapsing, based on the concept of shape subsumption. Experimental results show that iso-subsumption gives a significant reduction on the number of states that have to be explored, thus improving both the running time and memory consumption of the tool. Furthermore, the experiments also show that the choice of the exploration strategy has a heavy influence on performance, with DFS giving much better results.

The work presented in Sections 7.5 and 7.6 was originally published in

[ZR12], and correspond to the second version of the implementation (V2). We see the improvements in (V2) as an important achievement over the original implementation (V1). As any tool developer can tell, performance improvements in programs that deal with highly combinatorial problems such as state space exploration usually involve a painstaking cycle of refactorings, experimentation and fine-tuning. Our case was no different, where version (V1) had to be rewritten from scratch in order to accommodate the new concept of iso-subsumption. A further improvement over the code from (V1) is that rules with NACs (negative application conditions) are now also supported in (V2), which increases rule expressivity.

The current code base for (V2) amounts to approximately 8,500 lines of Java code, properly integrated into the GROOVE source tree. Although GUI integration in the Simulator is still lacking, the tool set now provides a new text-based tool, called ShapeGenerator. This is the abstract counterpart of the Generator tool used for concrete state space exploration.

There are many directions where the current research/tool can be extended. Aside from the usual points, such as additional experimentation with more test cases and comparison with other tools, the main future work item we consider is state collapsing under subsumption. The iso-subsumption relation $\sqsubseteq$ presented in Section 7.5 gives an important reduction in state space sizes, but this reduction would certainly increase if the subsumption relation $\sqsubseteq$ could be used. However, as discussed previously in this chapter, efficient implementation of subsumption comparison poses quite a challenge. In particular, this would require additional code refactoring, since we would no longer be able to re-use the isomorphism checking package from GROOVE.

While neighbourhood abstraction can be used for several different classes of problems, it should be clear from the results presented in this chapter that the method actually does not fare so well in practice: overall, the grammars used in the experiments of Section 7.6 can all be still considered toy models; and even for some of the larger instances, tool performance was unfortunately sub-par when considering other existing graph-based abstraction tools (*e.g.*, AUGUR and hiralysis). In part, we believe that some of this inefficiency lies in the abstraction method itself, in particular in the way the neighbourhood abstraction relation is defined over a radius i ; after some consideration, it was clear that such radius is too coarse, in the sense of grouping too many nodes. Usually, we are not interested in the whole neighbourhood of a node but instead we want to look at different radii for different types of edges. This insight was the motivation behind the development of the method of *pattern abstraction*, presented at the next chapter.

Pattern abstraction

This chapter presents the newly developed theory of *pattern abstraction* for the exploration of graph transformation systems with infinite state spaces. As the name suggests, the method is based on *patterns*, simple graphs describing structures that should be preserved by the abstraction. The novelty of the approach lies in the flexibility for tuning the abstraction according to the patterns of interest, represented via a *type graph*.

This text is an extended and updated version of the theory that was originally published in [RZ12a]. Proofs for the propositions and theorems are given in an accompanying technical report [RZ12b].

8.1 Introduction

The contents of this chapter rely on the concepts presented in Chapter 5 but are completely independent from the neighbourhood abstraction theory given in Chapter 6. Similarities in notation between the two theories are intentional, in order to indicate equivalent concepts.

At the concrete level we work with *pattern graphs*, layered structures that describe the composition of patterns. The abstraction of pattern graphs gives rise to *pattern shapes*, bounded structures forming a finite universe. An informal introduction to pattern shapes was given in Section 2.4.2.

In this chapter we define how pattern graphs and pattern shapes are constructed from simple graphs, and we show how the application of graph transformation rules can be lifted to these new structures. This results in an overapproximation of the original system behaviour and thus enables verification on the abstract level.

Before moving to the presentation of pattern graphs, some additional notation is necessary. A *path* in a graph G is a non-empty sequence of edges $\pi = e_1 \cdots e_k$ such that $\text{tgt}_G(e_i) = \text{src}_G(e_{i+1})$ for $1 \leq i < k$. For convenience, we write $\text{src}(\pi) = \text{src}_G(e_1)$ and $\text{tgt}(\pi) = \text{tgt}_G(e_k)$. Paths π_1, π_2 are *parallel* if $\text{src}(\pi_1) = \text{src}(\pi_2)$ and $\text{tgt}(\pi_1) = \text{tgt}(\pi_2)$. Furthermore, v is a *predecessor* of w in G , denoted $v \leq_G w$, if either $v = w$ or there is a path π with $\text{src}(\pi) = v$ and $\text{tgt}(\pi) = w$.

8.2 Pattern graphs

Pattern graphs are the cornerstone graph representation in this work. A pattern graph is a rather rich structure, representing simple graphs and morphisms

between them. We first give a general definition, but in this paper we restrict ourselves to pattern graphs that are *well-formed*, a condition that will be presented shortly.

Definition 8.1 (Pattern graph). A pattern graph P is a graph extended with a labelling function $\text{lab}_P : N_P \cup E_P \rightarrow \text{SGraph} \cup \text{SMorph}$, where

- $\text{lab}_P(N_P) \subseteq \text{SGraph}$;
- $\text{lab}_P(E_P) \subseteq \text{SMorph}$; and
- for all $d \in E_P$, $\text{lab}_P(d) : \text{lab}_P(\text{src}_P(d)) \rightarrow \text{lab}_P(\text{tgt}_P(d))$ is an injective, non-surjective simple graph morphism. $\triangleleft$

Thus, the labelling function lab_P maps nodes of P to simple graphs and edges of P to simple graph morphisms that associate the source and target graphs. Elements of N_P and E_P are called *pattern nodes* and *pattern edges*, respectively. For $p \in N_P$, $\text{lab}_P(p)$ is the *pattern* of p . As with simple graphs, we may write $\langle p, f, q \rangle$ as a short-hand for a pattern edge d with $\text{src}_P(d) = p$, $\text{lab}_P(d) = f$, and $\text{tgt}_P(d) = q$.

Note that the restriction to non-surjective simple graph morphisms means that the pattern of $\text{tgt}_P(d)$ is always strictly larger than that of $\text{src}_P(d)$, which in turn implies that a pattern graph is always a dag. In categorical terms, a pattern graph corresponds to a diagram in the category SGraph .

We layer the nodes of P according to the number of simple edges in their patterns. So, for $i \geq 0$, the set of pattern nodes N_P for layer i is defined as

$$N_P^i = \{p \in N_P \mid |E_G| = i, G = \text{lab}_P(p)\}$$

and the set of nodes for layers $\geq i$ is denoted as

$$N_P^{i+} = \bigcup_{j \geq i} N_P^j.$$

We write $\text{depth}(P)$ to indicate the maximum layer of P that is not empty, *i.e.*, $\text{depth}(P) = i \in \mathbb{N}$ such that $|N_P^i| \neq 0$ and for all $j > i$, $|N_P^j| = 0$.

Example 8.2. Figure 8.1 shows an example of a pattern graph. Pattern nodes are drawn with dashed lines and the associated patterns are shown inside the node. Pattern edges are depicted as arrows labelled with their corresponding simple graph morphisms, except that embeddings are omitted to avoid clutter. Layers are indicated on the right. Note that there is no distinction between simple edges with unary or binary labels for the purpose of layer assignment. From here on we simplify the figures by showing only the patterns with labelled simple nodes. $\triangleleft$

Let $d = \langle p, f, q \rangle \in E_P$ be a pattern edge and let $G = \text{lab}_P(p)$. The *image* of d is defined as $\text{img}_P(d) = H$, where

- $N_H = f(N_G)$;
- $E_H = f(E_G)$;
- $\text{src}_H = f \circ \text{src}_G$;
- $\text{tgt}_H = f \circ \text{tgt}_G$; and
- $\text{lab}_H \circ f = \text{lab}_G$.

It is easy to see that $H \subset \text{lab}_P(q)$. We say that every pattern edge d *covers* the sub-graph $\text{img}_P(d)$. Furthermore, we call a set of pattern edges $E' \subseteq E_P$ *jointly surjective* if

Figure 8.1: Example of a pattern graph that is not well-formed and not commuting.

- $\text{tgt}_P(d_1) = \text{tgt}_P(d_2) = p$ for all $d_1, d_2 \in E'$; and
- $\bigcup_{d \in E'} \text{img}_P(d) = \text{lab}_P(p)$.

As an equivalent term we say that the pattern edges of E' *together cover* $\text{lab}_P(p)$.

Definition 8.3 (Well-formed pattern graph). A pattern graph P is called well-formed if (with $G = \text{lab}_P(p)$)

1. for all $p \in N_P^0$, $|N_G| = 1$;
2. for all $p \in N_P^1$ and the unique $e \in E_G$, $N_G = \{\text{src}_G(e), \text{tgt}_G(e)\}$ and $p \triangleleft_P$ together cover N_G ; and
3. for all $p \in N_P^{2+}$, $p \triangleleft_P$ together cover G . $\triangleleft$

In words, the patterns of level-0 pattern nodes consist of a single node, the patterns of level-1 pattern nodes consist of a single edge and its end nodes, and the patterns on any other level are determined by the combined images of their predecessors. Another consequence of pattern morphisms being non-surjective is that, on well-formed pattern graphs, there are no pattern edges between nodes of the same layer.

Level-0 pattern nodes are sometimes called *node patterns*, level-1 pattern nodes are called *edge patterns*, and nodes in layers ≥ 2 are called *sub-graph patterns*. The universe of (well-formed) pattern graphs is denoted PGraph . Note that the pattern graph in Figure 8.1 is not well-formed: pattern nodes p_5 , p_{12} , and p_{17} are not sufficiently covered by the incoming morphisms.

Analogously to simple graphs, pattern graphs are related by *pattern graph morphisms*, which respect the graph structure and the corresponding “labelling”.

Definition 8.4 (Pattern graph morphism). A pattern graph morphism between pattern graphs $P, Q \in \text{PGraph}$ is a graph morphism $m : P \rightarrow Q$, where,

1. for all $p \in N_P$, there exists a simple graph isomorphism $\varphi_p : \text{lab}_P(p) \rightarrow \text{lab}_Q(m(p))$; and
2. for all $d = \langle p, f, q \rangle \in E_P$, $\varphi_q \circ f = f' \circ \varphi_p$, with $f' = \text{lab}_Q(m(d))$. $\triangleleft$

The definition above states that m maps pattern nodes with isomorphic patterns (condition 1) and that m is compatible with the simple graph morphisms of pattern edges modulo isomorphism (condition 2). It is simple to see from the definition that pattern graph morphisms are composable: they are graph morphisms, which are composable, and the additional conditions in Definition 8.4 are transitive.

We now identify two important sub-classes of pattern graphs, called *commuting* and *concrete*, which can be flattened to equivalent simple graphs. Note that every path $\pi = d_1 \cdots d_k$ in a pattern graph P generates a composite simple graph morphism $f_\pi = \text{lab}_P(d_k) \circ \cdots \circ \text{lab}_P(d_1)$.

Definition 8.5 (Commuting & concrete pattern graph). *Let P be a pattern graph.*

- P is called *commuting* if it satisfies the following properties:
 1. all parallel paths give rise to the same morphism;
 2. for all distinct $p_1, p_2 \in N_P^{2+}$, the sets of proper level-0 or level-1 predecessors $\{q_i \in N_P^0 \cup N_P^1 \mid q_i <_P p_i\}$, for $i = 1, 2$, are also distinct; and
 3. for all $q \in N_P^{2+}$ and $x \in \text{lab}_P(q)$ (x a node or edge), there is a predecessor $p <_P q$ such that each $d \in q <_P$ with $x \in \text{img}_P(d)$ is the final edge of some path π from p to q with $x \in \text{img}_P(\pi)$.
- P is called *concrete* if it satisfies the following properties:
 1. for all distinct $p, q \in N_P$, $\text{lab}_P(p) \neq \text{lab}_P(q)$; and
 2. for all $\langle p, f, q \rangle \in E_P$, $f = \text{emb}(\text{lab}_P(p), \text{lab}_P(q))$. $\triangleleft$

Commutativity condition 3 essentially implies that common simple nodes and edges in non-trivial patterns always stem from a common ancestor. The pattern graph in Figure 8.1 is not commuting: the pattern associated with pattern node p_{16} cannot be constructed from its predecessors since there is no common ancestor for simple node v_2 .

The first concrete pattern graph condition states that all patterns are distinct and the second condition guarantees that identities of simple graph elements are preserved along the morphisms of pattern graph edges. The following proposition states that we can essentially always treat commuting pattern graphs as concrete.

Proposition 8.6. *Let P be a pattern graph.*

1. If P is concrete, then P is commuting.
2. If P is commuting, there is a concrete pattern graph Q isomorphic to P . $\triangleleft$

Clause 1 follows easily from Definition 8.5 and Definition 8.3. Clause 2 can be shown to hold with a simple algorithm that constructs Q from P , where the identities of simple graph elements in the patterns of Q are changed to ensure that the conditions for a concrete pattern graph are satisfied.

For concrete pattern graphs P , we define the *flattening* of P as

$$\text{flat}(P) = \bigcup_{p \in N_P} \text{lab}_P(p) ,$$

which is a simple graph. Clause 2 of Proposition 8.6 can then be used to justify the application of *flat* to arbitrary commuting pattern graphs (where the outcome is well-defined modulo simple graph isomorphism).

Figure 8.2: Example of a pattern type graph.

8.2.1 Typing

So far we have not restricted the patterns occurring in a pattern graph, but in practice we will only use *typed* patterns.

Definition 8.7 (Pattern type graph). A pattern type graph T is a pattern graph that complies to the following additional conditions:

1. for all distinct $p, q \in N_T$, $\text{lab}_T(p) \not\approx \text{lab}_T(q)$; and
2. for all $p \in N_T^{2+}$, $|p \triangleleft_T| = 2$.

◁

Nodes and edges of a pattern type graph are called *type nodes* and *type edges*, respectively. Condition 1 of the definition above ensures that a pattern type graph T has no isomorphic patterns. Thus, distinct type nodes have distinct patterns (and similarly for type edges), which allows us to use the elements of T as the *types* for pattern graph elements. Another consequence of this first condition is that pattern type graphs are not necessarily commuting.

The second condition of Definition 8.7 forces sub-graph patterns to be composed of precisely two ancestors. Note that this imposes no real restrictions since any pattern graph T' that does not satisfies this condition can be transformed into a pattern type graph T simply by adding intermediate patterns.

Example 8.8. Figure 8.2 gives an example of a pattern type graph with patterns taken from the elements of the simple graph grammar in Figure 5.1.

◁

A pattern graph P is considered *typable* by a certain pattern type graph T if it has a morphism to $t : P \rightarrow T$ that has an additional property on the image $t(P) \subseteq T$; namely, the image should be *left-closed*.

Definition 8.9 (Left-closed subgraph). Let G be a graph. A left-closed subgraph of G is a graph $H \subseteq G$ such that $e \in v \triangleleft_G$ for $v \in N_H$ implies $e \in E_H$.

◁

Type morphisms are formally defined as follows.

Definition 8.10 (Pattern type morphism). Let P be a pattern graph and T a pattern type graph. A typing morphism of P into T is a pattern graph morphism $t : P \rightarrow T$ such that $t(P)$ is a left-closed sub-graph of T .

◁

Figure 8.3: Concrete pattern graph lifted from the simple graph of Figure 5.1 according to the pattern type graph of Figure 8.2.

The following property is an important consequence of typing morphisms: if a morphism $t : P \rightarrow T$ exists, then we can always reconstruct the labelling of P (up to isomorphism) from t and (the labelling of) T . This means that, if we know the typing morphism, we may ignore the labelling of P . This will be used to good effect later in constructions over typed pattern graphs.

Proposition 8.11 (Pattern reconstruction). *Let G be an arbitrary graph with a non-label-preserving morphism $m : G \rightarrow T$, such that $m(G)$ is a left-closed sub-graph of T . There is a unique labelling function $\text{lab} : G \rightarrow \text{SGraph} \cup \text{SMorph}$ such that G extended with lab is a pattern graph P and $m : P \rightarrow T$ is a typing morphism.* $\triangleleft$

It is easy to see that for a given pattern type graph T , any pattern graph P has at most one morphism to T . The following definition shows how simple graphs can be *lifted* to pattern graphs according to a pattern type graph T .

Definition 8.12 (Lifting). *Let $G \in \text{SGraph}$ be a simple graph and T be a pattern type graph. The lifting $\text{lift}_T(G)$ is a concrete pattern graph P with a pattern graph morphism $t : P \rightarrow T$, defined as follows:*

- For every $H \subseteq G$ such that there exists an isomorphism $\varphi_H : H \rightarrow \text{lab}_T(p')$ for some $p' \in N_T$, let p_H be a fresh node of P , and let $\text{lab}_P : p_H \mapsto H$ and $t : p_H \mapsto p'$.
- For every $p_H \in N_P$ and $d' \in t(p_H) \triangleleft_T$, define $F = \varphi_H^{-1}(\text{img}_T(d'))$; let $d_{F,H}$ be a fresh edge of P , and let $\text{src}_P : d_{F,H} \mapsto p_F$, $\text{tgt}_P : d_{F,H} \mapsto p_H$, $\text{lab}_P : d_{F,H} \mapsto \text{emb}(F, H)$, and $t : d_{F,H} \mapsto d'$. $\triangleleft$

We first state that this construction gives a useful result: the outcome is a well-formed concrete pattern graph P with a morphism t to T .

Proposition 8.13 (Lifting effect). *For any simple graph $G \in \text{SGraph}$ and pattern type graph T , $P = \text{lift}_T(G)$ is a concrete pattern graph with morphism $t : P \rightarrow T$ (constructed as above).* $\triangleleft$

Example 8.14. Figure 8.3 shows the concrete pattern graph resulting from the lifting of the simple graph in Figure 5.1 according to the pattern type graph of Figure 8.2. Note that the identities of simple graph elements are preserved

Figure 8.4: Example of a closing pattern. (a) The closing pattern for the largest pattern of the type graph in Figure 8.2. (b) The pre-closing pattern of (a).

by the lifting. The dotted rectangles indicate the pattern equivalence relation $\equiv$ (Definition 8.36) over nodes of the pattern graph. This relation is used in Section 8.3 to construct pattern shapes. $\triangleleft$

From the concepts presented up to this point, a pattern graph P can be considered “type compatible” with a pattern type graph T if there exists a pattern graph morphism $m : P \rightarrow T$. However, this is not sufficient to consider P typed by T ; for that we also require a closure condition with respect to T . Closedness requires that any sub-graph pattern that can be commutatively constructed from some ancestors is included in P . This concept is formally defined in the following.

For each node type $p' \in N_T^{2+}$, we define the *closing pattern* of p' as the concrete pattern graph lifted from the pattern of p' . Formally, for every $p' \in N_T^{2+}$,

$$\text{cp}_T(p') = \text{lift}_T(\text{lab}_T(p')) .$$

Clearly, any such closing pattern has a maximal node, namely the one labelled by $\text{lab}_T(p')$. We use $\text{cp}_T^\triangleleft(p')$ to denote the *pre-closing pattern*, without this maximal node. Closure, then, is the property that every occurrence of a pre-closing pattern is embedded in an occurrence of the full closing pattern (where “occurrence of” means “image of some morphism from”). In other words, the pre-closing pattern describes the pattern graph structure which, when found in a pattern graph P , implies the existence of the closing pattern.

Example 8.15. Figure 8.4(a) shows the closing pattern for the lowermost pattern type in the type graph of Figure 8.2. The corresponding pre-closing pattern is given in Figure 8.4(b). $\triangleleft$

Definition 8.16 (Closed pattern graph). Let T be a pattern type graph and $P \in \text{PGraph}$ be a pattern graph.

1. P is called T -typable if there exists a pattern graph morphism $t : P \rightarrow T$ (the typing morphism).
2. P is called closed with respect to type $p' \in N_T^{2+}$ if for all $m : \text{cp}_T^\triangleleft(p') \rightarrow P$ there exists $m' : \text{cp}_T(p') \rightarrow P$, where m' is an extension of m .
3. P is called closed under T , or T -closed, if it is closed with respect to all $p' \in N_T^{2+}$.
4. P is called a T -pattern graph if it is T -typable and T -closed. $\triangleleft$

If a pattern graph P is T -typable but not T -closed, it is always possible to extend it to a T -pattern graph Q , namely by repeatedly adding the elements required by the closure condition (see Definition 8.22). (This terminates because, although one closure may generate a new occurrence of a pre-closing pattern, the average size of the remaining unclosed patterns always increases during this process, and this can never exceed the size of the largest closing pattern of the type graph.) We will return to this operation in the next subsection.

Returning to the lifting operation in Definition 8.12, it can be seen that the concrete pattern graph $P = \text{lift}_T(G)$ is always closed under T . Moreover, for concrete T -pattern graphs, lifting is left-inverse to flattening (modulo isomorphism).

Proposition 8.17. *Let T be a pattern type graph.*

1. *For any simple graph G , $\text{lift}_T(G)$ is a concrete T -pattern graph.*
2. *For any concrete T -typable pattern graph P , $Q = \text{lift}_T(\text{flat}(P))$ is a unique (modulo isomorphism) concrete T -pattern graph with $Q \supseteq P$.*
3. *For any concrete T -pattern graph P , $\text{lift}_T(\text{flat}(P)) \simeq P$.* $\triangleleft$

Note that the inverse of clause 3 above does not hold for arbitrary graphs G : lifting and then flattening G may result in a proper sub-graph of G , if there are edge labels in G that do not occur in any pattern of T .

From here on we assume that a pattern type graph T is always (implicitly) given. Typing of all pattern graphs is always taken with respect to this T , unless explicitly stated otherwise.

8.2.2 Manipulating a pattern graph

Having established a strong correspondence between simple graphs and concrete T -pattern graphs, we will proceed with defining the transformation of the latter. We will break down the application of a rule to a pattern graph into a number of steps:

$$P_0 \xrightarrow{\text{del}} \dots \xrightarrow{\text{del}} P_i \xrightarrow{\text{add}} \dots \xrightarrow{\text{add}} P_j \xrightarrow{\text{close}} \dots \xrightarrow{\text{close}} P_k .$$

Here, each **del** corresponds to the removal of a single node or edge, each **add** to the addition of the same, and **close** stands for a closure step. P_0 – P_i are concrete T -pattern graphs, whereas P_i – P_j are T -typable but may fail to be T -closed; closedness is reinstated during steps P_j – P_k .

We will now define the above operations formally. In all the constructions, we take advantage of Proposition 8.11 by omitting the explicit definition of the patterns for new nodes; instead we rely on the existence of a (typing) morphism.

Operation del

Since patterns represent both nodes and edges of simple graphs, erasing a pattern G from a concrete pattern graph P amounts to removing the pattern node p associated with G . However, given that G may be embedded into larger patterns, the removal of p triggers a “domino” effect that also deletes all successors of p in P .

Definition 8.18 (Pattern deletion). Let $P \in \text{PGraph}$ be a T -pattern graph with typing morphism $t_P : P \rightarrow T$, and let $p \in N_P$.

$$\begin{aligned} \text{del}(P, p) &:= Q \text{ where} \\ N_Q &= N_P \setminus \{q \in N_P \mid p \leq_P q\} \\ E_Q &= \{d \in E_P \mid \text{tgt}_P(d) \in N_Q\} \\ \text{src}_Q &= \text{src}_P|_{E_Q} \\ \text{tgt}_Q &= \text{tgt}_P|_{E_Q} . \end{aligned}$$

Moreover, $t_Q := t_P|_Q$. $\triangleleft$

Thus, deletion removes all pattern nodes in P that have p as an ancestor, together with their incident edges. This can be thought of as a kind of “domino effect”: the deletion of a pattern cascades to all larger patterns. Note that, due to Proposition 8.11, we do not have to specify the labels of Q since they can be reconstructed from t_Q . The following proposition states that the operation is “well-behaved”.

Proposition 8.19 (Deletion effect). Let P be a T -pattern graph, and let $p \in N_P$. Then $Q = \text{del}(P, p)$ is a T -pattern graph; moreover, if P is commuting then so is Q . $\triangleleft$

Operation add

The creation of a new pattern in a concrete pattern graph P amounts to the addition of a new pattern node into layer 0 (a node pattern) or 1 (an edge pattern). In an effect that is dual to domino deletion, the creation of new edge patterns may trigger a cascading inclusion of larger patterns, in order to ensure that the resulting pattern graph is closed under the type graph. This cascading inclusion is treated separately by operation `close`.

In contrast to deletion, addition is indexed by patterns from the pattern type graph.

Definition 8.20 (Pattern addition). Let $P \in \text{PGraph}$ be a pattern graph with typing morphism t_P , and let $p' \in N_T^0$ be a level-0 type node.

$$\begin{aligned} \text{add}_{p'}(P) &:= Q \text{ where} \\ N_Q &= N_P \cup \{p\} \quad \text{with } p \notin N_P \\ E_Q &= E_P \\ \text{src}_Q &= \text{src}_P \\ \text{tgt}_Q &= \text{tgt}_P . \end{aligned}$$

Moreover, $t_Q := t_P \cup \{p \mapsto p'\}$.

Now, let $q' \in N_T^1$ be a level-1 type node with incoming edges $d'_1, d'_2 \in q' \triangleleft_T$, and let $p_i \in t_P^{-1}(\text{src}_T(d'_i))$, for $i = 1, 2$.

$$\begin{aligned} \text{add}_{d'_1, d'_2}(P, p_1, p_2) &:= Q \text{ where} \\ N_Q &= N_P \cup \{q\} && \text{with } q \notin N_P \\ E_Q &= E_P \cup \{d_1, d_2\} && \text{with } d_1, d_2 \notin E_P \\ \text{src}_Q &= \text{src}_P \cup \{d_1 \mapsto p_1, d_2 \mapsto p_2\} \\ \text{tgt}_Q &= \text{tgt}_P \cup \{d_1 \mapsto q, d_2 \mapsto q\} . \end{aligned}$$

Moreover, $t_Q := t_P \cup \{q \mapsto q', d_1 \mapsto d'_1, d_2 \mapsto d'_2\}$. $\triangleleft$

Thus, the creation of a node or edge pattern simply amounts to adding a fresh, appropriately typed node to P , as well as (in the case of edge creation) two incoming edges. Note that **add** operations are indexed by type nodes p' and type edges d'_1, d'_2 . Also note that (as in Definition 8.18), due to Proposition 8.11 we do not have to specify the labels of Q since they can be reconstructed from t_Q . The following proposition states that these operations are “well-behaved”.

Proposition 8.21 (Addition effect). *Let P be a pattern graph with typing morphism t_P , and let $p' \in N_T^0$ and $q' \in N_T^1$, with $d'_1, d'_2 \in q' \triangleleft_T$.*

1. $Q = \text{add}_{p'}(P)$ is a well-defined (modulo isomorphism) pattern graph with typing morphism t_Q ; moreover, if P is a T -pattern graph then so is Q , and if P is commuting then so is Q .
2. For arbitrary $p_i \in t_P^{-1}(\text{src}_T(d'_i))$ ($i = 1, 2$), $Q = \text{add}_{d'_1, d'_2}(P, p_1, p_2)$ is a well-defined (modulo isomorphism) pattern graph with typing morphism t_Q ; moreover, if P is commuting then so is Q . $\triangleleft$

Note that one important property that is *not* preserved by edge addition is that of Q being a T -pattern graph: after adding a level-1 node, the resulting pattern graph can easily fail to be closed under T . In order to close it, we have to add more patterns, on higher levels, as dictated by the type graph. The fact that this can be done was already stated informally after Definition 8.16, but in order to facilitate the proofs for pattern shapes (Section 8.3) we will spell out the construction in the next operation.

Operation close

The closure of sub-graph patterns can occur after the addition of node and edge patterns, and is done to ensure that the resulting concrete pattern graph is closed under T .

Definition 8.22 (Pattern closure). *Let $P \in \text{PGraph}$ be a pattern graph with typing morphism $t_P : P \rightarrow T$, and let $q' \in N_T^{2+}$ be a type node with incoming edges $d'_1, d'_2 \in q' \triangleleft_T$.*

Furthermore, let $m : \text{cp}_T^{\triangleleft}(q') \rightarrow P$ be a pattern graph morphism from the pre-closing pattern of q' to a left-closed sub-graph of P , such that there does not exist an m -extension $m' : \text{cp}_T(q') \rightarrow P$; and let $p_i = m(\text{src}_T(d'_i))$, for $i = 1, 2$.

$$\begin{aligned} \text{close}_{d'_1, d'_2}(P, p_1, p_2) &:= Q \text{ where} \\ N_Q &= N_P \cup \{q\} && \text{with } q \notin N_P \\ E_Q &= E_P \cup \{d_1, d_2\} && \text{with } d_1, d_2 \notin E_P \\ \text{src}_Q &= \text{src}_P \cup \{d_1 \mapsto p_1, d_2 \mapsto p_2\} \\ \text{tgt}_Q &= \text{tgt}_P \cup \{d_1 \mapsto q, d_2 \mapsto q\} . \end{aligned}$$

Moreover, $t_Q := t_P \cup \{q \mapsto q', d_1 \mapsto d'_1, d_2 \mapsto d'_2\}$. $\triangleleft$

It can be immediately seen that the construction of a pattern closure is identical to that of edge pattern addition. The difference lies in the fact that q' in the above definition is a level- 2^+ node, and that **close** is only defined at places

where P is not closed with respect to q' (see Definition 8.16). Again note that, as before, due to Proposition 8.11 we do not have to specify the labels of Q since they can be reconstructed from t_Q .

Proposition 8.23 (Closure effect). *Let P be a pattern graph with typing morphism $t_P : P \rightarrow T$, and let $d'_i \in E_T$, $p_i \in N_P$ (for $i = 1, 2$) be such that $Q = \text{close}_{d'_1, d'_2}(P, p_1, p_2)$ is defined. Then Q is a well-defined (modulo isomorphism) pattern graph with typing morphism $t_Q : Q \rightarrow T$; moreover, if P is commuting then so is Q . $\triangleleft$*

Putting it together

Let us return to the notations informally introduced at the start of this subsection.

- $P \xrightarrow{\text{del}} Q$ denotes that $Q = \text{del}(P, p)$ for some $p \in N_P$.
- $P \xrightarrow{\text{add}} Q$ denotes that either $Q = \text{add}_{p'}(P)$ for some $p' \in N_T$, or $Q = \text{add}_{d'_1, d'_2}(P, p_1, p_2)$ for some $p_1, p_2 \in N_P$ and $d'_1, d'_2 \in E_T$.
- $P \xrightarrow{\text{close}} Q$ denotes that $Q = \text{close}_{d'_1, d'_2}(P, p_1, p_2)$ for some $p_1, p_2 \in N_P$ and $d'_1, d'_2 \in E_T$.

In addition to the “arrow” notations above, we also write $\xrightarrow{\text{op}^*}$ to denote a sequence of op operations. Furthermore, we use $\xrightarrow{\text{op}^*}$ to represent a *maximal* sequence, i.e.,

$$\xrightarrow{\text{op}^*} = \xrightarrow{\text{op}^*} \text{op}$$

The following properties are a consequence of Proposition 8.19 through Proposition 8.23.

Proposition 8.24. *Let P be a T -typable pattern graph.*

1. *For any sequence $P \xrightarrow{\text{del}^*} Q$, Q is T -typable; moreover, if P is T -closed then so is Q , and if P is commuting then so is Q .*
2. *For any sequence $P \xrightarrow{\text{add}^*} Q$, Q is a T -typable; moreover, if P is commuting then so is Q .*
3. *There exists a unique (modulo isomorphism) T -pattern graph Q with a maximal sequence $P \xrightarrow{\text{close}^*} Q$; moreover, if P is commuting then so is Q . $\triangleleft$*

We write $\text{close}_T(P)$ for the unique T -pattern graph Q such that $P \xrightarrow{\text{close}^*} Q$ whose existence is stated in Clause 3.

8.2.3 Pattern graph transformation

In this section we first define pattern graph transformation in a way that is analogous to ordinary graph transformation (Definition 5.8). Subsequently, we present an alternative characterisation based on a sequence of the operations introduced above. This characterisation is much closer to a practical implementation and will form the basis for the correctness proofs.

Concrete T -pattern graphs are transformed by lifting simple graph rules to pattern graph equivalents. We only consider simple rules whose left-hand sides are patterns in T , i.e., for any simple graph rule $r = \langle L, R \rangle$, $L \simeq \text{lab}_T(p)$ for

Figure 8.5: Two pattern graph transformation rules. (a) Rule equivalent to the `get` rule of Figure 5.1 and disjoint with pattern graph of Figure 8.6. (b) Equivalent `put` rule.

some $p \in N_T$. In this sense, one can say that a set of simple graph rules $\mathcal{R}$ constrains the choice of a pattern type graph T , or, equivalently, that T is extracted from set $\mathcal{R}$.

Definition 8.25 (Pattern graph transformation rule). A T -pattern graph rule $r = \langle \lambda, \rho \rangle$ consists of two concrete T -pattern graphs λ (the LHS) and ρ (the RHS), where for any $x_1 \in \lambda$ and $x_2 \in \rho$, $x_1 = x_2$ if and only if $\text{lab}_\lambda(x_1) = \text{lab}_\rho(x_2)$. $\triangleleft$

As done previously for simple graphs, the relation between λ and ρ is established by their common elements, with sets of pattern graph elements deleted (N^{del} , E^{del}) and created (N^{new} , E^{new}) by pattern graph rules defined similarly. In fact, due to the T -typability of λ and ρ , we have $E^{\text{del}} = \{d \in p \triangleright_\lambda \mid p \in N^{\text{del}}\}$ and $E^{\text{new}} = \{d \in p \triangleright_\rho \mid p \in N^{\text{new}}\}$. We use PRule to denote the universe of pattern graph transformation rules, with two of them shown in Figure 8.5.

We extend lifting and flattening operations to rules by defining $\text{lift}_T(r) = \langle \text{lift}_T(L), \text{lift}_T(R) \rangle$ for $r = \langle L, R \rangle$; and $\text{flat}(r') = \langle \text{flat}(\lambda), \text{flat}(\rho) \rangle$ for $r' = \langle \lambda, \rho \rangle$. Note that since, by assumption, L is a pattern in T , we have $\text{flat}(\text{lift}_T(L)) = L$ (Clause 3 of Proposition 8.17).

As with simple graph transformations, the application of a pattern graph transformation rule $r = \langle \lambda, \rho \rangle$ into a pattern graph P starts with a match $\mu : (\lambda \cup \rho) \rightarrow (P \uplus N^{\text{new}} \uplus E^{\text{new}})$ of r into P (where again $N^{\text{new}} \cup E^{\text{new}}$ are assumed to be fresh w.r.t. P). As before, such μ can be constructed by first finding an *injective* pattern graph morphism $\mu : \lambda \rightarrow P$ and then extending it with $\text{id}_{N^{\text{new}}} \cup \text{id}_{E^{\text{new}}}$.

Proposition 8.26. Let P be a commuting T -pattern graph and $r = \langle \lambda, \rho \rangle$ be

a T -pattern graph rule. There is a one-to-one correspondence between pattern graph matches $\mu : \lambda \rightarrow P$ and simple graph matches $m : \text{flat}(\lambda) \rightarrow \text{flat}(P)$. $\triangleleft$

Essentially, what happens when a rule is applied is that graph elements are removed and others are added. In a concrete pattern graph, each simple graph element x is represented by a pattern node in N_P^0 (if x is a node) or N_P^1 (if x is an edge), and it also contributes to all successor patterns; so when x is removed, all those pattern nodes disappear. Conversely, adding simple graph elements to a pattern graph means adding new pattern nodes to N_P^0 and N_P^1 , and then closing the resulting structure with respect to T .

Definition 8.27 (Pattern graph transformation). Let P be a commuting T -pattern graph with typing morphism $t_P : P \rightarrow T$, $r = \langle \lambda, \rho \rangle$ be a T -pattern graph rule, and $\mu : \lambda \rightarrow P$ be a match of r into P . The transformation of P by rule r according to μ is given by $Q' = \text{close}_T(Q)$, where Q is defined by

$$\begin{aligned} N_Q &= (N_P \setminus N') \cup N^{\text{new}} & \text{with } N' &= \{q \in N_P \mid p \in \mu(N^{\text{del}}), p \leq_P q\} \\ E_Q &= (E_P \setminus E') \cup E^{\text{new}} & \text{with } E' &= \{d \in p \triangleright_P \mid p \in N'\} \\ \text{src}_Q &= (\text{src}_P \cup (\mu \circ \text{src}_\rho))|_{E_Q} \\ \text{tgt}_Q &= (\text{tgt}_P \cup (\mu \circ \text{tgt}_\rho))|_{E_Q}, \end{aligned}$$

with a typing morphism $t_Q := (t_P \cup \mu \circ t_\rho)|_Q$. $\triangleleft$

So, a pattern graph transformation is similar to a simple graph transformation, except for the cascading deletion and addition. In the following we break up the transformation into small steps, corresponding to applications of the pattern graph operators defined in Section 8.2.2.

We write $P \xrightarrow{r, \mu} Q$ to denote that Q is the transformation of P by rule r , under match μ , or simply $P \xrightarrow{r} Q$ if there exists a match μ such that $P \xrightarrow{r, \mu} Q$.

The following proposition states that we can indeed (as announced at the start of Section 8.2.2) decompose a pattern graph transformation into a sequence of small steps.

Proposition 8.28. Let P and Q be T -pattern graphs. Every pattern graph transformation $P \xrightarrow{r, \mu} Q$ can be decomposed into a sequence of operations

$$P = P_0 \xrightarrow{\text{del}_1} \dots \xrightarrow{\text{del}_i} P_i \xrightarrow{\text{add}_{i+1}} \dots \xrightarrow{\text{add}_j} P_j \xrightarrow{\text{close}_{j+1}} \dots \xrightarrow{\text{close}_k} P_k = Q$$

where $P_0 - P_i$ are T -pattern graphs and $P_i - P_k$ are T -typable pattern graphs. $\triangleleft$

Example 8.29. Figure 8.6 shows an example of a pattern graph transformation. The pattern graph rule **get** of Figure 8.5 is matched to the concrete pattern graph in Figure 8.6(a), with the match shown in bold. Rule application removes pattern nodes p_2, p_5, p_6 and p_8 , and their incidents edges. The resulting pattern graph is given in Figure 8.6(b), where pattern node p_9 was created by the application. This new node p_9 can compose a larger pattern with p_7 and thus the closure operation adds node p_{10} . $\triangleleft$

We come now to the first major result of this paper: simple and pattern graph transformations are equivalent.

Figure 8.6: Example of a pattern graph transformation. (a) Pattern graph to be transformed, with match of rule `get` shown in bold. (b) Resulting pattern graph, with elements added by the closure operation shown in bold.

Theorem 8.30 (Transformation equivalence). *Let $G \in \text{SGraph}$ be a simple graph, $r \in \text{SRule}$ be a simple graph rule, and $r' = \text{lift}_T(r)$ an equivalent T -pattern graph rule.*

1. *If $G \xrightarrow{\tau} H$ is a simple graph transformation then there is a pattern graph transformation $\text{lift}_T(G) \xRightarrow{r'} Q$ with $Q \simeq \text{lift}_T(H)$.*
2. *If $\text{lift}_T(G) \xRightarrow{r'} Q$ is a pattern graph transformation then there is a simple graph transformation $G \xrightarrow{\tau} H$ with $Q \simeq \text{lift}_T(H)$.* $\triangleleft$

The intuition supporting this result is simple: concrete pattern graphs are nothing more than “blown-up” simple graphs, *i.e.*, each concrete pattern graph P has a unique (modulo isomorphism) corresponding simple graph that is isomorphic to $\text{flat}(P)$. The equivalence between simple graph and pattern graph transformations is used to show the equivalence between simple graph transition systems (SGTS, Definition 5.11) and pattern graph transition systems (PGTS, defined below).

Analogously to what was done to simple graph transformation rules, pattern graph rules can be grouped to form a *pattern graph grammar*.

Definition 8.31 (Pattern graph grammar). *A pattern graph grammar $\mathcal{P}_T = \langle \mathcal{R}_T, P_0 \rangle$ is formed by a set $\mathcal{R}_T$ of T -pattern graph rules and an initial T -pattern graph P_0 .* $\triangleleft$

We say that a simple graph grammar $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$ and a pattern graph grammar $\mathcal{P}_T = \langle \mathcal{R}'_T, P_0 \rangle$ are *equivalent* if for any simple graph rule $r \in \mathcal{R}$ there exists a T -pattern graph rule $r' \in \mathcal{R}'_T$ such that $r' = \text{lift}_T(r)$ and $r = \text{flat}(r')$; and, furthermore, $P_0 = \text{lift}_T(G_0)$.

In the same vein as was done with simple graph grammars, a pattern graph grammar *generates* a pattern graph transition system.

Definition 8.32 (PGTS). *A pattern graph transition system is a tuple $\text{PGTS} = \langle \mathcal{S}, \Rightarrow, \iota \rangle$ where*

- $\mathcal{S} \subseteq \text{PGraph}$ is a set of states;
- $\Rightarrow \subseteq \mathcal{S} \times \text{PRule} \times \mathcal{S}$ is a set of transitions; and
- $\iota \in \mathcal{S}$ is the initial state.

A pattern graph grammar $\mathcal{P}_T = \langle \mathcal{R}_T, P_0 \rangle$ generates a $\text{PGTS}_{\mathcal{P}}$ if $\mathcal{S}$ is a minimal set of pattern graphs such that

- $\iota = P_0$; and
- if $P \xrightarrow{r} Q$ for some $P \in \mathcal{S}$ and $r \in \mathcal{R}_T$, then there exists $Q' \in \mathcal{S}$ where $Q \simeq Q'$ and $\langle P, r, Q' \rangle \in \Rightarrow$. $\triangleleft$

We conclude this section with our second major result, which establishes the relation between a SGTS and a PGTS generated by equivalent grammars.

Theorem 8.33. *Let T be a pattern type graph, $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$ a simple graph grammar and $\mathcal{P}_T = \langle \mathcal{R}'_T, P_0 \rangle$ pattern graph grammar equivalent to $\mathcal{G}$, and take $\text{SGTS}_{\mathcal{G}} = \langle \mathcal{S}_{\mathcal{G}}, \rightarrow, G_0 \rangle$ as the SGTS generated by $\mathcal{G}$ and $\text{PGTS}_{\mathcal{P}} = \langle \mathcal{S}_{\mathcal{P}}, \Rightarrow, P_0 \rangle$ as the PGTS generated by $\mathcal{P}_T$. The two transition systems are bi-similar. $\triangleleft$*

Bi-similarity is a quite interesting result because it implies that satisfaction of μ -calculus formulæ (and thus also CTL*, CTL, and LTL formulæ) are preserved among the two systems (see Section 5.4). This in turn means that we can discard the SGTS and perform verification (model-checking) on the pattern graph level. However, a PGTS may still be infinite, effectively preventing its construction. In the next section we present an abstraction for pattern graphs.

8.3 Pattern shapes

The key reason for introducing pattern graphs is that their structure can be used to define an abstraction based on the patterns of interest (given in the pattern type graph T). Pattern graphs are abstracted into pattern shapes. As usual with structural abstraction, equivalent structures (patterns) are collapsed into an abstract representative, while keeping an approximate count of the number of concrete elements collapsed. Approximated counting is done using multiplicities, which were presented in Section 5.3.

8.3.1 Pattern shapes

Pattern shapes are the structures used in our abstract domain. As with pattern graphs, there are several variants of pattern shapes.

Definition 8.34 (Pattern shape). *A pattern shape is a tuple $S = \langle P_S, \text{mult}_S^n, \text{mult}_S^e \rangle$, where*

- P_S is a pattern graph with typing morphism $t : P_S \rightarrow T$;
- $\text{mult}_S^n : N_S \rightarrow \mathcal{M}^n$ is a node multiplicity function; and
- $\text{mult}_S^e : E_S \rightarrow \mathcal{M}^e$ is an edge multiplicity function.

Moreover, for all $p \in N_S$ and all $d' \in t(p) \triangleleft_T$, it is required that

$$\sum_{d \in C} (\text{mult}_S^n(\text{src}_S(d)) * \text{mult}_S^e(d)) \sqsubseteq \text{mult}_S^n(p)$$

where $C = \{d \in p \triangleleft_S \mid t(d) = d'\}$. $\triangleleft$

Function mult_S^n indicates how many concrete patterns were folded into an abstract pattern node, up to bound n . Function mult_S^e , on the other hand, counts locally, i.e., it indicates how many edges of a certain type *each* of the concrete

nodes had, up to bound $\mathbf{e}$. The side condition relating node and edge multiplicities effectively constrains (uniquely defines) the multiplicities for nodes in layers 1^+ . For instance, this in turn means that we can infer the multiplicities for nodes in layers 1^+ based on the edge multiplicities and on the multiplicities for the nodes in layer 0.

We write $\mathbf{PShape}$ to denote the universe of pattern shapes and we consider $\mathbf{mult}_S = \mathbf{mult}_S^n \cup \mathbf{mult}_S^e$.

Pattern graphs can be *trivially extended* to pattern shapes by associating multiplicity maps according to the kind of pattern graph. For a pattern type graph T we associate the most abstract multiplicity to all elements of T , *i.e.*, $\mathbf{mult}_T(x) \mapsto \mathbf{0}^+$, for all $x \in T$. For any other pattern graph P , its trivial extension is obtained by making $\mathbf{mult}_P(x) \mapsto \mathbf{1}$, for all $x \in P$.

From here on, we consider that trivial extensions of pattern graphs are taken when necessary. The distinct choice for multiplicities in pattern type graphs is motivated by the definition below.

Definition 8.35 ($\preceq$ -morphism). *Given a transitive relation $\preceq$ on multiplicities, a $\preceq$ -morphism between pattern shapes $X, Y \in \mathbf{PShape}$ is a pattern graph morphism $m : P_X \rightarrow P_Y$ that satisfies the following additional conditions,*

1. *for all $p' \in N_Y$,*

$$\sum_{p \in m^{-1}(p')} \mathbf{mult}_X^n(p) \preceq \mathbf{mult}_Y^n(p'); \text{ and}$$

2. *for all $d' \in E_Y$ and all $p \in N_X$,*

$$\sum_{d \in C} \mathbf{mult}_X^e(d) \preceq \mathbf{mult}_Y^e(d')$$

where $C = m^{-1}(d') \cap p \triangleright_X$.

A $\preceq$ -morphism is called a pattern shape morphism, and a $\sqsubseteq$ -morphism is called a subsumption morphism. $\triangleleft$

Since relation $\sqsubseteq$ is a refinement of $\preceq$, based on the definition above we see that all subsumption morphisms are also pattern shape morphisms. We write $m : X \rightarrow Y$ to indicate that m is a $\preceq$ -morphism from X to Y . It follows from Definition 8.35 that $\preceq$ -morphisms are composable. This is a direct consequence of $\preceq$ being transitive and the fact that pattern graph morphisms are also composable.

As done with pattern graphs, we can identify pattern shapes that are typed according to a given pattern type graph T . Shape S is called a *T-pattern shape* if P_S is *T-pattern graph*.

8.3.2 Canonical pattern shapes

Some more notation is necessary to define the notion of *pattern equivalence*, used to construct *canonical pattern shapes*. Let A be a set and $\equiv \subseteq A \times A$ be an equivalence relation over A . For $x \in A$, we write $[x]_\equiv$ to denote the equivalence class of x induced by $\equiv$, *i.e.*, $[x]_\equiv = \{y \in A \mid y \equiv x\}$ and we write $A/\equiv$ to denote the set of equivalence classes in A , *i.e.*, $A/\equiv = \{[x]_\equiv \mid x \in A\}$. For any $C_1, C_2 \subseteq A$, $C_1 \equiv C_2$ if for all $x_1 \in C_1$ there exists $x_2 \in C_2$ such that $x_1 \equiv x_2$, and vice versa.

Definition 8.36 (Pattern equivalence). Let S be a T -pattern shape and $t : S \rightarrow T$ be the typing morphism. The pattern equivalence $\equiv$ is defined as the smallest symmetrical relation over $N_S \times N_S$ and $E_S \times E_S$, where

- for any $p_1, p_2 \in N_S$, $p_1 \equiv p_2$ if
 1. $t(p_1) = t(p_2)$; and
 2. for all $C_1 \in (p_1 \triangleright_S) / \equiv$, there exists $C_2 \in (p_2 \triangleright_S) / \equiv$ such that $C_1 \equiv C_2$ and

$$\sum_{d_1 \in C_1} \text{mult}_S^e(d_1) = \sum_{d_2 \in C_2} \text{mult}_S^e(d_2);$$
- for any $d_1, d_2 \in E_S$, $d_1 \equiv d_2$ if
 3. $t(d_1) = t(d_2)$; and
 4. $\text{tgt}_S(d_1) \equiv \text{tgt}_S(d_2)$. $\triangleleft$

The definition above implies that only pattern nodes of the same type can be equivalent (condition 1), and that equivalent pattern nodes have the same number of outgoing edges of each type into the same classes (condition 2). Also, note that condition 2 is vacuously true for nodes in layer $\text{depth}(S)$, which gives a base case for the inductive definition. Given $\equiv$ we can derive a finer equivalence relation $\trianglelefteq$ that groups edges per source equivalence classes: $d_1 \trianglelefteq d_2$ if $d_1 \equiv d_2$ and $\text{src}_S(d_1) \equiv \text{src}_S(d_2)$. Using relations $\equiv$ and $\trianglelefteq$ we construct canonical pattern shapes.

Definition 8.37 (Canonical pattern shape). Let X be a T -pattern shape and let $t : X \rightarrow T$ be the typing morphism. The canonical pattern shape of X w.r.t. equivalence relation $\equiv$ is the T -pattern shape Y , where $N_Y = N_X / \equiv$ and $E_Y = E_X / \trianglelefteq$, and for all $[p]_{\equiv} \in N_Y$, $p \in N_X$, $[d]_{\trianglelefteq} \in E_Y$ and $d \in E_X$:

- $\text{src}_Y : [d]_{\trianglelefteq} \mapsto [\text{src}_X(d)]_{\equiv}$;
- $\text{tgt}_Y : [d]_{\trianglelefteq} \mapsto [\text{tgt}_X(d)]_{\equiv}$;
- $\text{lab}_Y : [p]_{\equiv} \mapsto \text{lab}_T(t(p))$;
- $\text{lab}_Y : [d]_{\trianglelefteq} \mapsto \text{lab}_T(t(d))$;
- $\text{mult}_Y^n : [p]_{\equiv} \mapsto \sum_{p' \in [p]_{\equiv}} \text{mult}_X^n(p')$; and
- $\text{mult}_Y^e : [d]_{\trianglelefteq} \mapsto \sum_{d' \in C} \text{mult}_X^e(d')$, where $C = (\text{src}_X(d) \triangleright_X) \cap [d]_{\trianglelefteq}$. $\triangleleft$

In words, nodes and edges of canonical pattern shape Y are the equivalence classes of elements of pattern shape X , the labelling of Y takes the type associated with each equivalence class of X , and the multiplicities of Y are the bounded sum of the multiplicities of elements in the equivalence classes of X .

Let P be a T -pattern graph and S be a T -pattern shape. We use $\text{abstract}(P)$ and $\text{normalise}(S)$ to denote the canonical pattern shape of P and S , respectively. Morphism $\alpha : P \rightarrow \text{abstract}(P)$ is called an *abstraction morphism* and morphism $\Omega : S \rightarrow \text{normalise}(S)$ is called a *normalisation morphism*. Both α and Ω are instances of subsumption morphisms. We use $\text{concr}(S)$ to denote the (possibly infinite) set of *concretisations* of S , i.e., $\text{concr}(S)$ is the set of all pattern graphs that have an abstraction morphism into S .

Example 8.38. Figure 8.7 shows the canonical pattern shape obtained when considering the pattern graph and equivalence relation of Figure 8.3. $\triangleleft$

Figure 8.7: Canonical pattern shape obtained when considering the pattern graph and equivalence relation of Figure 8.3. Pattern edge morphisms are not explicitly shown. Node multiplicities are given at the upper right corner of each pattern node. All edge multiplicities are $\mathbf{1}$ and are not shown.

We use $\text{CanPShape}_T^{\mathbf{n}, \mathbf{e}}$ to denote the universe of canonical shapes typable by T and bounded by $\mathbf{n}$ and $\mathbf{e}$. The following is our third major result, establishing the finiteness of the abstract state space.

Theorem 8.39. *Given a pattern type graph T and bounds $\mathbf{n}, \mathbf{e} \in \mathbb{N}$, the universe $\text{CanPShape}_T^{\mathbf{n}, \mathbf{e}}$ is finite (under isomorphism). $\triangleleft$*

Since the number of canonical shapes is limited, their transformation is guaranteed to yield a bounded state space. The first prerequisite for this transformation is the existence of a match from a pattern graph rule into the shape.

Definition 8.40 (Pre-match/match into a pattern shape). *Let $S \in \text{PShape}$ be a T -pattern shape and $r = \langle \lambda, \rho \rangle$ be a T -pattern graph transformation rule. A pre-match of r into S is a pattern shape morphism $\mu : \lambda \rightarrow S$. We call μ a match if, in addition, μ is injective, and for all $x \in \mu(\lambda)$, $\text{mult}_S(x) = \mathbf{1}$. $\triangleleft$*

Thus, a pre-match is a $\leq$ -morphism and therefore it ensures that the number of rule elements mapped into S satisfies the multiplicities maps. The additional condition on a match μ ensures that all elements of S in the image of μ are concrete.

Given a pre-match, we can “extract” a concrete copy of the rule LHS from the shape to obtain a match. This extraction is performed by operation **pull**.

Definition 8.41 (Pulling a match). *Let $X \in \text{CanPShape}_T^{\mathbf{n}, \mathbf{e}}$ be a canonical T -pattern shape with typing morphism $t_X : X \rightarrow T$, $r = \langle \lambda, \rho \rangle$ be a T -pattern graph rule disjoint from X and with typing morphism $t_\lambda : \lambda \rightarrow T$, and let $\mu : \lambda \rightarrow X$ be a pre-match.*

$\text{pull}(X, \mu) := \langle Y, \mu' \rangle$ where

$$N_Y = N_X \cup N_\lambda$$

$$E_Y = E_X \cup E_\lambda \cup E^\triangleright \text{ with}$$

$$E^\triangleright = \{ \langle p, d \rangle \mid p \in N_\lambda, d \in \mu(p) \triangleright_X, \text{mult}_X^{\mathbf{e}}(d) - |\mu^{-1}(d)| \neq \mathbf{0} \}$$

$$\text{src}_Y = \text{src}_X \cup \text{src}_\lambda \cup \{ \langle p, d \rangle \mapsto p \mid \langle p, d \rangle \in E^\triangleright \}$$

$$\text{tgt}_Y = \text{tgt}_X \cup \text{tgt}_\lambda \cup \{ \langle p, d \rangle \mapsto \text{tgt}_X(d) \mid \langle p, d \rangle \in E^\triangleright \}$$

$$\begin{aligned} \text{mult}_Y^n(p) &= \begin{cases} 1 & \text{if } p \in N_\lambda \\ \text{mult}_X^n(p) - |\mu^{-1}(p)| & \text{otherwise} \end{cases} \\ \text{mult}_Y^e(a) &= \begin{cases} 1 & \text{if } a \in E_\lambda \\ \text{mult}_X^e(d) - |\mu^{-1}(d)| & \text{if } a = \langle p, d \rangle \in E^\triangleright \\ \text{mult}_X^e(a) & \text{otherwise} \end{cases} \\ \mu' &= \text{id}_\lambda . \end{aligned}$$

Moreover, $t_Y := t_X \cup t_\lambda \cup \{\langle p, d \rangle \mapsto t_X(d) \mid \langle p, d \rangle \in E^\triangleright\}$. $\triangleleft$

In words, when constructing Y we keep the same structure of X and add a copy of the rule LHS λ . This copy is the sub-graph corresponding to the image of match μ' in Y , and this image is concrete. The multiplicities for the graph elements in the image of the pre-match μ are subtracted by the number of rule elements in the pre-image of μ . Furthermore, we add the set of extra edges $E^\triangleright$ to connect the match image to the rest of the shape.

Example 8.42. Figure 8.8 shows an example of a pulling operation. The canonical pattern shape X represents any arbitrary graph structure with two or more distinct sequences of three **C**-nodes connected by n-edges; this is the case, for instance, in a complete simple graph with four or more **C**-nodes.

The usual visual notation is used in Figure 8.8 to depict pattern shapes, with the exception that edge morphisms are not explicitly shown to avoid clutter. Instead, we use different line strokes to distinguish between edges with different types: a full line represents an edge that maps to the left part of the target pattern and a dotted line is used for an edge that maps to the right part.

The LHS λ stands for a single n-edge, with the mapping of pre-match μ indicated by dashed arrows. Although node and edge identities are not shown, λ and X are disjoint, due to the condition of Definition 8.41. The resulting shape Y and match μ' are shown at the bottom of Figure 8.8. $\triangleleft$

The following proposition characterises the effect of operation **pull**.

Proposition 8.43 (Effect of pulling). *Let $X \in \text{CanPShape}_T^{\text{n,e}}$ be a canonical T -pattern shape, $r = \langle \lambda, \rho \rangle$ be a T -pattern graph rule, and let $\mu : \lambda \rightarrow X$ be a pre-match. Then, for $\langle Y, \mu' \rangle = \text{pull}(X, \mu)$, Y is a T -pattern shape and $\mu' : \lambda \rightarrow Y$ is a match.* $\triangleleft$

We write $X \xrightarrow{\text{pull}} Y$ to denote that $\langle Y, \mu' \rangle = \text{pull}(X, \mu)$, for some pre-match μ . It is immediate to see that the resulting shape Y is no longer canonical. Nevertheless, the set of concretisations for X and Y remains the same.

Proposition 8.44 (Correctness of pulling). *Let X be a canonical T -pattern shape and let $X \xrightarrow{\text{pull}} Y$ be a pulling operation. Then $\text{concr}(X) = \text{concr}(Y)$.* $\triangleleft$

Unfortunately, after obtaining a match, the transformation of a pattern shape Y cannot be carried out directly, because the preconditions for operations **del**, **add** and **close** may not hold. For example, closure requires the image of the pre-closing pattern to be a left-closed subgraph of Y , a condition that

Figure 8.8: Example of a pull operation performed on a canonical pattern shape X with a pre-match $\mu : \lambda \rightarrow X$, resulting in the pattern shape Y and match $\mu' : \lambda \rightarrow Y$. Edge morphisms are not explicitly shown; different line strokes are used to indicate the edge type: a full line represents an edge with an image corresponding to the left part of the target pattern, whereas a dotted edge maps to the right part.

may be violated by some collector nodes of Y . Thus, in order to repair this problem, we are forced to take a detour through *quasi-shapes*, presented in the following section.

8.4 Quasi-shapes

Quasi-shapes are auxiliary structures used to represent the intermediate states of a pattern shape during its transformation by a rule application. As the name implies, a quasi-shape is not a pattern shape per se, but instead it formalises the *possibility* of a certain pattern shape configuration. In fact, a quasi-shape Q actually represents a *finite* set of possible pattern shapes, which are referred

to as the *materialisations* of $\mathcal{Q}$ (see Definition 8.66).

A quasi-shape has a pattern graph structure and a set of *multiplicity constraints*. In contrast to pattern shapes, where each node or edge has an associated multiplicity, nodes and edges of quasi-shapes have an associated *multiplicity variable*. The constraints of the quasi-shape then group these variables in equations that limit the possible *multiplicity values* that each variable can have.

We assume the existence of an infinite universe of multiplicity variables, denoted by $\mathbb{X}$. Furthermore, we also assume that every graph element is associated with a variable, *i.e.*, for any node p or edge d occurring in a quasi-shape, $x_p, x_d \in \mathbb{X}$ are their corresponding variables. We call these, node and edge variables, respectively.

Definition 8.45 (Quasi-shape). A quasi-shape is a tuple $\mathcal{Q} = \langle P_{\mathcal{Q}}, \mathcal{C}_{\mathcal{Q}} \rangle$, where

- $P_{\mathcal{Q}}$ is a pattern graph; and
- $\mathcal{C}_{\mathcal{Q}} \subseteq (\mathbb{X}^+ \times \mathcal{M})$ is a finite set of multiplicity constraints, where a constraint $\langle x_1, \dots, x_k, m \rangle \in \mathcal{C}_{\mathcal{Q}}$ stands for the equation $x_1 + \dots + x_k = m$. $\triangleleft$

Since quasi-shapes have a pattern graph structure, they can be related via pattern graph morphisms. So, as done with pattern shapes, we can consider quasi-shapes *typed* according to a type graph T .

Given that the order of terms is irrelevant in a sum, the same applies to variable ordering in a constraint. In fact, we usually abuse notation and treat a constraint $c \in \mathcal{C}_{\mathcal{Q}}$ as a set of variables, and we write $x \in c$ to indicate that x occurs in some position of tuple c . Similarly, we consider addition and removal of variables from a constraint as set operations.

Let $\mathcal{Q}$ be a quasi-shape, $c \in \mathcal{C}_{\mathcal{Q}}$ be a constraint, and $x \in c$ be a variable. Furthermore, let $\mathcal{D} \subset \mathbb{X}$ be a finite set of variables. The *substitution of x by $\mathcal{D}$ in c* is defined as

$$c[x := \mathcal{D}] = (c \setminus \{x\}) \cup \mathcal{D}.$$

It is important to note that while there always exists a variable for every element of a quasi-shape $\mathcal{Q}$, it is not always the case that such variable occurs in the constraint set $\mathcal{C}_{\mathcal{Q}}$. The dual of this mismatch is also possible, *i.e.*, there are situations where a variable occurs in a constraint but the associated graph element is no longer in $\mathcal{Q}$. This is the case, for instance, when a pattern edge is deleted from a quasi-shape. When progressing along this section, we try to highlight all occurrences of these idiosyncrasies, pointing, in particular, to the conditions that led to such cases, while arguing on why these mismatches are necessary.

We are interested only in *well-formed* quasi-shapes, a property that imposes restrictions on the constraint set $\mathcal{C}_{\mathcal{Q}}$, as discussed in the following.

Firstly, each constraint in $\mathcal{C}_{\mathcal{Q}}$ must have a single type of variable, *i.e.*, a constraint $c \in \mathcal{C}_{\mathcal{Q}}$ can only be (entirely) composed of node variables or edge variables. In this case, we can represent sub-sets of $\mathcal{C}_{\mathcal{Q}}$ as below.

- $\mathcal{C}_{\mathcal{Q}}^N \subseteq \mathcal{C}_{\mathcal{Q}}$ is the sub-set of *node constraints*, *i.e.*, for any $\langle x_1, \dots, x_k, m \rangle \in \mathcal{C}_{\mathcal{Q}}^N$ and $1 \leq i \leq k$, x_i is a node variable.

- $\mathcal{C}_Q^E \subseteq \mathcal{C}_Q$ is the sub-set of *edge constraints*, i.e., all variables in all constraints of $\mathcal{C}_Q^E$ are edge variables.
- $\mathcal{C}_Q^N(p) \subseteq \mathcal{C}_Q^N$ is the sub-set of node constraints where the variable associated with p occurs, i.e., for $p \in N_Q$, $\mathcal{C}_Q^N(p) = \{c \in \mathcal{C}_Q^N \mid x_p \in c\}$.
- $\mathcal{C}_Q^E(p) \subseteq \mathcal{C}_Q^E$ is the sub-set of edge constraints where the variables associated with edges outgoing from p occur, i.e., for $p \in N_Q$, $\mathcal{C}_Q^E(p) = \{c \in \mathcal{C}_Q^E \mid x_d \in c, d \in p \triangleright_Q\}$.

Secondly, we assume that node constraints only have variables from node patterns, i.e., for any node constraint $c \in \mathcal{C}_Q^N$ and for all node variables $x_p \in c$, it holds that $p \in N_Q^0$. This requirement is needed to allow some simplifications on the definitions for the operations on quasi-shapes. However, as discussed in Definition 8.34, node constraints are not strictly necessary for pattern nodes in layers 1^+ because the multiplicities of these nodes are implicitly constrained by the multiplicity of the incoming edges and source patterns.

Thirdly, every edge constraint must be composed of variables with associated edges that have the same source node. Thus, for any $\langle x_{d_1}, \dots, x_{d_k}, m \rangle \in \mathcal{C}_Q^E$ and $1 \leq i, j \leq k$, $\text{src}_Q(d_i) = \text{src}_Q(d_j)$.

We use QShape to denote the universe of (well-formed) quasi-shapes.

An *assignment* $\sigma : \mathbb{X} \rightarrow (\mathcal{M}^n \cup \mathcal{M}^e)$ is a function that maps variables to bounded multiplicities. We say that σ *solves* a constraint c if the values associated to the variables of c by σ give rise to a correct equation. In the same vein, σ *satisfies* a quasi-shape Q if all constraints of Q are solved by σ .

Definition 8.46 (Solving assignment). *Let $\sigma : \mathbb{X} \rightarrow (\mathcal{M}^n \cup \mathcal{M}^e)$ be an assignment and $Q \in \text{QShape}$ be a quasi-shape.*

- σ solves a constraint $\langle x_1, \dots, x_k, m \rangle \in \mathcal{C}_Q$ if

$$\sum_{i=1}^k \sigma(x_i) = m .$$

- σ satisfies Q if for all constraints $c \in \mathcal{C}_Q$, σ solves c . $\triangleleft$

As with pattern shapes, a quasi-shape also has a set of concretisations, formed by the pattern graphs that satisfy the constraints of the quasi-shape. This is based on the existence of *satisfying morphisms*, defined in the following.

Definition 8.47 (Satisfying morphism). *Let $P \in \text{PGraph}$ be a pattern graph, $Q \in \text{QShape}$ be a quasi-shape, and $m : P \rightarrow P_Q$ be a pattern graph morphism. We call m a satisfying morphism if the assignment σ_m satisfies Q , where $\sigma_m : \mathbb{X} \rightarrow (\mathcal{M}^n \cup \mathcal{M}^e)$ is defined as*

- for all $p \in N_Q$, $\sigma_m : x_p \mapsto \beta^n(\langle k, k \rangle)$, where $k = |m^{-1}(p)|$; and
- for all $d \in E_Q$, $\sigma_m : x_d \mapsto \beta^e(\langle k, k \rangle)$, where $k = |m^{-1}(d)|$. $\triangleleft$

We say that a pattern graph P *satisfies* Q if there exists a satisfying morphism from P into Q . Thus, the set of *concretisations* of Q , denoted $\text{concr}(Q)$ is composed of all pattern graphs $P \in \text{PGraph}$ that satisfy Q .

Any pattern shape $S \in \text{PShape}$ can be *trivially devolved* into a quasi-shape $Q \in \text{QShape}$, written as $Q = \text{devol}(S)$ or $S \xrightarrow{\text{devol}} Q$, by keeping the graph structure of S and creating singleton constraints for each graph element of S .

Formally, take $P_Q = P_S$ and define

$$\mathcal{C}_Q = \{\langle x_a, m \rangle \mid a \in (N_S^0 \cup E_S), m = \text{mult}_S(a)\} .$$

Note that the devolution of a shape into a quasi-shape does not modify the set of concretisations.

Proposition 8.48 (Devolution correctness). *Let $S \in \text{PShape}$ be a pattern shape and let $Q = \text{devol}(S)$ be the trivial devolution of S into a quasi-shape. Then $\text{concr}(S) = \text{concr}(Q)$.* $\triangleleft$

8.4.1 Manipulating quasi-shapes

As done with pattern graphs, the manipulation of quasi-shapes is broken down into a sequence of steps:

$$Q_0 \xrightarrow{\text{da}^*} Q_i \xrightarrow{\text{del}^*} Q_j \xrightarrow{\text{add}^*} Q_k \xrightarrow{\text{close}^*} Q_l \xrightarrow{\text{sp}^*} Q_m \xrightarrow{\text{br}} \{S_n\}_{n \in I} .$$

Here, *del*, *add* and *close* (called the *transformation* operations) are the counterparts for quasi-shapes of the homonymous pattern graph operations; the other steps are exclusive for quasi-shapes and are necessary to establish the pre- and post-conditions for the transformation operations.

Operation *da* is called *disambiguation*; it modifies the T -quasi-shape Q_0 in order to ensure that each pattern node p has only one incoming edge of each type, hence making the composition of p from ancestors *unambiguous* (see Definition 8.49). The maximal sequence of *da* operations yields an unambiguous quasi-shape Q_i , which is ready to be transformed by deletion, addition and closure.

Disambiguation eliminates “sharing” of incoming edges but introduces sharing of outgoing edges, expressed in the quasi-shape by non-singular edge constraints. Since this situation cannot be represented in pattern shapes, it prevents the evolution of a quasi-shape back to a shape. Thus, we first apply the *split* operation *sp*, which can be seen as the dual of *da*. The maximal sequence of *sp* operations produces a *deterministic* quasi-shape Q_m (see Definition 8.62), meaning that it no longer has shared outgoing edges.

Note, as mentioned in the beginning of this section, that a quasi-shape represents a *set* of shapes. Thus, in order to return to the realm of pattern shapes, we need to compute the materialisation set $\{S_n\}_{n \in I}$ (I being some index set) for quasi-shape Q_m , which is done by the *branching* operation *br*. This branching into several shapes should come as no surprise since the transformation of canonical pattern shapes is expected to be a non-deterministic operation.

In the following we formally define all quasi-shape operations. For each of them, we present two propositions: one describing the operation effect, *i.e.*, what it accomplishes; and another showing that the operation is correct, in the sense of being compatible with the notion that the abstraction is an over-approximation of the original pattern graph transformation.

Operation da

Let $\mathcal{Q}$ be a T -quasi-shape with typing morphism $t : \mathcal{Q} \rightarrow T$. A pattern node $p \in N_{\mathcal{Q}}$ is said to be *uniquely covered*, denoted $\text{uc}_{\mathcal{Q}}(p)$, if for all edge types $d' \in t(p) \triangleleft_T$, there exists a unique edge $d \in p \triangleleft_{\mathcal{Q}}$ such that $t(d) = d'$. In other words, if p is uniquely covered, then there exists precisely one edge of each type incoming into p . If this is not the case, an offending edge type d' is said to be *ambiguous* in p . Quasi-shape $\mathcal{Q}$ is called *unambiguous* if for all $p \in N_{\mathcal{Q}}$, $\text{uc}_{\mathcal{Q}}(p)$ holds.

Ambiguity is an unavoidable side-effect of canonicity; to ensure that the universe of canonical shapes is finite we need to employ a lossy abstraction that “forgets” some structural information. This loss of information corresponds to the ambiguous coverage of some collector pattern nodes in the canonical shape.

Definition 8.49 (Disambiguation). Let $\mathcal{X} \in \text{QShape}$ be a T -quasi-shape with typing morphism $t_{\mathcal{X}} : \mathcal{X} \rightarrow T$, and let $p \in N_{\mathcal{X}}$ be a pattern node such that $\text{uc}_{\mathcal{X}}(p)$ does not hold.

Furthermore, let $d' \in t_{\mathcal{X}}(p) \triangleleft_T$ be an edge type ambiguous in p and define $E^{\text{da}} = \{d \in p \triangleleft_{\mathcal{X}} \mid t(d) = d'\}$ as the set of edges to be disambiguated and $N^{\text{da}} = \{p_d \mid d \in E^{\text{da}}\}$ as the set of nodes created from p .

$\text{da}_{d'}(\mathcal{X}, p) := \mathcal{Y}$ where

$$\begin{aligned}
 N_{\mathcal{Y}} &= (N_{\mathcal{X}} \setminus \{p\}) \cup N^{\text{da}} \\
 E_{\mathcal{Y}} &= E_{\mathcal{X}} \setminus (p \triangleleft_{\mathcal{X}} \cup p \triangleright_{\mathcal{X}}) \cup \{E_d^{\triangleleft} \mid d \in p \triangleleft_{\mathcal{X}}\} \cup \{E_d^{\triangleright} \mid d \in p \triangleright_{\mathcal{X}}\} \text{ with} \\
 E_d^{\triangleleft} &= \begin{cases} \{\langle d, p_d \rangle\} & \text{if } d \in E^{\text{da}} \\ \{\langle d, q \rangle \mid q \in N^{\text{da}}\} & \text{otherwise} \end{cases} \\
 E_d^{\triangleright} &= \{\langle q, d \rangle \mid q \in N^{\text{da}}\} \\
 \text{src}_{\mathcal{Y}}(a) &= \begin{cases} \text{src}_{\mathcal{X}}(d) & \text{if } a = \langle d, q \rangle \in E_d^{\triangleleft} \\ q & \text{if } a = \langle q, d \rangle \in E_d^{\triangleright} \\ \text{src}_{\mathcal{X}}(a) & \text{otherwise} \end{cases} \\
 \text{tgt}_{\mathcal{Y}}(a) &= \begin{cases} q & \text{if } a = \langle d, q \rangle \in E_d^{\triangleleft} \\ \text{tgt}_{\mathcal{X}}(d) & \text{if } a = \langle q, d \rangle \in E_d^{\triangleright} \\ \text{tgt}_{\mathcal{X}}(a) & \text{otherwise} \end{cases} \\
 \mathcal{C}_{\mathcal{Y}}^N &= \mathcal{C}_{\mathcal{X}}^N \\
 \mathcal{C}_{\mathcal{Y}}^E &= \mathcal{C}_{\mathcal{X}}^E \setminus (\mathcal{C}^{\triangleleft} \cup \mathcal{C}^{\triangleright}) \cup \mathcal{D}^{\triangleleft} \cup \mathcal{D}^{\triangleright} \text{ with} \\
 \mathcal{C}^{\triangleleft} &= \{\mathcal{C}_d^{\triangleleft} \mid d \in p \triangleleft_{\mathcal{X}}\} \text{ with } \mathcal{C}_d^{\triangleleft} = \mathcal{C}_{\mathcal{X}}^E(\text{src}_{\mathcal{X}}(d)) \\
 \mathcal{C}^{\triangleright} &= \mathcal{C}_{\mathcal{X}}^E(p) \\
 \mathcal{D}^{\triangleleft} &= \{\mathcal{D}_d^{\triangleleft} \mid d \in p \triangleleft_{\mathcal{X}}\} \text{ with } \mathcal{D}_d^{\triangleleft} = \{c[x_d := \{x_a \mid a \in E_d^{\triangleleft}\}] \mid c \in \mathcal{C}_d^{\triangleleft}\} \\
 \mathcal{D}^{\triangleright} &= \{c[x_d := \{x_{\langle q, d \rangle}\}] \mid x_d \in c, c \in \mathcal{C}^{\triangleright}, q \in N^{\text{da}}\} .
 \end{aligned}$$

Moreover, for all $q \in N_{\mathcal{Y}}$,

$$t_{\mathcal{Y}}(q) = \begin{cases} t_{\mathcal{X}}(p) & \text{if } q \in N^{\text{da}} \\ t_{\mathcal{X}}(q) & \text{otherwise} \end{cases}$$

and for all $a \in E_{\mathcal{Y}}$,

$$t_{\mathcal{Y}}(a) = \begin{cases} t_{\mathcal{X}}(d) & \text{if } a = \langle d, q \rangle \in E_d^{\triangleleft} \\ t_{\mathcal{X}}(d) & \text{if } a = \langle q, d \rangle \in E_d^{\triangleright} \\ t_{\mathcal{X}}(a) & \text{otherwise} . \end{cases} \quad \triangleleft$$

So, when disambiguating node p according to edge type d' , we remove p from the quasi-shape and create as many new nodes (set N^{da}) as the number of ambiguous edges with type d' (set E^{da}). Each new node is associated with only one of the edges in E^{da} (first case of $E_d^{\triangleleft}$), thus making type d' unambiguous. The remaining incoming and outgoing edges of p are all duplicated (second case of $E_d^{\triangleleft}$ and sets $E_d^{\triangleright}$).

Since nodes in layer 0 are always unambiguous (they have no incoming edges), the set of node constraints remains the same in the definition above. As for the edge constraints, we duplicate the ones associated with the duplicated outgoing edges (set $\mathcal{D}^{\triangleright}$) and we *expand* the constraints of the incoming edges by replacing a single edge variable with a set (see $\mathcal{D}_d^{\triangleleft}$ above). This constraint expansion represents the introduction of shared outgoing edges.

Operation **da** is undefined if the quasi-shape $\mathcal{X}$ is unambiguous. The most effective way to perform **da** applications in $\mathcal{X}$ is in a top-down manner according to the layers. This is the case because each disambiguation of a node in layer i causes node splits in this layer, and may cause nodes in layers $> i$ to also become ambiguous. Note, however, that the ordering for these operations is irrelevant to the correctness of the maximal sequence of applications, because this said sequence is confluent (see Proposition 8.68).

Example 8.50. Figure 8.9 shows an example of a **da** operation. Quasi-shape $\mathcal{X}$ is a structure similar to the one resulting from operation **pull** in Figure 8.8. The same visual notations from that figure are also used here, with the addition of grey arcs to indicate shared outgoing edges in the resulting quasi-shape $\mathcal{Y}$. The quasi-shape constraints are presented below the shape structure.

In this example we take $p = p_3$ as the node to be disambiguated, with a dotted edge type d' . According to Definition 8.49, this gives rise to the sets

$$\begin{aligned} E^{\text{da}} &= \{d_3, d_4\} & N^{\text{da}} &= \{p_{d_3}, p_{d_4}\} \\ E_{d_1}^{\triangleleft} &= \{a = \langle d_1, p_{d_3} \rangle, b = \langle d_1, p_{d_4} \rangle\} & E_{d_2}^{\triangleleft} &= \{c = \langle d_2, p_{d_3} \rangle, d = \langle d_2, p_{d_4} \rangle\} \\ E_{d_3}^{\triangleleft} &= \{e = \langle d_3, p_{d_3} \rangle\} & E_{d_4}^{\triangleleft} &= \{f = \langle d_4, p_{d_4} \rangle\} \\ E_{d_5}^{\triangleright} &= \{g = \langle p_{d_3}, d_5 \rangle, i = \langle p_{d_4}, d_5 \rangle\} & E_{d_6}^{\triangleright} &= \{h = \langle p_{d_3}, d_6 \rangle, j = \langle p_{d_4}, d_6 \rangle\} . \end{aligned}$$

Further applications of **da** are necessary to disambiguate the new nodes p_{d_3}, p_{d_4} with respect to the other incoming edge type. $\triangleleft$

The following proposition establishes the effect of a disambiguation.

Proposition 8.51 (Disambiguation effect). *Let $\mathcal{X} \in \text{QShape}$ be a T -quasi-shape with typing morphism $t_{\mathcal{X}} : \mathcal{X} \rightarrow T$, and let $p \in N_{\mathcal{X}}$ and $d' \in t_{\mathcal{X}}(p) \triangleleft_T$ be such that $\mathcal{Y} = \text{da}_{d'}(\mathcal{X}, p)$ is defined. Then $\mathcal{Y}$ is a T -quasi-shape and $\text{da}_{d'}(\mathcal{Y}, p)$ is undefined.* $\triangleleft$

Figure 8.9: Example of a da operation performed on quasi-shape $\mathcal{X}$. The same visual notations are used to indicate edge types (see Figure 8.8). The quasi-shape constraints are shown below the shape structure. Shared outgoing edges in the resulting quasi-shape $\mathcal{Y}$ are marked with a grey arc.

The last statement of the proposition above indicates that after performing the operation, edge type d' is no longer ambiguous in p . This provides a termination criterion: after finitely many steps the operation da can no longer be applied because there is nothing left to disambiguate.

The correctness requirement for operation da is that the set of concretisations of the transformed quasi-shape remains the same.

Proposition 8.52 (Disambiguation correctness). *Let $\mathcal{X} \in \text{QShape}$ be a T -quasi-shape and let $\mathcal{Y} = \text{da}_{d'}(\mathcal{X}, p)$ for some $p \in N_{\mathcal{X}}$ and $d' \in t_{\mathcal{X}}(p) \triangleleft_T$. Then $\text{concr}(\mathcal{X}) = \text{concr}(\mathcal{Y})$.* $\triangleleft$

Operation del

In order to delete elements from a quasi-shape $\mathcal{X}$, operation del requires $\mathcal{X}$ to be unambiguous. This is a necessary condition to ensure that the domino deletion can be safely performed on the pattern graph structure of $\mathcal{X}$.

Definition 8.53 (Pattern deletion). *Let $\mathcal{X} \in \text{QShape}$ be an unambiguous T -quasi-shape and let $p \in N_{\mathcal{X}}$.*

$$\begin{aligned}
 \text{del}(\mathcal{X}, p) &:= \mathcal{Y} \text{ where} \\
 P_{\mathcal{Y}} &= \text{del}(P_{\mathcal{X}}, p) \\
 \mathcal{C}_{\mathcal{Y}} &= \mathcal{C}_{\mathcal{X}} \setminus \mathcal{C}_{\mathcal{X}}^N(p) .
 \end{aligned}
 \quad \triangleleft$$

In words, the new quasi-shape $\mathcal{Y}$ is obtained by deleting p from the pattern graph underlying $\mathcal{X}$ and adjusting the constraints. Node constraints in $\mathcal{Y}$ are

taken from $\mathcal{X}$, except that we remove the constraints associated to the deleted node p .

In the definition above, all edge constraints from $\mathcal{X}$ are copied to $\mathcal{Y}$, and therefore the variables associated with deleted edges are still present in $\mathcal{C}_{\mathcal{Y}}^E$. We need to keep these variables around for the subsequent operations performed on $\mathcal{Y}$. For instance, when computing multiplicities for the variables in an edge constraint, the possible multiplicities of the deleted edges must also be taken into account, since they may influence the multiplicities of the remaining edges in the shape.

The fact that the operation above is “well-behaved” is captured by the following proposition.

Proposition 8.54 (Deletion effect). *Let $\mathcal{X} \in \text{QShape}$ be an unambiguous T -quasi-shape, and let $p \in N_{\mathcal{X}}$. Then $\text{del}(\mathcal{X}, p)$ is an unambiguous T -quasi-shape. $\triangleleft$*

The correctness requirement for the deletion on quasi-shapes is that every concrete operation on pattern graphs can be mimicked by the quasi-shape counterpart.

Proposition 8.55 (Deletion correctness). *Let $\mathcal{X} \in \text{QShape}$ be an unambiguous T -quasi-shape and let $P \in \text{concr}(\mathcal{X})$ be a concretisation of $\mathcal{X}$ with satisfying morphism $m_P : P \rightarrow P_{\mathcal{X}}$.*

If $Q = \text{del}(P, p)$ is a pattern graph deletion then there exists a corresponding quasi-shape deletion $\mathcal{Y} = \text{del}(\mathcal{X}, m_P(p))$, such that

- $Q \in \text{concr}(\mathcal{Y})$ with satisfying morphism $m_Q : Q \rightarrow P_{\mathcal{Y}}$; and
- $m_Q = m_P|_Q$. $\triangleleft$

Operation add

This operation is rather simple; since pattern graph elements created by a rule application can only be concrete, we only need to add proper constraints to the quasi-shape to reflect this fact.

A singleton constraint $\langle x, m \rangle$ is called *concrete* if $m = \mathbf{1}$. The creation of an edge pattern in a quasi-shape requires the ancestor node patterns to have associated concrete constraints, i.e., given $p_1, p_2 \in N_{\mathcal{X}}^0$, $\langle x_{p_i}, \mathbf{1} \rangle \in \mathcal{C}_{\mathcal{X}}^N$, for $i = 1, 2$. The creation of node patterns has no pre-requisites.

Definition 8.56 (Pattern addition). *Let $\mathcal{X} \in \text{QShape}$ be an unambiguous quasi-shape with typing morphism $t_{\mathcal{X}}$, and let $p' \in N_T^0$ be a level-0 type node.*

$$\begin{aligned} \text{add}_{p'}(\mathcal{X}) &:= \mathcal{Y} \text{ where} \\ P_{\mathcal{Y}} &= \text{add}_{p'}(P_{\mathcal{X}}) \quad \text{let } p \in N_{\mathcal{Y}} \setminus N_{\mathcal{X}} \\ \mathcal{C}_{\mathcal{Y}} &= \mathcal{C}_{\mathcal{X}} \cup \{\langle x_p, \mathbf{1} \rangle\} . \end{aligned}$$

Now, let $q' \in N_T^1$ be a level-1 type node with incoming edges $d'_1, d'_2 \in q' \triangleleft_T$, and let $p_i \in t_{\mathcal{X}}^{-1}(\text{src}_T(d'_i))$ such that $\langle x_{p_i}, \mathbf{1} \rangle \in \mathcal{C}_{\mathcal{X}}^N$, for $i = 1, 2$.

$$\begin{aligned} \text{add}_{d'_1, d'_2}(\mathcal{X}, p_1, p_2) &:= \mathcal{Y} \text{ where} \\ P_{\mathcal{Y}} &= \text{add}_{d'_1, d'_2}(P_{\mathcal{X}}, p_1, p_2) \quad \text{let } d_1, d_2 \in E_{\mathcal{Y}} \setminus E_{\mathcal{X}} \\ \mathcal{C}_{\mathcal{Y}} &= \mathcal{C}_{\mathcal{X}} \cup \{\langle x_{d_1}, \mathbf{1} \rangle, \langle x_{d_2}, \mathbf{1} \rangle\} . \end{aligned} \quad \triangleleft$$

Thus, in the definition above we simply operate on the pattern graph structure of $\mathcal{X}$ and add a new concrete constraint for the created node p . Note that, although we do not add a concrete constraint for the edge pattern $\mathbf{tgt}_{\mathcal{Y}}(d_1) = \mathbf{tgt}_{\mathcal{Y}}(d_2) = q$ created by the operation, we can implicitly derive that $x_q = \mathbf{1}$, since $x_{d_i} = \mathbf{1}$ and $x_{p_i} = \mathbf{1}$.

The following proposition establishes the effect of an addition operation.

Proposition 8.57 (Addition effect). *Let $\mathcal{X} \in \mathbf{QShape}$ be an unambiguous quasi-shape with typing morphism $t_{\mathcal{X}}$, and let $p' \in N_T^0$ and $q' \in N_T^1$ with $d'_1, d'_2 \in q' \triangleleft_T$.*

1. $\mathcal{Y} = \mathbf{add}_{p'}(\mathcal{X})$ is an unambiguous quasi-shape with typing morphism $t_{\mathcal{Y}}$; moreover, if $\mathcal{X}$ is a T -quasi shape then so is $\mathcal{Y}$.
2. For arbitrary $p_i \in t_{\mathcal{X}}^{-1}(\mathbf{src}_T(d'_i))$ ($i = 1, 2$), $\mathcal{Y} = \mathbf{add}_{d'_1, d'_2}(\mathcal{X}, p_1, p_2)$ is an unambiguous quasi-shape with typing morphism $t_{\mathcal{Y}}$. $\triangleleft$

The correctness requirement for addition is equivalent to the one for deletion.

Proposition 8.58 (Addition correctness). *Let $\mathcal{X} \in \mathbf{QShape}$ be an unambiguous T -quasi-shape and let $P \in \mathbf{concr}(\mathcal{X})$ be a concretisation of $\mathcal{X}$ with satisfying morphism $m_P : P \rightarrow P_{\mathcal{X}}$.*

1. If $Q = \mathbf{add}_{p'}(P)$ is a pattern graph addition of node $p \in N_Q \setminus N_P$ then there exists a corresponding quasi-shape addition $\mathcal{Y} = \mathbf{add}_{p'}(\mathcal{X})$, such that
 - $p \in N_{\mathcal{Y}} \setminus N_{\mathcal{X}}$;
 - $Q \in \mathbf{concr}(\mathcal{Y})$ with satisfying morphism $m_Q : Q \rightarrow P_{\mathcal{Y}}$; and
 - $m_Q = m_P \cup \{p \mapsto p\}$.
2. If $Q = \mathbf{add}_{d'_1, d'_2}(P, p_1, p_2)$ is a pattern graph addition of node $q \in N_Q \setminus N_P$ and edges $d_1, d_2 \in q \triangleleft_Q$ then there exists a corresponding quasi-shape addition $\mathcal{Y} = \mathbf{add}_{d'_1, d'_2}(\mathcal{X}, m_P(p_1), m_P(p_2))$, such that
 - $q \in N_{\mathcal{Y}} \setminus N_{\mathcal{X}}$ and $d_1, d_2 \in q \triangleleft_{\mathcal{Y}}$;
 - $Q \in \mathbf{concr}(\mathcal{Y})$ with satisfying morphism $m_Q : Q \rightarrow P_{\mathcal{Y}}$; and
 - $m_Q = m_P \cup \{q \mapsto q, d_1 \mapsto d_1, d_2 \mapsto d_2\}$. $\triangleleft$

Operation close

As done previously with pattern graphs, the closure of quasi-shapes amounts to performing a series of `close` operations with increasingly larger pattern types.

Similarly to the cases for operations `del` and `add`, operation `close` is built upon the closure operation defined on pattern graphs. However, in this case, the update of the constraint set poses a challenge. While in some cases we can derive a precise value for the constraints added for the new closing edges, in the general case, no precise information can be established based solely on the constraints of the pre-closing pattern. For this reason, the new edges added by the closure remain unconstrained. Alternatively, one can think of a new constraint $\langle x_d, \mathbf{0}^+ \rangle$ for all new edges d , which effectively imposes no restrictions on the multiplicity for d . This reasoning justifies the use of multiplicity $\mathbf{0}^+$ in the definition below.

Definition 8.59 (Pattern closure). Let $\mathcal{X} \in \text{QShape}$ be an unambiguous quasi-shape with typing morphism $t_{\mathcal{X}} : \mathcal{X} \rightarrow T$, and let $q' \in N_T^{2+}$ be a type node with incoming edges $d'_1, d'_2 \in q' \triangleleft_T$.

Furthermore, let $m : \text{cp}_T^{\triangleleft}(q') \rightarrow \mathcal{X}$ be a pattern graph morphism from the pre-closing pattern of q' to a left-closed sub-graph of $\mathcal{X}$, such that there does not exist an m -extension $m' : \text{cp}_T(q') \rightarrow \mathcal{X}$; and let $p_i = m(\text{src}_T(d'_i))$, for $i = 1, 2$.

$$\begin{aligned} \text{close}_{d'_1, d'_2}(\mathcal{X}, p_1, p_2) &:= \mathcal{Y} \text{ where} \\ P_{\mathcal{Y}} &= \text{close}_{d'_1, d'_2}(P_{\mathcal{X}}, p_1, p_2) && \text{let } d_1, d_2 \in E_{\mathcal{Y}} \setminus E_{\mathcal{X}} \\ C_{\mathcal{Y}} &= C_{\mathcal{X}} \cup \{\langle x_{d_1}, \mathbf{0}^+ \rangle, \langle x_{d_2}, \mathbf{0}^+ \rangle\} . \end{aligned} \quad \triangleleft$$

Leaving the closing edges unconstrained in the definition above has an undesirable effect later on when trying to evolve a quasi-shape back into its materialisations: it causes operation **br** to branch more, since all possible multiplicity values have to be instantiated in some materialisation. However, note that this never produces invalid shape configurations, because the multiplicities of the materialised shape are required to be consistent, per Definition 8.34.

Proposition 8.60 (Closure effect). Let $\mathcal{X}$ be an unambiguous quasi-shape with typing morphism $t_{\mathcal{X}} : \mathcal{X} \rightarrow T$, and let $d'_i \in E_T$, $p_i \in N_{\mathcal{X}}$ (for $i = 1, 2$) be such that $\mathcal{Y} = \text{close}_{d'_1, d'_2}(\mathcal{X}, p_1, p_2)$ is defined. Then $\mathcal{Y}$ is an unambiguous quasi-shape with typing morphism $t_{\mathcal{Y}} : \mathcal{Y} \rightarrow T$. $\triangleleft$

The correctness requirement for closure is the same as for addition.

Proposition 8.61 (Closure correctness). Let $\mathcal{X} \in \text{QShape}$ be an unambiguous T -quasi-shape and let $P \in \text{concr}(\mathcal{X})$ be a concretisation of $\mathcal{X}$ with satisfying morphism $m_P : P \rightarrow P_{\mathcal{X}}$.

If $Q = \text{close}_{d'_1, d'_2}(P, p_1, p_2)$ is a closure operation of node $q \in N_Q \setminus N_P$ and edges $d_1, d_2 \in q \triangleleft_Q$ then there exists a corresponding quasi-shape closure $\mathcal{Y} = \text{close}_{d'_1, d'_2}(\mathcal{X}, m_P(p_1), m_P(p_2))$, such that

- $q \in N_{\mathcal{Y}} \setminus N_{\mathcal{X}}$ and $d_1, d_2 \in q \triangleleft_{\mathcal{Y}}$;
- $Q \in \text{concr}(\mathcal{Y})$ with satisfying morphism $m_Q : Q \rightarrow P_{\mathcal{Y}}$; and
- $m_Q = m_P \cup \{q \mapsto q, d_1 \mapsto d_1, d_2 \mapsto d_2\}$. $\triangleleft$

Operation **sp**

Let $\mathcal{Q}$ be a quasi-shape. A pattern node $p \in N_{\mathcal{Q}}$ is said to be *deterministic*, denoted $\text{det}_{\mathcal{Q}}(p)$, if for all constraints $c \in \mathcal{C}_{\mathcal{Q}}^E(p)$, c is of the form $\langle x, m \rangle$. In other words, if p is deterministic, then no edges outgoing from p are shared, i.e., the edges do not have variables in non-singular constraints. If this is not the case, an offending constraint c is said to be *non-singular* in p . Quasi-shape $\mathcal{Q}$ is called *deterministic* if for all $p \in N_{\mathcal{Q}}$, $\text{det}_{\mathcal{Q}}(p)$ holds.

Note that splitting undoes the sharing of outgoing edges introduced by disambiguation. Being the dual of **da**, operation **sp** reintroduces ambiguity.

Definition 8.62 (Splitting). Let $\mathcal{X} \in \text{QShape}$ be a T -quasi-shape with typing morphism $t_{\mathcal{X}} : \mathcal{X} \rightarrow T$, and let $p \in N_{\mathcal{X}}$ be a pattern node such that $\text{det}_{\mathcal{X}}(p)$ does not hold.

Furthermore, let $c \in \mathcal{C}_Q^E(p)$ be a non-singular constraint in p and define $E^{\text{sp}} = \{e \in E_{\mathcal{X}} \mid x_e \text{ occurs in } c\}$ as the set of shared edges to be split and $N^{\text{sp}} = \{\langle p, \sigma \rangle \mid \sigma \text{ solves } c\}$ as the set of nodes created from p .

$\text{sp}(\mathcal{X}, p, c) := \mathcal{Y}$ where

$$\begin{aligned}
 N_{\mathcal{Y}} &= (N_{\mathcal{X}} \setminus \{p\}) \cup N^{\text{sp}} \\
 E_{\mathcal{Y}} &= E_{\mathcal{X}} \setminus (p \triangleleft_{\mathcal{X}} \cup p \triangleright_{\mathcal{X}}) \cup \{E_d^{\triangleleft} \mid d \in p \triangleleft_{\mathcal{X}}\} \cup \{E_d^{\triangleright} \mid d \in p \triangleright_{\mathcal{X}}\} \text{ with} \\
 E_d^{\triangleleft} &= \{\langle d, q \rangle \mid q \in N^{\text{sp}}\} \\
 E_d^{\triangleright} &= \begin{cases} \{\langle q, d \rangle \mid q = \langle p, \sigma \rangle \in N^{\text{sp}}, \sigma(x_d) \neq \mathbf{0}\} & \text{if } d \in E^{\text{sp}} \\ \{\langle q, d \rangle \mid q \in N^{\text{sp}}\} & \text{otherwise} \end{cases} \\
 \text{src}_{\mathcal{Y}}(a) &= \begin{cases} \text{src}_{\mathcal{X}}(d) & \text{if } a = \langle d, q \rangle \in E_d^{\triangleleft} \\ q & \text{if } a = \langle q, d \rangle \in E_d^{\triangleright} \\ \text{src}_{\mathcal{X}}(a) & \text{otherwise} \end{cases} \\
 \text{tgt}_{\mathcal{Y}}(a) &= \begin{cases} q & \text{if } a = \langle d, q \rangle \in E_d^{\triangleleft} \\ \text{tgt}_{\mathcal{X}}(d) & \text{if } a = \langle q, d \rangle \in E_d^{\triangleright} \\ \text{tgt}_{\mathcal{X}}(a) & \text{otherwise} \end{cases} \\
 \mathcal{C}_{\mathcal{Y}}^N &= (\mathcal{C}_{\mathcal{X}}^N \setminus \mathcal{C}_p^N) \cup \mathcal{D}_p^N \text{ with} \\
 \mathcal{C}_p^N &= \begin{cases} \mathcal{C}_{\mathcal{X}}^N(p) & \text{if } p \in N_{\mathcal{X}}^0 \\ \emptyset & \text{otherwise} \end{cases} \\
 \mathcal{D}_p^N &= \{c'[x_p := \{x_q \mid q \in N^{\text{sp}}\}] \mid c' \in \mathcal{C}_p^N\} \\
 \mathcal{C}_{\mathcal{Y}}^E &= \mathcal{C}_{\mathcal{X}}^E \setminus (\mathcal{C}^{\triangleleft} \cup \mathcal{C}^{\triangleright}) \cup \mathcal{D}^{\triangleleft} \cup \mathcal{D}^{\triangleright} \text{ with} \\
 \mathcal{C}^{\triangleleft} &= \{\mathcal{C}_d^{\triangleleft} \mid d \in p \triangleleft_{\mathcal{X}}\} \text{ with } \mathcal{C}_d^{\triangleleft} = \mathcal{C}_{\mathcal{X}}^E(\text{src}_{\mathcal{X}}(d)) \\
 \mathcal{C}^{\triangleright} &= \mathcal{C}_{\mathcal{X}}^E(p) \\
 \mathcal{D}^{\triangleleft} &= \{\mathcal{D}_d^{\triangleleft} \mid d \in p \triangleleft_{\mathcal{X}}\} \text{ with} \\
 \mathcal{D}_d^{\triangleleft} &= \{c'[x_d := \{x_a \mid a \in E_d^{\triangleleft}\}] \mid c' \in \mathcal{C}_d^{\triangleleft}\} \\
 \mathcal{D}^{\triangleright} &= \{\mathcal{D}_{c'}^{\triangleright} \mid c' \in \mathcal{C}^{\triangleright}\} \text{ with} \\
 \mathcal{D}_{c'}^{\triangleright} &= \begin{cases} \{\langle x_a, \sigma(x_d) \rangle \mid d \in E^{\text{sp}}, a = \langle q, \sigma \rangle \in E_d^{\triangleright}\} & \text{if } c' = c \\ \{c'[x_d := \{x_{\langle q, d \rangle}\}] \mid x_d \in c', q \in N^{\text{sp}}\} & \text{otherwise} \end{cases}
 \end{aligned}$$

Moreover, for all $q \in N_{\mathcal{Y}}$,

$$t_{\mathcal{Y}}(q) = \begin{cases} t_{\mathcal{X}}(p) & \text{if } q \in N^{\text{sp}} \\ t_{\mathcal{X}}(q) & \text{otherwise} \end{cases}$$

and for all $a \in E_{\mathcal{Y}}$,

$$t_{\mathcal{Y}}(a) = \begin{cases} t_{\mathcal{X}}(d) & \text{if } a = \langle d, q \rangle \in E_d^{\triangleleft} \\ t_{\mathcal{X}}(d) & \text{if } a = \langle q, d \rangle \in E_d^{\triangleright} \\ t_{\mathcal{X}}(a) & \text{otherwise} \end{cases} .$$

◁

In words, we split node p into as many new copies as the possible solutions for constraint c (set N^{sp}). Similarly to operation **da**, when splitting we also duplicate incoming and outgoing edges of p , except for the edges in E^{sp} . Node constraints are modified if p is a layer-0 node, otherwise they remain unchanged.

Figure 8.10: Example of a sp operation, following the same visual notations introduced in Figures 8.8 and 8.9.

Edges constraints are also properly adjusted, possibly leading to the introduction of additional shared edges.

Operation sp is undefined if the quasi-shape $\mathcal{X}$ is deterministic. Since sp is the dual of da , splitting should be performed in $\mathcal{X}$ in the opposite direction, in a bottom-up manner according to the layers. This is the case because each splitting of a node in layer i may introduce additional sharing of edges outgoing from nodes in layers $< i$.

Example 8.63. Figure 8.10 shows an example of a sp operation with $p = p_2$ as the node to be split and $c = x_{d_3} + x_{d_4} = 0^+$ as the non-singular constraint in p . Constraint c admits the following three solutions,

	x_{d_3}	x_{d_4}
σ_1	0^+	0
σ_2	0^+	0^+
σ_3	0	0^+

and therefore p is split into three new nodes p_5, p_6, p_7 . The sets computed

according to Definition 8.62 are listed below.

$$\begin{aligned}
E^{\text{sp}} &= \{d_3, d_4\} \\
N^{\text{sp}} &= \{p_5 = \langle p, \sigma_1 \rangle, p_6 = \langle p, \sigma_2 \rangle, p_7 = \langle p, \sigma_3 \rangle\} \\
E_{d_1}^{\triangleleft} &= \{a = \langle d_1, p_5 \rangle, b = \langle d_1, p_6 \rangle, d = \langle d_1, p_7 \rangle\} \\
E_{d_2}^{\triangleleft} &= \{e = \langle d_2, p_5 \rangle, f = \langle d_2, p_6 \rangle, g = \langle d_2, p_7 \rangle\} \\
E_{d_3}^{\triangleright} &= \{h = \langle p_5, d_3 \rangle, i = \langle p_6, d_3 \rangle\} \\
E_{d_4}^{\triangleright} &= \{j = \langle p_6, d_4 \rangle, k = \langle p_7, d_4 \rangle\} \\
E_{d_5}^{\triangleright} &= \{l = \langle p_5, d_5 \rangle, m = \langle p_6, d_5 \rangle, n = \langle p_7, d_5 \rangle\} \\
E_{d_6}^{\triangleright} &= \{o = \langle p_5, d_6 \rangle, q = \langle p_6, d_6 \rangle, r = \langle p_7, d_6 \rangle\}
\end{aligned}$$

The constraint sets are shown to the right of Figure 8.10. Note that although node p_6 has two outgoing edges i and j of the same type, these edges are no longer shared and further splitting on quasi-shape $\mathcal{Y}$ does not reintroduce sharing for these edges, since they are duplicated on all split nodes. $\triangleleft$

The proposition below establishes the effect of a split operation, and similarly to disambiguation, it gives a termination criterion for a sequence of splits.

Proposition 8.64 (Effect of splitting). *Let $\mathcal{X} \in \text{QShape}$ be a T -quasi-shape, and let $p \in N_{\mathcal{X}}$ and $c \in \mathcal{C}_{\mathcal{Q}}^E(p)$ such that $\mathcal{Y} = \text{sp}(\mathcal{X}, p, c)$ is defined. Then $\mathcal{Y}$ is a T -quasi-shape and $\text{sp}(\mathcal{Y}, p, c)$ is undefined.* $\triangleleft$

The correctness requirement for operation sp is that the set of concretisations of the transformed quasi-shape remains the same.

Proposition 8.65 (Correctness of splitting). *Let $\mathcal{X} \in \text{QShape}$ be a T -quasi-shape and let $\mathcal{Y} = \text{sp}(\mathcal{X}, p, c)$ for some $p \in N_{\mathcal{X}}$ and $c \in \mathcal{C}_{\mathcal{Q}}^E(p)$. Then $\text{concr}(\mathcal{X}) = \text{concr}(\mathcal{Y})$.* $\triangleleft$

Operation br

Once a deterministic quasi-shape is obtained we are ready to return to a pattern shape representation. Recall, however, that a quasi-shape $\mathcal{Q}$ stands for a set of pattern shapes (the materialisations of $\mathcal{Q}$), and therefore when we want to *evolve* $\mathcal{Q}$ back into a shape we have to consider all possible materialisations.

It is important to note that, at this point, only node constraints are left to be solved since all non-singular edge constraints were already solved previously by split operations.

Definition 8.66 (Branching). *Let $\mathcal{Q} \in \text{QShape}$ be a deterministic T -quasi-shape. The branching on $\mathcal{Q}$ is defined as*

$$\text{br}(\mathcal{Q}) = \{S \in \text{PShape} \mid S \text{ satisfies } \mathcal{Q}\} . \quad \triangleleft$$

Note that since $\mathcal{Q}$ is T -typed and T -closed, this is also the case for all materialisations in $\text{br}(\mathcal{Q})$. The correctness criterion for operation br is that any concretisation of $\mathcal{Q}$ is also the concretisation of *some* materialisation S .

Proposition 8.67 (Correctness of branching). *Let $\mathcal{Q} \in \text{QShape}$ be a deterministic T -quasi-shape. Then $\text{concr}(\mathcal{Q}) = \bigcup_{S \in \text{br}(\mathcal{Q})} \text{concr}(S)$.* $\triangleleft$

Putting it together

As done with pattern graphs, we now return to the notations introduced at the beginning of this sub-section. For each operation we have the following.

- $\mathcal{X} \xrightarrow{\text{da}} \mathcal{Y}$ denotes that $\mathcal{Y} = \text{da}_{d'}(\mathcal{X}, p)$ for some $p \in N_{\mathcal{X}}$ and $d' \in E_T$.
- $\mathcal{X} \xrightarrow{\text{del}} \mathcal{Y}$ denotes that $\mathcal{Y} = \text{del}(\mathcal{X}, p)$ for some $p \in N_{\mathcal{X}}$.
- $\mathcal{X} \xrightarrow{\text{add}} \mathcal{Y}$ denotes that either $\mathcal{Y} = \text{add}_{p'}(\mathcal{X})$ for some $p' \in N_T$ or $\mathcal{Y} = \text{add}_{d'_1, d'_2}(\mathcal{X}, p_1, p_2)$ for some $p_1, p_2 \in N_{\mathcal{X}}$ and $d'_1, d'_2 \in E_T$.
- $\mathcal{X} \xrightarrow{\text{close}} \mathcal{Y}$ denotes that $\mathcal{Y} = \text{close}_{d'_1, d'_2}(\mathcal{X}, p_1, p_2)$ for some $p_1, p_2 \in N_{\mathcal{X}}$ and $d'_1, d'_2 \in E_T$.
- $\mathcal{X} \xrightarrow{\text{sp}} \mathcal{Y}$ denotes that $\mathcal{Y} = \text{sp}(\mathcal{X}, p, c)$ for some $p \in N_{\mathcal{X}}$ and $c \in \mathcal{C}_{\mathcal{X}}^E(p)$.
- $\mathcal{X} \xrightarrow{\text{br}} \{S_i\}_{i \in I}$ denotes that $\{S_i\}_{i \in I} = \text{br}(\mathcal{X})$.
- $\mathcal{X} \xrightarrow{\text{br}} S$ denotes that $S \in \text{br}(\mathcal{X})$.

The following proposition shows that the pre-conditions of each operation correspond to the post-conditions of its predecessor operation, thus establishing that we can indeed concatenate the steps as intended.

Proposition 8.68. *Let $\mathcal{X}$ be a quasi-shape.*

1. *The maximal sequence $\mathcal{X} \xrightarrow{\text{da}^*} \mathcal{Y}$ is confluent, leading to the unique (modulo isomorphism) unambiguous T -quasi-shape $\mathcal{Y}$.*
2. *For any sequence $\mathcal{X} \xrightarrow{\text{del}^*} \mathcal{Y}$, $\mathcal{Y}$ is unambiguous and T -typable; moreover, if $\mathcal{X}$ is T -closed then so is $\mathcal{Y}$.*
3. *For any sequence $\mathcal{X} \xrightarrow{\text{add}^*} \mathcal{Y}$, $\mathcal{Y}$ is unambiguous and T -typable.*
4. *The maximal sequence $\mathcal{X} \xrightarrow{\text{close}^*} \mathcal{Y}$ is confluent, leading to the unique (modulo isomorphism) unambiguous T -quasi-shape $\mathcal{Y}$.*
5. *The maximal sequence $\mathcal{X} \xrightarrow{\text{sp}^*} \mathcal{Y}$ is confluent, leading to the unique (modulo isomorphism) deterministic T -quasi-shape $\mathcal{Y}$.* $\triangleleft$

8.5 Pattern shape transformation

We are finally at the stage where we have all the ingredients to define the transformation of canonical pattern shapes. As done with pattern graphs, a rule application is decomposed in several operations, which were introduced in the previous section. The difference with pattern graphs is that here we use the operations to *define* the canonical pattern shape transformation.

Definition 8.69 (Canonical pattern shape transformation). *Let X be a canonical T -pattern shape, $r = \langle \lambda, \rho \rangle$ be a T -pattern graph rule, and $\mu : \lambda \rightarrow X$ be a pre-match of r into X . A canonical pattern shape transformation of X is defined as the canonical T -pattern shape $Y = \text{normalise}(Y')$, where*

$$\begin{aligned}
 X &\xrightarrow{\text{pull}} X' \xrightarrow{\text{devol}} Q_0 \xrightarrow{\text{da}^*} Q_i \\
 Q_i &\xrightarrow{\text{del}_{i+1} \cdots \text{del}_j} Q_j \xrightarrow{\text{add}_{j+1} \cdots \text{add}_k} Q_k \xrightarrow{\text{close}_{k+1} \cdots \text{close}_l} Q_l \\
 Q_l &\xrightarrow{\text{sp}^*} Q_m \xrightarrow{\text{br}} Y'
 \end{aligned}$$

where

$$P_{Q_i} \xrightarrow{\text{del}_{i+1} \cdots \text{del}_j} P_{Q_j} \xrightarrow{\text{add}_{j+1} \cdots \text{add}_k} P_{Q_k} \xrightarrow{\text{close}_{k+1} \cdots \text{close}_l} P_{Q_l}$$

is the pattern graph transformation sequence as characterised in Proposition 8.28, under match $\mu' : \lambda \rightarrow P_{Q_i}$ obtained by $X \xrightarrow{\text{pull}} X'$. $\triangleleft$

Thus, given a canonical shape $X \in \text{CanPShape}_T^{\text{n,e}}$ and a pre-match μ into X , we first extract a concrete match μ' from X , using operation pull . Subsequently, we move to a quasi-shape representation by means of devol . The resulting quasi-shape Q_0 is then modified by a sequence of da operations, to produce a disambiguated quasi-shape Q_i , which is transformed with deletion, addition and closing operations. Finally, the resulting quasi-shape Q_l is split to eliminate shared edges, leading to the final quasi-shape Q_m , for which we compute the set of materialisations, thus returning to a pattern shape representation. A resulting materialised shape Y' is then normalised back to a canonical shape Y , which is the result of the transformation.

Note that the sequence of transformation operations (*i.e.*, del , add , and close applications) for quasi-shapes is in one-to-one correspondence to the sequence of pattern graph operations. The justification for this comes from the fact that we have an identity morphism from P_{Q_i} into Q_i and that this identity is preserved along all transformation operations on quasi-shapes. This is a consequence of the correctness propositions for these operations.

Furthermore, note that the side condition for addition on quasi-shapes, namely, that the connecting nodes have to be concrete (see Definition 8.56) is always satisfied since we only connect nodes that are part of the match image $\mu'(\rho)$, which are all concrete due to the match condition (Definition 8.40).

The last argument on the correctness of the definition above is that the match $\mu' : \lambda \rightarrow X'$ obtained by operation pull is taken along unchanged until Q_i . This can be seen immediately for operation devol , since $P_{X'} = P_{Q_0}$. For the sequence of disambiguation applications, note that $\mu'(\lambda)$ is concrete and therefore unambiguous, and thus $\mu'(\lambda)$ is not modified by the da operations.

We write $X \Rightarrow^r Y$ to denote a canonical pattern shape transformation, which is illustrated with an example in Figure 8.11.

The next theorem states that any concrete pattern graph transformation is captured by a canonical pattern shape transformation.

Theorem 8.70. *Let X be a canonical T -pattern shape, $P \in \text{concr}(X)$ be a pattern graph concretisation of X , and let r be a T -pattern graph rule. If $P \xrightarrow{r} Q$ is a pattern graph transformation then there is a canonical pattern shape transformation $X \Rightarrow^r Y$, with $Q \in \text{concr}(Y)$.* $\triangleleft$

Similarly to what was done with simple graphs and pattern graphs, rule applications on pattern shapes produce a *pattern shape transition system* (PSTS).

Definition 8.71 (PSTS). *A pattern shape transition system is a tuple $\text{PSTS} = \langle \mathcal{S}, \Rightarrow, \iota \rangle$ where*

- $\mathcal{S} \subseteq \text{CanPShape}_T^{\text{n,e}}$ is a set of states;
- $\Rightarrow \subseteq \mathcal{S} \times \text{PRule} \times \mathcal{S}$ is a set of transitions; and
- $\iota \in \mathcal{S}$ is the initial state.

Figure 8.11: Pattern shape resulting from the application of rule `put` of Figure 8.5 to the pattern shape of Figure 8.7. Relation $\equiv$ is shown with dotted rectangles. After normalisation the canonical pattern shape obtained is isomorphic to the one given in Figure 8.7.

A pattern graph grammar $\mathcal{P}_T = \langle \mathcal{R}_T, P_0 \rangle$ generates a $PSTS_{\mathcal{P}}$ if $\mathcal{S}$ is a minimal set of canonical pattern shapes such that

- $\iota = \text{abstract}(P_0)$; and
- if $X \Rightarrow^r Y$ for some $X \in \mathcal{S}$ and $r \in \mathcal{R}_T$, then there exists $Y' \in \mathcal{S}$ where $Y \simeq Y'$ and $\langle X, r, Y' \rangle \in \Rightarrow$.

◁

The following is our final major result, establishing the connection between concrete and abstract transition systems.

Theorem 8.72. *Given a pattern graph grammar $\mathcal{P}_T = \langle \mathcal{R}_T, P_0 \rangle$, let $PGTS_{\mathcal{P}} = \langle \mathcal{S}_{\mathcal{P}}, \Rightarrow, P_0 \rangle$ and $PSTS_{\mathcal{P}} = \langle \mathcal{S}'_{\mathcal{P}}, \Rightarrow, \text{abstract}(P_0) \rangle$ be the PGTS and PSTS generated by $\mathcal{P}_T$, respectively. Transition system $PSTS_{\mathcal{P}}$ simulates $PGTS_{\mathcal{P}}$.*

◁

An immediate consequence of PSTS being an over-approximation is that verification can then be carried out on the abstract domain, which is always finite. As usual with over-approximations, if a property holds at the pattern shape level then it is guaranteed to hold at the pattern graph level; however if a property is false for pattern shapes then we cannot be sure that it is also false for pattern graphs, since the abstraction can introduce spurious behaviour. The results at the pattern graph level transfer directly to simple graphs due to the bi-simulation result of Theorem 8.33.

8.6 Conclusion

This chapter presented a new method for a property-driven abstraction of graphs, based on a pre-defined collection of patterns of interest, represented as a pattern type graph. Pattern-based abstraction leads to a finite over-approximation of (infinite-state) graph transformation systems and as such the abstraction can be used for system verification. Furthermore, the technique lends itself nicely to abstraction refinement: one can start with a rather minimal type graph and then add more patterns to it to make the abstraction more precise when necessary.

An interesting side-effect of developing a theory of pattern graph transformation is that structures used in incremental pattern matching like RETE networks [GJR10] can now also be formalised as pattern graphs. To the best of our knowledge, this is the first time such structures were considered under a more formal focus.

The main advantage of pattern abstraction over neighbourhood abstraction is the flexibility in specifying which structures should be collapsed. Precision of the neighbourhood method can be adjusted via the radius parameter but this radius is always the same when analysing the equivalence of nodes and edges. On the other hand, the pattern-based method is much more fine-grained: patterns of various sizes can be represented in the type graph, and are considered independently by the abstraction. Roughly speaking, a radius i neighbourhood abstraction can be simulated by a pattern abstraction using a type graph T with depth i , where all possible simple graphs of size smaller or equal to i occur as patterns in T .

The implementation of pattern abstraction into GROOVE is currently ongoing. As evidenced in Chapter 7, experimental evaluation is essential to gauge how suitable a proposed abstraction is in practice. An interesting point to note is that the lessons learned with neighbourhood abstraction carry over to this new setting: the exploration of pattern shapes should stick to a DFS strategy, and an equivalent collapsing of states under iso-subsumption is also necessary. Knowing this we can plan the code development accordingly.

As a final note we would like to point that while pattern shapes may seem more complex than neighbourhood shapes, the first lend themselves to a practical implementation much easily than the second. The key reason for this advantage is that, from a algorithmic point of view, updating a pattern shape is simpler than updating a neighbourhood shape; the latter has a grouping relation that is quite hard to keep up-to-date and consistent upon every modification of the neighbourhood shape structure.

Furthermore, during the entire development of the pattern abstraction theory we always kept in mind its future implementation. This is reflected in the text of this chapter, where most definitions are constructive instead of declarative. Thus, due to these facts, we are confident that state space exploration with pattern abstraction will perform better than with neighbourhood abstraction. Nevertheless, checking whether this claim indeed holds remains as future work.

Conclusion

This chapter provides an overall conclusion of the thesis, built upon some of the individual conclusions given at the end of each chapter. We give a summary of the chapters contents, intertwined with a discussion on the contributions of this thesis. We conclude with an indication of possible future work.

9.1 Thesis summary and discussion of contributions

In this thesis, we presented two abstraction techniques that can be used for the analysis of infinite-state graph transformation systems. One of such methods, namely, neighbourhood abstraction, is covered under both a theoretical and a practical light, providing a complete evaluation of its characteristics. The other abstraction method proposed, *i.e.*, pattern abstraction, is given a detailed theoretical presentation, while still keeping practical implementation aspects in sight.

We also investigated and discussed another aspect revolving around graph transformation, namely the connection between imperative programming languages and graph transformation systems. Our work focused on the translation from the former to the latter, in order to bridge the gap between standard programming and modelling by graph transformation.

In the following we present individual summaries for each chapter and highlight its contributions.

Chapter 1. In this chapter we give an overall introduction to the concept of system verification and explain the setting where the contents of this thesis fit. This setting is depicted as a complete verification cycle for graph transformation systems, where its key ingredient is state space exploration. We then motivate the need for graph abstractions, to allow the exploration of infinite-state systems. In this chapter we also give a bird's-eye view of the translation process between a program source code and its corresponding graph-based model.

Contributions. A layperson may profit from the big pictures provided in this chapter. In particular, the general explanation given for the verification cycle of graph transformation systems serves as an introduction for understanding the design behind GROOVE.

Chapter 2. This chapter provides an introduction on essential points necessary for the rest of the thesis. We present the two most widely-known algebraic

approaches to graph transformation, DPO and SPO, and we discuss their differences, advantages and disadvantages.

We then continue with a detailed presentation of the GROOVE functionalities. All the main components of the verification cycle presented in Chapter 1 that are implemented in GROOVE are properly illustrated with a complete example, showing the verification of safety and liveness properties of a leader election protocol for communicating processes.

The last part of the chapter provides a introduction to neighbourhood and pattern abstraction, with examples of shapes and their corresponding concretisations.

Contributions. The section about GROOVE serves as a high-level explanation on the functionalities of the tool. A reader interested in a brief overview on graph transformation and graph abstraction may profit from this chapter.

Chapter 3. This chapter presents other researchers' work that is related to the contents of this thesis. We cite other publications that also provide a graph-based representation for Java, or perform model checking of Java programs. Another research area mentioned is guided search, which makes use of abstraction techniques. The largest part of the chapter is devoted to the presentation of related work on abstraction methods. The techniques covered include shape analysis, Petri graph unfolding, hyper-edge replacement-based abstraction, and partner/star abstraction. A summary comparing these abstractions is given at the end of the chapter.

Contributions. Although this chapter is not an exhaustive literature review, it serves as a important list of references, in particular on graph abstractions. The comparison between the different abstractions may provide a quick overview to a person that is not an expert in the field.

Chapter 4. We present two of the steps for the translation of Java source code to GROOVE graphs. First, we discuss in detail the construction of a graph compiler based on the existing Eclipse Java compiler, and we show how a GROOVE type graph can be constructed from the compiler code. The second part of the chapter presents a graph grammar that captures the control flow semantics of Java.

Contributions. The major contributions of the work in this chapter come in the form of tool support. The graph compiler, the Java syntax type graph, and the flow construction grammar are all available for use, and can be employed without much learning effort from the user. Apart from tooling, the methodology for the translation from code to graphs described in this chapter can serve as a basis for similar tasks involving other programming languages.

Chapter 5. In this chapter we formally define simple graphs and their transformations, and we establish our semantic representation for simple graph grammars in terms of transition systems. The main algorithm for state space exploration, as currently implemented in GROOVE, is also presented in a high-level

form. The definition for multiplicities and the associated operations are also given here. The chapter concludes with a discussion about how abstract and concrete transition systems are related by simulations, and about which kind of properties these simulations preserve.

Contributions. The discussion on the algorithm for state space exploration serve as a gentle introduction for someone that is interested in learning the inner workings of GROOVE.

Chapter 6. This chapter presented the main concepts and results of the theory of neighbourhood abstraction. We formally define neighbourhood shapes, their characteristics and how they can be transformed. The finiteness of the abstraction is guaranteed by using canonical shapes, which are computed up to a certain radius and form a finite universe. The chapter concludes with the simulation relation that is established between simple graph and neighbourhood shape transition systems.

Contributions. The contents of this chapter serve as a good summary of neighbourhood abstraction. Most of the text in this chapter was previously available only as technical reports. Here, the presentation is properly unified with the notation used in the rest of the thesis.

Chapter 7. In this chapter we discussed the main points of the neighbourhood abstraction implementation. We first give an overview of the main exploration algorithm for abstract state spaces and we present all its steps, with focus on the materialisation operation. We also propose a new method for state collapsing, based on the iso-subsumption relation. The chapter concludes with an experimental analysis of the ShapeGenerator tool that was developed.

Contributions. The final product of the work described in this chapter is the new ShapeGenerator tool that was incorporated in the GROOVE tool set. The main implementation points discussed in the text and the results obtained from experimentation are actually transferable to similar exploration methods. As an example, given that we have a similar setting in pattern graph abstraction, we know already that its future implementation should incorporate the same concept of iso-subsumption for pattern shapes.

Chapter 8. This chapter gave a full presentation of the theory for pattern abstraction. We first show how simple graphs can be lifted to pattern graphs and that their corresponding transition systems are bi-similar. We then discuss the construction of pattern shapes and how they can be transformed by rule applications. As done with neighbourhood abstraction, we show that the abstract pattern shape transition system and the concrete pattern graph counterpart are related by a simulation, which implies that the abstraction is an over-approximation of the original system.

Contributions. The theory of pattern abstraction itself constitutes the main contribution of this chapter. As also pointed in the chapter conclusion, this

formalism could also be used to describe other hierarchical structures and their transformations, such as a RETE network and its updates.

9.2 Future work

Based on the work presented in this thesis, there are several directions that can be further explored. In the following, we discuss possible future work for the chapters that leave some points open.

Chapter 4. A question that was not addressed in this chapter is the performance evaluation of the translation from Java to graphs. So far, the tool was used only on simple input programs and thus an analysis with more realistic program sizes is in order. The performance of the graph compiler is not expected to differ much from the original performance of the normal Eclipse compiler, since the additional step of outputting the internal AST to a GROOVE graph is a rather lightweight operation. The performance of the flow construction, on the other hand, may constitute a problem when input graphs with thousands of nodes are used. In this case, some of the GROOVE algorithms, such as matching, will have to be adapted to the specifics of this setting.

The development of a graph grammar modelling the Java execution semantics is currently ongoing. This is the last step required in order to allow GROOVE to “run” Java programs.

Chapter 7. The most straightforward line of future work in this chapter is the experimentation with more graph grammars and a comparison with other tools.

Another interesting point, but now concerning the usability of the tool is the proper integration of the ShapeGenerator with the GUI of the GROOVE Simulator, which will allow for the visualisation of abstract transition systems.

A third item of future investigation is the use of proper subsumption collapsing, instead of iso-subsumption, during abstract state space exploration. As mentioned before, this may increase the tool performance but unfortunately requires additional code refactoring.

Chapter 8. Perhaps the most pressing line of future work, and the one we are already pursuing at the time of this writing, is the implementation of pattern abstraction in GROOVE. As done with neighbourhood abstraction, after the tool is developed we can carry out experiments to gauge how well the new proposed abstraction performs in practice.

We expect this new tool to outperform the original neighbourhood shape generator. This expectation is based on the fact that pattern shapes retain more structural information than neighbourhood shapes, which in turn will allow for a more precise materialisation step. This improvement in precision translates to less spurious abstract states and implies that the amount of work to be done in a pattern-based exploration will be smaller when compared to a neighbourhood-based exploration.

Bibliography

- [ALP07] M. Alanen, T. Lundkvist, and I. Porres. Creating and reconciling diagrams after executing model transformations. *Sci. Comput. Program.*, 68(3):155–178, 2007.
- [AP04] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS – Turku Centre for Computer Science, March 2004.
- [APV06] S. Anand, C. Păsăreanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *SPIN*, LNCS 3925, pages 163–181. Springer, 2006.
- [APV07] S. Anand, C. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS*, LNCS 4424, pages 134–138. Springer, 2007.
- [Bau06] J. Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität des Saarlandes, 2006.
- [BBKR08] J. Bauer, I. Boneva, M. Kurban, and A. Rensink. A modal-logic based graph abstraction. In Ehrig et al. [EHRT08], pages 321–335.
- [BCE⁺05] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying red-black trees. In *COSMICAH*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).
- [BCK01] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *CONCUR*, LNCS 2154, pages 381–395. Springer, 2001.
- [BCK02] P. Baldan, A. Corradini, and B. König. Static analysis of distributed systems with mobility specified by graph grammars – a case study. In *IDPT*. Society for Design and Process Science, 2002.
- [BCK04] P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: An unfolding-based approach. In *CONCUR*, LNCS 3170, pages 83–98. Springer, 2004.
- [BCK08] P. Baldan, A. Corradini, and B. König. A framework for the verification of infinite-state graph transformation systems. *Inf. Comput.*, 206(7):869–907, 2008.
- [BCKLL06] P. Baldan, A. Corradini, B. König, and A. Lluch-Lafuente. A temporal graph logic for verification of graph transformation systems. In *WADT*, LNCS 4409, pages 1–20. Springer, 2006.
- [BGK01] D. Baumer, E. Gamma, and A. Kiezun. Integrating refactoring support into a Java development tool. In *OOPSLA 2001 Companion*, 2001.
- [BHS07] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

- [BK02] P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *ICGT*, LNCS 2505, pages 14–29. Springer, 2002.
- [BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, New York, 2008.
- [BKK03] P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In *SAS*, LNCS 2694, pages 255–272. Springer, 2003.
- [BKR⁺12] I. Boneva, J. Kreiker, A. Rensink, M. Kurban, and E. Zambon. Graph abstraction and abstract graph transformation (amended version). Technical Report TR-CTIT-12-26, University of Twente, October 2012.
- [BLS04] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, LNCS 3362, pages 49–69. Springer, 2004.
- [BR10a] P. Backes and J. Reineke. Abstract topology analysis of the join phase of the merge protocol. In *TCC*, volume WP10-03 of *CTIT Workshop Proceedings*, pages 127–133. University of Twente, 2010.
- [BR10b] P. Backes and J. Reineke. A graph transformation case study for the topology analysis of dynamic communication system. In *TTC*, volume WP10-03 of *CTIT Workshop Proceedings*, pages 107–118. University of Twente, 2010.
- [BRKB07] I. Boneva, A. Rensink, M. Kurban, and J. Bauer. Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, University of Twente, July 2007.
- [BW07] J. Bauer and R. Wilhelm. Static analysis of dynamic communication systems by partner abstraction. In *SAS*, LNCS 4634. Springer, 2007.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [CDFR04] A. Corradini, F. Dotti, L. Foss, and L. Ribeiro. Translating Java code to graph transformation systems. In *ICGT*, LNCS 3256, pages 383–398. Springer, 2004.
- [CDH⁺00] J. Corbett, M. Dwyer, J. Hatchiff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.
- [CEM⁺06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors. *ICGT*, LNCS 4178. Springer, 2006.
- [CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
- [CHHK06] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-pushout rewriting. In Corradini et al. [CEM⁺06], pages 30–45.
- [Cou90] B. Courcelle. Graph rewriting: an algebraic and logic approach. In *Handbook of Theoretical Computer Science (vol. B)*, pages 193–242. MIT Press, 1990.
- [CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of ACM*, 22(5):281–283, 1979.

- [dBNR09] N. de Brugh, V. Nguyen, and T. Ruys. MoonWalker: Verification of .NET programs. In *TACAS*, LNCS 5505, pages 170–173. Springer, 2009.
- [DHHR05] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *CAV*, LNCS 3576, pages 148–152. Springer, 2005.
- [Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [dLV10] J. de Lara and D. Varro, editors. *GraBaTs*, volume 32 of *Electronic Communications of the EASST*, 2010.
- [DMSS01] J. Docks, K. Mertens, N. Smeets, and E. Steegmans. jnome: A Java meta model in detail. Technical Report CW 232, Katholieke Universiteit Leuven, December 2001.
- [DP08] D. Distefano and M. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.
- [Ecl] Eclipse Foundation. JDT core component development resources. <http://www.eclipse.org/jdt/core/dev.php>.
- [EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.
- [EHRT08] H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors. *ICGT*, LNCS 5214. Springer, 2008.
- [EJLL06] S. Edelkamp, S. Jabbar, and A. Lluch-Lafuente. Heuristic search for the analysis of graph transition systems. In Corradini et al. [CEM⁺06], pages 414–429.
- [FLL⁺02] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
- [Flo67] R. Floyd. Assigning meanings to programs. *Proceedings of the Symposium on Applied Mathematics*, 19(10):19–32, 1967.
- [FM07] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, LNCS 4590, pages 173–177. Springer, 2007.
- [GdMR⁺12] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *STTT*, 14(1):15–40, February 2012.
- [GJR10] A. Ghamarian, A. Jalali, and A. Rensink. Incremental pattern matching in graph-based state space exploration. In de Lara and Varro [dLV10].
- [Gro04] Object Management Group. Metamodel and UML profile for Java and EJB specification. Technical Report version 1.0, OMG, February 2004. <http://www.omg.org/docs/formal/04-02-02.pdf>.
- [GZR⁺11] I. Galvão, E. Zambon, A. Rensink, L. Wevers, and M. Aksit. Knowledge-based graph exploration analysis. In *AGTIVE*, LNCS 7233, pages 121–136. Springer, 2011.
- [HBJ11] J. Heinen, H. Barthels, and C. Jansen. Juggernaut - an abstract JVM. In *FoVeOOS*, LNCS 7421, pages 142–159. Springer, 2011.
- [HMZM96] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1–2):321–361, 1996.

- [HNR10] J. Heinen, T. Noll, and S. Rieger. Juggernaut: Graph grammar abstraction for unbounded heap structures. *ENTCS*, 266:93–107, 2010.
- [Hoa69] C. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, 1969.
- [JHKN11] C. Jansen, J. Heinen, J.-P. Katoen, and T. Noll. A local greibach normal form for hyperedge replacement grammars. In *LATA*, LNCS 6638, pages 323–335. Springer, 2011.
- [Kas08] H. Kastenbergh. *Graph-based Software Specification and Verification*. PhD thesis, University of Twente, October 2008.
- [KK05] B. König and V. Kozioura. AUGUR – a tool for the analysis of graph transformation systems. *Bulletin of the EATCS*, 87:126–137, 2005.
- [KK06] B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *TACAS*, LNCS 3920, pages 197–211. Springer, 2006.
- [KK08a] B. König and V. Kozioura. AUGUR 2 – a new version of a tool for the analysis of graph transformation systems. *ENTCS*, 211:201–210, 2008.
- [KK08b] B. König and V. Kozioura. Towards the verification of attributed graph transformation systems. In Ehrig et al. [EHRT08], pages 305–320.
- [KKR06] H. Kastenbergh, A. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. In *FMOODS*, LNCS 4037, pages 186–201. Springer, 2006.
- [Kön09] B. König. Case study: Leader election. <http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009verification.pdf>, 2009.
- [KR08] A. Kleppe and A. Rensink. On a graph-based semantics for UML class and object diagrams. In *GT-VMT*, volume 10 of *Electronic Communications of the EASST*, 2008.
- [LV01] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN*, pages 80–102. Springer, 2001.
- [McK81] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [RD06] A. Rensink and D. Distefano. Abstract graph transformation. *ENTCS*, 157(1):39–59, 2006.
- [RDHI03] Robby, M. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. *ENTCS*, 89(3), 2003.
- [Ren03a] A. Rensink. The GROOVE simulator: A tool for state space generation. In *AGTIVE*, LNCS 3062, pages 479–485. Springer, 2003.
- [Ren03b] A. Rensink. Towards model checking graph grammars. In *AVoCS*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160. University of Southampton, 2003.
- [Ren04] A. Rensink. Canonical graph shapes. In *ESOP*, LNCS 2986, pages 401–415. Springer, 2004.
- [Ren07] A. Rensink. Isomorphism checking in GROOVE. In *GraBaTs*, Electronic Communications of the EASST, 2007.

- [RN08] S. Rieger and T. Noll. Abstracting complex data structures by hyperedge replacement. In Ehrig et al. [EHRT08], pages 69–83.
- [Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
- [RZ09a] A. Rensink and E. Zambon. A type graph model for Java programs. In *FMOODS/FORTE 2009*, LNCS 5522, pages 237–242. Springer, 2009.
- [RZ09b] A. Rensink and E. Zambon. A type graph model for Java programs. Technical Report TR-CTIT-09-01, University of Twente, February 2009.
- [RZ10] A. Rensink and E. Zambon. Neighbourhood abstraction in GROOVE. In de Lara and Varro [dLV10].
- [RZ12a] A. Rensink and E. Zambon. Pattern-based graph abstraction. In *ICGT*, LNCS 7562, pages 66–80. Springer, 2012.
- [RZ12b] A. Rensink and E. Zambon. Pattern-based graph abstraction (extended version). Technical report, University of Twente, October 2012.
- [SBG10] M. Striewe, M. Balz, and M. Goedicke. Enabling graph transformations on program code. In *GraBaTs (Pre-Proceedings)*, number WP 10-06 in CTIT Workshop Proceedings, Enschede, 2010. CTIT.
- [SL08] C. Spemann and M. Leuschel. ProB gets Nauty: Effective symmetry reduction for B and Z models. In *TASE*, pages 15–22. IEEE Computer Society, 2008.
- [SRK06] R. Smelik, A. Rensink, and H. Kastenbergh. Specification and construction of control flow semantics. In *VL/HCC*, pages 65–72. IEEE Computer Society Press, September 2006.
- [SRW98] S. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.
- [SRW02] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
- [Sta10] T. Staijen. *Graph-based Specification and Verification for Aspect-Oriented Languages*. PhD thesis, University of Twente, June 2010.
- [Tae04] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *AGTIVE*, LNCS 3062, pages 446–453. Springer, 2004.
- [TLSB07] E. Turner, M. Leuschel, C. Spemann, and M. Butler. Symmetry reduced model checking for B. In *TASE*, pages 25–34. IEEE Computer Society, 2007.
- [Tur89] A. Turing. The early British computer conferences. chapter Checking a large routine, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.
- [VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, pages 3–12, 2000.
- [Zam10] E. Zambon. Using graph transformations and graph abstractions for software verification. In *ICGT*, LNCS 6372, pages 416–418. Springer, 2010.
- [ZR11] E. Zambon and A. Rensink. Using graph transformations and graph abstractions for software verification. *Electronic Communications of the EASST*, 38:1–13, 2011.

- [ZR12] E. Zambon and A. Rensink. Graph subsumption in abstract state space exploration. In *GRAPHITE*, volume 99 of *EPTCS*, pages 35–49, 2012.

Titles in the IPA Dissertation Series since 2007

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. Calamé.** *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. Kastenbergh. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.*

Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäuser. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty

of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Morali. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

- A.N. Tamalet.** *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20
- H.J.S. Basten.** *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21
- M. Izadi.** *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22
- L.C.L. Kats.** *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23
- S. Kemper.** *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24
- J. Wang.** *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25
- A. Khosravi.** *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01
- A. Middelkoop.** *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02
- Z. Hemel.** *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03
- T. Dimkov.** *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04
- S. Sedghi.** *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05
- F. Heidarian Dehkordi.** *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06
- K. Verbeek.** *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07
- D.E. Nadales Agut.** *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08
- H. Rahmani.** *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09
- S.D. Vermolen.** *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10
- L.J.P. Engelen.** *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11
- F.P.M. Stappers.** *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12
- W. Heijstek.** *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13
- C. Kop.** *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14
- A. Osaiweran.** *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15
- W. Kuijper.** *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16
- H. Beohar.** *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03