Verification of Well-Structured Graph Transformation Systems

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

genehmigte Dissertation

von

Jan Stückrath
aus
Gießen

1. Gutachterin: Prof. Dr. Barbara König
2. Gutachter: Prof. Dr. Giorgio Delzanno
Tag der mündlichen Prüfung: 2. März 2016

Abstract

The aim of this thesis is the definition of a high-level framework for verifying concurrent and distributed systems. Verification in computer science is challenging, since models that are sufficiently expressive to describe real-life case studies suffer from the undecidability of interesting problems. This also holds for the graph transformation systems used in this thesis. To still be able to analyse these system we have to restrict either the class of systems we can model, the class of states we can express or the properties we can verify. In fact, in the framework we will present, all these limitations are possible and each allows to solve different verification problems.

For modelling we use graphs as the states of the system and graph transformation rules to model state changes. More precisely, we use hypergraphs, where an edge may be incident to an arbitrary long sequence of nodes. As rule formalism we use the single pushout approach based on category theory. This provides us with a powerful formalisms that allows us to use a finite set of rules to describe an infinite transition system.

To obtain decidability results while still maintaining an infinite state space we use the theory of well-structured transition systems (WSTS), the main source of decidability results in the infinite case. We need to equip our state space with a well-quasi-order (wqo) which is a simulation relation for the transition relation (this is also known as compatibility condition or monotonicity requirement). If a system can be seen as a WSTS and some additional conditions are satisfied, one can decide the coverability problem, i.e., the problem of verifying whether, from a given initial state one can reach a state that covers a final state, i.e. is larger than the final state with respect to a chosen order. This problem can be used for verification by giving a finite set of minimal error states that represent an infinite class of erroneous states (i.e. all larger states). By checking whether one of these minimal states is coverable, we verify whether an error is reachable. The theory of WSTS provides us with a generic backwards algorithm to solve this problem.

For graphs we will introduce three orders, the minor ordering, the subgraph ordering and the induced subgraph ordering, and investigate which graph transformation systems form WSTS with these orders. Since only the minor ordering is a wqo on all graphs, we

will first define so-called $\mathcal{Q}$-restricted WSTS, where we only require that the chosen order is a wqo on the downward-closed class $\mathcal{Q}$. We examine how this affects the decidability of the coverability problem and present appropriate classes $\mathcal{Q}$ such that the subgraph ordering and induced subgraph ordering form $\mathcal{Q}$-restricted WSTS. Furthermore, we will prove the computability of the backward algorithm for these $\mathcal{Q}$-restricted WSTS. More precisely, we will do this in the form of a framework and give necessary conditions for orders to be compatible with this framework. For the three mentioned orders we prove that they satisfy these conditions. Being compatible with different orders strengthens the framework in the following way: On the one hand error specifications have to be invariant wrt. the order, meaning that different orders can describe different properties. On the other hand, there is the following trade-off: coarser orders are wqos on larger sets of graphs, but fewer GTS are well-structured wrt. coarse orders (analogously the reverse holds for fine orders).

Finally, we will present the tool UNCOVER which implements most of the theoretical framework defined in this thesis. The practical value of our approach is illustrated by several case studies and runtime results.

Acknowledgements

This thesis is the result of my work in the theoretical computer science group at the University of Duisburg-Essen. Before I present my research I want to thank those who supported me – directly or indirectly – with this subject.

First of all I want to thank my supervisor Prof. Dr. Barbara König for her support and guidance while working on the research presented in this thesis. I am grateful that she was always there when I had questions and shared her wide knowledge in different fields of theoretical computer science with me, which lead to the many fruitful discussions we had. In fact, the idea of combining graph transformation and well-structured transition systems in the way presented in this thesis originates in a paper published by her and Salil Joshi. I further pursued this idea to finally define a general framework.

I also want to thank my colleagues at the university Christoph Blume, Sander Bruggink, Mathias Hülsbusch, Henning Kerstan, Sebastian Küpper and Dennis Nolte for the great working atmosphere and interesting discussions about a lot of different topics. A special thanks goes to my officemate Sander Bruggink with whom I had the most enlightening discussions about programming, graph problems and mathematical "tricks" in general. I also give my thanks to Marvin Heumüller, a student who I did not meet, but who provided the first quick prototype of what would later become the tool Uncover.

Furthermore, I thank the computer scientists with which I have collaborated in several publications. In particular I thank Giorgio Delzanno for our discussions about methods of verification and how these can be applied in the context of graph transformation.

Last but not least I am grateful to my family and friends for their understanding and support beyond my research. It is a great pleasure to know and be friends with such wonderful people.

Contents

Abstract iii

Acknowledgements v

1. Introduction 1
1.1. Graph Transformation and Verification . . . 1
1.2. Contributions by Publication . . . 4
1.3. Thesis Outline . . . 5

2. Transition Systems 9
2.1. Basic notation . . . 9
2.2. General Transition Systems . . . 12
2.3. Well-Structured Transition Systems . . . 13
2.4. Q-Restricted Well-Structured Transition Systems . . . 17

3. Graph Transformation Systems 23
3.1. Category Theory . . . 24
3.2. Category of Graphs . . . 27
3.3. Graph Transformation Systems . . . 30
3.4. Construction of Pushouts . . . 34
3.5. Construction of Pushout Complements . . . 36

4. Decidability Results for Graph Transformation 45
4.1. Restrictions on the Deletion and Creation of Nodes . . . 46
4.2. Non-Deleting Graph Transformation Systems . . . 50
4.3. General Graph Transformation Systems with Minor Rules . . . 56
4.4. Relabelling Rules . . . 62

4.5. Overview . . . 68

5. Well-Structured Graph Transformation Systems 71
5.1. Minor Ordering . . . 72
5.2. Subgraph Ordering . . . 81
5.3. Induced Subgraph Ordering . . . 88
5.4. Further Interesting Orders . . . 94

6. Backward Analysis 97
6.1. A General Backward Procedure . . . 97
6.2. Minor Ordering . . . 111
6.3. Subgraph Ordering . . . 114
6.4. Induced Subgraph Ordering . . . 119
6.5. Optimizations . . . 123
6.6. Universally Quantified Rules . . . 127
6.7. Summary . . . 135

7. Implementation and Case Studies 141
7.1. The Uncover Tool . . . 141
7.2. Termination Detection . . . 149
7.3. Leader Election . . . 152
7.4. Access Rights Management . . . 154
7.5. Dining Philosophers . . . 158
7.6. Public-Private Server Communication . . . 159

8. Conclusion 163
8.1. Summary . . . 163
8.2. Related Work . . . 164
8.3. Future Work . . . 166

A. Related Formalism 171
A.1. Petri Nets . . . 171
A.2. Turing Machines . . . 172
A.3. Minsky Machines (Two-Counter Machines) . . . 173

B. Proofs of Chapter 3 175
B.1. Proofs of Section 3.4 . . . 175
B.2. Proofs of Section 3.5 . . . 178

C. Proofs of Chapter 6 191
C.1. Proofs of Section 6.1 . . . 191
C.2. Proofs of Section 6.2 . . . 192

C.3. Proofs of Section 6.6 198

Bibliography **207**

List of Symbols **221**

Index **225**

Chapter 1
Introduction

1.1. Graph Transformation and Verification

Verification in computer science is a challenging task which can be performed at very different abstraction levels ranging from hardware verification, over program verification and verified compilers to proving correctness of protocols. Substantial research is performed on all of these abstraction levels and each level is necessary to fully prove the correctness and reliability of a complex computer system. Of course, many verification problems are undecidable but by restricting to verifiable models, by employing algorithms without guaranteed termination or by using over-approximations, good results can be achieved.

In my research I focused on the highest abstraction level, i.e. the verification of protocols or dynamic systems in general. One possibility to describe these systems are *graphs* and *graph transformation rules* [Roz97], called graph transformation systems (GTS). Graphs are here used to model the current state of a system and graph transformation rules are used to model state changes. Graph transformation rules are effectively a transformation schema which can be applied to possibly infinitely many graphs and can therefore finitely represent infinitely large transition systems.

Graph transformation systems have been used for verification in many different application areas. In [KMP00; KMP02; KMP05] the authors model role-based access control policies by using graphs and with the goal of proving safety properties. The same goal is pursued in [ADR09] for directory-based consistency protocols and in [PE02] for safety-critical systems (using invariants). Ad hoc networks and routing protocols are modelled and analysed in [SWJ08; DSZ10; DSZ11]. For analysing pointer-manipulating programs, data structures such as heaps are abstracted by graph transformation systems in [HJ+15a]. In a wider context, graphs have also been used for analysing vulnerability of computer networks [PS98; AWK02] and as so-called attack graphs representing attack

possibilities against such networks [OBM06]. There are also approaches to use graphs and graph transformations for programming [AE+99; HP01] which lead to a language for graph programs (GP) [Ste07] which was further extended to GP 2 in [Plu12]. Monadic second-order logic has also been defined for graphs [Cou90].

In our analysis approach we use hypergraphs, a generalization of directed graphs, where each edge need not connect only two nodes, but can be connected to an arbitrarily long, but finite sequence of nodes. Furthermore, for rewriting we use the so-called *single pushout approach (SPO)* [EH+97] based on category theoretical constructions using partial morphisms, i.e. partial mappings from graphs to graphs. In this approach deletion is preferred over preservation, i.e. when removing a node all incident edges are removed as well, even if their removal was not explicitly stated in the rule. For comparison, in the *double pushout approach (DPO)* [CM+97] such rule applications would be blocked. Both approaches are thoroughly covered in the Handbook of Graph Grammars and Computing by Graph Transformation that spans over three volumes [Roz97; EE+99; EK+99].

The basic problem we will build our framework on is the *coverability problem* wrt. some order $\preceq$, i.e. given two graphs G, G_f, is there a graph G' reachable (by application of the rules) from G such that $G_f \preceq G'$? The main idea is to choose an appropriate order $\preceq$ such that an error in the system can be described by a set of minimal graphs. The order $\preceq$ must hence preserve the error in the sense that if some graph G is erroneous, then all $G' \succeq G$ are erroneous as well. Assume for instance a graph transformation system modelling a multi-user access rights management system, where there are users and objects, and users can have read or write access to objects. The graph shown in Figure 1.1 shows an error graph, since two users having write access to the same object is an undesired state in the system. Provided the use of an appropriate order, every graph "containing" this graph, i.e. every larger graph, is also erroneous, since the problem of two write access rights to the same object persists. We can prove that no error is reachable – and thus the correctness of the system – by showing that none of the minimal error graphs is coverable from the initial graph (or graphs). Note that this approach can also be used to model desired states and check whether they are coverable. However, the coverability problem only states that they *can* be covered, although usually one is interested in checking whether some states *will* eventually be covered on all possible paths.

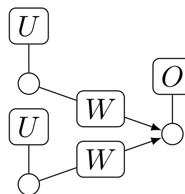


Figure 1.1.: An undesired state in a multi-user system

Unfortunately, it is a well-known result that the coverability problem is undecidable for graph transformation systems in general (see for instance [BD+12b]). To still obtain decidability results while keeping a possibly infinite state space we use the theory of *well-structured transition systems (WSTS)* [AČ+96; FS01]. These transition systems $\langle S, \Rightarrow, \preceq \rangle$ are equipped with a partial order $\preceq$ on the set of states S which has to be a well-quasi-order and a simulation relation for the transition relation, i.e. if there are states with $s_1 \Rightarrow s_2$ and $s_1 \preceq t_1$, then there is a state t_2 with $t_1 \Rightarrow^* t_2$ and $s_2 \preceq t_2$ (this is called the compatibility condition). A *well-quasi-order (wqo)* is a transitive and reflexive relation where there is no infinite, strictly decreasing sequence $s_1 \succ s_2 \succ \ldots$ and no infinite antichain, i.e. a sequence of pairwise incomparable elements. Direct consequences of this property are that every upward closed set wrt. to a wqo is finitely representable by its minimal elements and that every infinite increasing sequence $I_1 \subseteq I_2 \subseteq \ldots$ of upward closed set I_i becomes stationary. Together, these properties give rise to a backward algorithm for solving the coverability problem for WSTS. The algorithm is guaranteed to terminate due to $\preceq$ being a wqo and is correct due to $\preceq$ being a simulation relation for $\Rightarrow$. There are large classes of infinite-state systems that are well-structured and therefore can benefit from this algorithm, for instance (unbounded) Petri nets and certain lossy systems.

The main contribution of this thesis is the investigation of well-structured graph transformation systems and the application of the backward algorithm to these. This has first been done by König and Joshi in [JK08] using the minor ordering. A graph G is a *minor* of a graph G' if G can be obtained by a sequence of node deletions, edge deletions and edge contractions. The *contraction* of an edge deletes the edge and merges its incident nodes according to an arbitrary partition on these nodes. The minor ordering is a well-quasi-order on all graphs [RS04; RS10], but only forms a WSTS if the GTS is lossy [JK08], i.e. messages (modelled via edges) can be removed by the GTS. In this thesis we present a general framework (previously published in [KS14b]) which is compatible with different orders satisfying certain constraints. So far we have shown that the minor ordering, the subgraph ordering and the induced subgraph ordering are compatible with the framework, but additional orders such as the topological minor ordering or the induced minor ordering [FHR12] are conceivable. Using multiple orders enhances the approach in two ways. On the one hand error specifications have to be invariant wrt. the order, meaning that different orders can describe different properties. On the other hand, there is the following trade-off: coarser orders are wqos on larger sets of graphs, but fewer GTS are well-structured wrt. coarse orders (analogously the reverse holds for fine orders). For instance, we will later see that the minor ordering is a well-quasi-order on all graphs, but not well-structured wrt. all graph transformation systems, whereas the subgraph ordering is a well-quasi-order only on a restricted class of graphs, but well-structured wrt. all graph transformation systems. If all graphs reachable by applying rules of a graph transformation system are in the class of graphs $\mathcal{Q}$ on which the chosen order is a well-quasi-order, then the coverability problem is decidable. How-

ever, if this is not the case, we can not simply ignore rule applications that generate graphs outside of $\mathcal{Q}$, since this will violate the compatibility condition. In that case we can still use the backward algorithm to obtain useful partial decidability results or apply the backward algorithm without restricting to $\mathcal{Q}$. In the latter case we no longer have a guarantee of termination, but can fully decide coverability for every terminating instance. We will make these considerations precise by introducing *$\mathcal{Q}$-restricted* WSTSs, where the order need only be a wqo on $\mathcal{Q}$, a subset of the state space. In general, one wants $\mathcal{Q}$ to be as large as possible to obtain stronger statements. We will also present the tool UNCOVER, which implements our framework for graph transformation systems, and present some case studies we were able to verify using the tool.

1.2. Contributions by Publication

The following list shows the main publications which lead to this thesis. Their content has been revised and extended to form the basis of several chapters.

[BD+12b] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. “On the Decidability Status of Reachability and Coverability in Graph Transformation Systems”. In: *Proceedings of RTA ’12*. Vol. 15. LIPIcs. Schloss Dagstuhl – Leibniz Center for Informatics, 2012, pp. 101–116.

[DS14b] Giorgio Delzanno and Jan Stückrath. “Parameterized Verification of Graph Transformation Systems with Whole Neighbourhood Operations”. In: *Proceedings of RP ’14*. Ed. by Joël Ouaknine, Igor Potapov, and James Worrell. Vol. 8762. LNCS. Springer, 2014, pp. 72–84.

[HJ+11] Marvin Heumüller, Salil Joshi, Barbara König, and Jan Stückrath. “Construction of Pushout Complements in the Category of Hypergraphs”. In: *Selected Revised Papers from the Workshop on Graph Computation Models (GCM 2010)*. Vol. 39. Electronic Communications of the EASST. 2011.

[KS12b] Barbara König and Jan Stückrath. “Well-Structured Graph Transformation Systems with Negative Application Conditions”. In: *Proceedings of ICGT ’12*. Vol. 7562. LNCS. Springer, 2012, pp. 89–95.

[KS14b] Barbara König and Jan Stückrath. “A General Framework for Well-Structured Graph Transformation Systems”. In: *Proceedings of CONCUR ’14*. Ed. by Paolo Baldan and Daniele Gorla. Vol. 8704. LNCS. Springer, 2014, pp. 467–481.

[KS16] Barbara König and Jan Stückrath. “Well-Structured Graph Transformation Systems”. In: *Information and Computation* (2016). Accepted for publication.

[Stü15] Jan Stückrath. “Uncover: Using Coverability Analysis for Verifying Graph Transformation Systems”. In: *Proceedings of ICGT ’15*. Ed. by Francesco Parisi-Presicce and Bernhard Westfechtel. Vol. 9151. LNCS. Springer, 2015, pp. 266–274.

Note that some of these publications are also available as extended versions [BD+12a; DS14a; KS14a; KS12a], containing proofs also presented in this thesis.

With [BD+12b] we published a survey on the current results for reachability and coverability for graph transformation systems. Both problems are undecidable in the general case, but we presented several subclasses for which these problems become decidable. One of these subclasses are graph transformation systems with minor rules. Minor rules can be either node-deleting (rules deleting a single node), edge-deleting (rules deleting a single edge) or edge-contracting (rules deleting a single edge and merging some of its incident nodes). If a graph transformation system contains edge-contraction rules for each label, the coverability problem becomes decidable, and if it contains all minor rules, the reachability problem is decidable as well. We will later obtain both results as a consequence of applying the theory of well-structured transition systems to graph transformation systems.

In [KS12b] we further developed the idea of using well-structured transition systems based on the minor ordering, first introduced in [JK08]. We added negative application conditions and presented necessary conditions for correctness as well as an adapted backward search. In [KS14b; KS16] we further developed the idea and defined a general framework which is also able to handle other orders, such as the subgraph and the induced subgraph ordering. However, we had to introduce so-called Q-restricted well-structured transition systems, since not every order is a well-quasi-order on all graphs, an important condition for WSTS. We had to weaken this condition to also allow finer orders, which of course also affects the decidability of coverability for these systems. We obtained interesting partial decidability results which allow us to analyse these systems as well. In [DS14b] we added so-called universally quantified rules to our framework and adequately modified the backward search. Most of the framework is implemented in the tool Uncover [Stü15] with which we analysed several case studies in several papers. The algorithmic enumeration of pushout complement, essential for applying rules backward, has already been investigated in [HJ+11].

1.3. Thesis Outline

In the following I will briefly summarize the content of each chapter and give some reading advice.

Chapter 2 – Transition Systems In this chapter we will first fix some basic notations for mathematical clarification of the subsequent chapters. Furthermore, we will

define the notion of ordinary as well as well-structured transition systems. The latter will be extended to so-called Q-restricted well-structured transition systems for which we will later show that graph transformation systems form an instance of these. We also define the reachability and coverability problems and recall how well-structuredness affects the decidability of these problems.

Readers familiar with the theory of well-structured transition systems can skip most of this chapter and can limit themselves to Section 2.4 where we define Q-restricted well-structured transition systems and state decidability results in Theorem 2.21.

Chapter 3 – Graph Transformation Systems This chapter first introduces some basic notions of category theory, such as pushouts and pushout complements, and proves abstract properties for these. We then introduce two categories for graphs and define graph transformation systems as well as the application of rules based on these categories. Finally we present constructions for pushouts and pushout complements in both categories. For ease of reading, long proofs have been moved to Appendix B.

Readers familiar with SPO graph rewriting can skip this chapter. However, the construction of pushouts and pushout complements will be used in some proofs of subsequent chapters.

Chapter 4 – Decidability Results for Graph Transformation An overview of decidability of reachability and coverability for graph transformation systems is given in this chapter. Since these problems are undecidable in general, we investigate subclasses such as systems without node deletion and creation, non-deleting graph transformation systems and systems which contain special rules. Furthermore, we also examine simple relabelling GTS. In Appendix A we give definitions of other well-known computation models used for the reductions in this chapter.

This chapter relies on the notions presented in Chapter 3. Since it is an overview chapter, it does not contain any notions or results used by later chapters.

Chapter 5 – Well-Structured Graph Transformation Systems In this chapter we will prove that graph transformation systems can form (Q-restricted) well-structured transition systems and introduce a general framework for such systems. We investigate three main orders which we will integrate into our framework, the minor ordering, the subgraph ordering and the induced subgraph ordering. Further interesting orders are discussed at the end.

This chapter uses some definitions of Chapters 2 and 3 and is essential for the understanding of Chapter 6.

Chapter 6 – Backward Analysis The main decidability procedures of this thesis are defined in this chapter. Here we introduce a specialization of the general backward search for well-structured transition systems for graph transformation systems. We also give necessary and sufficient correctness conditions and show that the minor ordering, subgraph ordering and induced subgraph ordering all satisfy these conditions. Furthermore, we introduce some optimizations of the backward search and define so-called universally quantified rules for use with the subgraph ordering. For ease of reading, some of the proofs of this chapter have been moved to Appendix C.

This chapter extensively uses results and definitions of Chapters 2, 3 and 5.

Chapter 7 – Implementation and Case Studies To prove practicality of our approach, we implemented the framework defined in Chapters 5 and 6 in the tool UNCOVER. Basic functionality and design choices of this tool are described in this chapter. Furthermore, five different case studies using the minor and subgraph orderings are presented to measure performance of the tool.

Although this chapter can be read independently, a good understanding of the framework in Chapters 5 and 6 is helpful.

Chapter 8 – Conclusion Finally, in this chapter we summarize our approach and compare with other existing approaches using well-structured transition systems. We also discuss some interesting points for future extension of our framework.

For better readability a list of symbols and an index is provided at the end of this thesis.

Chapter 2
Transition Systems

In the first part of this chapter I will fix some basic definitions and notations which will be needed in this and the following chapters. The second part describes classical well-structured transition systems and I will introduce a generalisation of these, so-called Q-restricted well-structured transition systems.

2.1. Basic notation

2.1.1. Sets and Relations

We use $\mathbb{N}_0$ to denote the set of *natural numbers* $\{0, 1, 2, \ldots\}$, including zero, and use $\mathbb{N}$ to denote the natural numbers without zero. For distinction, we use $A \subseteq B$ to denote that A is a subset of B or equal to B and we use $A \subset B$ to denote that A is a strict subset of B, i.e $A \neq B$. For any $n \in \mathbb{N}_0$ we use A^n to denote the *Cartesian product* $A \times A \times \cdots \times A$ of length n.

In this work *binary relations* $R \subseteq A \times A$ are of special interest and will later be used to describe equivalences, orders and transition relations. A binary relation is called *reflexive* if $\langle a, a \rangle \in R$ holds for every $a \in A$, *symmetric* if for every $\langle a, b \rangle \in R$ it also holds that $\langle b, a \rangle \in R$, *antisymmetric* if for every $a, b \in A$ with $\langle a, b \rangle \in R$ and $\langle b, a \rangle \in R$ it holds that $a = b$, and *transitive* if for every $a, b, c \in A$ with $\langle a, b \rangle \in R$ and $\langle b, c \rangle \in R$ it holds that $\langle a, c \rangle \in R$.

We will often write relations in infix notation, i.e. we will write $a \mathrel{R} b$ if $\langle a, b \rangle \in R$.

Definition 2.1 (Equivalence)**.** Let A be a set. A relation $\equiv \subseteq A \times A$ is an *equivalence relation* if it is reflexive, transitive and symmetric. We use $[a]_\equiv$ to denote the *equivalence class* of a, i.e. $[a]_\equiv = \{a' \mid a \equiv a'\}$. We write $A/{\equiv}$ for the *quotient set*, the set of all equivalence classes of $\equiv$ on A.

Let $R \subseteq A \times A$ be a binary relation. We use $\overline{R}$ to denote the *equivalence closure* of R, i.e. the smallest equivalence containing R.

The equivalence closure of R can be obtained by performing the reflexive closure, the symmetric closure and the transitive closure, in this order. It always exists and is unique.

2.1.2. Sequences

Given some set A we use A^* to denote the set of all *finite sequences* consisting of elements of A. For a finite sequence usually we write $s_1, s_2, \ldots, s_n$ or alternatively $s_1 \sim s_2 \sim \ldots \sim s_n$ to denote that two consecutive elements are related by some binary relation $\sim$. If the elements of the sequence are simple objects, e.g. numbers or symbols, we use $s_1 s_2 \ldots s_n$ as an abbreviation of the sequence $s_1, s_2, \ldots, s_n$. We use $\mathbf{s}[i] = s_i$ to denote the i-th element of a sequence $\mathbf{s}$ as well as $|\mathbf{s}|$ to denote the *length* of $\mathbf{s}$. We write infinite sequences as $s_1, s_2, \ldots$ or use one of the alternative notations in the same way. Note that in some cases for convenience we will start numbering sequences with zero instead of one.

2.1.3. Functions

We use $f\colon A \to B$ to denote a (total) *function* mapping each element of $a \in A$ to some element of $f(a) \in B$. We call A the *domain* and B the *codomain* or *image* of f. Similarly we call $f(a)$ the *image* of a and say that $b \in B$ has a *preimage* under f if there exists an $a \in A$ with $f(a) = b$. A function is *injective* if $a \neq a'$ implies $f(a) \neq f(a')$ for all $a, a' \in A$ and it is *surjective* if every $b \in B$ has a preimage. If a function is injective and surjective, it is *bijective*.

If not stated otherwise we extend a function $f\colon A \to B$ directly to sequences $a_1 \ldots a_n \in A^*$ by $f(a_1 \ldots a_n) = f(a_1) \ldots f(a_n)$ and to tuples $\langle a_1, \ldots, a_n \rangle \in A^n$ by $f(a_1, \ldots, a_n) = \langle f(a_1), \ldots, f(a_n) \rangle$.

2.1.4. Quasi-Orders and Well-Quasi-Orders

Well-structured transition systems extensively use the theory of well-quasi-orders and upward-closed sets, as defined below.

Definition 2.2 (Quasi-order and partial order)**.** A *quasi-order* $\preceq \subseteq A \times A$ is a transitive, reflexive relation. A *partial order* is a quasi-order which is also antisymmetric.

Note that quasi-orders are sometimes also called preorders.

Definition 2.3 (Well-quasi-order)**.** A quasi-order $\preceq$ is a *well-quasi-order (wqo)* if for any infinite sequence $a_1, a_2, a_3, \ldots$ of elements of A, there exist indices $i < j$ with $a_i \preceq a_j$.

Well-quasi-orders give rise to interesting properties, especially with respect to upward-closed sets.

Definition 2.4 (Upward and downward closure)**.** Let A be a set and let $\preceq$ be a quasi-order on A. The *upward closure* of a set $B \subseteq A$ is the set $\uparrow B = \{a \in A \mid \exists b \in B \colon b \preceq a\}$. We call a set B *upward-closed* if $B = \uparrow B$. A *basis* of an upward-closed set B is a set C such that $B = \uparrow C$.

Analogously, the *downward closure* of a set $B \subseteq A$ is the set $\downarrow B = \{a \in A \mid \exists b \in B \colon a \preceq b\}$. We call B *downward-closed* if $B = \downarrow B$. Finally, a *basis* of a downward-closed set B is a set C such that $B = \downarrow C$.

Due to Higman, a multitude of properties and definitions equivalent to Definition 2.3 exist.

Proposition 2.5 ([Hig52])**.** *Let A be a set and let $\preceq$ be a quasi-order on A. The following statements are equivalent:*

(i) Every upward-closed subset of A has a finite basis, also called the finite basis property*.*

(ii) For any infinite ascending sequence of upward-closed sets $I_1 \subseteq I_2 \subseteq I_3 \subseteq \dots$ there exists an index $k \in \mathbb{N}$ such that $I_i = I_{i+1}$ for all $i \geq k$.

(iii) If B is a subset of A, then there exists a finite set B_0 such that $B_0 \subseteq B \subseteq \uparrow B_0$.

(iv) Every infinite sequence of elements of A has an infinite ascending subsequence.

(v) The quasi-order $\preceq$ is a well-quasi-order.

(vi) There exists neither an infinite strictly descending sequence of elements of A, nor an infinite sequence of pairwise incomparable elements of A (a so-called anti-chain).

In the literature case (vi) can often be found as an alternative definition of well-quasi-orders. Of special interest are the cases (i) and (ii) which we will exploit later to obtain decidability results for various problems.

There are two consequences of the properties above which are also of interest for this work. The first is a generalization of Dickson's Lemma and states that the lifting of a collection of wqos to a quasi-order on tuples is also a wqo. This was proven for natural numbers and $\leq$ by Dickson in [Dic13], but directly extends to general wqos.

Lemma 2.6. *Let A_i with $1 \leq i \leq k$ be a (finite) family of sets and let $\preceq_i \subseteq A_i \times A_i$ with $1 \leq i \leq k$ be a family of well-quasi-orders. Then $\preceq$ is a well-quasi-order on $A_1 \times \ldots \times A_k$ where $\langle a_1, \ldots, a_k \rangle \preceq \langle b_1, \ldots, b_k \rangle$ if and only if $a_i \preceq_i b_i$ for all $1 \leq i \leq k$.*

This lemma can be easily proven by using case (iv) of Proposition 2.5. Let $s = a_1, a_2, a_3, \ldots$ be an infinite sequence, where $a_i = \langle a_{i,1}, \ldots, a_{i,k} \rangle$ for every i. We can form the sequence $a_{1,1}, a_{2,1}, \ldots$ by taking only the first component of every tuple. Since $\preceq_1$ is a wqo, there is an infinite ascending subsequence $a_{j_1,1} \preceq_1 a_{j_2,1} \preceq_1 \ldots$ leading to an infinite sequence $s_1 = a_{j_1}, a_{j_2}, \ldots$, which is a subsequence of s and increasing in the first component. This step can be repeated for every component proving that s has an infinite ascending subsequence and which in turn shows that $\preceq$ is a wqo by Proposition 2.5.

Another interesting result is that sequences of well-quasi-ordered elements can also be well-quasi-ordered by a "disconnected subsequence" relation. This generalizes Higman's Lemma [Hig52] which states the result for languages over a finite alphabet.

Lemma 2.7. *Let $\preceq$ be a well-quasi-order on a set A. The order $\preceq^*$ is a well-quasi-order on A^*, where $a_1 \ldots a_k \preceq^* b_1 \ldots b_\ell$ with $k \leq \ell$ if and only if there are $j_1 < j_2 < \ldots < j_k$ such that $a_i = b_{j_i}$ for $1 \leq i \leq k$.*

This lemma can be proven by the same ideas used for the previous lemma. The proof and additional similar properties can be found in [SS12].

2.2. General Transition Systems

A transition system is generated naturally when modelling state-based systems, such as algorithms or protocols. Our main focus lies on the analysis of infinite state systems, for which we will address reachability problems.

Definition 2.8 (Transition system)**.** A *transition system* is a pair $\langle S, \Rightarrow \rangle$, where S is a (possibly infinite) set of states and $\Rightarrow \subseteq S \times S$ is the transition relation.

In the following we will often be interested in arbitrary (but finitely) long sequences of transitions. For this we use $\Rightarrow^*$ to denote the transitive closure of $\Rightarrow$, i.e. $s \Rightarrow^* s'$ if and only if there exists a sequence of transitions $s \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n \Rightarrow s'$. Moreover, as an extension of $\Rightarrow$ to sets of states, we introduce successor and predecessor sets. For any $I \subseteq S$ the set $\mathit{Succ}(I) = \{s' \in S \mid s \in I \wedge s \Rightarrow s'\}$ is the set of direct successors of I and $\mathit{Pred}(I) = \{s' \in S \mid s \in I \wedge s' \Rightarrow s\}$ is the set of direct predecessors of I. Finally, we denote the sets of indirect successors and predecessors by $\mathit{Succ}^*(I) = \{s' \in S \mid s \in I \wedge s \Rightarrow^* s'\}$ and $\mathit{Pred}^*(I) = \{s' \in S \mid s \in I \wedge s' \Rightarrow^* s\}$, respectively. The problem of computing indirect successors is often called the reachability problem.

Definition 2.9 (Reachability problem)**.** Let $\langle S, \Rightarrow \rangle$ be a transition systems. The *reachability problem* is the question whether, given two states $s, s' \in S$, does $s \Rightarrow^* s'$ hold, i.e. is $s' \in \mathit{Succ}^*(\{s\})$? If such a sequence of transitions exist we say that s' is *reachable* from s, or s can reach s'.

A problem similar to reachability is the coverability problem, where the state s' need not be reachable, but some other state "containing" s'. Effectively we want to check if some state of a set of states is reachable instead of just a particular state. This set of states is modelled by a quasi-order in the sense that a state is in the set if and only if it is larger than s' according to the order.

Definition 2.10 (Coverability problem)**.** Let $\langle S, \Rightarrow \rangle$ be a transition systems and let $\preceq \subseteq S \times S$ be a quasi-order. The *coverability problem* is the question whether, given two states $s, s' \in S$, does there exist a state s'' with $s \Rightarrow^* s''$ and $s' \preceq s''$? If such an s'' exists, we say that s' is *coverable* by s, or s can cover s'.

Reachability and coverability are crucial problems in the context of verification of safety properties in different formalisms, such as Petri nets [Mur89], the π-calculus [Mil82; SW01] or graph transformation systems [Roz97].

The reachability problems is decidable for basic Petri nets as first shown by Mayr in [May81] and revisited by Leroux in the context of vector addition systems [Ler11]. The decidability of the coverability problem has been shown by Karp and Miller [KM69] for Petri nets and by Rackoff for vector addition systems [Rac78]. In fact, Dufourd et al. showed that this result also holds for Petri nets with reset and transfer arcs, where reachability is undecidable, but both problems are undecidable if inhibitor arcs are used [DFS98].

Since the general π-calculus and unrestricted graph transformation systems are Turing-complete, reachability and coverability are both undecidable. For depth-bounded processes, a restriction of the π-calculus, Meyer showed the decidability of coverability [Mey09], in this context called control reachability problem. In this work we will show similar decidability results for fragments of graph transformation systems.

The majority of the results above come from the fact that the described systems, or fragments thereof, are so called well-structured transition systems. These systems naturally possess necessary properties for the existence of an algorithm for the coverability problem. Note that there are other interesting verification problems such as termination or boundedness for which the theory of well-structured transition systems may be used as well. However, this work will focus on the coverability problem and only briefly discuss extensions to other problems.

2.3. Well-Structured Transition Systems

In this section I will present the theory of well-structured transition systems as introduced independently by Finkel and Schnoebelen [FS01] as well as Abdulla et al. [AČ+96]. These well-structured transition systems equip a transition system with a well-quasi-order and require the transition system to be monotonic wrt. the wqo. Larger states have to be able to simulate smaller states in the sense that every transition from the smaller state can be imitated by a sequence of transitions from the larger state.

Definition 2.11 (Well-structured transition system)**.** Let S be a set of states and let $\preceq$ be a quasi-order. A *well-structured transition system (WSTS)* is a transition system $\langle S, \Rightarrow, \preceq \rangle$, where the following conditions hold:

Ordering: $\preceq$ is a well-quasi-order.

Compatibility: For all $s_1 \preceq t_1$ and transitions $s_1 \Rightarrow s_2$, there exists a sequence $t_1 \Rightarrow^* t_2$ of transitions such that $s_2 \preceq t_2$.

$$\begin{array}{ccc} t_1 & \Longrightarrow^* & t_2 \\ \curlyvee\!| & & \curlyvee\!| \\ s_1 & \Longrightarrow & s_2 \end{array}$$

Example 2.12. Petri nets [Mur89] are one of the most well-known formalisms for modelling concurrent systems and a widely used example of a well-structured transition system. It is easy to see that $\leq$ is a well-quasi-order on $\mathbb{N}_0$, since every upward-closed set of natural numbers has exactly one minimal element (see Proposition 2.5). By Dickson's Lemma (Lemma 2.6) it immediately follows that the usual order on markings is also a well-quasi-order. The compatibility condition is satisfied due to the even stronger monotonicity of Petri nets by which adding tokens to a marking does not deactivate a previously active transition. In fact, the transition fired in the step $s_1 \Rightarrow s_2$ can be fired from the marking t_1 to obtain a valid t_2 in just one step. This still holds when reset and transfer arcs are used, but inhibitor arcs violate the compatibility condition [DFS98], since they can be used to model negative application conditions.

2.3.1. A Backward Search for Solving Coverability

The two properties of Definition 2.11 can be used to obtain decidability for several problems. The most important wrt. to this work is the coverability problem and we will state necessary conditions for a backward algorithm to exist for this problem.

We observe that due to $\preceq$ being a well-quasi-order the finite basis property holds, i.e. every upward-closed set is finitely representable by its minimal elements (see Proposition 2.5 case (i)). Let $S' \subseteq S$ be an upward-closed set of states and let S'_f be a finite basis of S', i.e. $\uparrow S'_f = S'$. Although $\mathit{Pred}(\uparrow S'_f)$ is not necessarily an upward-closed set, we can prove that $\uparrow \mathit{Pred}(\uparrow S'_f) \subseteq \mathit{Pred}^*(\uparrow S'_f)$. For every $t \in \uparrow \mathit{Pred}(\uparrow S'_f)$ there is an $s \in \mathit{Pred}(\uparrow S'_f)$ with $s \preceq t$ and $s \Rightarrow s'$ for some $s' \in \uparrow S'_f$, thus, by the compatibility condition there exists a t' with $t \Rightarrow^* t'$ and $s' \preceq t'$. By transitivity t' is an element of $\uparrow S'_f$ implying $t \in \mathit{Pred}^*(\uparrow S'_f)$. In fact, by using this argument inductively we can show that $\mathit{Pred}^*(\uparrow S'_f)$ itself is upward-closed for every set S'_f.

This proves that the set of direct predecessors is finitely representable. However, for a backward algorithm to exist the predecessors have to be computable as well, i.e. a so-called effective pred-basis must exist.

Definition 2.13 (Effective pred-basis)**.** A WSTS has an *effective pred-basis* if there exists an algorithm accepting any state $s \in S$ and returning $pb(s)$, a finite basis of $\uparrow \mathit{Pred}(\uparrow \{s\})$.

As shown by Finkel and Schnoebelen as well as Abdulla et al. the existence of an effective pred-basis is one of two conditions which are together sufficient for the coverability problem to be decidable.

Theorem 2.14 ([FS01; AČ+96]). *The coverability problem is decidable for WSTS with an effective pred-basis and a decidable wqo* $\preceq$.

The decision procedure implied by Theorem 2.14 is an iterative backward search for which the correctness is guaranteed by the compatibility condition and termination is guaranteed by $\preceq$ being a wqo.

Algorithm 2.15 (General backward search for WSTS).
Input: A well-structured transition system $T = \langle S, \Rightarrow, \preceq \rangle$ with an effective pred-basis $pb()$ and a finite set of states $S' \subseteq S$.
Output: A finite basis of the set of all states from which a state of S' is coverable.

1: $W_{old} \leftarrow \emptyset$
2: $W_{new} \leftarrow S'$
3: **while** $\uparrow W_{old} \neq \uparrow W_{new}$ **do**
4: $\quad W_{old} \leftarrow W_{new}$
5: $\quad$ **for all** $s \in W_{old}$ **do**
6: $\quad\quad W_{new} \leftarrow W_{new} \cup pb(s)$
7: $\quad$ **end for all**
8: **end while**
9: **return** W_{old}

We observe that all steps in Algorithm 2.15 are *computable*, because the involved sets W_{new}, W_{old}, S' and $pb(s)$ are always finite. The condition in line 3 is decidable as well, since it is satisfied if and only if: for all $s \in W_{old}$ there is an $s' \in W_{new}$ with $s' \preceq s$ and for all $s \in W_{new}$ there is an $s' \in W_{old}$ with $s' \preceq s$. Note that this requires $\preceq$ to be decidable.

The *termination* of the algorithm is guaranteed by the fact that $\preceq$ is a wqo. In every iteration of the while-loop a new W_{new} is computed, which is larger or equal to the previous W_{new} (stored in W_{old}). Effectively, a sequence of working sets $W_1, W_2, W_3, \ldots$ is computed where $W_1 = S'$ and $W_{i+1} = W_i \cup pb(W_i)$ (we extended $pb()$ to sets of states in the straight-forward way). The sequence $\uparrow W_1, \uparrow W_2, \uparrow W_3, \ldots$ is increasing and becomes stationary due to case (ii) of Proposition 2.5 at some index $k \in \mathbb{N}$, thus, the condition in line 3 will be violated after at most k iterations.

A simple argument can be used to prove the *correctness* of this algorithm. We observe that if a state t_1 can cover a state $t \in S'$, then there exists a sequence $t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow t_m$ with $t \preceq t_m$, i.e. t_1 can cover t in m steps. Since $pb(t)$ is a finite basis of $\uparrow Pred(\uparrow\{t\})$ and $t_{m-1} \in Pred(\uparrow\{t\})$, we know that $t_{m-1} \in \uparrow pb(t)$. By iterating this argument we can

show that t_1 will be in the upward closure of a computed working set after at most m steps, thus, being also represented by the final working set.

On the other hand, for every state $s'_0 \in \uparrow W_*$, where W_* is the working set returned by Algorithm 2.15, there is a sequence of backward steps generating states $s \multimap s_n \multimap s_{n-1} \multimap \ldots \multimap s_1$ with $s_1 \preceq s'_1$ (see Figure 2.1). We use $s_{i+1} \multimap s_i$ to denote a backward step, i.e. $s_i \in pb(s_{i+1})$ holds. For every $1s \leq i \leq n$ it is guaranteed that there is an s'_{i+1} with $s_i \Rightarrow s'_{i+1}$ and $s_{i+1} \preceq s'_{i+1}$, where $s_{n+1} = s$ and $s'_{n+1} = s'$. By applying the compatibility condition at each backward step, we can show the existence of an s'' with $s'_1 \Rightarrow^* s''$ and $s \preceq s''$, as shown in Figure 2.1. Hence, s'_1 can cover s.

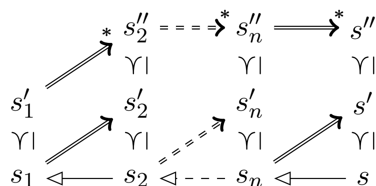


Figure 2.1.: Shows how the compatibility condition ensures the correctness of the backward search

Note that if we want to solve the coverability problem for some specific $s \in S$, we can stop the algorithm as soon as $s \in \uparrow W_i$ for some i. Since $\uparrow W_i \subseteq \uparrow W_{i+1}$ it is guaranteed that $s \in \uparrow W$ where W is the result returned by the algorithm.

Complexity of the Backward Search

The exact complexity of Algorithm 2.15 depends highly on the concrete WSTS to which it is applied. Although this field is not extensively researched yet, there are results of Schmitz and Schnoebelen [SS13] showing non-primitive recursive upper-bounds for termination and coverability and even some non-primitive recursive completeness results. However, in specific WSTS the complexity may be better, e.g. for vector addition systems [BG11]. Despite these results we will show in Chapter 7 that the runtimes in practice are better than expected.

2.3.2. Forward Approaches to Coverability

This work focuses mainly on the backward approach to solve coverability, which is pleasant because of its simplicity, but forward search based algorithms also exist. One of the first was the Karp and Miller algorithm for Petri nets [KM69], which introduces limit elements and accelerates paths in the reachability tree to achieve termination. The idea was extended to general well-structured transition systems in the form of the Expand, Enlarge and Check (EEC) algorithm [GRV06]. One problem of the forward algorithm is

that it uses not just upward-closed sets of states, but also downward-closed sets, which are not necessarily finitely representable, even for WSTS. This problem is solved by the introduction of a so-called adequate domain of limits, which have to be found for each concrete WSTS. However, there are approaches of Finkel and Goubault-Larrecq to determine these limits by ideal completion [FG09a; FG09b]. These algorithms have already been applied to the π-calculus [WZH10] and even to graph transformation systems [BK+13], but these techniques are currently still incomplete in the sense that they either do not guarantee termination or are approximating.

It is worth noting that the forward and backward algorithms are not fully comparable. The backward algorithm computes the upward-closed set of all states from which a given state is coverable, yielding a stronger result that just solving the coverability problem as defined in Definition 2.10. The analogous result of the forward algorithm would be the covering set, the downward-closed set of states which are coverable from a given set of states. However, this covering set is not necessary computable, even if coverability is decidable, as shown for depth-bounded systems [BK+13].

2.4. Q-Restricted Well-Structured Transition Systems

In Chapters 5 and 6 we will observe that the basic definition of well-structured transition systems, as described in the previous section, is not sufficient for defining a general framework for graph transformation systems. The problem is that we use different orders, which are not necessarily a wqo on the complete set of states, but only on a restricted set. We cannot simply restrict ourselves to these sets of states, since this would likely violate the compatibility condition. Thus, we introduce a weaker definition of Q-restricted well-structured transition systems in [KS14a; KS14b] and studied the decidability of coverability in this new setting.

Definition 2.16 (Q-Restricted well-structured transition system)**.** Let S be a set of states, let $\preceq\, \subseteq S \times S$ be an quasi-order on S and let Q be a downward-closed subset of S wrt. $\preceq$, where membership is decidable. A *Q-restricted well-structured transition system (Q-restricted WSTS)* is a transition system $\langle S, \Rightarrow, \preceq \rangle$, where the following conditions hold:

Ordering: $\preceq$ is a quasi-order on S and a wqo on Q.

Compatibility: For all $s_1 \preceq t_1$ and transitions $s_1 \Rightarrow s_2$, there exists a sequence $t_1 \Rightarrow^* t_2$ of transitions such that $s_2 \preceq t_2$.

$$\begin{array}{ccc} t_1 & \Longrightarrow^* & t_2 \\ \curlyvee\!\mid & & \curlyvee\!\mid \\ s_1 & \Longrightarrow & s_2 \end{array}$$

We use $\Rightarrow_Q$ to denote the *restricted transition relation* $\Rightarrow_Q = (\Rightarrow \cap\, Q \times Q)$.

These Q-restricted WSTS are a generalization of WSTS and are identical to classical WSTS if $Q = S$. We will show that decidability results differ, but that Algorithm 2.15

can still be applied if some necessary conditions are met.

We observe that for Q-restricted WSTS there are two coverability problems of interest. In this context we will call the coverability problem as defined in Definition 2.10 the *general coverability problem* to distinguish it from the *restricted coverability problem*, which is the question whether a state s can be covered only by transitions within Q.

Definition 2.17 (Restricted coverability problem)**.** Let $\langle S, \Rightarrow, \preceq \rangle$ be a Q-restricted WSTS. The *restricted coverability problem* is the question whether, given two states $s, s' \in Q$, does there exist a state $s'' \in Q$ with $s \Rightarrow_Q^* s''$ and $s' \preceq s''$? If such an s'' exists, we say that s' is *coverable within* Q by s.

Both coverability problems are undecidable in the general case. For the general coverability problem this follows from the fact that the problem corresponds to the coverability problem of general transition systems if $Q = \emptyset$, which is undecidable for instance for graph transformation systems. The restricted coverability problem is undecidable, because it is possible to encode inhibiting conditions by using the restricting set Q. Even if inhibiting conditions are not directly expressible by the transition system, one can ensure that an inhibiting condition is violated if and only if a state not in Q was reached. We show how this can be done for graph transformation systems in Proposition 5.18.

For any $I \subseteq Q$ the restriction to Q gives rise to a new notion of a *Q-restricted direct successor set* $\mathit{Succ}_Q(I) = \mathit{Succ}(I) \cap Q = \{s' \in Q \mid s \in I \wedge s \Rightarrow_Q s'\}$ and a *Q-restricted indirect successor set* $\mathit{Succ}_Q^*(I) = \{s' \in Q \mid s \in I \wedge s \Rightarrow_Q^* s'\}$. Note that $\mathit{Succ}^*(I) \cap Q$ and $\mathit{Succ}_Q^*(I)$ are not necessarily equal, since $\mathit{Succ}_Q^*(I)$ only contains states which are reachable by a sequence of transitions where every intermediate state reached is an element of Q.

We also introduce the notion of a *Q-restricted direct predecessor set*, but in a slightly different way as $\mathit{Pred}_Q(I) = \mathit{Pred}(I) \cap Q = \{s' \in Q \mid s \in I \wedge s' \Rightarrow s\}$ for $I \subseteq S$. We will see that this more general definition is necessary for transferring results for WSTS to Q-restricted WSTS. It is worth pointing out that $\uparrow\mathit{Pred}_Q(I)$ usually contains states not in Q, but is still finitely representable – even if $\uparrow\mathit{Pred}(I)$ is not – because $\preceq$ is a wqo on Q. Effectively, in our algorithms we will not restrict our state space to Q, but only require that the minimal elements to represent upward-closed sets are elements of Q. This also gives rise to the so-called effective Q-pred-basis, which is a strictly stronger effective pred-basis.

Definition 2.18 (Effective Q-pred-basis)**.** A Q-restricted WSTS has an *effective Q-pred-basis* if there exists an algorithm accepting any state $q \in Q$ and and returning $pb_Q(q)$, a finite basis of $\uparrow\mathit{Pred}_Q(\uparrow\{q\})$.

In the following lemma we show that Definition 2.18 is really a stricter condition than Definition 2.13.

Lemma 2.19. *Let T be a Q-restricted WSTS. If T has an effective pred-basis $pb()$, it also has an effective Q-pred-basis $pb_Q()$ with $pb_Q(q) = pb(q) \cap Q$ for $q \in Q$.*

Proof. Let $pb_Q(q) = pb(q) \cap Q$ for $q \in Q$. Since membership is decidable for Q and $pb()$ is effective, $pb_Q()$ is effective as well. Thus, we only have to show that $pb()$ finitely represents the restricted predecessors, i.e. $\uparrow Pred_Q(\uparrow\{q\}) = \uparrow pb_Q(q)$ holds for every $q \in Q$.

We can show that $pb_Q(q)$ is a subset of $\uparrow Pred_Q(\uparrow\{q\})$ by the following implications, using mainly the downward closure of Q and the properties of $pb()$.

$$\begin{aligned}
x \in pb_Q(q) &\implies x \in pb(q)\\
&\implies x \in \uparrow pb(q)\\
&\implies x \in \uparrow Pred(\uparrow\{q\})\\
&\implies \exists x' : (x' \preceq x \wedge x' \in Pred(\uparrow\{q\}))\\
&\overset{x\in Q}{\implies} \exists x' : (x' \preceq x \wedge x' \in Pred_Q(\uparrow\{q\}))\\
&\implies x \in \uparrow Pred_Q(\uparrow\{q\})
\end{aligned}$$

Furthermore, the same argument can be used for the other direction, finally proving $\uparrow Pred_Q(\uparrow\{q\}) = \uparrow pb_Q(q)$.

$$\begin{aligned}
x \in \uparrow Pred_Q(\uparrow\{q\}) &\implies \exists x' \in Q : (x' \preceq x \wedge x' \in Pred_Q(\uparrow\{q\}))\\
&\implies \exists x' \in Q : (x' \preceq x \wedge x' \in Pred(\uparrow\{q\}))\\
&\implies \exists x' \in Q : (x' \preceq x \wedge \exists x'' : (x'' \preceq x' \wedge x'' \in pb(q)))\\
&\overset{x'\in Q}{\implies} \exists x' \in Q : (x' \preceq x \wedge \exists x'' : (x'' \preceq x' \wedge x'' \in pb_Q(q)))\\
&\implies \exists x' \in Q : (x' \preceq x \wedge x' \in \uparrow pb_Q(q))\\
&\implies x \in \uparrow pb_Q(q)
\end{aligned}$$

□

We can apply Algorithm 2.15 to Q-restricted WSTS in two different variants. If an effective pred-basis exists, the unchanged algorithm can be used to solve the general coverability problem. However, it is not guaranteed to terminate, but is correct if it terminates. On the other hand, if an effective Q-pred-basis exists, we can apply the algorithm while calling $pb_Q()$ instead of $pb()$ in line 6 of the algorithm. We can prove that the algorithm will terminate, although it only partially solves the general and the restricted coverability problem.

Lemma 2.20. *Let T be a Q-restricted WSTS with an effective Q-pred-basis $pb_Q()$ and let $F \subseteq Q$ be a finite set. Algorithm 2.15 terminates for the input T and F, if $pb_Q()$ is used instead of an effective pred-basis.*

Proof. Let $W_1 = F$, then Algorithm 2.15 computes the sequence $W_1, W_2, W_3, \ldots$ with $W_{i+1} = W_i \cup pb_Q(W_i)$ for $i \in \mathbb{N}$. Since $\preceq$ is a wqo on Q, it is guaranteed that the increasing sequence $(\uparrow W_1 \cap Q) \subseteq (\uparrow W_2 \cap Q) \subseteq (\uparrow W_3 \cap Q) \subseteq \ldots$ becomes stationary,

i.e. $\uparrow W_n \cap Q = \uparrow W_{n+1} \cap Q$ for some $n \in \mathbb{N}$. We can use this to show that the sequence $\uparrow W_1 \subseteq \uparrow W_2 \subseteq \uparrow W_3 \subseteq \dots$ becomes stationary as well. Assume $\uparrow W_n \neq \uparrow W_{n+1}$, more precisely $\uparrow W_n \subset \uparrow W_{n+1}$. Then there is a $q \in W_{n+1}$ which is not in $\uparrow W_n$. Without loss of generality due to the upward closure we can assume that q is a minimal element of W_{n+1} (otherwise we could take the minimal element which is smaller). By definition $pb_Q(W_i) \subseteq Q$ and therefore $W_{n+1} \subseteq Q$ and $q \in Q$. However, this violates our assumption, since $q \notin (\uparrow W_n \cap Q) \neq (\uparrow W_{n+1} \cap Q) \ni q$ and the first sequence would not have become stationary at the index n. Thus, the sequence $\uparrow W_1 \subseteq \uparrow W_2 \subseteq \uparrow W_3 \subseteq \dots$ becomes stationary and the termination condition of Algorithm 2.15 is satisfied at some point. □

Using the non-terminating variant of Algorithm 2.15 and the terminating variant existing according to Lemma 2.20 we can state partial decidability results for both coverability problems.

Theorem 2.21 (Coverability problems)**.** *Let $T = \langle S, \Rightarrow, \preceq \rangle$ be a Q-restricted WSTS with a decidable order $\preceq$.*

(i) If T has an effective pred-basis and $S = Q$, the general and restricted coverability problems coincide and both are decidable.

(ii) If T has an effective Q-pred-basis, the restricted coverability problem is decidable if Q is closed under reachability.

(iii) Let $Q' \subseteq Q$ be a finite set. If T has an effective Q-pred-basis and applying Algorithm 2.15 to Q' using the effective Q-pred-basis returns the set W, then: if $s \in \uparrow W$, then s can cover a state of Q' in $\Rightarrow$ (general coverability). If $s \notin \uparrow W$, then s can not *cover a state of Q' in $\Rightarrow_Q$ (no restricted coverability).*

(iv) Let $S' \subseteq S$ be a finite set. If T has an effective pred-basis and Algorithm 2.15 applied to S' terminates and returns the set W, then: a state s can cover a state of S' if and only if $s \in \uparrow W$.

Proof. Case (i) is just a reformulation of the decidability results for WSTS in Theorem 2.14.

Case (ii) is similar, since the restricted coverability problem corresponds to the coverability problem of the transition system $T_Q = \langle Q, \Rightarrow_Q, \preceq \cap\, Q \times Q \rangle$. Because T is closed under reachability, T_Q satisfies the compatibility condition and is in fact a WSTS.

We now consider *case (iii)* where Q is not required to be closed under reachability. Let $W_1, W_2, W_3, \dots$ with $Q' = W_1$ be the sequence computed by Algorithm 2.15 using $pb_Q()$ and let $n \in \mathbb{N}$ be the smallest index such that $\uparrow W_n = \uparrow W_{n+1}$, which is guaranteed to exist due to Lemma 2.20. Assume that $s \in \uparrow W_n$. By induction we show the existence of a sequence of transitions leading from s to some state in $\uparrow W_1$. Obviously there is an $q_n \in W_n$ with $q_n \preceq s$ and by definition either $q_n \in W_{n-1}$ or there is a $q_{n-1} \in W_{n-1}$

and a $q'_{n-1} \in S$ with $q_n \Rightarrow q'_{n-1}$ and $q_{n-1} \preceq q'_{n-1}$. In the latter case, because of the compatibility condition there is a $q''_{n-1} \in S$ with $s \Rightarrow^* q''_{n-1}$ and $q_{n-1} \preceq q'_{n-1} \preceq q''_{n-1}$, i.e. s can reach an element of $\uparrow W_{n-1}$. Since this argument holds for q''_{n-1} as well, the state s can ultimately reach a state $q''_1 \in \uparrow W_1$. Note that it is possible that $s = q''_1$, but it is not guaranteed that $q''_i \in Q$ for every i.

For the other statement assume that $s \notin \uparrow W_n$ and assume that there exists a path $s = q_1 \Rightarrow_Q q_2 \Rightarrow_Q \ldots \Rightarrow_Q q_k \in \uparrow W_1$. Note that the second assumption is trivially false, if $s \notin Q$. We can show by induction and by definition of $pb_Q()$ that $q_i \in \uparrow W_{k-i}$ and hence $q_1 \in \uparrow W_k \subseteq \uparrow W_n$, which leads to a contradiction.

The proof of *case (iv)* is straightforward by observing that the set W is an exact representation of all predecessors of S'. □

Chapter 3 Graph Transformation Systems

Graphs have established themselves as an intuitive and powerful modelling language for concurrent or distributed systems and have been used for very different topologies. The introduction of graph transformations enables the modelling of dynamic changes to the topology or the system in general. This induces a transition system, where states are graphs and state changes are modelled by transformation rules of some kind. These transformation systems are usually finite, although their induced transition system may be infinite, as we will see later.

Over the years several different notions of graph transformations have been defined. The algorithmic approaches such as [Nag79; Göt88] directly define how rules are applied, while algebraic approaches such as [EPS73; Ros75] or [BC87] (using also logical concepts) focus on generalized rewriting definitions applied to graphs. There are also purely logical approaches where graphs are modelled by predicates [Cou90]. In many approaches edge labels – besides the graph structure itself – play an important role for defining which transformations are possible (e.g. hyperedge replacement systems [Hab92]), but there are also so-called node-label controlled graph grammars [ER97]. A good overview is given in the Handbook of Graph Grammars and Computing by Graph Transformation series [Roz97; EE+99; EK+99] with the latter two volumes focusing on applications and concurrency.

In this thesis I use an algebraic approach based on categorical constructions [EE+06] as foundation of graph transformation, the so-called single pushout approach [EH+97]. I will first introduce the necessary categorical notions (Section 3.1) and then concretize the constructions for our notion of graphs (Section 3.2).

3.1. Category Theory

Category theory aims to generalize concepts and to provide a high-level description in which properties are proved. The results can then be used in any concrete category (satisfying the necessary properties).

Definition 3.1 (Category)**.** A category $\mathbf{C}$ consists of:

- a class of $\mathbf{C}$-*objects*,
- a class of $\mathbf{C}$-*arrows* (later called *morphisms*),
- two functions *dom* and *cod* assigning to each $\mathbf{C}$-arrow $f\colon A \to B$ the *domain* $dom(f) = A$ and the *codomain* $cod(f) = B$, which are both $\mathbf{C}$-objects,
- a *composition operator* $\circ$ assigning to each two $\mathbf{C}$-arrows f and g with $cod(f) = dom(g)$ the composed $\mathbf{C}$-arrow $g \circ f\colon dom(f) \to cod(g)$ satisfying the following:
  - for every three $\mathbf{C}$-arrows $f\colon A \to B$, $g\colon B \to C$ and $h\colon C \to D$ it holds that $h \circ (g \circ f) = (h \circ g) \circ f$ and,
  - for every two $\mathbf{C}$-object A, B there are *identity* $\mathbf{C}$-arrows $id_A\colon A \to A$ and $id_B\colon B \to B$ such that for every $\mathbf{C}$-arrow $f\colon A \to B$ it hold that $id_B \circ f = f$ and $f \circ id_A = f$.

For two $\mathbf{C}$-objects we use $\mathbf{C}(A, B)$ to denote the class of all $\mathbf{C}$-arrows f with $dom(f) = A$ and $cod(f) = B$.

Example 3.2. One of the simplest categories is the category **Set** where the objects are sets and the arrows are total functions. It is easy to show that **Set** satisfies the properties of Definition 3.1. The domain and codomain of each arrow are the domain and codomain of the function and the composition operator is the standard composition of functions.

A more complex category it the category of hypergraphs which we will introduce in Section 3.2.

A special kind of arrows, which we will often use, are isomorphisms. These can be formalized for arbitrary categories in the following way.

Definition 3.3 (Isomorphism)**.** Let $\mathbf{C}$ be a category and let $f\colon A \to B$ be an arrow. We call f an *isomorphism* if there exists an arrow $g\colon B \to A$ such that $f \circ g = id_B$ and $g \circ f = id_A$.

Also important for this work is the notion of commuting diagrams in a category. In fact, in our proofs we will extensively show and use that specific diagrams commute.

Definition 3.4 (Diagram)**.** A *diagram* in a category **C** is a subclass of **C**-objects $\mathcal{O}$ and a subclass of **C**-arrows $\mathcal{A}$, where for every $f \in \mathcal{A}$ it holds that $dom(f) \in \mathcal{O}$ and $cod(f) \in \mathcal{O}$.

We say that the (whole) diagram *commutes* if for every two sequences of compositions $f_1 \circ \ldots \circ f_n$ and $g_1 \circ \ldots \circ g_m$ of arrows of $\mathcal{A}$ with $dom(f_n) = dom(g_m)$ and $cod(f_1) = cod(g_1)$ it holds that $f_1 \circ \ldots \circ f_n = g_1 \circ \ldots \circ g_m$.

Example 3.5. An example of a diagram can be seen in Figure 3.1. The diagram commutes if and only if $f_1 = f_4 \circ f_3 \circ f_2 = f_6 \circ f_5$ holds.

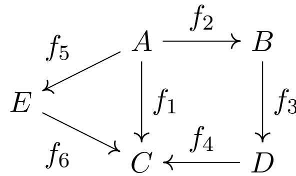


Figure 3.1.: A diagram consisting of objects $\{A, B, C, D, E\}$ and arrows $\{f_1, \ldots, f_6\}$

A special form of diagram is the pushout diagram which will later serve as the base construction for rewriting a graph. A pushout is especially suitable for this purpose, since it generalizes the disjoint union of the codomain of two arrows.

Definition 3.6 (Pushout)**.** Let **C** be a category and let $f\colon A \to B$, $g\colon A \to C$ be arrows of **C** as shown in Figure 3.2. The *pushout* of f and g is the triple $\langle D, f', g' \rangle$ consisting of the *pushout object* D and two arrows $f'\colon C \to D$, $g'\colon B \to D$ such that the following conditions are satisfied:

- the diagram commutes, i.e. $g' \circ f = f' \circ g$, and
- for every other object D' with arrows $f''\colon C \to D'$ and $g''\colon B \to D'$ such that $g'' \circ f = f'' \circ g$ there is a unique arrow $h\colon D \to D'$ (up to isomorphism) such that the diagram commutes, i.e. $f'' = h \circ f'$ and $g'' = h \circ g'$.

A pushout defines a unique way to "close" a diagram. Requiring the existence of h, in the following often called mediating arrow, guarantees that the pushout is unique up to isomorphism if it exists. It also ensures that D is more universal than any other object D' which is part of a commuting square (but no pushout).

There are a few general properties of pushouts (in any category) which we will use frequently in proofs in later chapters. These results are well-known (see [Mac78; Pie91; LS04]), but due to their importance we restate them in Lemma 3.7.

Lemma 3.7 ([LS04])**.** *Let* **C** *be a category and let A, B, C, D, E, F be objects with arrows as given in the diagram in Figure 3.3. The following two statements are true:*

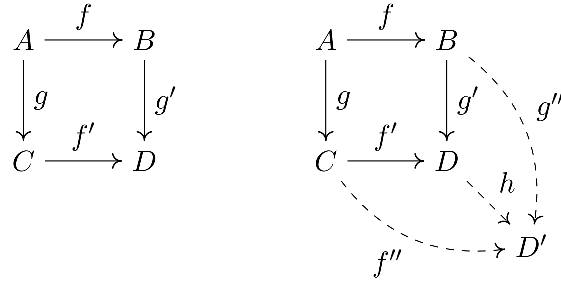


Figure 3.2.: A pushout square (on the left) including the mediating morphism (on the right)

- *If $\langle E, f', g' \rangle$ is the pushout of f, g (left square) and $\langle F, j', g'' \rangle$ is the pushout of j, g' (right square), then $\langle F, j' \circ f', g'' \rangle$ is the pushout of $j \circ f$ and g (outer rectangle).*
- *If $\langle E, f', g' \rangle$ is the pushout of f, g (left square), $\langle F, j' \circ f', g'' \rangle$ is the pushout of $j \circ f$ and g (outer rectangle) and the diagram commutes, then $\langle F, j', g'' \rangle$ is the pushout of j, g' (right square).*

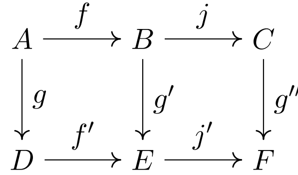


Figure 3.3.: Diagram showing all arrows of Lemma 3.7

The previous lemma leads to a third property we will need in our proofs.

Lemma 3.8. *Let* **C** *be a category and let A, B, C, D, F be objects with the arrows f, j, g, g'' as shown in the diagram in Figure 3.3 and the arrow $k\colon D \to F$, such that $\langle F, k, g'' \rangle$ is the pushout of $j \circ f$ and g (outer rectangle). If the pushout of f and g exists, then there is a unique way to split k into morphisms f' and j' such that $\langle E, f', g' \rangle$ is the pushout of f, g (left square) and $\langle F, j', g'' \rangle$ is the pushout of j, g' (right square).*

Proof. Let $\langle E, f', g' \rangle$ be the pushout of f and g (which is unique). Since $g'' \circ j \circ f = k \circ g$ commutes, by Definition 3.6 there is a unique mediating arrow $j'\colon E \to F$, commuting with the diagram. By Lemma 3.7 the right square is a pushout, since the left square and the outer rectangle are both pushouts. □

Another important concept for graph rewriting is the notion of pushout complements. These are especially important for applying rules backwards and thus essential for computing Algorithm 2.15.

Definition 3.9 (Pushout complement)**.** Let $\mathbf{C}$ be a category and let $f\colon A \to B$ and $g'\colon B \to D$ be two arrows (as shown in Figure 3.2). A *pushout complement* of f and g' is the triple $\langle C, g, f' \rangle$ with arrows $g\colon A \to C$ and $f'\colon C \to D$ such that $\langle D, f', g' \rangle$ is a pushout of f and g.

Neither the existence nor the uniqueness of pushout complements is guaranteed, even in categories where both holds for pushouts, but we will state sufficient conditions for it to exists in the category of graphs.

3.2. Category of Graphs

The hypergraphs used in this thesis are a generalization of directed graphs. They consist of a set of (unlabelled) nodes which are connected by labelled hyperedges. A hyperedge can be connected to an arbitrary large, but finite, number of nodes and can also be connected to the same node multiple times. This class of graphs is very suitable for modelling, since not just binary but also n-nary relations can be modelled directly by corresponding hyperedges. Furthermore, many decidability results for directed graphs can be extended to hypergraphs.

Definition 3.10 (Hypergraph)**.** Let Λ be a *finite set of (edge) labels* and let $ar\colon \Lambda \to \mathbb{N}_0$ be a function that assigns an *arity* to each label. A *(Λ-)hypergraph* is a tuple $\langle V_G, E_G, c_G, l_G \rangle$ where V_G is a finite set of nodes, E_G is a finite set of edges, $c_G\colon E_G \to V_G^*$ is a connection function and $l_G\colon E_G \to \Lambda$ is the labelling function for edges. We require that $|c_G(e)| = ar(l_G(e))$ for each edge $e \in E_G$.

We say that an edge e is *incident* to a node v (and vice versa) if v occurs at least once in $c_G(e)$. We call two nodes *adjacent* if there is an edge incident to both and we call two edges *adjacent* if there is a node incident to both.

As abbreviation we will write $x \in G$ instead of $x \in V_G \cup E_G$ and will use V, E, c and l when the corresponding graph is unambiguously determined. Additionally, we will refer to hypergraphs simply as graphs, since the main results of this thesis are all stated for hypergraphs. In some places we also use *directed graphs*, which are hypergraphs, where every edge has the arity two, i.e. $|ar(\ell)| = 2$ for every label $\ell \in \Lambda$. However, we will use directed graphs also to denote hypergraphs where every arity is at most two, since edges with arity zero or one can be easily modelled by loops.

In the following we will often refer to the class of all graphs with a fixed label set Λ and denote this by $\mathcal{G}(\Lambda)$.

Example 3.11. A visual representation of some graphs is shown in Figure 3.4. In general we depict edges of arity at least three (e.g. the A-labelled edge) by a rounded rectangle containing the label and a numbered connection to all incident nodes, showing their position in the node sequence. In examples we will often use binary edges (e.g. the B-labelled edge), which we draw for simplicity by an arrow pointing from the first element in the node sequence to the second element. Unary and zero edges are drawn as shown by the C- and D-labelled edges respectively. For visual clarity we may omit the numberings of nodes, if they are of little interest for the specific example.

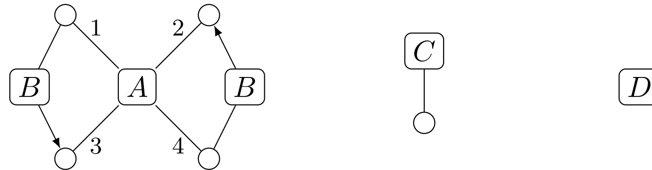


Figure 3.4.: Visual representation of a hypergraph, containing general hyperedges (A), binary edges (B), unary edges (C) and zero edges (D)

Hypergraphs will serve as objects in the category we use for graph rewriting. As arrows we use structure preserving mappings between graphs, so-called graph morphisms. The structure preserving properties include the preservation of edge labels and a compatibility condition for node and edge mappings. If an edge is mapped by a morphism, its incident nodes must be mapped to the nodes incident to the image of the edge.

Definition 3.12 (Graph morphism)**.** Let G, H be (Λ-)graphs. A *partial graph morphism* (or simply *morphism*) $f\colon G \rightharpoonup H$ consists of a pair of partial functions $\langle f_V \colon V_G \rightharpoonup V_H, f_E \colon E_G \rightharpoonup E_H\rangle$ such that for every $e \in E_G$ it holds that $l_G(e) = l_H(f_E(e))$ and $f_V(c_G(e)) = c_H(f_E(e))$ whenever $f_E(e)$ is defined. Furthermore, if a morphism is defined on an edge $e \in E_G$, it must be defined on all nodes incident to e. We use $f(x)$ for $x \in G$ to refer to $f_V(x)$ or $f_E(x)$, as appropriate, and write $f(G)$ to denote the subgraph of H where every element has a preimage in G. We compose two morphisms $f\colon G \rightharpoonup H$ and $g\colon H \rightharpoonup I$ by composing both partial functions, i.e. $g \circ f = \langle g_V \circ f_V, g_E \circ f_E\rangle$.

A morphism f is *total*, if it is defined for every $x \in G$. A morphisms is *injective*, if for every $x_1, x_2 \in G$ on which f is defined, it holds that $f(x_1) = f(x_2) \implies x_1 = x_2$. A morphisms is *surjective*, if for every $x \in H$ there is an $x' \in G$ such that $f(x') = x$. We denote total morphisms by the arrow $\rightarrow$ and total, injective morphisms by the arrow $\rightarrowtail$.

Note that we will often use the extension of f_V to a sequence of nodes, as defined in Section 2.1.3.

Example 3.13. Figure 3.5 shows an example of a graph morphism. We normally represent the two mappings f_V and f_E either by numbers (as seen in the figure) or

by position. The morphism is not total, since the B-labelled edge has no image, not injective, since two nodes are mapped to the same node in H, and not surjective, since the lower A-labelled edge (in H) has no preimage in G.

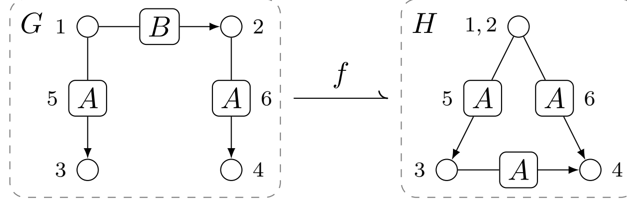


Figure 3.5.: Example of a graph morphisms which is neither total, nor injective, nor surjective

We can easily prove that the composition, as defined in Definition 3.12, is well-defined.

Lemma 3.14. *The composition of two graph morphisms is a graph morphism and preserves totality, injectivity and surjectivity.*

Proof. Let $f\colon G \rightharpoonup H$ and $g\colon H \rightharpoonup I$ be two graph morphisms. Obviously $g_V \circ f_V$ and $g_E \circ f_E$ exist and it remains to be shown that the additional restrictions for morphisms are satisfied.

By using that f and g are morphisms we obtain $l_I(g_E(f_E(e))) = l_H(f_E(e)) = l_G(e)$ and $g_V(f_V(c_G(e))) = g_V(c_H(f_E(e))) = c_I(g_E(f_E(e)))$ for all $e \in E_G$. Furthermore, if $g(f(e))$ is defined, then all nodes of $f(e)$ and ultimately $g(f(e))$ have an image under $g \circ f$. Morphisms preserve totality, injectivity and surjectivity, since functions preserve totality, injectivity and surjectivity. □

Using the notion of graph morphisms we can define the category of graphs used as basis for our rewriting approach. In fact, we define two categories, one with total morphisms and one with partial morphisms. Since the constructions of pushouts and pushout complements are more involved for partial morphisms, we first define them for total morphisms and then extend them to work for partial morphisms.

Definition 3.15 (Category of graphs)**.** We use **Λ-HGp** to denote the category of all Λ-hypergraphs and partial graph morphisms and **Λ-HGt** to denote the category of Λ-hypergraphs and total graph morphisms.

As already shown in Lemma 3.14, the composition is well-defined and it is easy to see that it satisfies the necessary conditions. The commutativity follows from the commutativity of the functions f_E, f_V and the identity morphism of a graph consists of two identities on the node and edge sets respectively.

3.3. Graph Transformation Systems

There are two major algebraic approaches to graph transformation using category theory, the single pushout approach (SPO) [EH+97] and the double pushout approach (DPO) [CM+97]. In the SPO approach a rewriting step is performed by one pushout in the category Λ-**HGp**, while the DPO approach uses a pushout and a pushout complement in Λ-**HGt**. The main difference between the two approaches lies in their handling of conflicts, e.g. when a rule should delete a node incident to an edge, which is not specified to be deleted. In the SPO approach all such edges are implicitly deleted as well, while in the DPO approach the rule would not be applicable. Since the handling in the DPO approach resembles a negative application condition, the SPO approach is more suitable for WSTS, where negative conditions usually interfere with the compatibility condition of Definitions 2.11 and 2.16. We therefore base our general framework on the SPO approach.

Definition 3.16 (Rule and match)**.** A *rewriting rule* is a partial morphism $r\colon L \rightharpoonup R$. A *match* of the rule r in some graph G is a total morphism $m\colon L \to G$. We say that a match m is *conflict-free wrt.* r if for every two $x_1, x_2 \in L$ with $m(x_1) = m(x_2)$, the images $r(x_1)$ and $r(x_2)$ are either both defined or both undefined. A rule is *applicable* to a graph G if there exists at least one match in G.

In this work we will mainly use conflict-free matches or injective matches. Note that every injective match is naturally conflict-free, but conflict-free matches may be non-injective. The application of a rule is performed by the computation of a pushout of the rule and the match.

Definition 3.17 (Graph rewriting)**.** Let $r\colon L \rightharpoonup R$ be a rule and let $m\colon L \to G$ be a match to some graph G. A *rewriting step* is obtained by taking the pushout of m and r in the category Λ-**HGp**. Then G is rewritten to the pushout object H (written as $G \overset{r,m}{\Rightarrow} H$ or simply $G \Rightarrow H$).

Example 3.18. The application of a rule can change a graph G in three different ways, of which two are shown in Figure 3.6. For every element in L for which r is undefined, in this case the node 2 and the edges 4 and 5, its match in G is deleted. Note that this implicitly deletes the B-edge incident to the deleted node 2 in G. Furthermore, all elements of R which have no preimage in L, here just a single A-edge, are added by the rule application. Finally, if the rule is non-injective, the images in G of elements in L with the same image in R, are merged.

It has been shown that in the categories Λ-**HGt** and Λ-**HGp** pushouts exist for arbitrary morphisms [EH+97], thus a rewriting step is always possible if a match exists. Moreover, the pushout of two total morphisms is the same in both categories. We show this also in Section 3.4 by giving a construction for pushouts of partial and total morphisms

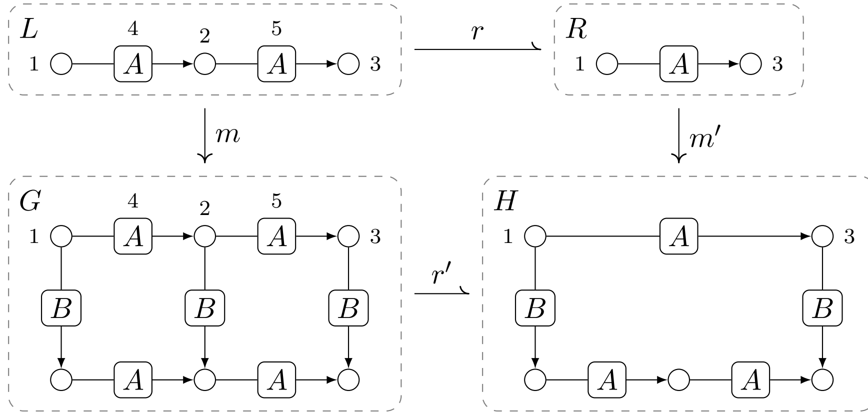


Figure 3.6.: Example of an application of the rule r using the match m

respectively. In the following we will refer to the morphism m', as shown in the previous example, as the *co-match*.

The graph transformation systems (GTS) used in this thesis are simply a set or rules, not necessarily associated with initial graphs, since these are not required for our analysis (but can be used for premature termination). Given a class of graphs $\mathcal{G}$, these GTS give rise to a transition system on $\mathcal{G}$. If $\mathcal{G}$ is infinite, the induced transition system is also infinite – except for some artificial examples – and the GTS is effectively a finite representation of the transition system.

Definition 3.19 (Graph transformation system)**.** A *graph transformation system* (GTS) $\mathcal{T}$ is a finite set of rules.

Let $\mathcal{G}$ be a class of graphs. A *graph transition system* on $\mathcal{G}$ generated by $\mathcal{T}$ is a pair $\mathcal{T}_\mathcal{G} = \langle \mathcal{G}, \Rightarrow \rangle$, where $\mathcal{G}$ is the set of states and a transition $G \Rightarrow G'$ exists if and only if $G, G' \in \mathcal{G}$ and G can be rewritten to G' using a rule of $\mathcal{T}$. We write $\mathcal{T}_\mathcal{G}^i$ for the transition system induced by using only injective matches and we write $\mathcal{T}_\mathcal{G}^c$ for the transition system induced by using only conflict-free matches.

Ideally we want $\mathcal{G}$ to be the class of all graphs, since analysing a larger transition system yields a stronger result. However, at some point we will have to restrict $\mathcal{G}$ to ensure the termination of our analysis. In contexts where $\mathcal{G}$ is either fixed or not relevant, we will use graph transformation systems and graph transition systems synonymously and say that a transformation system has a property if its generated transition system (on $\mathcal{G}$) has the property.

Our analysis makes it necessary to apply a rule backwards. This backward application can be done by computing the pushout complement, that is, given a rule and a co-match we compute G and a match m such that G is rewritten to H via m. However, contrary to pushouts, pushout complements in Λ-**HGt** and Λ-**HGp** do not necessarily

exist and are not necessarily unique if they exist. In Λ-**HGt** the number of pushout complements is always finite, but in Λ-**HGp** two morphisms may have infinitely many pushout complements, as shown in Example 3.20. For simplification we only cover the computation of pushout complements in Λ-**HGp** where the co-match is total. This is sufficient for the backward application of rules, as we will show in Chapter 6.

Example 3.20. Figure 3.7 shows a rule r and a co-match m' for which the number of pushout complements is infinite. The graphs G_1 and G_2 are two of these pushout complements. The deletion of node 1 causes all incident edges (even if not matched) to be deleted, such that both graphs are rewritten to H. More precisely, every pushout complement of r and m' can be obtained by adding an arbitrary large number of edge to G_1, as long as every such edge is incident to node 1 (including unary edges and edges of higher arity).

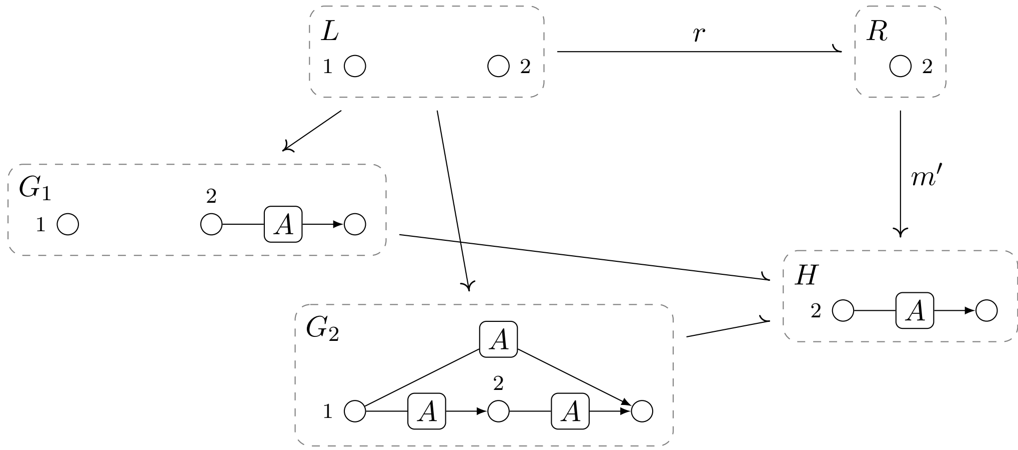


Figure 3.7.: Two morphisms for which the number of pushout complements is infinite and two of those pushout complements

For the existence of pushout complements two well-known conditions exist for total morphisms, together called the gluing condition [CM+97]. These conditions are also sufficient for partial morphisms if the co-match is total.

Proposition 3.21 (Existence of pushout complements)**.** *Let $r\colon L \rightharpoonup R$ be a partial morphisms and let $m'\colon R \rightarrow H$ be a total morphism. There exists a pushout complement $\langle G, r', m\rangle$ in* Λ-**HGp** *with $r'\colon G \rightharpoonup H$ and $m\colon L \rightharpoonup G$ if and only if:*

- *there exists no edge $e \in E_H$ with $e \notin m'(R)$ which is incident to a node $m'(v)$ for some $v \in R$ with $v \notin r(L)$ (dangling condition) and,*
- *for every $x_1, x_2 \in R$ with $x_1 \neq x_2$ and $m'(x_1) = m'(x_2)$ it holds that $x_1, x_2 \in r(L)$ (identification condition).*

If r is total, there exists a pushout complement in Λ-**HGt** *if and only if the previous two conditions hold.*

Proof. That the dangling condition and identification condition are necessary and sufficient in Λ-**HGt** is a well-known result [CM+97]. If both conditions are satisfied, we show that at least one pushout complement exists by giving a construction in Propositions 3.30 and 3.33 in Section 3.5. So it remains to be shown that no pushout complement exists if one of the conditions is not satisfied.

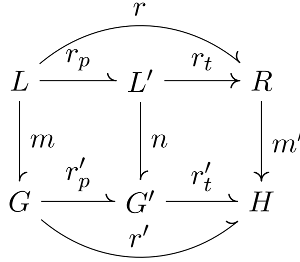


Figure 3.8.: Diagram showing all arrows of the proof of Proposition 3.21

Let r be a rule, let m' be a co-match and let $\langle G, m, r' \rangle$ be a pushout complement (in Λ-**HGp**) as shown in Figure 3.8. We assume that at least one condition does not hold and will derive a contradiction. Obviously we can split r into morphism r_p, r_t with $r_t \circ r_p = r$, where r_t is total and r_p is injective, surjective and partial, i.e. r_p is effectively the identity for every element for which it is defined. According to Lemma 3.8 there is a unique way to split r' such that $\langle G', r'_p, n \rangle$ is the pushout of r_p, m, and $\langle H, r'_t, m' \rangle$ is the pushout of r_t, n. Since $m' \circ r_t = r'_t \circ n$ and r_t, m' are total, n must be total as well. Since a pushout of two total morphisms in Λ-**HGp** is the same as their pushout in Λ-**HGt**, we now that r'_t is total as well. This implies that G' is a pushout complement of r_t and m' in Λ-**HGt**. However, if one of the conditions is violated for r, m', it is also violated for r_t, m, since for every $x \in R$ it holds that $x \in r(L)$ if and only if $x \in r_t(r_p(L))$. Thus, no pushout complement G' can exist. □

Both conditions of Proposition 3.21 are a direct consequence of Definition 3.6. Due to the universality, every element in the pushout has to have a preimage in R or G (or both). Thus, an edge $e \notin m'(R)$ has to have a preimage in G and therefore all its incident nodes have to have a preimage in G as well. Again because of universality, if two elements x_1, x_2 in R have the same image in H, they have to have preimages in L. If this would not be the case, then there would be another commuting diagram, where x_1 and x_2 are not merged, but there is no mediating morphism from H to the diagram. The construction of pushout complements is covered in Section 3.5 in greater detail.

3.4. Construction of Pushouts

Pushouts for graphs can be computed by performing pushouts on the set of nodes and edges. If all involved morphisms are total and injective, the pushout of two morphisms $f\colon G \rightarrowtail H$ and $g\colon G \rightarrowtail I$ is the union of H and I, merging elements which have a common preimage in G and being otherwise disjoint. If f or g (or both) are non-injective or partial, this will result in additional mergings or deletions, respectively. We will now first construct pushouts in Λ-**HGt** and then extend the construction to Λ-**HGp**.

Proposition 3.22 (Pushout in Λ-**HGt**). *Let G, H, I be graphs with pairwise disjoint node and edge sets*[1] *and let $f\colon G \rightarrow H$, $g\colon G \rightarrow I$ be total graph morphisms. Let $\sim$ be the relation on $V_H \cup V_I \cup E_H \cup E_I$, where $f(x) \sim g(x)$ and $g(x) \sim f(x)$ for all $x \in G$ and let $\eqsim$ be the equivalence closure of $\sim$. The pushout object $J = \langle V_J, E_J, c_J, l_J\rangle$ can be constructed as follows:*

- $V_J = (V_H \cup V_I)/\eqsim$,
- $E_J = (E_H \cup E_I)/\eqsim$,
- $c_J\colon E_J \rightarrow V_J^*$ *where* $c_J([e]_\eqsim) = [v_1]_\eqsim \dots [v_k]_\eqsim$ *and* $v_1 \dots v_k = \begin{cases} c_H(e) & \text{if } e \in E_H \\ c_I(e) & \text{if } e \in E_I \end{cases}$
- $l_J\colon E_J \rightarrow \Lambda$ *where* $l_J([e]_\eqsim) = \begin{cases} l_H(e) & \text{if } e \in E_H \\ l_I(e) & \text{if } e \in E_I \end{cases}$

The resulting morphisms are $f'\colon I \rightarrow J$, $g'\colon H \rightarrow J$ with

$$f'(x) = [x]_\eqsim \qquad\qquad g'(y) = [y]_\eqsim$$

for $x \in I$ and $y \in H$, respectively. The object J together with the morphisms f', g' is the pushout of f, g.

Proof. See Appendix B.1. □

Example 3.23. An example of a pushout for two morphisms f and g can be seen in Figure 3.9. Both morphisms are non-injective: nodes 1 and 2 are merged via f, while the nodes 2 and 3 are merged via g. Due to the equivalence closure $\eqsim$, all three nodes are merged in J, although 1 and 3 are neither merged by f nor by g. Note that this merging is necessary for the diagram to commute. The two C-labelled edges in H and I are not merged in J, since they do not share a common preimage in G.

[1]Disjointness can be easily achieved by renaming.

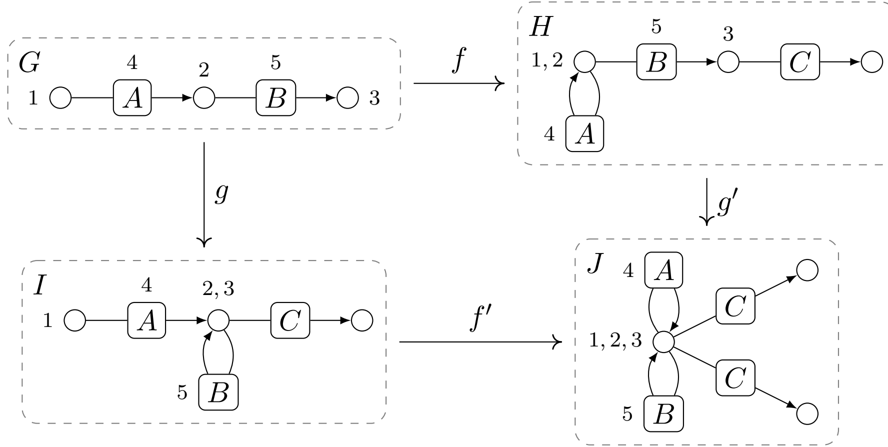


Figure 3.9.: Shows the pushout J of f and g in Λ-**HGt**

The pushout for partial morphisms can be computed with a similar construction. Mergings as well as new nodes and edges are handled as in the total case, with the exception that some equivalence classes must be deleted. More precisely, a class has to be deleted if it contains a element $f(x)$ which should be equivalent to $g(x)$, but $g(x)$ is undefined (or vice versa).

Proposition 3.24 (Pushouts in Λ-**HGp**)**.** *Let G, H, I be graphs with pairwise disjoint node and edge sets and let $f\colon G \rightharpoonup H$, $g\colon G \rightharpoonup I$ be partial graph morphisms. Let $\sim$ be the relation on $V_H \cup V_I \cup E_H \cup E_I$, where $f(x) \sim g(x)$ and $g(x) \sim f(x)$ for all $x \in G$ for which $f(x)$ and $g(x)$ are both defined and let $\overline{\sim}$ be the equivalence closure of $\sim$.*

We say that an equivalence class on nodes is valid *if and only if it does contain no element $f(x)$ for which $g(x)$ is undefined and no element $g(x)$ for which $f(x)$ is undefined. An equivalence class on edges is* valid*, if the previous condition holds for the class and the equivalence class of every incident node is valid as well.*

We can construct the pushout J by the same means as Proposition 3.22 with the exception that V_J and E_J contain only the valid equivalence classes and $f'(x)$, $g'(x)$ are undefined if the equivalence class of x is not valid.

Proof. See Appendix B.1. □

Example 3.25. An example of pushouts in Λ-**HGp** is shown in Figure 3.10. The nodes $2, 3$ in I and 2 in H are in the same equivalence class, since they have the common preimage 2 in G. However, since 3 is also a preimage of $2, 3$ in G, the equivalence class is not valid and must be deleted in J. Note that this means that the node 2 must be deleted as well, although it has an image under f as well as g. It becomes evident that this must be done when assuming that $2, 3$ has an image under f'. In this case the node

3 has an image under $f' \circ g$ but not under $g' \circ f$, i.e. the diagram does not commute. The A-labelled edge is incident to a node with a non-valid equivalence class, i.e. 2, and must be deleted as well. Thus, the node 1 is the only remaining element of the pushout.

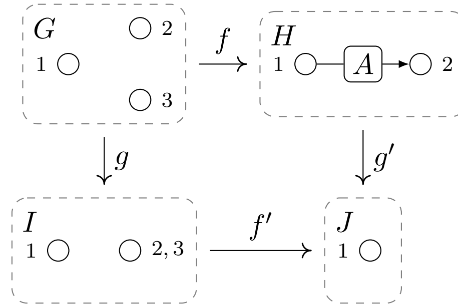


Figure 3.10.: Shows the pushout of two morphism f and g of which one is partial

3.5. Construction of Pushout Complements

The construction of pushout complements is considerably more difficult than the construction of pushouts. This is also due to the fact that pushout complements are not unique if they exist and for partial morphisms there may even be infinitely many. For total, possibly non-injective morphisms we published a construction in [HJ+10] which I will restate in the following. Based on this I will also present the construction for partial morphisms we used in [JK08; KS14b]. However, since for the backward analysis all pushout complements must be computed, we are bound to compute a finite subset which is sufficient wrt. some order. We will therefore revisit this construction in Chapter 6, where we introduce the backward analysis for graph transformation systems.

Similar to the pushout construction, the construction below uses equivalences and equivalence closure as a core concept. Intuitively the construction works as follows. We first copy G and add a copy of every element in J that has no preimage in H. Then we compute two equivalence relations, $\equiv_f$ relating all elements merged by f and $\equiv_{\overline{f}}$ relating all elements merged by $g' \circ f$. As a last step we are searching for equivalences $\equiv$, relating all elements merged by g, such that the equivalence closure of $\equiv$ and $\equiv_f$ is $\equiv_{\overline{f}}$. By factoring through $\equiv$ we assure that g merges exactly those elements necessary for the pushout to be J.

Proposition 3.26 (Pushout complements in Λ-**HGt**). *Let $f\colon G \rightarrow H$ and $g'\colon H \rightarrow J$ be total morphisms as shown in Figure 3.11. We construct a pushout complement I with morphisms $g\colon G \rightarrow I$ and $f'\colon I \rightarrow J$ as follows:*

1. *Construct a graph $\tilde{J}$ as follows:*

- *For every node* $v \in V_J$ *that is not in the range of* g'*, add a copy of* v *to* $\tilde{J}$*. The copy of* v *will be denoted by* v_c*.*
- *For every edge* $e \in E_J$ *that is not in the range of* g'*, add a copy of* e *with the same arity, incident to fresh nodes, to* $\tilde{J}$*. The copy of* e *will be denoted by* e_c *and the fresh nodes by* $\langle e_c, i\rangle$ *for* $1 \le i \le ar(l_J(e))$*.*

This means that $\tilde{J}$ *is a collection of disconnected nodes and edges.*

2. *Now construct* $G \uplus \tilde{J}$*, the disjoint union of* G *and* $\tilde{J}$*, with morphisms* $\overline{g}\colon G \to G \uplus \tilde{J}$*,* $\overline{f}\colon G \uplus \tilde{J} \to J$ *as follows:*

$$\overline{g}(x) = x$$
$$\overline{f}(x) = \begin{cases} g'(f(x)) & \text{if } x \in G \\ y & \text{if } x = y_c \\ c_J(e)[i] & \text{if } x = \langle e_c, i\rangle \end{cases}$$

Clearly $\overline{f} \circ \overline{g} = g' \circ f$*.*

3. *Define two equivalences on elements of* $G \uplus \tilde{J}$*:*
   - $x \equiv_{\overline{f}} y$ *if and only if* $\overline{f}(x) = \overline{f}(y)$*.*
   - $x \equiv_f y$ *if either* $x = y$ *or* $x, y \in G$ *and* $f(x) = f(y)$*.*

   It can easily be seen that $\equiv_f$ *is a refinement of* $\equiv_{\overline{f}}$*, i.e.,* $x \equiv_f y$ *implies* $x \equiv_{\overline{f}} y$*.*

4. *Now let* $\equiv$ *be an equivalence on* $G \uplus \tilde{J}$ *such that* $\equiv_{\overline{f}} = \overline{\equiv \cup \equiv_f}$ *and whenever* $e_1 \equiv e_2$ *for two edges* e_1, e_2 *we require that* $c_{G\uplus\tilde{J}}(e_1)[i] \equiv c_{G\uplus\tilde{J}}(e_2)[i]$ *for all* $1 \le i \le ar(l_{G\uplus\tilde{J}}(e_1)) = ar(l_{G\uplus\tilde{J}}(e_2))$*. We construct the pushout complement* $I = (G \uplus \tilde{J})/\equiv$ *with morphisms* $g\colon G \to I$*,* $f'\colon I \to J$ *as specified below:*

$$g(x) = [\overline{g}(x)]_\equiv \qquad\qquad f'([x]_\equiv) = \overline{f}(x)$$

Note that f' *is well-defined since* $\equiv$ *refines* $\equiv_{\overline{f}}$*.*

Proof. See Lemmas 3.28 and 3.29. □

Example 3.27. Let the morphisms $f\colon G \to H$ and $g'\colon H \to J$ be given as shown in Figure 3.12. In J there is a single A-labelled edge and the mappings are indicated by numbers. On nodes we have the equivalences $\equiv_{\overline{f}}$ and $\equiv_f$, represented by their equivalence classes:

- $\equiv_{\overline{f}}$: $\{1, 2, 3, 4, \langle e_c, 1\rangle, \langle e_c, 2\rangle\}$
- $\equiv_f$: $\{1, 2\}, \{3, 4\}, \{\langle e_c, 1\rangle\}, \{\langle e_c, 2\rangle\}$

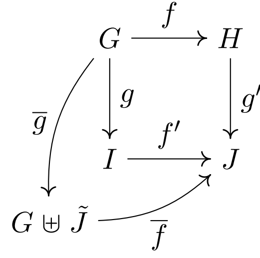


Figure 3.11.: Pushout complement diagram for the construction in Proposition 3.26

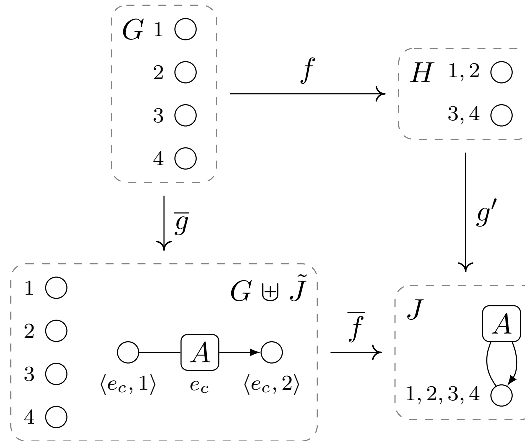
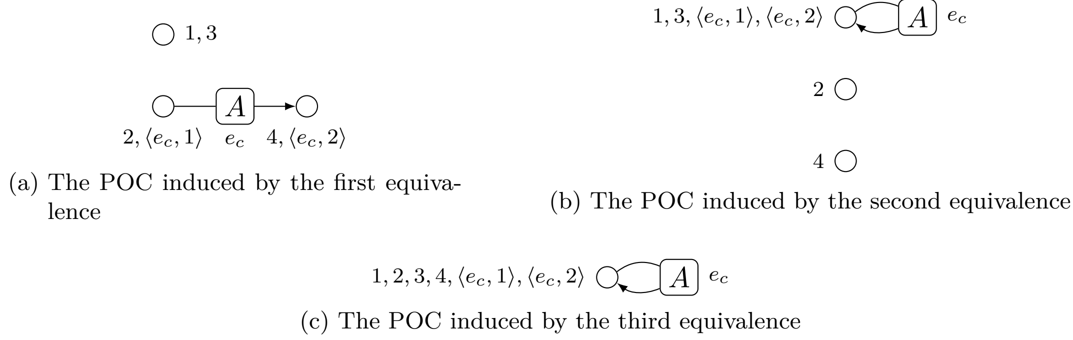


Figure 3.12.: Given morphisms f and g', shows the first two steps of Proposition 3.26

Every possible equivalence $\equiv$ has to "connect" equivalence classes of $\equiv_f$ if and only if they are contained in the same equivalence class of $\equiv_{\overline{f}}$. This is done by relating at least one node of each such equivalence class with a node of another class until all classes are (indirectly) connected. In this case we have to relate at least one element of $\{1, 2\}$ with one of $\{3, 4\}$ and both nodes $\langle e_c, 1\rangle$ and $\langle e_c, 2\rangle$ have to be related to one of the nodes 1, 2, 3 or 4. For instance the following three equivalences $\equiv$ are all permissible and the pushout complements induced by them are shown in Figure 3.13:

- $\{1, 3\}, \{2, \langle e_c, 1\rangle\}, \{4, \langle e_c, 2\rangle\}$
- $\{1, 3, \langle e_c, 1\rangle, \langle e_c, 2\rangle\}, \{2\}, \{4\}$
- $\{1, 2, 3, 4, \langle e_c, 1\rangle, \langle e_c, 2\rangle\}$

But there are many more possibilities. In order to enumerate them more systematically we can first consider all 15 equivalences on the set $\{1, 2, 3, 4\}$, given by equivalence classes. The ones that do not satisfy the requirement above are crossed out.



(a) The POC induced by the first equivalence

(b) The POC induced by the second equivalence

(c) The POC induced by the third equivalence

Figure 3.13.: The three pushout complements induced by the equivalences of Example 3.27

$\{1,2,3,4\}$	$\{1\},\{2,3,4\}$	$\{2\},\{1,3,4\}$	$\{3\},\{1,2,4\}$	$\{4\},\{1,2,3\}$
~~$\{1,2\},\{3,4\}$~~	$\{1,3\},\{2,4\}$	$\{1,4\},\{2,3\}$	~~$\{1,2\},\{3\},\{4\}$~~	$\{1,3\},\{2\},\{4\}$
$\{1,4\},\{2\},\{3\}$	$\{2,3\},\{1\},\{4\}$	$\{2,4\},\{1\},\{3\}$	~~$\{3,4\},\{1\},\{2\}$~~	~~$\{1\},\{2\},\{3\},\{4\}$~~

Now for k equivalence classes there are k^2 possibilities to associate $\langle e_c, 1\rangle$ and $\langle e_c, 2\rangle$ to these equivalence classes. Hence, in total there are $1+6\cdot 2^2+4\cdot 3^2 = 61$ equivalences. Note that although some of the resulting pushout complement objects might be isomorphic, none of these isomorphisms commutes with the diagram. The pushout complements, including the two morphisms g and f', are therefore not isomorphic.

Before we extend the previously shown construction to partial morphisms, we will first prove the correctness of Proposition 3.26. Due to its length we split the proof into Lemma 3.28 stating the soundness and Lemma 3.29 stating the completeness.

Lemma 3.28. *Let $f\colon G \rightarrow H$ and $g'\colon H \rightarrow J$ be two morphisms satisfying the conditions of Proposition 3.21, i.e. at least one pushout complement exists. Then every equivalence relation $\equiv$ created by the construction in Proposition 3.26 generates a pushout complement.*

Proof. See Appendix B.2. □

Lemma 3.29. *Assume that $f\colon G \rightarrow H$ and $g'\colon H \rightarrow J$ are given. Then every pushout complement $\langle I, g\colon G \rightarrow I, f'\colon I \rightarrow J\rangle$ of f, g' can be obtained via the construction of Proposition 3.26. Furthermore two isomorphic pushout complements which commute with the isomorphism give rise to the same equivalence $\equiv$.*

Proof. See Appendix B.2. □

The fact that two isomorphic pushout complements give rise to the same equivalence means that the number of generated equivalences is exactly the number of different pushout complements. However, if we consider only isomorphisms on the pushout object I – without requiring commutativity of the triangles consisting of morphisms j, g_1, g_2 and j, f_1', f_2' – there will usually be fewer different pushout complements.

The construction of pushout complements in **Λ-HGp** is more complex than in **Λ-HGt**. We will therefore restrict ourselves to the computation of pushout complements where f is partial and g' is total. This setting is sufficient, since in Chapter 6 we will only use total co-matches. However, this does not bound the number of pushout complements, such that we will additionally need to define (and compute) a finite representation of all pushout complements. This representation will be introduced in Chapter 6 and depend on which order is used.

In this more restricted setting we can compute pushouts by first splitting the rule into two morphisms, a total one and a partial one, satisfying certain conditions. We can ensure that the partial morphism is injective as well as surjective and will show in Proposition 3.30 how a pushout complement for such an morphisms can be computed. By using the construction of Proposition 3.26 we will then extend this approach to a construction for arbitrary partial morphisms (still using a total co-match) in Proposition 3.33.

The construction in Proposition 3.30 works in three steps. First we generate a copy of J and add all elements of G for which f is undefined. In a second step choose an equivalence that merges some of the elements added from G. This is necessary to cover all possible conflict-free matches. In the last step we add an arbitrary number of edges which are incident to at least one node which will be deleted when forming the pushout, since such edges will be implicitly deleted as well.

Proposition 3.30 (Pushout complements in **Λ-HGp** I)**.** *Let $f\colon G \rightharpoonup H$ be a injective and surjective, partial morphism and let $g'\colon H \to J$ be a total morphism such that f and g' satisfy Proposition 3.21, i.e. a pushout complement exists. We can compute a pushout complement $\langle I, g, f'\rangle$ with $g\colon G \to I$ and $f'\colon I \rightharpoonup J$ as follows (see also Figure 3.14):*

1. *Generate $\tilde{J}$ by taking a copy of J and adding a copy v_c for every $v \in V_G$ for which $f(v)$ is undefined. Then add a copy e_c for every $e \in E_G$ for which $f(e)$ is undefined with $l_{\tilde{J}}(e_c) = l_G(e)$ and*

$$c_{\tilde{J}}(e_c)[i] = \begin{cases} g'(f(c_G(e)[i])) & \text{if } f(c_G(e)[i]) \text{ is defined} \\ v_c & \text{with } v = c_G(e)[i] \text{ if } f(v) \text{ is undefined} \end{cases}$$

   for $1 \leq i \leq ar(l_{\tilde{J}}(e_c))$.

We define the morphisms $\tilde{g}\colon G \rightarrow \tilde{J}$ *and* $\tilde{f}'\colon \tilde{J} \rightharpoonup J$ *as follows:*

$$\tilde{g}(x) = \begin{cases} g'(f(x)) & \text{if } f(x) \text{ is defined} \\ x_c & \text{if } f(x) \text{ is undefined} \end{cases}$$

$$\tilde{f}'(x) = \begin{cases} x & \text{if } x \in J \\ \text{undefined} & \text{else} \end{cases}$$

2. *Now let* $\equiv$ *be any equivalence on* $\tilde{J}$*, where*
   - *if* $x \equiv y$*, then either* $x, y \in V_{\tilde{J}}$ *or* $x, y \in E_{\tilde{J}}$*,*
   - *if* $x \equiv y$ *and* $\tilde{f}'(x)$ *is defined, then* $x = y$ *holds, and*
   - *if* $x \equiv y$ *for* $x, y \in E_{\tilde{J}}$*, then* $c_{\tilde{J}}(x)[i] \equiv c_{\tilde{J}}(y)[i]$ *for* $1 \leq i \leq ar(l_{\tilde{J}}(x)) = ar(l_{\tilde{J}}(y))$*.*

3. *We obtain a pushout complement* I *by taking a copy of* $\tilde{J}/\!\equiv$ *and adding an arbitrary (but finite) number of edges. For each such edge* e *there has to be an index* i *such that* $f'(c_I(e)[i])$ *is undefined. The morphisms* g *and* f' *are defined as follows:*

$$g(x) = [\tilde{g}(x)]_{\equiv}$$

$$f'(x) = \begin{cases} \tilde{f}'(x') & \text{if } x = [x']_{\equiv} \in \tilde{J}/\!\equiv \\ \text{undefined} & \text{else} \end{cases}$$

If we do not add any edges in step 3, this construction generates only finitely many pushout complements.

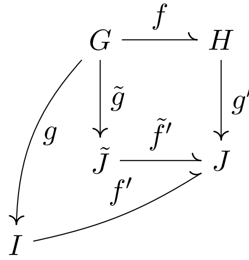


Figure 3.14.: Pushout complement diagram for the construction in Proposition 3.30

Proof. See Appendix B.2. □

The construction presented in Proposition 3.30 does not generate all pushout complements, but only pushout complement where the morphism g is conflict-free wrt. f. Since

in our analysis matches will always be conflict-free, or even injective, these pushout complements are a sufficient subset. Before we can state the completeness of the construction wrt. this criteria in Lemma 3.32, we have to prove that injectivity is preserved by pushouts in the sense that f' is injective, if f is. Note that in any category monomorphisms are preserved in this sense, but monomorphisms in Λ-**HGp** are all morphisms which are injective and total.

Lemma 3.31. *Let $f\colon G \rightharpoonup H$ be a partial, injective morphism and let $g\colon G \rightharpoonup I$ be any partial morphism. The morphism f' of the pushout $\langle J, f', g'\rangle$ is injective.*

Proof. See Appendix B.2. □

Using this lemma we can finally state the completeness result of the construction of Proposition 3.30.

Lemma 3.32. *Let $f\colon G \rightharpoonup H$ be a injective and surjective, partial morphism and let $g'\colon H \rightarrow J$ be a total morphism. Every pushout complement $\langle I, g, f'\rangle$ with $g\colon G \rightarrow I$ and $f'\colon I \rightharpoonup J$ where g is conflict-free wrt. f can be obtained by the construction of Proposition 3.30.*

Proof. See Appendix B.2. □

Using previous results we can now state a procedure for the construction of pushout complements (where the match is conflict-free) for arbitrary rules and total co-matches.

Proposition 3.33 (Pushout complements in Λ-**HGp** II)**.** *Let $f\colon G \rightharpoonup H$ be a partial morphism and let $g'\colon H \rightarrow J$ be a total morphisms, as shown in Figure 3.15. We can construct every pushout complement I' with morphisms $k\colon G \rightharpoonup I$ and $f'\colon I \rightharpoonup J$ where k is conflict-free wrt. f as follows:*

1. *Split f into two morphisms $f_1\colon G \rightharpoonup G'$ and $f_2\colon G' \rightarrow H$ with $f = f_2 \circ f_1$ where f_1 is injective and surjective, and f_2 is total.*
2. *Use the construction of Proposition 3.26 to compute $\langle I', g, f_2'\rangle$, a pushout complement of f_2, g' with $g\colon G' \rightarrow I'$ and $f_2'\colon I' \rightarrow J$.*
3. *Use the construction of Proposition 3.30 to compute $\langle I, k, f_1'\rangle$, a pushout complement of f_1, g with $k\colon G \rightarrow I$ and $f_1'\colon I \rightharpoonup I'$.*
4. *We define f' as the composition $f' = f_2' \circ f_1'$.*

This construction will generate finitely many pushout complements if and only if the construction of Proposition 3.30, will compute finitely many pushout complements.

Proof. See Appendix B.2. □

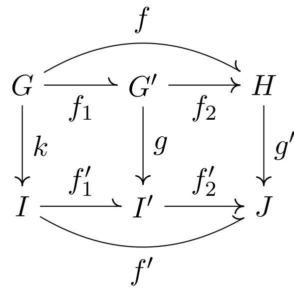


Figure 3.15.: Pushout complement diagram for the construction in Proposition 3.33

Chapter 4
Decidability Results for Graph Transformation

Although the decidability of classical verification problems, e.g. reachability, has not been studied as extensively for graph transformation systems as it has been done for related formalism like Petri nets [DFS98], some results exist. Unfortunately, due to the fact that graph transformation systems are Turing-complete, many interesting problems are undecidable. However, some of these problems can become decidable by imposing restrictions on the rules or graphs. In this chapter I will present some possible restrictions and show their effect on the decidability of the reachability and coverability problems. Note that all these restrictions can still generate infinite state systems. For the coverability we use the subgraph ordering, i.e. a graph G_1 is smaller than a graph G_2, if G_1 can be obtained by deleting nodes and edges of G_2 (where a node can only be deleted if all incident edges are deleted as well). We cover this and alternative orders in Chapter 5 in greater detail.

Most of the results of this chapter were previously published in a survey paper at RTA '12 [BD+12b; BD+12a]. However, the proofs were only for injective matches and have been extended in this chapter to general and conflict-free matches, whenever possible. Note that for any transition system induced by a GTS using general or conflict-free matches, we can define a GTS inducing the same transition system when using injective matches. The effect of non-injective matches can be encoded into the rule set. Unfortunately, we cannot automatically use this encoding to infer results for all types of matches, since it does not guarantee that the resulting GTS satisfies the same restrictions the original system does.

4.1. Restrictions on the Deletion and Creation of Nodes

General graph transformation systems are more expressive than Petri nets [BC+10] (see Appendix A.1 for a brief introduction to Petri nets). We can for instance model a Petri net by a graph using edges labelled with place names [BCM05]. The graph contains one edge labelled with a place p for each token in p. A GTS simulating the Petri net can be obtained by adding one transformation rule for each transition t. The rule deletes a p-labelled edge for each token t consumes from p and creates a p-labelled edge for each token t creates in p. For basic P/T nets the edges need not be incident to any nodes, i.e. the encoding needs no nodes at all. For more complex nets, such as nets with transfer or reset arc, we can simulate a place by a single node which has on incident unary edge for each token in the place. By merging or deleting the incident nodes, we can transfer or delete all tokens of a place respectively. It is straightforward to show that a rule behaves the same as the transition it represents and that a finite net and marking are represented by a finite graph and GTS.

The main reason why GTS are more expressive than Petri nets lies in the structure of graphs and the ability to freely change this structure by rule applications. A vital element is the deletion, creation and fusion of nodes. Inspired by [BCM05] we can obtain GTSs which can be simulated by Petri nets, if we introduce restrictions on how rules may handle nodes. If the number of nodes is fixed, then there are only finitely many possibilities for edges to be incident to these nodes (there are only finitely many labels and each has a fixed arity). We can use this to introduce a place for each possibility and mark the existence of edges by tokens in the corresponding places. Transitions can adequately model graph transformation rules in this setting.

Proposition 4.1 (GTS without node deletion or creation)**.** *Let $\mathcal{T}$ be a graph transformation systems where every rule morphism $r \in \mathcal{T}$ is a bijection on nodes. Then the reachability problem and the coverability problem are both decidable for the induced transition systems $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$.*

Proof. Let G be the initial graph (for the reachability or coverability problem). In this setting V_G remains unchanged by any graph rewriting step and only the edges may change. To reduce reachability and coverability for GTS to reachability and coverability for Petri nets, we construct a Petri net from the GTS as described below. We will assume that every match is injective and later extend this proof to general and conflict-free matches.

The places of the Petri net are defined as $P = \{\langle \ell, s\rangle \in \Lambda \times V_G^* \mid |s| = ar(\ell)\}$, where Λ is the label set of G. A token in a place $\langle \ell, s\rangle$ represents an edge e with $l(e) = \ell$ and $c(e) = s$. Note that P is finite since the label set is finite and the arity of each label is fixed. The initial graph can be transformed straightforwardly into a marking of the Petri net by adding one token in the appropriate place for each edge in the graph.

To simulate the graph transformation system, we need to simulate the rule application mechanism by transitions. For each rule $r\colon L \rightharpoonup R$ we need to take every possible match of r to any possible graph with $|V_G|$ nodes into account. In fact, to cover all such matches it is sufficient to search for matches into a complete graph with $|V_G|$ nodes. However, this graph needs to contain parallel edges if L contains parallel edges.

Let k be the largest number of parallel edges in L and let n be the number of nodes of G. We define a complete graph $K_{n,k}$ on the nodes of G which has every possible edge up to k times (in parallel), i.e. $K_{n,k} = \langle V_G, E_K, c_K, l_K\rangle$ with $E_K = \{\langle \ell, s, i\rangle \in \Lambda \times V_G^* \times \mathbb{N}_0 \mid |s| = ar(\ell) \wedge 0 \leq i < k\}$ where an edge is connected and labelled by $c_K(\langle \ell, s, i\rangle) = s$ and $l_K(\langle \ell, s, i\rangle) = \ell$, respectively. Note that this means that a node v may for instance be incident to $3 \cdot k$ different binary A-edge, k times as the source (only), k times as the target (only) and k times as the source and target. We can now simulate the application of r by computing all injective matches of r to $K_{n,k}$ and adding one transition to the Petri net for each match. No graph larger than $K_{n,k}$ needs to be considered, since for any match to a larger graph there is a match to $K_{n,k}$ generating the same transition. We generate the transition as follows.

Let $m\colon L \rightarrowtail K_{n,k}$ be a injective match and let $\langle H, r'\colon K_{n,k} \rightharpoonup H, m'\colon R \rightharpoonup H\rangle$ be the pushout of r and m (see Figure 4.1). For any place $p = \langle \ell, s\rangle$ of the Petri net we define $noe(p, X)$ as the number of edges in X with are represented by p, i.e. $noe(\langle \ell, s\rangle, X) = |\{e \in E_X \mid c(e) = s \wedge l(e) = \ell\}|$. We now add the transition $t_{r,m}$ to the Petri net which, for each place p, consumes $noe(p, m(L))$ tokens from p and generates $noe(p, m(L)) - noe(p', K_{n,k}) + noe(p, H)$ tokens in p. Note that for $noe(p, H)$ we use $V_H = V_G$, which we can safely assume since r is bijective on nodes.

The transition $t_{r,m}$ consumes all matched edges, making sure that the concrete, matched graph contains at least enough edges for the match to be valid. On the other hand by definition the exact number of edges deleted by the rule (with this specific match) is deleted from the corresponding places as well. Thus, the Petri net precisely simulates the GTS.

$$
\begin{array}{ccc}
\overline{L} & \xrightharpoonup{\;\overline{r}\;} & \overline{R} \\
{\scriptstyle \overline{m}}\downarrow & & \downarrow{\scriptstyle \overline{m}'} \\
L & \xrightharpoonup{\;r\;} & R \\
{\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\
K_{n,k} & \xrightharpoonup{\;r'\;} & H
\end{array}
$$

Figure 4.1.: Shows the two pushout squares used in the proof of Proposition 4.1

In general a GTS using non-injective matches can be simulated by a GTS using injective matches by a simple extension of the rule set. For each original rule $\overline{r}\colon \overline{L} \rightharpoonup \overline{R}$ and every total surjective morphism $\overline{m}\colon \overline{L} \to L$ to any graph L, as seen in Figure 4.1,

we add r to the new rule set, where $\langle R, r, \overline{m}' \rangle$ is the pushout of $\overline{r}$ and $\overline{m}$. Any match from $\overline{L}$ to $K_{n,k}$ can be split into $\overline{m}, m$ such that the non-injective part of the match is encoded into the rule r. Note that $\overline{m}'$ may be partial, but is total on nodes, since $\overline{r}$ and $\overline{m}$ are. Thus, every rule r is a bijection on nodes and r can be simulated according to the previous part of this proof. The same approach is possible for conflict-free matches, taking all conflict-free $\overline{m}$.

Hence, since coverability and reachability are decidable for P/T nets, we obtain the same for this variant of GTS, regardless of the types of matches used. □

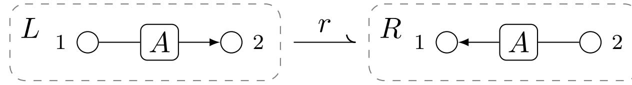
Example 4.2. An example of the encoding used in the proof of Proposition 4.1 is shown in Figure 4.2. The GTS consists of only one rule r shown in Figure 4.2a and we assume that the initial graph has (exactly) two nodes and consists only of A-labelled edges. There are four possibilities for A-labelled edges to be incident to these two nodes (in the following named 1 and 2) and each possibility is represented by its own place in Figure 4.2b. The initial graph is unambiguously defined by a marking where the number of tokens of a place is the number of parallel edges of the graph with the corresponding sequence of incident nodes.

To simulate the application of r we add four transitions, each representing a different matching of the rule into a graph with two nodes. Note that the two left transitions are only added if we allow non-injective matchings. In that case we would first generate a second rule r' identical to r with the exception that the two nodes in the left and right-hand sides are merged, and then compute injective matches for both r and r'. For graphs with more than two nodes this encoding works in the same way, but the necessary number of places and transitions will depend exponentially (or worse) on the number of nodes.

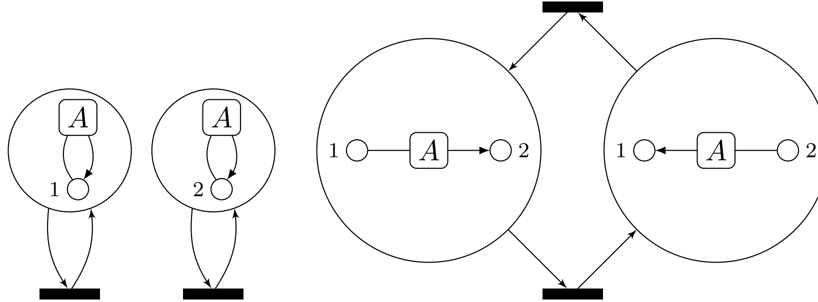
If we allow node fusion and node deletion (including the deletion of incident dangling edges) while recreating the same number of nodes, we are equivalent in expressiveness to Petri nets with transfer arcs. This allows us to prove the decidability of coverability, but reachability is undecidable. Contrary to the previous proposition we restrict ourselves to injective matches to ensure that a rule does not accidentally increase the number of nodes by using a non-injective match.

Proposition 4.3 (GTS with a constant number of nodes)**.** *Let $\mathcal{T}$ be a graph transformation systems where every rule morphism $(r\colon L \rightharpoonup R) \in \mathcal{T}$ satisfies $|V_L| = |V_R|$. Then the reachability problem is undecidable for the class of all $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$, but the coverability problem is decidable.*

Proof. Injective matches ensure that the number of nodes of a graph stays constant during rewriting, thus we can use an encoding of GTS into Petri nets similar to the one in the proof of Proposition 4.1. In order to deal with partial morphisms (i.e. node deletion) and non-injective ones (i.e. node fusion), we introduce transitions with transfer



(a) Rule used in Example 4.2



(b) Exemplary encoding of the rule in Figure 4.2a into a Petri net

Figure 4.2.: Example of encoding GTS into Petri nets

arcs that can transfer all tokens contained in a given set of places into another place. Reset arcs [DFS98] are a special case in which the transferred tokens are moved to a sink place (which only deletes its tokens).

Node deletion and subsequent recreation can be simulated via reset arcs. Whenever a node $v \in G$ is deleted, all places $\langle \ell, \alpha v \beta \rangle$ of the encoding shown in the proof of Proposition 4.1 have to be reset, where α, β are any sequences of nodes. These are all the places that represent edges incident to v. Note that from some of the places reset by a transition we may still need to ensure the existence of a certain number of tokens to ensure that enough edges exist for the corresponding match to be valid.

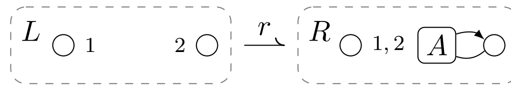
Similarly, node fusion can be simulated by transfer arcs. Let N be a set of nodes which are merged and choose any $v \in N$ to merge the other nodes into. We need to transfer the tokens of all places $\langle \ell, \alpha v' \beta \rangle$ for any $v' \in N$ with $v \neq v'$, where α, β are any sequences of nodes, to places $\langle \ell, \alpha v \beta \rangle$. This effectively changes all edges incident to any v' at an index i to an edge incident to v at the same index and with an otherwise unchanged node sequence.

Hence we can encode all GTS conforming to the restrictions into transfer nets, inheriting the decidability result from coverability of transfer nets. On the other hand, every reset net can be encoded into a GTS with the above restrictions (see [BCM05]). Hence reachability is undecidable for this class of GTS. □

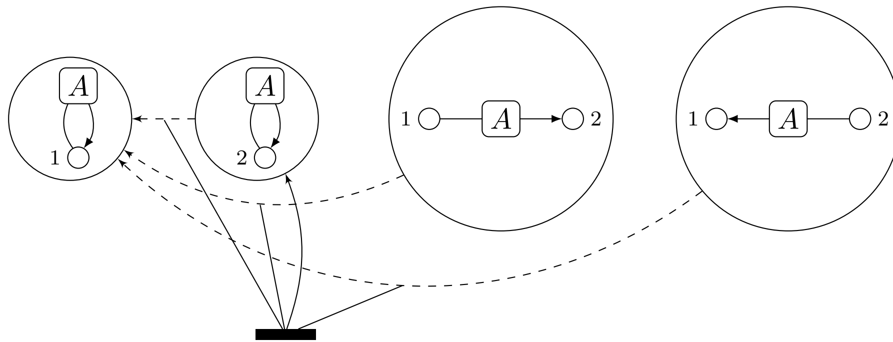
Note that already for conflict-free matches it is not guaranteed that a rule preserves the number of nodes of a graph, even if the rule satisfies $|V_L| = |V_R|$. In fact, it may increase the number of nodes if the match is non-injective on a set of nodes which are merged by the rule. General matches may also decrease the number of nodes whenever there is a

conflict.

Example 4.4. Assume we have a graph with two nodes and only binary A-labelled edges, as in Example 4.2 and let the rule r be given, as shown in Figure 4.3a. The rule merges two nodes and generates a new one with an incident A-labelled loop, such that the overall number of nodes remains constant. Note that if we would use a match where both nodes of the left-hand side are mapped to the same node, the application of this rule would actually increase the number of nodes, thus we restricted to injective matches.



(a) Rule used in Example 4.4



(b) Exemplary encoding of the rule in Figure 4.3a into a Petri net

Figure 4.3.: Example of encoding GTS with node fusion and deletion into Petri nets

The Petri net simulating this rule by using the encoding of Proposition 4.3 is shown in Figure 4.3b. The transition merges node 2 into node 1 by transforming all edges incident to node 2 to edges incident to node 1. Thus, we need three transfer arcs each transforming a different kind of edge incident to node 2. Note that after the transfer arcs removed all edges incident to node 2 (i.e. node 2 is no longer part of the old graph encoding), we can "reuse" node 2 to store the loop incident to the newly created node.

For r a total of two injective matches exist to a graph with two nodes. The transition added for the second match is symmetric to the shown one, i.e. all tokens are transferred to the second place and then one token is added to the first place.

4.2. Non-Deleting Graph Transformation Systems

Now consider GTS that are *non-deleting*, i.e. every rule morphism r is total and injective. This means that every rule application either increases the rewritten graph in size (nodes

or edges), regardless of the match types used, or r is an isomorphism. Since isomorphisms need not be applied – they do not change the graph – we obtain a monotonicity that causes the reachability problem to be decidable.

Proposition 4.5 (Reachability for non-deleting GTS)**.** *Let $\mathcal{T}$ be a non-deleting GTS. The reachability problem for $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ is decidable.*

Proof. In this setting reachability is decidable, due to the monotonicity of the rules. Let $r\colon L \rightarrowtail R$ be a rule, let $m\colon L \to G$ be a (possibly non-injective) match and let $\langle H, r', m'\rangle$ be the pushout of r and m. Since pushouts preserve monomorphism, r' is total and injective as well. This means that H contains a subgraph isomorphic to G, namely the image of r'. Thus, H contains at least as much nodes and edges as G.

Now let G_0 be the initial graph and let G_f be the graph we want to reach from G_0. We start with G_0 and apply all (finitely many) rules to derive all possible graphs. We drop every graph which is larger that G_f, i.e. has more nodes or more edges, and we apply all rules to reached graphs, which are still smaller. We stop either if we reach the desired graph G_f or if can no longer derive graphs smaller than G_f. The last condition will be satisfied at some point, since the number of graphs smaller than G_f is finite (up to isomorphism). □

Interestingly, the monotonicity used in the previous proof has no effect on coverability and we can show that coverability is in fact undecidable, although it is often considered easier than reachability.

Proposition 4.6 (Coverability for non-deleting GTS)**.** *The coverability problem is undecidable for the classes of all $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$, respectively, where Let $\mathcal{T}$ is a non-deleting GTS.*

Proof. We will prove this proposition by encoding a Turing machine into a non-deleting GTS, such that the Turing machine holds if and only if a certain graph is coverable in $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$. We will then see that this encoding can also be used to show the undecidability for $\mathcal{T}_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$. For this let a deterministic Turing machine (TM) with the initial state z_0, the input word $w = w_1 \dots w_n$ and the blank symbol $\Box$ be given (see Appendix A.2 for a brief definition of a Turing machine). We define the initial graph G_0 for the coverability problem as shown in Figure 4.4.

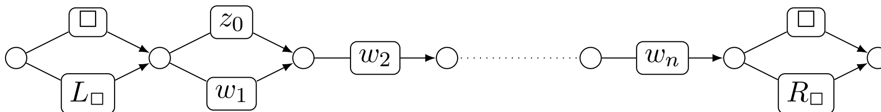


Figure 4.4.: Initial graph for the Turing machine encoding

The GTS consist of so-called δ-rules simulating the TMs transition function and some auxiliary rules. We add a δ-rule for every input of the TMs transition function $\delta(z, \alpha) = \langle z', \alpha', \beta \rangle$ where Γ is the tape alphabet, as shown in Figure 4.5.

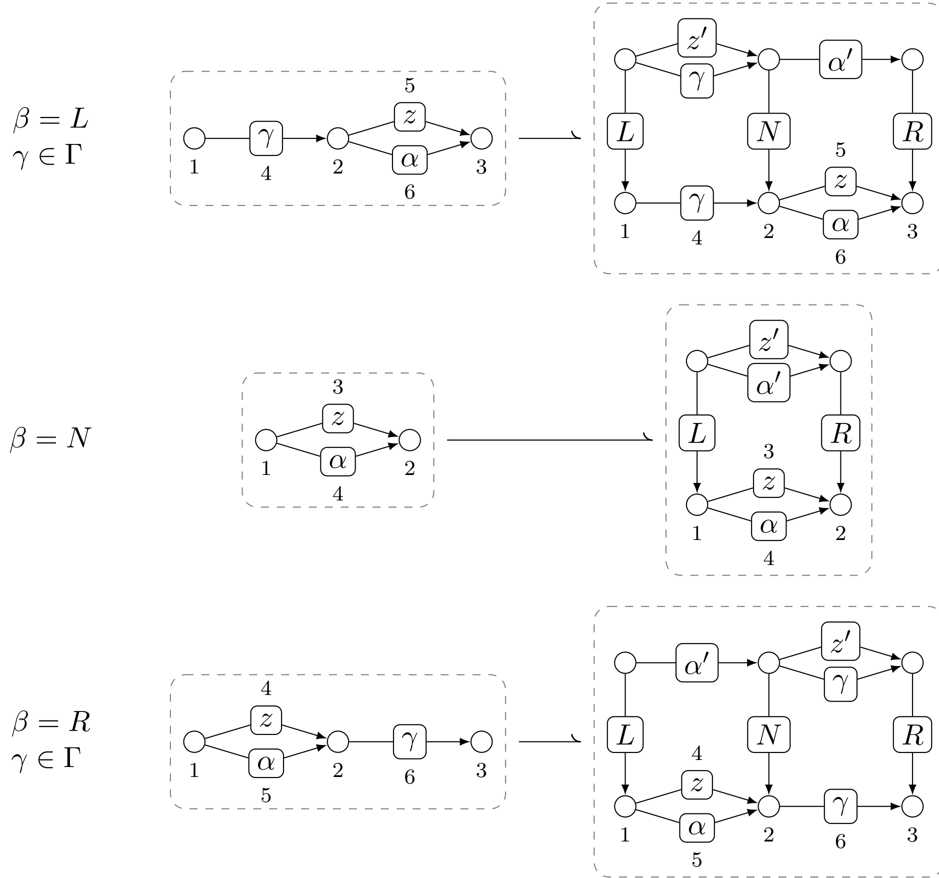


Figure 4.5.: Shows δ-rules added for each input $\delta(z, \alpha)$ of the transition function

Each application of a δ-rule adds a new level to the generated graph, later resulting in a grid-like structure, where higher levels have edges labelled L, R or N pointing to lower levels. To generate an arbitrary number of blanks at both ends of the tape as well as to copy a tape edge (edges labelled with a tape symbol) from a lower level to a higher level, we introduce auxiliary rules (so-called copy rules) as shown in Figure 4.6.

We now prove that the reduction is correct by first showing that *if the TM reaches a final state, an edge labelled with a final state is coverable in the GTS*. Without loss of generality we can assume that all used matches are injective and we will later show that non-injective matches in fact do not exist.

We define the level of a tape edge to be zero if it belongs to the initial graph or is a

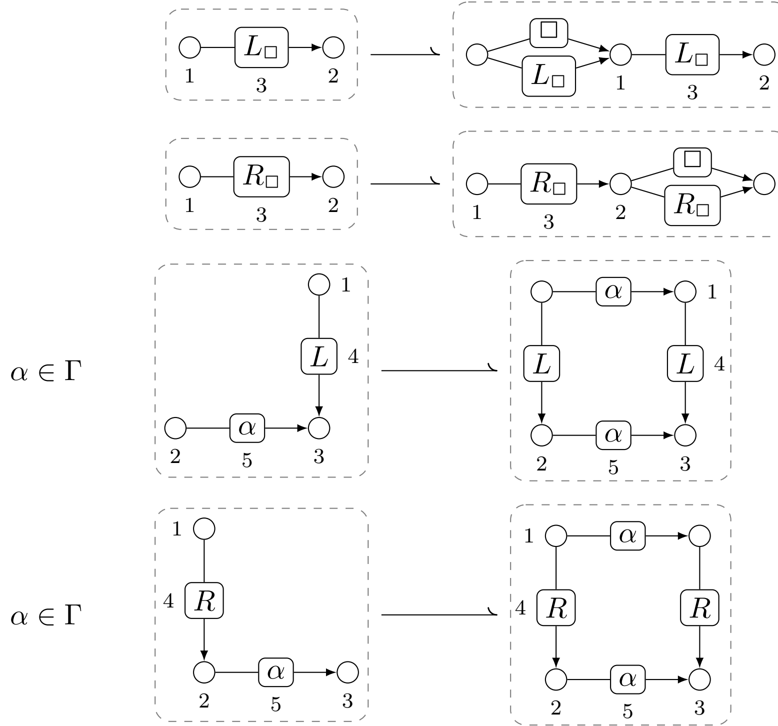


Figure 4.6.: Copy-rules are added to generate an empty tape and copy tape symbols from lower levels to higher levels

$\square$-edge parallel to a $R_\square$ or $L_\square$-labelled edge (i.e. was generated by one of the first two copy rules). If the edge was generated by any other rule application, the level is one higher than the level of any tape edges connected by outgoing edges labelled with L, R or N. Taking into account the structure of the δ- and copy-rules it can be shown by induction over the levels that any two adjacent tape edges have the same level.

Assume the TM reaches a final state. Then there exists a sequence of transition function applications leading to the final state. Because there is a GTS rule for any transition rule of the TM, this sequence has a corresponding sequence of rule applications in the GTS. However, copy rules have to be used to copy the tape to higher levels between each step of the TM computation. Hence an edge labelled with a final state is generated by the last rule application and is therefore coverable.

We now show that *if an edge labelled with a final state is coverable in the GTS, the TM reaches a final state.*

Let $\mathcal{R}$ be set of GTS rules and G_0 the initial graph. Moreover let G be the graph covering the final state z_f, generated out of G_0 by the set of rule applications $\mathcal{A}$. A rule application $a \in \mathcal{A}$ is represented by a tuple $a = \langle \rho_a, \ell_a, r_a \rangle$, where $(\rho_a \colon L_a \rightarrowtail R_a) \in \mathcal{R}$

is a rule and $\ell_a\colon L_a \rightarrowtail G$, $r_a\colon R_a \rightarrowtail G$ are total, injective morphisms. Note that these morphism are total and injective (and exist) because the rules are non-deleting. For two rule applications a, b we call b directly dependent on a if $r_a(R_a) \setminus \ell_a(L_a) \cap \ell_b(L_b) \neq \emptyset$. We define $\leqslant$ to be the smallest partial order satisfying $a \leqslant b$ if b directly depends on a and call b (indirectly) dependent on a if $a \leqslant b$ holds.

Let a_f be the rule application generating z_f. Without loss of generality we assume that $\mathcal{A}$ is minimal, i.e. for all rule applications $a \in \mathcal{A}$ it holds that $a \leqslant a_f$. Although the initial graph is a directed path, a tape generated by the GTS may be a so-called *multipath*, as shown in Figure 4.7. However, we can prove that if $\mathcal{A}$ is minimal, no multipath is generated. This especially implies that two branches of a multipath do not both contribute to the Turing machines simulation, i.e. the generation of at least one branch was superfluous.

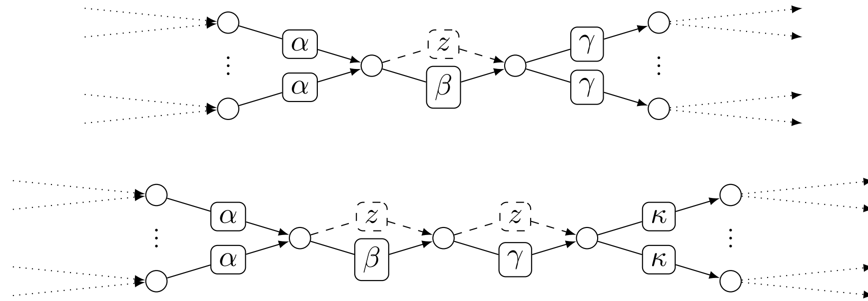
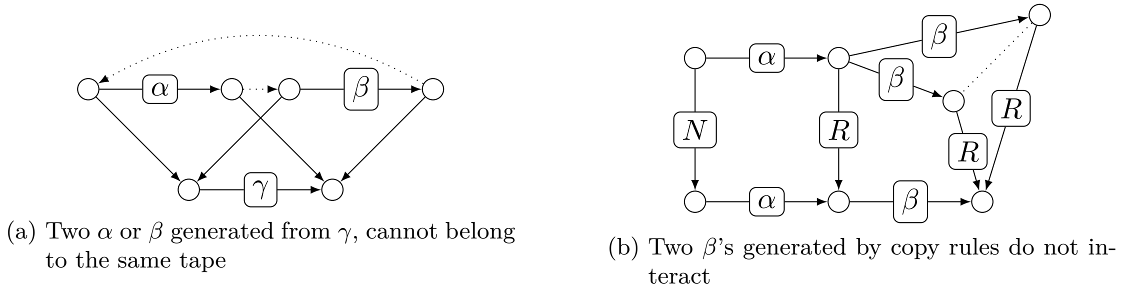


Figure 4.7.: Path-like structures (multipaths) generated by the given GTS; dashed edges indicate where state edges can be

On higher levels, depending on the rule, a δ-rule application generates one (upper graph) or two (lower graph) connected tape edges with one state edge attached (either on the left or right middle edge) in the centre. Multiple applications of the last two copy rules can then generate branching tree-like structures to the left and right. Different edges at the same depth of a tree have the same label, i.e. different paths within the multipath are labelled equally. It is important to remark that in the lower case in Figure 4.7 neither β nor γ can be copied multiple times to the current level because the middle node is connected to the lower level through an N-labelled edge. Multiple applications of δ-rules will lead to more than one multipath tape at the same level, however these tapes do not intersect and cannot be connected by any rule.

The given GTS rules are defined such that *two rule applications $a, b \in \mathcal{A}$ where the matches intersect on edges, i.e $\ell_a(L_a) \cap \ell_b(L_b) \cap E_G \neq \emptyset$, apply the same rule (i.e. $\rho_a = \rho_b$).* This is clear for different δ-rules, because the TM is deterministic and by definition there is exactly one δ-rule applicable on any tape containing exactly one state-edge. Also the match of a copy rule cannot intersect with that of a δ-rule, because copy rules can only copy tape edges to a higher level which were not already copied by a δ-rule application.

We will now show that $\mathcal{A}$ contains no rule applications where matches intersect on edges, because *whenever two rule applications $a, b \in \mathcal{A}$ intersect on edges, either $a \leqslant a_f$ or $b \leqslant a_f$ does not hold* (possibly none holds). This is the case for two δ-rule applications, as seen in Figure 4.8a, because they generate two unconnected tapes and existing tapes cannot be connected by any rule, i.e. the dotted edges cannot exist.



(a) Two α or β generated from γ, cannot belong to the same tape

(b) Two β's generated by copy rules do not interact

Figure 4.8.: Both situations in this figure can only occur in non-minimal rule application sets $\mathcal{A}$

The application of two copy rules leads to a branching in the tape (see Figure 4.8b), but tape edges of different branches cannot be part of the same match because no rule has tape edges directed in this way. The branching can be copied to higher levels, but this is not a problem as long as no δ-rule is applied to edges in different branches. However, any δ-rule applied to one of the copied edges generates an N-labelled edge which blocks the other copied edge from being copied to the tape just created, as illustrated by Figure 4.9. None of the two copy rules in question is applicable, because none can match the N-labelled edge. The second (lower) β-labelled edge could, however, be copied to a higher level by the application of aδ-rule, creating a second tape or multi-path, but this does not interfere with the first tape. Hence the minimal set $\mathcal{A}$ contains no branching and its application will result in a structure similar to one depicted in Figure 4.10.

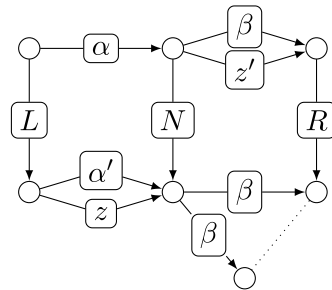


Figure 4.9.: N-labelled edges block adjacent tape-edges from being copied to higher levels

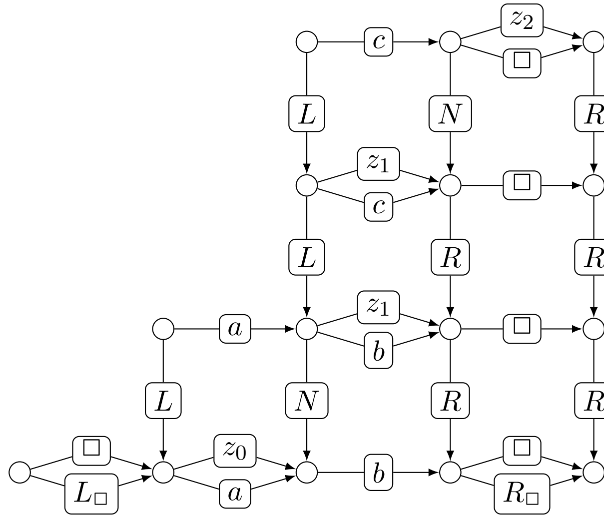


Figure 4.10.: Example of an initial graph after computing some steps of the simulation

The TM computation can be obtained from a minimal set of rule applications $\mathcal{A}$ by ordering the δ-rule applications by dependence. The TM will therefore reach a final state if the final state is coverable in the GTS.

Finally, it remains to be shown that the assumption that all matches are injective is valid even for $\mathcal{T}_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$. Since no left-hand side of a rule contains parallel edges with the same label, a non-injective match must be non-injective on nodes. For all but one rule such a match requires the existence of a loop or directed circle in the graph representing the Turing machines computation. However, the initial graph does not contain such structures and no rule can generate them. The only possibility to apply a rule non-injectively without generating a loop or directed circle is by using the left copy rule (see Figure 4.6) and matching the nodes 1 and 2 (only) to same target. In this case however, the graph would need to have an edge parallel to an L-labelled edge, which is also never the case. Thus, the type of matches used does not affect this reduction. $\square$

4.3. General Graph Transformation Systems with Minor Rules

It is a well-known result, that reachability and coverability are undecidable for general graph transformation systems. In this chapter we will prove that this still holds for some restricted classes of GTS, namely those containing a strict subset of what we call *minor rules*. A minor rule is a rule that either deletes a node, deletes an edge (without deleting the nodes) or contracts an edge, i.e. delete the edge and merge some of the incident nodes. At first this result seems to be unnecessary, but as a consequence of results we will prove in Chapters 5 and 6, we will see that reachability as well as coverability are

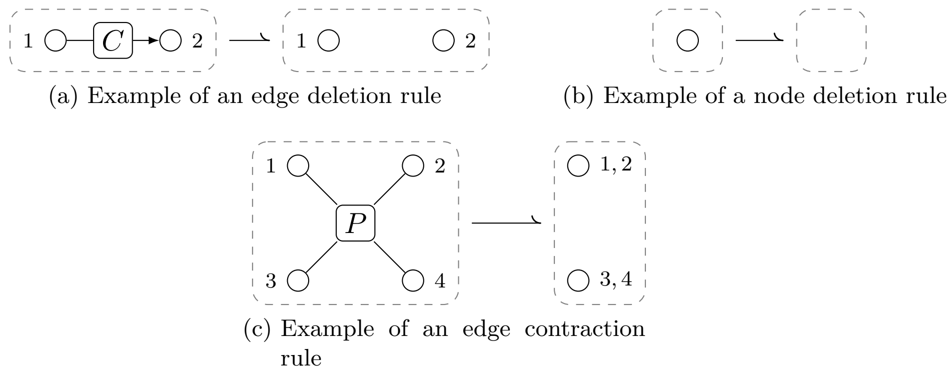
decidable for GTS containing all minor rules, but undecidable if they contain only a strict subset of these rules.

Definition 4.7. Let $r\colon L \rightharpoonup R$ be a rule. We say that r is an *edge contraction rule* if L consists of a single edge as well as its incident nodes, r is surjective, r is undefined on the edge and total as well as non-injective on nodes, i.e. there are two different nodes which have the same image under r. We call r an *edge deletion rule*, if r satisfies the conditions of an edge-contraction rule, but is injective on nodes. Finally, r is a *node deletion rule* if L is a single node and R is the empty graph.

Let $\mathcal{T}$ be a graph transformation system. We say that $\mathcal{T}$ is

- *edge-contracting* if the set of rules contains all edge contraction rules for each edge label,
- *edge-deleting* if the set of rules contains edge deletion rules for each edge label, and
- *node-deleting* if the set of rules contains a node deletion rule.

Example 4.8. An example of a edge deletion rule is shown in Figure 4.11a and an example of an edge contraction rule is shown in Figure 4.11c. In both cases the left-hand side is connected and contains (exactly) one edge, and the right-hand side consists only of nodes. Note that the only difference between these two rule types is the injectivity on nodes. Since we do not used node labels, Figure 4.11b shows the only node deletion rule possible.



(a) Example of an edge deletion rule

(b) Example of a node deletion rule

(c) Example of an edge contraction rule

Figure 4.11.: Examples of minor rules

All these minor rules appear quite naturally when modelling lossy systems. For instance the edge deletion rule could model the connection loss in a network and the node deletion rule could model the exiting of a machine or process from the network.

Even when a GTS satisfies any two of the conditions of the previous definition, the reachability problem remains undecidable, since it is still possible to encode a two counter machine into such a GTS.

Proposition 4.9. *The reachability problem is undecidable for the classes of all* $\mathcal{T}_{\mathcal{G}(\Lambda)}$*,* $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ *and* $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$*, respectively, where* $\mathcal{T}$ *is either*

- *edge-deleting and node-deleting,*
- *edge-contracting and node-deleting, or*
- *edge-deleting and edge-contracting.*

Proof. We prove this proposition by encoding a Minsky machine (a two counter machine) into a graph transformation system, such that a configuration is reachable by the machine if and only if a graph representing the configuration is reachable in the GTS (see Appendix A.3 for a brief definition of a Minsky machine). We will do the encoding such that the addition of either edge deletion and node deletion rules, edge contraction and node deletion rules, or edge deletion and edge-contraction rules will not affect the reduction. Furthermore, we define the rules in such a way that only injective matches are applicable to valid configurations. The undecidability of reachability for two counter machines [Min67] then implies the undecidability of reachability for the three classes of GTS.

Let $\langle Q, \Delta, \langle q_0, m, n\rangle\rangle$ be a Minsky machine, where Q is the set of states, $\Delta \subseteq Q \times \mathit{Cmd} \times Q$ is the set of instructions and $\langle q_0, m, n\rangle$ defines the initial state and counter values. For the GTS we use the label set $Q \cup \{c_1, B_1, E_1, c_2, B_2, E_2\}$. As shown in Figure 4.12 a graph representing a configuration contains (exactly) one zero-edge with a label in Q and two connected components representing the two counters. For each counter the number of c_i edges represents its value, B_i marks the beginning and E_i marks the end of the counter.

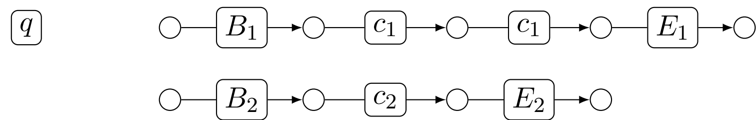


Figure 4.12.: Encoding of a Minsky machine in state q, where the value of c_1 is 2 and the value of c_2 is 1

The rules of the GTS are shown in Figure 4.13. For each $\langle q, \mathit{cmd}, q'\rangle \in \Delta$ we add one rule to the GTS. Note that the rules for incrementation and decrementation are dual in the sense that the increment rule can be obtained from the decrement rule by swapping the left and right rule sides (and inverse the morphism), and vice versa.

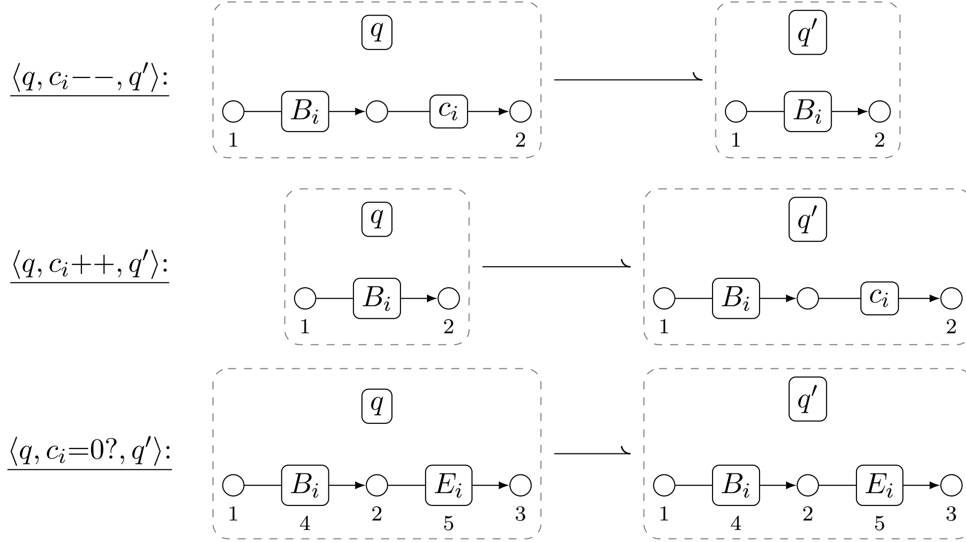


Figure 4.13.: Shows the rules for decrement, increment and zero test

We will now prove the correctness of this encoding. First if we assume that a configuration $\langle q', k', l' \rangle$ is reachable in the Minsky machine from an initial configuration $\langle q, k, l \rangle$ then it is easy to see that the graph encoding $\langle q', k', l' \rangle$ is reachable from the graph $\langle q, k, l \rangle$ in the graph transformation system described in Figure 4.13. We can simply mimic the used instructions, since if the instructions are applicable, the counters have sufficient values and the rules are applicable as well. For this we need not apply edge deletion, node deletion or edge contraction rules. Hence this direction of the proof holds for the three classes of graph transformation systems described in the statement of this proposition.

The other way is also straightforward, since we can also convert a sequence of rule applications to the instructions from which the corresponding rules where generated. The left-hand side of the decrement rule ensures that the counter has at least the value 1 and the zero test rule only matches if beginning and end markers of a counter are adjacent, i.e. the counter has the value zero. However, we also need to take node deletion, edge deletion and edge contraction rules into account. In the following we will prove the correctness of this proposition for all the three cases separately and prove each time, that as soon as a node deletion, edge deletion or edge contraction rule is applied, a valid configuration can no longer be reached. For the later two cases we will need to modify the GTS slightly. Before we prove the three cases, we need to emphasize some properties of the rules.

General observations. The number of edges labelled by an element of Q or B_i, E_i is invariant wrt. all rules. This means that a valid configuration graph has exactly one edge

labelled by an element of Q and exactly one edge for each of the labels B_1, B_2, E_1 and E_2. Furthermore, a valid configuration contains neither loops nor directed circles and every non-injective match requires the existence of such structures. A valid configuration consists of three (weakly) connected components and no rule of the GTS can connect elements of two different components. This also holds for node deletion, edge deletion and edge contraction rules.

Node and edge deletion. Now suppose that the graph G_f encoding a configuration $\langle q', k', l' \rangle$ of the Minsky machine is reachable from the graph G_0 encoding a configuration $\langle q, k, l \rangle$ in the graph transformation system described in Figure 4.13 extended with node deletion and edge deletion rules. Clearly, if an edge deletion rule removes an edge labelled by an element of Q or by B_i, E_i, we obtain an invalid configuration, which will remain invalid for the rest of the computation (the deleted edge cannot be added again). If a c_i-labelled edge is deleted, the component containing the B_i- and E_i-labelled edges is split into two components each containing one of those edges. This two components cannot be connected again and the configuration will remain invalid. If a node deletion rule is used, by construction at least one edge is deleted resulting in an invalid configuration as shown above. Thus, a valid configuration can only be reached if neither node deletion nor edge deletion rules where used.

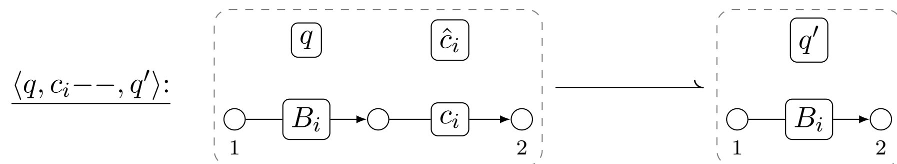


Figure 4.14.: Variant of the decrement rule for node-deleting and edge-contracting GTS

Node deletion and edge contraction. Now let the graph G_f encoding a configuration $\langle q', k', l' \rangle$ of the Minsky machine be reachable from the graph G_0 encoding a configuration $\langle q, k, l \rangle$ in the graph transformation system described in Figure 4.13 extended with node deletion and edge contraction rules. As with edge deletion rules, the contraction of an edge labelled with an element of Q or B_i, E_i leads to an invalid configuration. However, contracting an c_i-labelled edge does not. We therefore modify the increment and decrement rules slightly by adding a $\hat{c}_i$-labelled edge without incident nodes (extending the label alphabet as well) as shown in Figure 4.14. The number of $\hat{c}_i$-labelled edges in a valid configuration is always the same as the number of c_i-labelled edges (they are simultaneously incremented and decremented). If we now contract a c_i-labelled edge, the $\hat{c}_i$-labelled edge remains unchanged. Note also that edge contraction rules can by definition not be used to delete edges with an arity of less than two. Contracting a c_i-labelled edge will therefore irreversibly cause the numbers of c_i and $\hat{c}_i$ to differ. By adding the same number of $\hat{c}_i$-labelled edges as there are c_i-labelled edges to the initial

and final configuration, we ensure that no contraction rules can be applied in a valid computation. For node deletion rules the same argument holds as in the previous case.

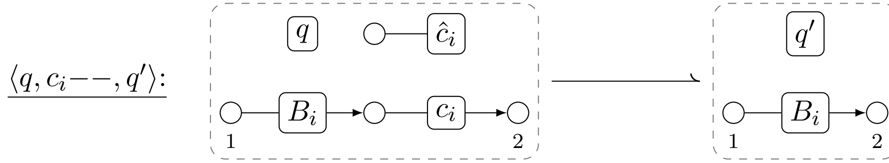


Figure 4.15.: Variant of the decrement rule for edge-deleting and edge-contracting GTS

Edge deletion and contraction. Now let the graph G_f encoding a configuration $\langle q', k', l' \rangle$ of the Minsky machine be reachable from the graph G_0 encoding a configuration $\langle q, k, l \rangle$ in the graph transformation system described in Figure 4.13 extended with edge deletion and edge contraction rules. Again we modify the increment and decrement rules as shown in Figure 4.15 to counter the problem of edge contractions. Note that the change we have done in the node deletion and edge contraction case shown in Figure 4.14 is not sufficient, since an $\hat{c}_i$-labelled edge without incident nodes can simply be deleted by an edge deletion rule. In this case the $\hat{c}_i$-labelled edge can still be deleted, but the node cannot, resulting in an invalid configuration. By the same argument as in the first case, no other edges may be deleted in a valid computation. Note that in this case a node incident to a $\hat{c}_i$-labelled edge may be matched non-injectively to one of the other nodes without requiring a directed circle, but this requires a $\hat{c}_i$-labelled edge to be adjacent to a c_i-, B_i- or E_i-labelled edge, which is no the case in a valid configuration.

Thus, we have proven that our GTS simulated a Minsky machine even when adding node deletion and edge deletion rules, node deletion and edge contraction rules or edge deletion and contraction rules. This holds regardless of the types of matches used.

Concluding remark. The reduction used in this proof is not directly transferable to directed graphs, since we use edges without incident nodes especially the case with node deletion and edge contraction rules. In [BD+12a] we published a more complex proof which only uses binary edges. However, this proof could be extended by using a modification for the node deletion and edge contraction case similar to the one shown in Figure 4.15. Instead of adding and deleting the nodes incident to $\hat{c}_i$-labelled edges, we need to have one designated node to which we add and remove both $\hat{c}_1$- and $\hat{c}_2$-labelled edges. This node also needs to be incident to the state-edge (which then has an arity of one). In this way, if we remove $\hat{c}_i$-labelled edges by a node deletion rule, we also delete the state-edge, permanently invalidating the configuration. □

The existence of edge contraction rules is sufficient for the coverability problem to be decidable, as we will show in Chapters 5 and 6. On the other hand, even with node deletion and edge deletion rules, coverability is undecidable.

Proposition 4.10. *The coverability problem is undecidable for the classes of all $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$, respectively, where $\mathcal{T}$ is a node-deleting and edge-deleting GTS.*

Proof. In this proof we use the same encoding as in the proof of Proposition 4.9 for node-deleting and edge-deleting GTS, but use the control state reachability problem for reduction (which is also undecidable [May03; Sch10]). This problem is the question whether, given an initial configuration $\langle q, k, l\rangle$ and a final state q', a configuration $\langle q', k', l'\rangle$ is reachable for any values of k', l'. We will show that a state q' is reachable if and only if the graph consisting only of a q'-labelled edge (without nodes) is coverable by the GTS.

Clearly, if a configuration $\langle q', k', l'\rangle$ is reachable, it can also be reached in the GTS. The graph representing $\langle q', k', l'\rangle$ contains a q'-labelled edge and, thus, q' is coverable. Obviously this also holds when we add node deletion and edge deletion rules.

On the other hand, a sequence of rule applications covering q' might contain applications of node or edge deletion rules. However, by removing all such applications from the sequence we obtain a valid sequence which also covers q'. This comes from a monotonicity which we will cover in Chapter 5 in greater detail: if we can cover a graph G from a graph G_0, we can also cover G from any graph larger than G_0. By not applying deletion rules, we automatically obtain a larger graph and thus q' is coverable in the GTS if and only if it is reachable by the Minsky machine. □

4.4. Relabelling Rules

The rewriting formalism we defined in Chapter 3 only allows “relabelling” of edges by deleting and recreating them. This is normally not a problem, since deletion and creation of edges has no side effects. However, some formalisms also label nodes for which the deletion and recreation will implicitly delete all incident edges.

For the DPO approach there are formalisms which allow unlabelled elements in rules [HP02; Ros75]. An element of the graph is relabelled if its corresponding preimage in the rule is unlabelled and the rule specifies its label to be set. Alternatively there are approaches which use ordered label sets to restrain the relabelling process by a partial order [PEM87]. We also used this approach in [SW14a; SW14b] for rewriting complex label structures of coloured Petri nets.

Unfortunately, not many extensions of SPO exist that are capable of relabelling. We will therefore define a simple extension in this section which is only capable of relabelling, but not changing the graph structure. Note that for general GTS with relabelling the undecidability results in the previous sections still hold. We also introduce node labels (in addition to edge labels) for more flexibility and will see that in some cases node and edge labels cannot be simulated by each other.

Definition 4.11 (Doubly-labelled hypergraph)**.** Let Λ_e and Λ_n be two disjoint, finite sets of labels. A $\langle \Lambda_V, \Lambda_E\rangle$*-hypergraph* G is a tuple $\langle V_G, E_G, c_G, l^V_G, l^E_G\rangle$, where V_G is the set

of nodes, E_G is the set of edges, $c_G \colon E_G \to V_G^*$ is the connection function, $l_G^V \colon V_G \to \Lambda_V$ is the node labelling function and $l_G^E \colon E_G \to \Lambda_E$ is the edge labelling function.

Example 4.12. Figure 4.16 shows how a relabelling rule for doubly-labelled graph may be used for modelling. In this case the nodes are machines or processes in a network and edges are network connections. Each element is labelled with its current state. Currently, data is *send* from the left machine and received (*rec*) by the right machine. Intuitively, when this rule is applied, the network connection is freed (*free*), the left machine starts to *wait* for other connections and the right machine processes the data (*pro*). Note that the actual morphism can easily be indicated by position, since the rule does not change the graphs structure.

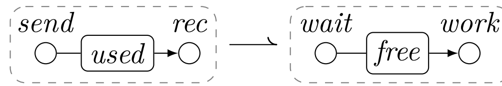


Figure 4.16.: Shows a network connection being freed after data has been transferred

For our relabelling approach we only use total morphism, since we need not change the structure of graphs. However, for rules we have to drop the restriction that morphisms preserve labels. This makes it necessary to distinguish between label-preserving morphisms and relabelling morphisms.

Definition 4.13 (Relabelling morphisms)**.** Let G, H be $\langle \Lambda_V, \Lambda_E \rangle$-graphs. A *relabelling morphism* $f \colon G \to H$ consists of two total functions $\langle f_V \colon V_G \to V_H, f_E \colon E_G \to E_H \rangle$ such that $f_V(c_G(e)) = c_H(f_E(e))$ hold for all $e \in E_G$.

We call a relabelling morphism *label-preserving* if $l_G^V(v) = l_H^V(f_V(v))$ and $l_G^E(e) = l_H^E(f_E(e))$ hold for all $v \in V_G$ and $e \in E_G$.

We use $\langle \Lambda_V, \Lambda_E \rangle$-**HGtr** to denote the category of $\langle \Lambda_V, \Lambda_E \rangle$-hypergraphs (in the following only called graphs) and relabelling morphisms. It is easy to see that in $\langle \Lambda_V, \Lambda_E \rangle$-**HGtr** pushouts always exist and are (by definition) unique up to isomorphism. However, in this category two graphs are isomorphic if they have the same structure, regardless of their labels! To obtain a relabelling formalism we have to require that the match as well as the co-match are label-preserving. Furthermore, we restrict to bijective rules (not necessarily label-preserving) to ensure that the graph structure remains unchanged and restrict to injective matches to prevent relabelling conflicts.

Definition 4.14 (Relabelling formalism)**.** A relabelling morphism $r \colon L \to R$ is called a *relabelling rule* if r is a bijective (but not necessarily label-preserving). A node or edge x is said to be relabelled by r if $l_L(x) \neq l_R(r(x))$. A *match* of a relabelling rule is a total, injective and label-preserving morphism.

We say that a graph G can be relabelled to a graph H if there is a match $m\colon L \rightarrowtail G$, $\langle H, r', m' \rangle$ is a pushout of r, m, the co-match $m'\colon R \rightarrowtail H$ is label-preserving and for all $x \in G$, if $l_G(x) \neq l_H(r'(x))$, then $x \in m(L)$.

Example 4.15. Figure 4.17 shows an application of the rule in Figure 4.16. We compute H by the standard pushout construction. However, any graph with the structure of H (regardless of labels) is a pushout. To obtain unique rewriting, Definition 4.14 states two special conditions. The co-match m' must be label-preserving, which ensures that the part of G matched by m is in fact relabelled. The second condition ensures that any part of G not matched by m is not relabelled. Note that this two conditions are sufficient for a rewriting step to be unique up to the standard definition of graph isomorphisms.

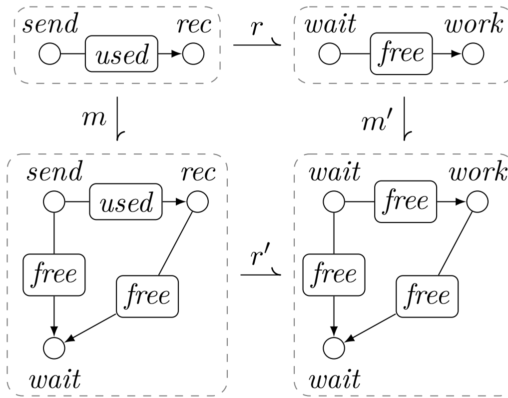


Figure 4.17.: An exemplary application of the rule in Figure 4.16

In the following we will use $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ to denote the transition system induced by $\mathcal{T}$ (we restricted to injective morphisms already), where $\mathcal{G}(\Lambda)$ is the class of all graphs with node labels Λ_V and edge labels Λ_E.

Clearly, both reachability and coverability from a fixed initial graph are decidable in this setting, since the set of derivable graphs is finite. However, by not fixing the initial graph, we obtain an existential coverability problem, which is interesting in its own right.

Definition 4.16. Let $\mathcal{T}$ be a relabelling graph transformation system and let I_V, I_E be two subsets of the node and edge labels, respectively. The *existential coverability problem* is the following: given a graph G_f, is there a graph G_0 labelled only by elements of I_V, I_E and a graph H, such that $G_0 \Rightarrow^* H$ and G_f is a subgraph of H?

The existential coverability is of interest when analysing distributed systems with a static network structure and where every component has a finite internal state. By modelling an algorithm for a distributed system using a GTS, one effectively obtains a relabelling GTS because the systems topology remains unchanged during execution of

the algorithm. When G_f represents an error configuration, the existential coverability problem transforms to the question: is there a distributed system where the modelled algorithm produces the specified error? A slightly different approach using node labels is pursued in [CMZ04], where the set of labels may be infinite and problems more specific for distributed systems are studied.

We first examine the existential coverability problem for relabelling GTS with only edge labels, i.e. the set of node labels is a singleton. It turns out that the problem can be solved by a very simple algorithm.

Proposition 4.17 (Edge Relabelling)**.** *Let $\mathcal{T}$ be a relabelling GTS with the set of node labels Λ_V. The existential coverability problem is decidable for $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ if Λ_V is a singleton, i.e., $|\Lambda_V| = 1$.*

Proof. We will show that the existential coverability problem can be decided by a simple fixed-point computation which determines the set of “reachable” edge labels. Intuitively a label is reachable if it occurs on the right-hand side of a rule, where all labels in the left-hand side are reachable or initial.

Let $\mathcal{R}$ be the rule set, H the graph to cover and I the set of initial (edge) labels. We define $\mathbf{Lab}(G)$ to be the set of all labels the edges of a graph G are labelled with. Let the label sets Lab_i with $i \in \mathbb{N}_0$ be recursively defined as follows:

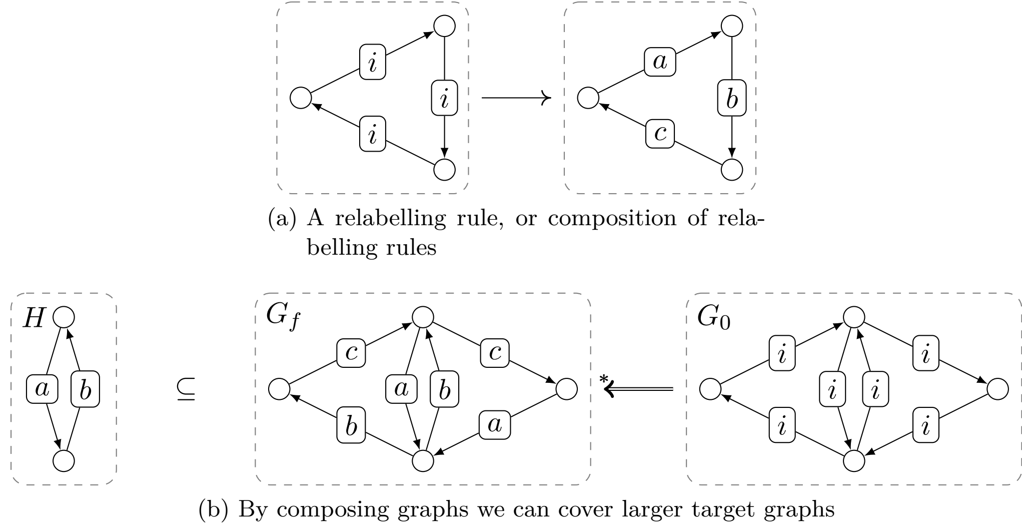
$$\begin{aligned} Lab_0 &= I \\ Lab_{i+1} &= Lab_i \cup \{\ell \in \mathbf{Lab}(R) \mid \exists(\rho\colon L \rightarrow R) \in \mathcal{R} : (\mathbf{Lab}(L) \subseteq Lab_i)\} \end{aligned}$$

Because the set of labels is finite, there is some $n \in \mathbb{N}_0$ such that the sequence becomes stationary, i.e. $Lab_m = Lab_{m+1}$ for all $m \geq n$. We will use Lab_* to denote this limit of the sequence. We now show that the graph H is existentially coverable by the GTS if and only if $\mathbf{Lab}(H) \subseteq Lab_*$.

If H is coverable, then there is an initial graph G_0, a graph G_f larger than H and a sequence of rule applications leading from G_0 to G_f. Because G_0 contains only initial labels, every rule applied to generate G_f satisfies the condition of Lab_i for some $i \in \mathbb{N}_0$, hence $\mathbf{Lab}(H) \subseteq \mathbf{Lab}(G_f) \subseteq Lab_*$ holds.

Now assume $\mathbf{Lab}(H) \subseteq Lab_*$ holds, hence $\mathbf{Lab}(H) \subseteq Lab_i$ for some $i \in \mathbb{N}_0$. By induction we show that any graph G satisfying $\mathbf{Lab}(G) \subseteq Lab_i$ for some i, is coverable. The idea is, that if we can derive a graph covering a single edge by a sequence of rule applications (for instance Figure 4.18a), we can do this multiple times on an arbitrary large initial graph (see Figure 4.18b) to cover graphs with more than one edge. For each edges in H we derive a graphs and compose all such graphs to cover H.

This holds for $i = 0$, because any graph containing only initial labels is coverable without applying any rules (G_0 is not fixed). Let G be a graph with $\mathbf{Lab}(G) \subseteq Lab_{i+1}$ and assume $Lab_i \neq Lab_{i+1}$. By definition of Lab_{i+1} there is a rule $\rho_\ell\colon L \rightarrow R$ for every label $\ell \in Lab_{i+1} \setminus Lab_i$, where $\mathbf{Lab}(L) \subseteq Lab_i$ and $\ell \in \mathbf{Lab}(R) \subseteq Lab_{i+1}$. We construct



(a) A relabelling rule, or composition of relabelling rules

(b) By composing graphs we can cover larger target graphs

Figure 4.18.: We can use rules (Figure 4.18a) to build larger graphs (Figure 4.18b) which cover any graph with reachable labels

a graph G' with $\mathbf{Lab}(G') \subseteq \mathit{Lab}_{i+1}$ by adding sufficient nodes and edges to G such that ρ_ℓ can be applied backwards matching any occurrence of a label ℓ in G. After all backwards applications we obtain a graph G'' with $\mathbf{Lab}(G'') \subseteq \mathit{Lab}_i$, hence G'' is coverable by the induction hypothesis. By application of the rule ρ_ℓ to G'' we obtain G', hence its subgraph G is coverable. □

The situation is similar for GTS which use only node labels and we can even use the same algorithm.

Proposition 4.18 (Node Relabelling)**.** *Let $\mathcal{T}$ be a relabelling GTS with the set of edge labels Λ_E. The existential coverability problem is decidable for $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ if Λ_E is a singleton, i.e., $|\Lambda_E| = 1$.*

Proof. The proof for this proposition is analogous to the proof of Proposition 4.17. The sets Lab_i are defined in the same way, while $\mathbf{Lab}(G)$ is now the set of node labels of G. Let H be the graph to be covered. Again by induction it can be shown that, an α-labelled node is coverable if $\alpha \in \mathit{Lab}_i$. If $\mathbf{Lab}(H) \subseteq \mathit{Lab}_*$, we can generate one graph H_v for each node $v \in V_H$ such that H_v can be reached from a graph G_v consisting only of initial labels, and there is a node $\hat{v} \in V_{H_v}$ and $l^V_H(v) = l^V_{H_v}(\hat{v})$. Note that $\hat{v}$ is also an element of G_v, since rules are bijective. Thus, we can generate an initial graph which covers H by first forming the disjoint union of all G_v and then adding an edge incident to $\hat{v}_1 \dots \hat{v}_n$ for each edge of H incident to $v_1 \dots v_n$. The correctness of this follows from the fact that any match of a rule is still applicable after adding nodes and edges. □

Now assume that we use node and edge labels and both can be modified. Interestingly, it turns out that the existential coverability problem is undecidable, since we can no longer disregard the graph structure (which is a prerequisite of the proofs of Propositions 4.17 and 4.18). We can now use both labels to encode a Turing machine into the relabelling system.

Proposition 4.19 (Node and Edge Relabelling)**.** *The existential coverability problem is undecidable for the class of all* $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$*, where* $\mathcal{T}$ *is an (unrestricted) relabelling GTS.*

Proof. We encode a Turing Machine into a node and edge relabelling GTS, such that the Turing machine halts if and only if the GTS covers an edge labelled with a final state (see Appendix A.2 for a brief definition of a Turing machine).

Let $\Lambda_V = \{I_n, L, R, F\}$ be the set of node labels and $\Lambda_E = \{I_e\} \cup ((Z \cup \{x\}) \times \Gamma)$ be the set of edge labels, where Z is the set of states and Γ is the tape alphabet of the TM with blank symbol $\square$. Only the labels I_n and I_e are initial. With the exception of I_e the edge labels denote the head position (where $z \in Z$ denotes the presence and x the absence of the head) as well as the current state of the TM and the tape symbol for each cell of the tape. Before the GTS can simulate the TM, we need to extract a tape out of the initial graph by applying the rules shown in Figure 4.19.

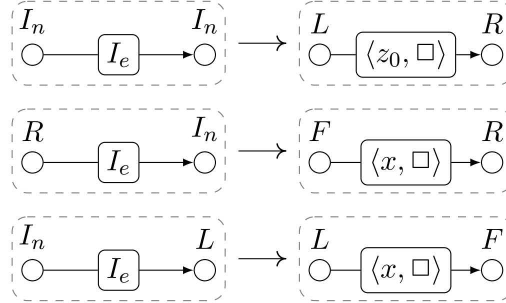


Figure 4.19.: Rules for creating a valid tape for the Turing machine

The first rule is used to start the extraction of a tape out of the initial graph (z_0 is the initial state of the TM) and the next two rules are used to extend the tape to the left and right. Note that the labels L and R mark the left and right end of the tape, and the labels F marks the middle nodes blocking any extension of the tape there. This ensures that the extracted tape on which the Turing machine is simulated, although it might be connected to its own or other nodes by I_e-labelled edges, is always a path. On a (possibly partially) extracted tape we can then apply the rules shown in Figure 4.20. These rules are a direct translation of the Turing machines transition function.

It is possible that the initial graph has an insufficient size and the simulation is blocked at some point. However, if there is a terminating computation of the Turing machine, there will be a graph of sufficient size to contain the Turing machines tape. Furthermore

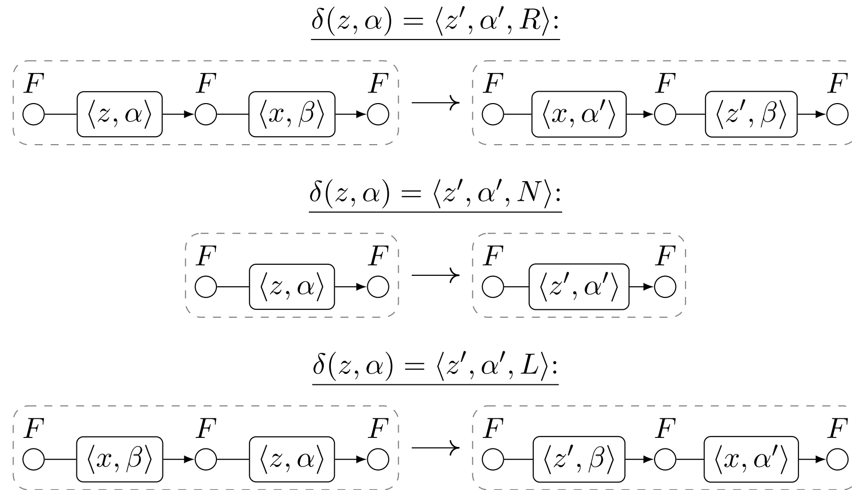


Figure 4.20.: A direct translation of the Turing machines transition function

this construction can generate multiple tapes, but these tapes can only be connected by I_e-labelled edges and there is exactly one state in each tape. □

We arrive at the surprising conclusion that while for edge or node relabelling existential coverability is easily decidable, their combination results in an undecidable problem. The fact that graphs with edge and node labels can be encoded into graphs with edge labels or node labels only, does not help, since the encoding is not surjective. Thus, the corresponding relabelling problems cannot be reduced to each other: there could always be a graph outside of the image of the encoding from which we can cover the given subgraph.

4.5. Overview

The decidability results for reachability and coverability (excluding the relabelling cases) we have shown in this chapter are summarised in Figure 4.21. The solid lines show which lower classes of GTS are subclasses of which upper classes. With the exception of the node deletion and recreation case, we could show that these results hold independent of the match types used. Coverability for bounded path graphs is decidable if all reachable graphs have bounded paths, as we will show in Chapters 5 and 6. Unfortunately this cannot be ensured by a syntactic restriction of the rules.

All results in this chapter are for the SPO approach defined in Chapter 3. Another possibility would be to use double-pushout (DPO) rewriting [CM+97]. In DPO a node cannot be deleted if it is connected to edges that are not explicitly deleted. This imposes a form of negative application condition and indeed affects the decidability results. The

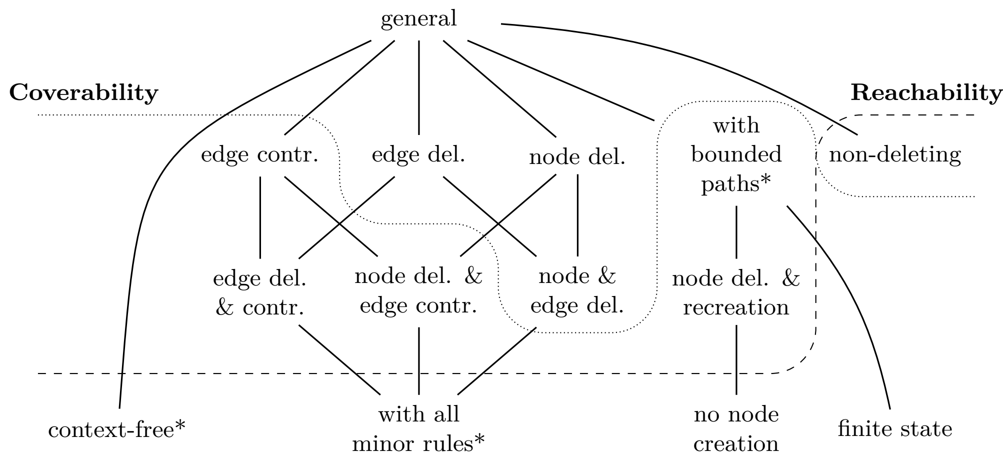


Figure 4.21.: Decidability and undecidability boundaries; we prove the cases marked with * in Chapters 5 and 6

relation between Petri nets and DPO GTS has been studied in [BCM05], where the encoding of nets into GTS deletes and recreates nodes in order to simulate the effects of inhibitor arcs from which we get undecidability of reachability and coverability even for rules which leave the number of nodes always constant.

In Chapters 5 and 6 we will extensively study how the theory of well-structured transition systems can be used to achieve decidability for the coverability problem. A few of the results are already shown in Figure 4.21 (marked by *). We will see that even if the general problem is undecidable, partial decidability results can be achieved. More precisely, we will present an algorithm which can compute whether a given graph is coverable in the class of all graphs or not coverable within a restricted class of graphs, but not the respective negated statements. To show that these results are not just of theoretical interest, we present an implementation and case studies in Chapter 7.

Chapter 5

Well-Structured Graph Transformation Systems

In Chapter 2 we introduced the theory of well-structured transition systems and there have been some approaches which use this theory in the context of graph transformation systems, mostly for the verification of protocols or distributed systems. In [AB+08] well-structured graph rewriting is used to verify programs with dynamic heaps, but they have to restrict to graphs with at most one outgoing edge – which is enough to represent heaps – to achieve well-structuredness. As ordering they use a variant of the minor order we present later in this section. In [BK+13] the authors consider the subgraph ordering, but use a forward algorithm to solve weakly fair termination (not coverability) for depth bounded systems. Finally, in [DSZ10] well-structured GTS are use for the verification of broadcast protocols, which are not directly expressible by the formalism we defined in Chapter 3. They use the induced subgraph ordering, a finer order than the standard subgraphs ordering, since it seems to be more suitable for broadcast rules.

The approach we pursue in this chapter is to transfer the generalization of WSTS, as introduced in Chapter 2, to graph transformation systems. The motivation of this generalization was to be able to use orders which are well-quasi-orders only on a restricted set of states. For this we consider three different orders, the minor ordering examined by Robertson and Seymour [RS04; RS10], the subgraph ordering and the induced subgraph ordering. Each order is a well-quasi-order, but possibly on a restricted class of graphs. An advantage of using multiple orders is that each order gives rise to a different notion of coverability. A class of graphs may be upward-closed according to one order, but not to another. Furthermore, we will see in this chapter that there is a trade-off: a coarser order is usually a well-quasi-order on a larger class of graphs, while a finer order is usually well-structured with a larger class of transformation systems. In addition to the three main orders, we will also briefly discuss other possible orders, such as the induced

or topological minor orderings covered in [FHR09; FHR12].

We published the majority of the results in this chapter in [KS14b; KS14a]. The backwards algorithm described in these papers is covered in Chapter 6, where we also state necessary and sufficient conditions for compatible orders.

In most orders a smaller graph can be obtained from a larger graph by deleting or merging nodes and edges, possibly with constraints on this operations. Since this can easily be done by partial graph morphisms, we can use classes of morphisms to represent orders. This enables us to describe rewriting and the orders in a unified categorical setting, which greatly simplifies our proofs. Although not all orders may be represented in this way, we do not consider this a strong restriction, and in fact all orders we examined so far satisfy this property.

Definition 5.1 (Representable by morphisms)**.** Let $\preceq$ be a quasi-order on graphs that is *antisymmetric up to isomorphism*, i.e. for graphs G_1, G_2 both $G_1 \preceq G_2$ and $G_2 \preceq G_1$ hold if and only if G_1 and G_2 are isomorphic.

We call $\preceq$ *representable by morphisms* if there is a class of (partial) morphisms $\mathcal{M}_{\preceq}$ such that for two graphs G_1, G_2 it holds that $G_2 \preceq G_1$ if and only if there is a morphism $\mu\colon G_1 \rightarrowtail G_2$ in $\mathcal{M}_{\preceq}$. Furthermore, for $(\mu_1\colon G_1 \rightarrowtail G_2), (\mu_2\colon G_2 \rightarrowtail G_3)$ in $\mathcal{M}_{\preceq}$ it holds that $\mu_2 \circ \mu_1$ is also contained in $\mathcal{M}_{\preceq}$, i.e. $\mathcal{M}_{\preceq}$ is closed under composition. Finally we require that for every graph G there are only finitely many $\mu \in \mathcal{M}_{\preceq}$, up to isomorphism, with the domain G. We call such morphisms μ *order morphisms*.

5.1. Minor Ordering

The first order we consider is the minor ordering. It is the coarsest order of the three orders we consider and it was shown by Robertson and Seymour in a seminal result that the minor ordering is a wqo on the set of all graphs [RS04], even for their variant of hypergraphs [RS10]. We will extend their results to our minor ordering, a slight variant of their definition for hypergraphs, and will show that case (i) of Theorem 2.21 applies, i.e. the general coverability problem is decidable for every GTS which is well-structured wrt. the minor ordering. The minor ordering was first used for well-structured GTS by Joshi and König in [JK08; JK12], but not yet as part of a more general framework.

Definition 5.2 (Minor)**.** A graph G_1 is a *minor* of a graph G_2 (written $G_1 \sqsubseteq G_2$), if G_1 can be obtained from G_2 by a sequence of deletions of nodes (including all incident edges) and contractions of edges. An *edge contraction* deletes an edge, chooses an arbitrary equivalence relation on the nodes incident to the edge and merges all nodes in each equivalence class. This includes edge deletion as a special case.

Example 5.3. Let the graph G be given as shown in the middle of Figure 5.1. Both the left and the right graphs are minors of G. The left graph can be obtained by contracting one B-labelled edge and deleting the other B-labelled edge. The right graph can be

obtained by contracting the A-labelled edge, merging the incident nodes 1, 2 and 3, 4 respectively. Note that using any other partition on these four nodes would also result in a valid contraction of the A-labelled edge.

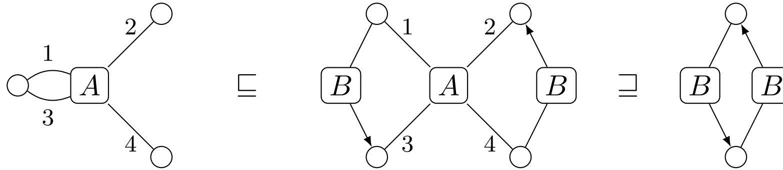


Figure 5.1.: Shows a graph (middle) and two of its minors (left and right)

The minor ordering was originally used to classify a class of graphs by forbidden minors, i.e. given a class of graphs $\mathcal{G}$ we can define $\mathcal{G}'$ as the class of all graphs which do not contain a graph of $\mathcal{G}$ as a minor. By proving that the minor ordering is a wqo, Robertson and Seymour also showed that for every downward-closed $\mathcal{G}'$ there is a finite class $\mathcal{G}$ of forbidden minors generating $\mathcal{G}'$. Although the problem of checking if a given graph G_1 is a minor of another given graph G_2 is NP-complete [LR80], they could show that the problem can be solved in polynomial time if G_1 is fixed [RS95], a bound that was reduced to quadratic time in [KKR12]. Together with the previous result this implies that for every class of graphs that can be characterized by forbidden minors the membership problem is decidable in polynomial time.

In this thesis, however, we will only use the dual property, that every class of graphs which is upward-closed wrt. the minor ordering can be characterized by its finitely many minimal elements. We define minor morphisms to represent the minor ordering.

Definition 5.4 (Minor morphism)**.** We call a partial morphism $\mu\colon G_1 \rightharpoonup G_2$ a *minor morphism* (written $\mu\colon G_1 \mapsto G_2$) if it is surjective (on nodes and edges), injective on edges and, whenever $\mu(v) = \mu(w) = z$ for some $v, w \in V_{G_1}$ and $z \in V_{G_2}$, there exists an undirected path between v and w in G_1 where all nodes on the path are mapped to z and μ is undefined on every edge of the path.

Note that this definition does not explicitly require that two paths used to merge nodes are node or edge disjoint. However, for v_1, v_2, w_1, w_2 with $\mu(v_1) = \mu(w_1) \neq \mu(v_2) = \mu(w_2)$ the two paths for v_1, w_1 and v_2, w_2 are implicitly node disjoint, since all nodes of the first path are mapped to $\mu(v_1)$ and all nodes of the second path are mapped to $\mu(v_2)$. On the other hand, the paths need not be edge disjoint, in fact, an edge of higher arity can be used in multiple paths as shown by the right minor in Figure 5.1. There the A-labelled edge merges by contraction the upper two and the lower two nodes and thus contributes to two different (node-disjoint) paths.

We prove that the class of all minor morphisms in fact represents the minor ordering by the following lemma.

Lemma 5.5. *The minor ordering is representable by minor morphisms (cf. Definition 5.1).*

Proof. We first show that minor morphisms are closed under composition. Let $\mu\colon G_1 \mapsto G_2$, $\mu'\colon G_2 \mapsto G_3$ be two minor morphisms. Obviously the composition $\mu' \circ \mu$ is surjective and injective on edges. It remains to show that the third property is also satisfied.

So let v, w be two nodes of G_1 with $\mu'(\mu(v)) = \mu'(\mu(w)) = z$. Since μ' is a minor morphism there exists a path $v'_0, e'_1, v'_1, \dots, e'_n, v'_n$ between $\mu(v) = v'_0$ and $\mu(w) = v'_n$ in G_2, where e'_i is incident to nodes v'_{i-1}, v'_i for $0 < i \leq n$. Furthermore all nodes v'_i are mapped to z by μ' and the image of all edges is undefined.

The morphism μ is surjective, hence there exist edges $e_1, \dots, e_n$ in G_1 such that $\mu(e_i) = e'_i$, where e_i is incident to nodes v_{i-1}, w_i (i.e. v_{i-1}, e_i, w_i is a path) with $\mu(v_i) = v'_i = \mu(w_i)$. Since μ is a minor morphism, v_i and w_i (for $0 < i < n$) are connected by a path $x^i_0, f^i_1, x^i_1, \dots, f^i_{m_i}, x^i_{m_i}$ in G_1 where $w_i = x^i_0$ and $v_i = x^i_{m_i}$. Furthermore, since $\mu(v_0) = v'_0 = \mu(v)$ there exists a path $x^0_0, f^0_1, x^0_1, \dots, f^0_{m_0}, x^0_{m_0}$ from $v = x^0_0$ to $v_0 = x^0_{m_0}$ and analogously a path $x^n_0, f^n_1, x^n_1, \dots, f^n_{m_n}, x^n_{m_n}$ from $w_n = x^n_0$ to $w = x^n_{m_n}$. Also, the image of all these edges under μ is undefined. So the combined path

$$x^0_0, f^0_1, \dots, f^0_{m_0}, x^0_{m_0}, e_1, x^1_0, f^1_1, \dots, f^{n-1}_{m_{n-1}}, x^{n-1}_{m_{n-1}}, e_n, x^n_0, f^n_1, \dots, f^n_{m_n}, x^n_{m_n}$$

connects v and w and it satisfies all the requirements of Definition 5.4.

It remains to be shown that a minor morphisms $\mu\colon G_2 \mapsto G_1$ exists if and only if $G_1 \sqsubseteq G_2$. Assume $G_1 \sqsubseteq G_2$, then G_1 can be obtained from G_2 by deleting nodes and contracting edges as specified in Definition 5.2. Clearly each of these operations can be separately specified by a minor morphism. If such operations are applied repeatedly, the result follows from the fact that minor morphisms are closed under composition.

Conversely, let $\mu\colon G_2 \mapsto G_1$ be a minor morphism. Now perform the following operations on G_2. First, determine all nodes in G_1 which have more than one preimage under μ. Since all preimages have to be connected by paths in G_2, where μ is undefined on the edges in the paths, we can contract all such edges, resulting in a graph G' with a minor morphism $\mu'\colon G_2 \mapsto G'$, where nodes are merged by μ' if and only if they are merged by μ. Afterwards, if an edge in G_2 has no image under μ and was not already deleted by μ', we can delete it from G' to obtain an G'' with $\mu''\colon G' \mapsto G''$. Then we can delete every node without an image under μ and obtain G_1. Note that all nodes deleted in the last step are isolated, since a node may only be undefined under a morphisms if all incident edges are undefined. Thus, all incident edges have been deleted already in one of the first two steps. Since we have restricted ourselves to edge contractions, edge deletions and node deletions, it is clear that G_1 is a minor of G_2. □

Although their notions of minors are slightly different, we can transfer the results of Robertson and Seymour to our setting

Proposition 5.6 ([RS10]). *The minor ordering is a wqo on the class of all graphs.*

Proof. We can obtain this result as a corollary of the results in [RS10], but we have to translate between the two settings. In Proposition 1.6 of [RS10] Robertson and Seymour use hypergraphs where nodes are ordered with respect to an edge (as in our case), but where edges can only be incident to the same node once, i.e., the sequence of nodes incident to an edge does not contain duplicates. This difference can be remedied by replacing edges which are incident to some node more than once with new edges of lower arity (and a new label). We will now first do this translation and then use Proposition 1.6 of [RS10] to show that our ordering is a wqo.

Assume that we have a sequence $G_1, G_2, G_3, \ldots$ of graphs. In order to make sure that the nodes incident to an edge are all distinct, we transform graphs as follows: let Λ be the label alphabet and for each label $\ell \in \Lambda$ with $ar(\ell) = k$ we enumerate all partitions on the set $\{1, \ldots, k\}$. For each such partition we fix an arbitrary order on the equivalence classes. The new label set Λ' now consists of pairs $\langle \ell, E_1 \ldots E_n \rangle$ where $E_1 \ldots E_n$ is one of the chosen sequences of equivalence classes. We set $ar(\langle \ell, E_1 \ldots E_n \rangle) = n$. Now transform a graph G into a graph G' by replacing every edge with label ℓ of arity k by a corresponding edge e' with label $\langle \ell, E_1 \ldots E_n \rangle$, where n is the number of distinct nodes incident to e. Here $E_1, \ldots, E_n$ are the equivalence classes induced by the equivalence $i \equiv j \iff c_G(e)[i] = c_G(e)[j]$. The new edge e' is incident to a node sequence $v'_1 \ldots v'_n$, where $v'_i = c_G(e)[j]$ for an arbitrary index $j \in E_i$. Note that two graphs G, H are isomorphic if and only if their transformed graphs G', H' are isomorphic.

Concerning the second requirement (well-quasi order on the labels): since we have only finitely many labels in Λ the set Λ' is finite as well and we can choose the identity as well-quasi order.

We now consider the sequence $G'_1, G'_2, G'_3, \ldots$ of transformed graphs. According to Proposition 1.6 there exists indices $i < j$ such that there is a collapse of G'_j to G'_i. More precisely, there exists a function η with domain $V_{G'_i} \cup E_{G'_i}$ such that:

1. If $v \in V_{G'_i}$, then $\eta(v)$ is a non-empty connected subgraph of $K_{V_{G'_j}}$ (where K_V is the undirected complete graph on the node set V) and the graphs $\eta(u), \eta(v)$ are pairwise disjoint for distinct $u, v \in V_{G'_i}$.

2. $\eta(e) \in E_{G'_j}$ for all $e \in E_{G'_i}$ and η is injective on edges and label-preserving.

3. For $e \in E_{G'_i}$ if $c_{G'_i}(e) = v_1 \ldots v_n$, then $c_{G'_j}(\eta(e)) = u_1 \ldots u_n$ and u_i is contained in the subgraph $\eta(v_i)$ for every $i \in \{1, \ldots, n\}$.

4. For each $v \in V_{G'_i}$ and each (undirected) edge f in $\eta(v)$, connecting $x, y \in V_{G'_j}$, there exists an edge $e \in E_{G'_j}$ which is incident to x, y. Furthermore e is not in the image of η. (The latter can be assumed since our label alphabet is finite and each label is associated with an arity. Hence every edge is bounded, i.e., has a finite neighbourhood.)

Now define a minor morphism $\mu\colon G'_j \mapsto G'_i$ as follows:

- An edge e' of G'_j is mapped to e in G'_i whenever $\eta(e) = e'$. If no such edge exists $\mu(e)$ is undefined. This is well-defined since η is injective on edges (Condition 2). Furthermore μ is injective and surjective on edges.
- Whenever a node v' of G'_j is contained in a subgraph $\eta(v)$ we map v' to v. Otherwise $\mu(v')$ is undefined. Clearly due to Condition 1 the μ obtained in this way is well-defined and surjective on nodes.

We now verify that μ is a partial morphism. Assume that $\mu(e') = e$ with $c_{G'_i}(e) = v_1 \dots v_n$ and $c_{G'_j}(e') = u_1 \dots u_n$: then $\eta(e) = e'$ and u_i is contained in $\eta(v_i)$ (Condition 3). Hence u_i is mapped to v_i, which means that the image of all nodes is defined and the map μ is structure-preserving.

Finally assume that $\mu(v') = \mu(w') = z$. This means that v', w' are both contained in $\eta(z)$. Since the subgraph $\eta(z)$ is connected there exists a path from v' to w' in $\eta(z)$. Let us denote this path by $v'_0, f_1, v'_1, \dots, v'_{n-1}, f_n, v'_n$ with $v' = v'_0$ and $w' = v'_n$. Without loss of generality we assume that the path is minimal. By Condition 4 we can require that there exists edges $e'_k \in E_{G'_j}$, which are not in the image of η and adjacent to v'_{k-1}, v'_k. This implies the existence of a path $v'_0, e'_1, v'_1, \dots, v'_{n-1}, e'_n, v'_n$ such that $\mu(e'_k)$ is undefined and $\mu(v'_k) = z$ (since all nodes $v'_0, \dots, v'_n$ are within the subgraph $\eta(z)$ and are hence mapped to z). Note that we can assume that all edges e'_l are pairwise non-equal, since otherwise there is a shorter path connecting v' and w'. In this case we can safely assume that $\eta(z)$ is complete in the following sense: if there is an edge g incident to nodes $x, x', x'' \in \eta(z)$, then there are three edges in $\eta(z)$ connecting $\{x, x'\}$, $\{x, x''\}$ and $\{x', x''\}$. Thus, if the path contains $\{x, x'\}$ and $\{x', x''\}$, we can shorten the path by using $\{x, x''\}$ instead.

This means that $\mu\colon G'_j \mapsto G'_i$ is a minor morphism. It is easy to see that the existence of μ also implies the existence of a minor morphism $\nu\colon G_j \mapsto G_i$.

Note that the collapse relation of [RS10] is finer than the minor ordering of Definitions 5.2 and 5.4. Especially a minor morphism might map straight edges to loops, which is not allowed in the collapse. However, this only means that we might "miss" some pairs of related graphs, but we will always find one. □

Since the minor ordering is a well-quasi-order on all graphs, we can use it not just for Q-restricted WSTS, but also for general WSTS. Unfortunately, since the minor ordering is so coarse, it does not form a WSTS with all graph transformation systems, since the compatibility condition of Definition 2.11 is not automatically satisfied.

Lossy Systems

A large class of systems for which the compatibility condition is often satisfied, are lossy systems [FS01], e.g. lossy counter machines [May98; Sch10]. Lossy versions of different

formalisms are often used for the verification of safety properties of protocol [AC+04] and can also be used for model checking [BM99]. Even in the context of probabilistic systems lossyness may be used for verification [Sch04; AB+05] and model checking [BS03].

In the setting of graph transformation a system is lossy when it contains edge contraction rules for every label (rules deleting an edge and merging its incident nodes according to some partition). In fact, it is sufficient to have contraction rules for edges of arity larger than one. These rules can be used to rewrite a graph to a minor to make rules applicable, thus the compatibility condition is satisfied. If a GTS does not contain these rules, they can be added, but this affects the induced transition system and results in an over-approximation of the original GTS.

Proposition 5.7 (WSTS wrt. the minor ordering)**.** *Let $\mathcal{T}$ be a GTS that contains edge contraction rules for every edge label of arity larger than one. More precisely, for $\ell \in \Lambda$ with $ar(\ell) > 1$ we require rules that delete this edge and contract the incident nodes according to every possible partition, disregarding only the trivial partition where each node is only equivalent to itself.*

Then the transition systems $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ induced by $\mathcal{T}$ on the class of all graphs using general, conflict-free and injective matches, are all WSTS.

Proof. Assume that G_2 is a minor of G_1, witnessed by a minor morphism μ, and there exists a total match $m\colon L \rightarrow G_2$ of a rule $r\colon L \rightharpoonup R$ as shown in Figure 5.2.

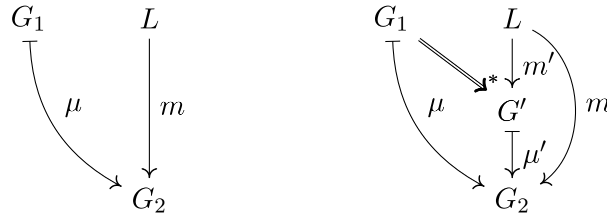


Figure 5.2.: If r is applicable to a minor of G_1 (left) we need to contract edges to generate a graph to which the rule is applicable (right)

Then, by mimicking the edge contractions performed by μ (but not the edge and node deletions), we can rewrite G_1 to G' via the contraction rules in $\mathcal{T}$ and we obtain the situation in the right of Figure 5.2. Note that G_2 is a subgraph of G', m' is conflict-free whenever m is conflict-free and analogously m' is injective if m is injective.

The pushout square can be split into two pushouts (see Lemma 3.8) as shown in Figure 5.3. We can show that μ'' is a minor morphism. First of all, according to Lemma 3.31 μ'' is injective since μ' is injective and thus, trivially a path exists whenever nodes are merged. Since surjectivity is also preserved by pushouts, μ'' is also surjective. Since μ' does not contract any edges, the deletions performed by μ' are either done by

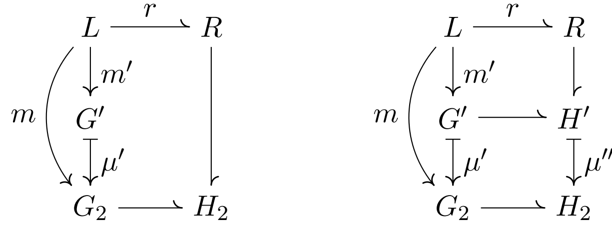


Figure 5.3.: Shows how rewriting G_2 via m results in a minor of H', the graph obtained from rewriting G' via m'

the rule (e.g. as dangling edges) or are performed by μ''. In fact, H_2 is again a subgraph of H'.

Hence, if G_2 can be rewritten to H_2, we can first rewrite any larger G_1 in possibly multiple step to G' and then rewrite G' in one step to H', with $H_2 \sqsubseteq H'$. Since the minor ordering is a wqo on all graphs, $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ are all WSTS. Note that matches of edge contraction rules are always conflict-free, since no nodes are deleted and there is only one edge. In the case of $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ we need to add edge contraction rules with partially merged left-hand sides, i.e. we merge nodes of the left-hand side according to any partition and adjust the rule and right-hand side accordingly. In this way we ensure that edge contraction is always possible, even when edges are incident to the same nodes multiple times. □

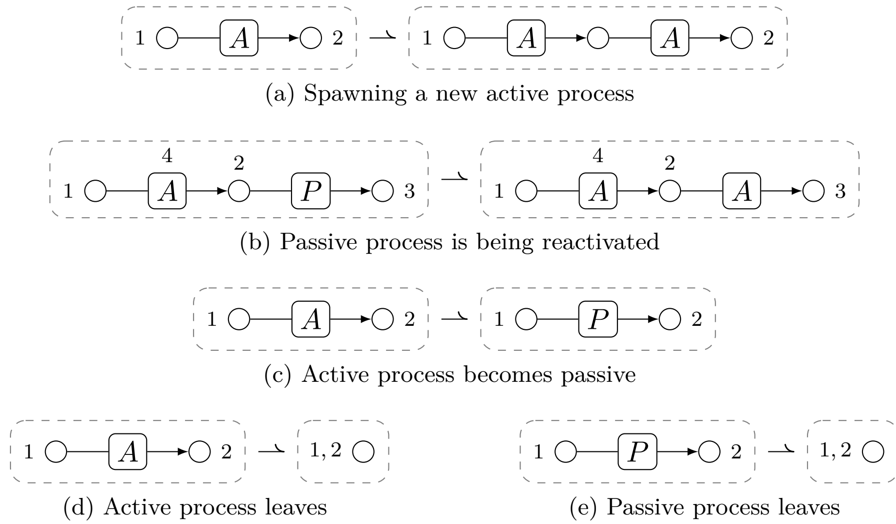


Figure 5.4.: Rules for a simple system of processes in a ring structure

Example 5.8. An example of a lossy system is shown in Figure 5.4. There we have a ring of active (A) and passive (P) processes, where each edge corresponds to a process. An active process can spawn new active processes (Figure 5.4a), reactivate a subsequent passive process (Figure 5.4b) or become passive itself (Figure 5.4c). Both active and passive processes can choose to leave the ring (Figures 5.4d and 5.4e). Note that all rules maintain the ring structure. Due to the last two rules, the GTS satisfies the conditions of Proposition 5.7 and therefore induces a WSTS. The rule set is also sufficient when using injective matches, since a rule contracting an A- or P-labelled loop is effectively a deletion rule and therefore not necessary for the well-structuredness.

Context-Free Graph Transformation Systems

Another class of well-structured graph transformation systems (wrt. the minor ordering) are context-free GTS. These GTS are the result of transferring the well-known classification for string grammars by Chomsky to graph grammars – graph transformation systems with initial graphs – and base upon work of Courcelle about recognizable (i.e. regular) languages [BC87; Cou90]. The *context-free graph grammars* in this setting are better known as *hyperedge replacement systems* which where introduced by Habel and Kreowski [DKH97] and extensively studied in Habels PHD thesis [Hab89; Hab92]. In hyperedge replacement systems the left-hand side of a rule consists of a single hyperedge with a distinct sequence of nodes and a rule may replace this hyperedge by any graph. However, a rule neither deletes nor merges nodes.

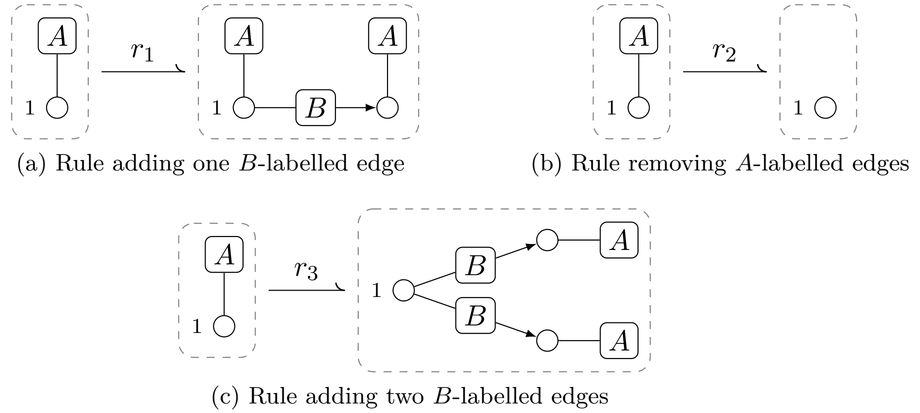
Hyperedge replacement systems are usually used for language generation, distinguishing between terminal and non-terminal edge labels. Words of the grammar (which are graphs in this case) are generated by applying rules beginning with a single edge with a designated non-terminal label until the word contains only edges with terminal labels. The membership problem of context-free graph grammars corresponds to the reachability problem when fixing the initial graph. In the following we will not distinguish between terminal and non-terminal labels, since these have no effect on the decidability result.

Definition 5.9 (Context-free graph transformation system)**.** Let r with $r\colon L \rightharpoonup R$ be are rule. We say that r is *context-free* if it satisfies the following restrictions:

- The left-hand side L has the form $L = \langle \{v_1, \dots, v_n\}, \{e\}, c_L, l_L \rangle$ with $c_L(e) = v_1 \dots v_n$, i.e., L consists of a single hyperedge, which is incident to a duplicate-free sequence of nodes.
- The rule r is defined and injective on $v_1, \dots, v_n$. Furthermore r is undefined on e.

A graph transformation system $\mathcal{T}$ is called *context-free* if every rule $r \in \mathcal{T}$ is context-free.

Example 5.10. Examples of context-free rules are shown in Figure 5.5. In a context-free grammar one would declare A a non-terminal and B a terminal label and would be interested in the graphs generated by the grammar which contain only B-labelled edges. A graph transformation system consisting of the rules r_1 and r_2 will generate all trees with B-labelled edges. A graph transformation system consisting of the rules r_2 and r_3 will only generate all binary trees.



(a) Rule adding one B-labelled edge (b) Rule removing A-labelled edges

(c) Rule adding two B-labelled edges

Figure 5.5.: Example of three context-free rules

Although context-free GTS do not necessarily satisfy the conditions of Proposition 5.7, we can still show that they satisfy the compatibility condition wrt. the minor ordering.

Proposition 5.11. *Let $\mathcal{T}$ be a context-free GTS. Then $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ are all WSTS wrt. the minor ordering.*

Proof. We can prove this proposition similar to Proposition 5.7 with the difference that just one rule application is necessary.

Let $r\colon L \rightharpoonup R$ be a (context-free) rule and let $m\colon L \to G_2$ be a match. Since the rule does not delete nodes and there is just one edge in the left-hand side (which cannot be matched non-injectively), every match m is automatically conflict-free. Now let G_1 be a graph with $G_2 \sqsubseteq G_1$, i.e. there is a minor morphism $\mu\colon G_1 \mapsto G_2$, as shown in Figure 5.6. We first show that there is an $m'\colon L \to G_1$ with $m = \mu \circ m'$ for any m, μ.

Let $e \in L$ be the edge of L. Since μ is surjective and injective on edges, there is exactly one $e' \in G_1$ with $\mu(e') = m(e)$. We define m' as $m'(e) = e'$ and $m'(c_L(e)[i]) = c_{G_1}(e')[i]$ for $1 \leq i \leq ar(l(e))$. Since L only consists of e and its incident nodes, m' is obviously well-defined and we only need to show that $m = \mu \circ m'$. By definition $\mu(m'(e)) = \mu(e') = m(e)$ and for the nodes we obtain

$$\mu(m'(c_L(e))) = c_{G_2}(\mu(m'(e))) = c_{G_2}(\mu(e')) = c_{G_2}(m(e)) = m(c_L(e)).$$

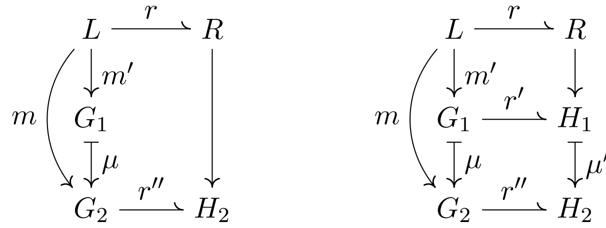


Figure 5.6.: For any μ, the match m can be split into $\mu \circ m'$

Thus, the morphisms commute. Note that m' is automatically conflict-free and m' is injective if m is injective, since both are total. It only remains to be shown that μ' is a minor morphism.

Since μ is injective on edges and injectivity is preserved by pushouts, μ' is injective on edges as well (see Lemma 3.31). The same argument also holds for surjectivity, so only the last condition remains to be shown. Assume there are two nodes $v_1, v_2 \in H_1$ with $v_1 \neq v_2$ and $\mu'(v_1) = \mu'(v_2)$. There must be preimages $v_1', v_2' \in G_1$ of v_1, v_2, since otherwise theses nodes cannot be merged in the pushout (the graph not merging them would be more general). Note that r is by definition injective, implying that r' and r'' are injective as well. This means that due to commutativity not just $r''(\mu(v_1')) = r''(\mu(v_2'))$, but also $\mu(v_1') = \mu(v_2')$ holds and the images of v_1' and v_2' under μ are defined. By definition there must be a path $x_0, e_1, x_1, \ldots, e_n, x_n$ in G_1 with $v_1' = x_0$ and $v_2' = x_n$. Since $\mu(e_i)$ is undefined for all i, none of the e_i can be e' and thus $r'(e_i)$ is defined. Analogously, $r'(x_i)$ is also defined for all i, since both r and m' are total on nodes. Hence, the path $r'(x_0), r'(e_1), r'(x_1), \ldots, r'(e_n), r'(x_n)$ with $r'(x_0) = v_1$, $r'(x_n) = v_2$ exists in H_1 and due to commutativity $\mu'(r'(e_i))$ is undefined and $\mu'(r'(x_i)) = \mu'(v_1)$ for all i. Thus, the last condition for minor morphisms is satisfied.

This means that $G_2 \sqsubseteq G_1$ implies $H_2 \sqsubseteq H_1$ and the compatibility condition is satisfied. □

Habel already showed the decidability of reachability for context-free graph transformation systems in her PHD thesis [Hab89]. In later work they could show that the problem is NP-complete [DKH97]. More precisely: there are context-free graph grammars for which the membership problem is NP-complete (in the size of the graph to be reached). Our result in Proposition 5.11 suggests that coverability wrt. the minor ordering is decidable and we will show that a backward algorithm exists in Chapter 6.

5.2. Subgraph Ordering

An order even more commonly used than the minor ordering is the subgraph ordering. It is similar to the minor ordering, but does not allow edge contraction, which obviously

causes the subgraph ordering to be finer. It was already used for instance by Meyer for the π-calculus [Mey09] or by Bansal et al. for deciding weakly fair termination [BK+13]. Unfortunately, since it is finer, the subgraph ordering is not a well-quasi-order on the class of all graphs, but going back to a result of Ding [Din92] we will be able to show that it is a wqo when bounding the length of paths by a constant. We first used this order in a general survey about decidability of graph transformation [BD+12b; BD+12a] and integrated it into our general framework in [KS14b; KS14a].

Definition 5.12 (Subgraph)**.** Let G_1, G_2 be graphs. We say that G_1 is a *subgraph* of G_2 (written $G_1 \subseteq G_2$) if G_1 can be obtained from G_2 by a sequence of deletions of edges and isolated nodes.

Note that $G_1 \subseteq G_2$ implies $G_1 \sqsubseteq G_2$, but not vice versa. Just like the minor ordering, we can represent the subgraph ordering by morphisms.

Definition 5.13 (Subgraph morphism)**.** We call a partial morphism $\mu\colon G_1 \rightharpoonup G_2$ a *subgraph morphism* (written $\mu\colon G_1 \rightarrowtail G_2$) if and only if it is injective on all elements on which it is defined and surjective.

Obviously every subgraph morphisms is also a minor morphisms, which allows us to transfer some of the properties of minor morphisms.

Lemma 5.14. *The subgraph ordering is representable by subgraph morphisms (cf. Definition 5.1).*

Proof. Let G, G' be two graphs with $G \subseteq G'$, then by definition G can be obtained from G' by a sequence of node and edge deletions of length n. W.l.o.g. we can assume that we first delete all edges and then all (isolated) nodes. For each deletion we can define a subgraph morphism $\mu_i\colon G_i \rightarrowtail G_{i+1}$, where μ_i is undefined on x – the node or edge deleted by the i-th deletion in the sequence – and the identity on all other elements. We can then compose all morphisms $\mu = \mu_n \circ \ldots \circ \mu_1$ with $G_1 = G'$ and $G_{n+1} = G$. Since injectivity and surjectivity are preserved by composition, $\mu\colon G' \rightarrowtail G$ is again a subgraph morphism.

Let $\mu\colon G' \rightarrowtail G$ be a subgraph morphism. Since μ is surjective and injective, the inverse of μ is a total, injective morphism $\mu^{-1}\colon G \rightarrowtail G'$. The image of μ^{-1} is therefore isomorphic to G and a subgraph of G', thus $G \subseteq G'$ holds. ☐

We will show that the subgraph ordering is a wqo on the class of all graphs where the longest path is bounded. For this we first generalize the notion of undirected paths for hypergraphs.

Definition 5.15 (Undirected path)**.** Let G be a graph. An (elementary) *undirected path* of length n in G is an alternating sequence $v_0, e_1, v_1, \ldots, v_{n-1}, e_n, v_n$ of nodes and edges

(i.e. $e_j \in E_G$ and $v_j \in V_G$) such that for every index $1 \leq i \leq n$ the nodes v_{i-1} and v_i are incident to e_i and the undirected path contains all nodes and edges at most once.

We use $\mathcal{G}_n$ to denote the class of all graphs, where the longest undirected path has length n.

Note that there is no established notion of directed paths for hypergraphs, but our definition gives rise to undirected paths in the setting of directed graphs (which are a special form of hypergraphs). Since we will not use any notion of direction in paths, we will simply speak of paths instead of undirected paths.

Using a result from Ding [Din92] we can show that the class $\mathcal{G}_n$ is well-quasi-ordered by the subgraph relation. A similar result was shown by Meyer for depth-bounded systems in [Mey09]. Note that we bound undirected path lengths instead of directed path lengths. For the class of graphs with bounded directed paths there exists a sequence of graphs violating the wqo property (a sequence of circles of increasing length, where the edge directions alternate along the circle). This violating sequence will still be violating for any proper generalization of directed paths to hypergraphs.

Proposition 5.16. *The subgraph ordering is a wqo on the class $\mathcal{G}_n$ for every natural number n.*

Proof. In [Din92] Ding showed that this proposition holds for undirected, simple graphs with node labels. We will now give an encoding f of hypergraphs to such graphs satisfying the following conditions:

- There is a function $g\colon \mathbb{N}_0 \to \mathbb{N}_0$ such that, if the longest undirected path in a hypergraph G has length n, then the longest undirected path in $f(G)$ has length $g(n)$.
- For every two hypergraphs G_1, G_2 if $f(G_1) \subseteq f(G_2)$ then $G_1 \subseteq G_2$.

If these two properties hold, every infinite sequence $G_1, G_2, \ldots$ of hypergraphs with bounded undirected paths can be encoded into an infinite sequence $f(G_1), f(G_2), \ldots$ of undirected graphs with bounded paths. Since Ding has shown that there are indices $i < j$ such that $f(G_i) \subseteq f(G_j)$, the same holds for the original sequence, i.e. $G_i \subseteq G_j$ holds, and $\subseteq$ is a wqo on $\mathcal{G}_n$. Note that Ding measures the length of paths by the number of nodes. However, in this proof we measure path lengths by the number of edges, both for hypergraphs as well as undirected graphs.

Let $G = \langle V, E, c, l\rangle$ be a Λ-hypergraph. We define its encoding as an undirected graph $f(G) = G' = \langle V', E', l'\rangle$ where E' consists of subsets of V' with (exactly) two elements

and $l' \colon V' \to \Lambda'$ where the components are defined as follows:

$$V' = V \cup E \cup \{\langle v, i, e\rangle \in V \times \mathbb{N} \times E \mid c(e)[i] = v\}$$
$$E' = \{\{x, y\} \mid x = \langle v, i, e\rangle \in V' \wedge (y = v \vee y = e)\}$$
$$\Lambda' = \Lambda \cup \{N\} \cup \{m \in \mathbb{N} \mid \exists \ell \in \Lambda : (m \leq ar(\ell))\}$$
$$l'(x) = \begin{cases} N & \text{if } x \in V \\ l(x) & \text{if } x \in E \\ i & \text{if } x = \langle v, i, e\rangle \end{cases}$$

Note that we assume that $N \notin \Lambda$ and $\Lambda \cap \mathbb{N} = \emptyset$ (we could easily fix this by renaming). An example of such an encoding can be seen in Figure 5.7, where the hypergraph on the left is encoded in the graph on the right-hand side.

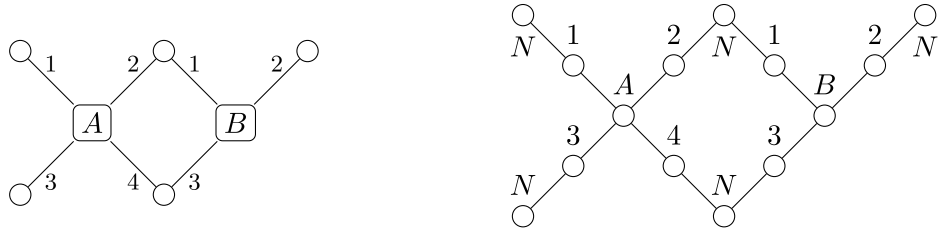


Figure 5.7.: Shows the encoding of hypergraphs into node-labelled, undirected graphs

We now show that the encoding satisfies the two necessary properties. First we observe, that every (undirected) graph generated by this encoding can be transformed back to a unique hypergraph, up to isomorphism.

Now let G be a hypergraph, where the longest undirected path is bounded by n, we show by contradiction that in $f(G)$ there can not be a path of length $4n + 10$ or longer. Assume there is such a path in $f(G)$. Apart from the first and last node, all nodes labelled with N or $\ell \in \Lambda$ on this path are adjacent to (exactly) two nodes labelled with $m \in \mathbb{N}$ and all nodes labelled with $m \in \mathbb{N}$ are adjacent to (exactly) one node labelled with N and one node labelled with $\ell \in \Lambda$ (this can again be violated in the start and end of the path). We now take the longest subpath which starts and ends with nodes labelled with N. Since this possibly shortens the path by 3 at both ends, the new path has at least length $4n + 4$. This new path can be translated back to a sequence $v_0, e_1, v_1, \ldots, v_n, e_{n+1}, v_{n+1}$ since every node labelled with N is a node of (the hypergraph) G and every node labelled with $\ell \in \Lambda$ is an edge of G. Since the path in $f(G)$ does not contain a node twice, the corresponding path in G does not contain nodes or edges twice. This violates our assumption, that the longest undirected path of G is bounded by n, thus, there is no path of length $4n + 10$ or longer in $f(G)$.

Let G_1, G_2 be hypergraphs such that $f(G_1) \subseteq f(G_2)$. Then there is a total, injective morphism $\mu \colon f(G_1) \rightarrowtail f(G_2)$. Since $f(G_i)$ contains (as nodes) all nodes and edges of G_i

(for $i \in \{1, 2\}$), we can restrict μ to $V_{G_1} \cup E_{G_1}$ and construct a total, injective morphism $\mu'\colon G_1 \rightarrowtail G_2$. The nodes of $f(G_i)$ labelled with natural numbers, ensures the morphism property on the hypergraphs. By inverting μ' we obtain an injective and surjective, but partial morphism from G_2 to G_1, i.e. a subgraph morphism. Hence $G_1 \subseteq G_2$ according to Lemma 5.14. □

The fact that the subgraph ordering is finer than the minor ordering causes it to be a wqo only on a restricted class of graphs. However, it also causes more GTS to satisfy the compatibility condition. In fact, any GTS without any form of negative application condition satisfies the compatibility condition wrt. the subgraph ordering. Any GTS we use (according to Definition 3.19) satisfies this property.

Proposition 5.17 (WSTS wrt. the subgraph ordering)**.** *Let $\mathcal{T}$ be an arbitrary GTS and let $\mathcal{G}$ be any downward-closed class of graphs on which the subgraph ordering is a wqo. The systems $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ are all $\mathcal{G}$-restricted WSTS wrt. the subgraph ordering.*

Proof. We have to show that whenever $G \Rightarrow H$ and $G \subseteq G'$, there exists H' with $G' \Rightarrow^* H'$ (here even $G' \Rightarrow H'$) and $G' \subseteq H'$.

Let $r\colon L \rightharpoonup R$ be a rule and $m\colon L \to G$ a matching that is such that G is rewritten to H, i.e. the upper square in Figure 5.8 is a pushout. Furthermore, let $\mu\colon G' \rightarrowtail G$ be a subgraph morphism (i.e. $G \subseteq G'$), then the inverse morphism μ^{-1} is total and injective. Let H' be the pushout object of r' and μ^{-1}, i.e. the lower square in Figure 5.8 is a pushout. According to Lemma 3.7 the outer square is a pushout as well and thus, we can use r to rewrite G' with the matching $m_\mu = \mu^{-1} \circ m$ (which is total) to H'. Furthermore, since μ^{-1} is injective and total, m_μ is conflict-free if m is conflict-free and m_μ is injective if m is injective.

$$
\begin{array}{ccc}
L & \xrightharpoonup{\;r\;} & R \\
{\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\
G & \xrightharpoonup{\;r'\;} & H \\
{\scriptstyle \mu^{-1}}\downarrow & & \downarrow{\scriptstyle \nu} \\
G' & \xrightharpoonup{\;r''\;} & H'
\end{array}
$$

Figure 5.8.: Using the inverse of a subgraph morphism μ, we can show that any GTS satisfies the compatibility condition wrt. the subgraph ordering

Also, μ^{-1} is a monomorphism and therefore preserved by pushout construction, i.e. ν is total and injective (see also Lemma 3.31). This means that the inverse of ν is a subgraph morphism and $H \subseteq H'$ holds. Thus, the compatibility condition holds, and since by definition $\subseteq$ is a wqo on $\mathcal{G}$, the systems $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ are $\mathcal{G}$-restricted WSTS wrt. the subgraph ordering. □

Since we already showed in Proposition 5.16 that $\subseteq$ is a wqo on the class $\mathcal{G}_n$, we obtain $\mathcal{G}_n$-restricted-WSTS wrt. the subgraph ordering.

Unfortunately, both the general and the restricted coverability problems are undecidable. For the general coverability problem (cf. Definition 2.10) this follows from our results in Chapter 4. We can show the undecidability of restricted coverability (cf. Definition 2.17), even for $\mathcal{G}_n$, by a simple reduction from the control state reachability problem for two-counter machines. Although we cannot directly simulate the zero test, i.e. negative application conditions are not possible, we can make sure that the rules simulating the zero test are applied correctly if and only if the bound n was not exceeded. A state in the two-counter machine is then reachable if and only if an appropriate graph is coverable without leaving $\mathcal{G}_n$.

Proposition 5.18. *Let $n \geq 2$ be a natural number. The restricted coverability problem for the $\mathcal{G}_n$-restricted WSTS $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ wrt. the subgraph ordering, is undecidable.*

Proof. We reduce the control state reachability problem of Minsky machines (see Appendix A.3 for a brief definition) to the restricted coverability problem using the subgraph ordering on the set of graphs $\mathcal{G}_2$, where the length of the longest undirected path is less than or equal to two. Let $\langle Q, \Delta, \langle q_0, m, k\rangle\rangle$ be the Minsky machine, where Q is the set of states, $\Delta \subseteq Q \times \mathit{Cmd} \times Q$ is the set of instructions with the set of command Cmd and $\langle q_0, m, k\rangle$ defines the initial state and counter values. We define a GTS using $\{q, q^B \mid q \in Q\} \cup \{c_1, c_2, X\}$ as the set of labels. The initial graph is shown in Figure 5.9 and illustrates how configurations of the Minsky machine are represented as graphs.

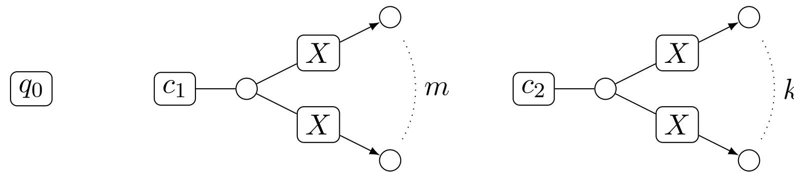


Figure 5.9.: The initial configuration $\langle q_0, m, k\rangle$ of the Minsky machine represented by a graph

For each transition rule of the Minsky machine, we add a graph transformation rule as shown in Figure 5.10. A counter is represented as a star-like structure with a main node as centre, where the value of the counter is the number of incident X-labelled edges. The centre nodes are marked by unary c_1- or c_2-labelled edges respectively. Incrementing and decrementing corresponds to creating and deleting X-labelled edges incident to the appropriate centre node, as shown in Figure 5.10. Regardless of the counters value, the longest undirected path of this structure has at most length two.

The zero test adds two X-labelled edges and "blocks" the state-edge, such that the rewritten graph has an undirected path of length three if and only if the counter was not

zero (i.e. had an X-labelled edge attached). The auxiliary rules (shown at the bottom of Figure 5.10) unblock the state to enable further computation. This ensures that the two X-labelled edges are deleted before another instruction can be applied, thus they will not affect the value of the counter.

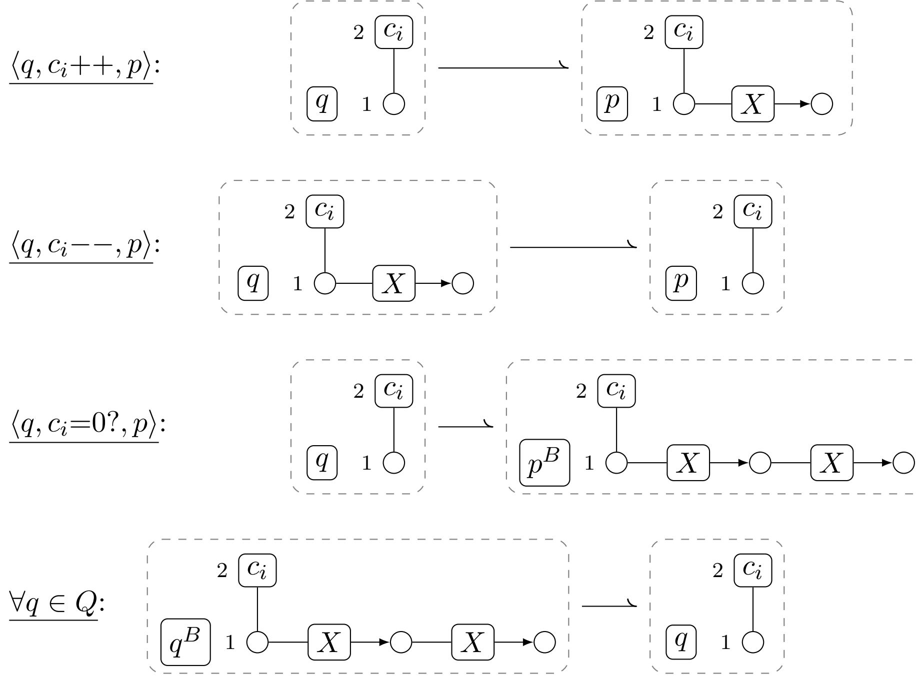


Figure 5.10.: Translation of Minsky rules to GTS rules

Obviously, if there is a sequence of transitions of the Minsky machine which leads from a configuration $\langle q_0, m, k\rangle$ to a state q_f, this sequence can be copied in the GTS and every graph generated through this sequence is in $\mathcal{G}_2$. On the other hand, if the graph consisting of a single q_f-labelled edge is $\mathcal{G}_2$-restricted coverable in the GTS, there is a sequence of rule applications corresponding to a sequence of transitions of the Minsky machine. Since this rule applications generate only graphs in $\mathcal{G}_2$, the zero-test-rule is only applied if the counters value is in fact zero and the sequence of transitions is valid (in the Minsky machine). Note that any non-injective match requires the existence of an X-labelled loop or a (directed) circle of X-labelled edges. Such structures do not exist in a valid configuration and cannot be produced by rules. Thus, this encoding is correct even when using conflict-free or general matches.

Instead of adding and removing a path of length two in the bottom two rules of Figure 5.10 one can add and remove a path of length n to show the undecidability for any $\mathcal{G}_n$. □

5.3. Induced Subgraph Ordering

Another useful order is the induced subgraph ordering, where a graph G is an induced subgraph of G' if every edge in G' where all incident nodes are present in G, is contained in G as well. This ordering was used in [DSZ10] to verify protocols using broadcast operations, for which it seems particularly suited. Broadcast operations are not directly expressible in our formalism, but in Section 6.6 we will introduce an extension we published in [DS14b; DS14a] to cover them and also instantiate the backward search for the new formalism. However, our broadcast operations are more general than those in [DSZ10], which causes our approach to be approximative.

Graph classes which are well-quasi ordered wrt. the induced subgraph ordering have been studied in [Dam90] and further by Ding [Din92]. Unfortunately, the induced subgraph ordering is not a wqo even when bounding the longest undirected path in a graph, so that we also have to bound the multiplicity of edges between two nodes. Note that this restriction is implicitly done in [Din92] since Ding uses simple graphs.

Furthermore, we do not know whether the induced subgraph ordering can be extended to a wqo on a class of hypergraphs. We therefore restrict ourselves to directed graphs which are sufficient for modelling many applications, including most of our examples. Note that this restriction will only affect Proposition 5.24, since the other definitions and lemmas can be stated for general hypergraphs.

At first, this order seems unnecessary, since it is stricter than the subgraph ordering and is a wqo on a more restricted set of graphs. On the other hand, it allows us to specify error graphs more precisely, since a graph G does not represent graphs with additional edges between nodes of G. Furthermore one can equip the rules with a limited form of negative application conditions, still retaining the compatibility condition of Definition 2.16.

Definition 5.19 (Induced subgraph)**.** Let G_1, G_2 be graphs. G_1 is an *induced subgraph* of G_2 (written $G_1 \trianglelefteq G_2$) if G_1 can be obtained from G_2 by deleting a subset of the nodes, including all edges incident to at least one of these nodes.

Note that $G_1 \trianglelefteq G_2$ implies $G_1 \subseteq G_2$, but not vice versa. Thus, we can represent the induced subgraph ordering by a restricted class of subgraph morphisms.

Definition 5.20 (Induced subgraph morphism)**.** We call a partial morphism $\mu\colon G_1 \rightarrowtail G_2$ an *induced subgraph morphism* if and only if it is injective on all elements on which it is defined, surjective, and if it is undefined on an edge e, it is undefined on at least one node incident to e.

Lemma 5.21. *The induced subgraph ordering is presentable by induced subgraph morphisms (cf. Definition 5.1).*

Proof. Let $\mu_1 \colon G_1 \rhd\!\!\rightarrow G_2$ and $\mu_2 \colon G_2 \rhd\!\!\rightarrow G_3$ be two induced subgraph morphisms. Induced subgraph morphisms are closed under composition, since injectivity and surjectivity are preserved and if $\mu_2 \circ \mu_1$ is undefined for some edge e, then μ_1 is undefined on e or μ_2 is undefined on $\mu_1(e)$ implying that $\mu_2 \circ \mu_1$ is undefined for at least one node of e.

For some graph G we can obtain any induced subgraph G' by a sequence of node deletions including all incident edges. Each morphisms $\mu_i \colon G_i \rhd\!\!\rightarrow G_{i+1}$ of this sequence, where G_{i+1} is obtained by deleting one node and all its incident edges from G_i, is an induced subgraph morphisms and since they are closed under composition, the entire sequence is as well.

On the other hand every induced subgraph morphism $\mu \colon G \rhd\!\!\rightarrow G'$ can be split into a sequence of node deletions (deleting all incident edges), since every edge deleted by μ is incident to a deleted node, hence $G' \trianglelefteq G$. □

We can again transfer Ding's results for the induced subgraph ordering to our setting. However, the simple encoding f used in the proof of Proposition 5.16 is not sufficient, since the second condition, i.e. $f(G_1) \trianglelefteq f(G_2) \implies G_1 \trianglelefteq G_2$, is not satisfied. A single node (without edges) is not an induced subgraph of a node with a loop, but the relationship does hold for the encoding. Thus, we need to modify Ding's proof directly, making it necessary to introduce the notion of a type of a graph, which we do for general hypergraphs.

Definition 5.22 (Type of a graph)**.** A graph which consists of at most a single node, possibly with incident edges, has type one. A connected graph containing at least two nodes has at most type n, if there is a node v so that the deletion of v and all incident edges splits the graph into components which each have at most type $n-1$. The type of a non-connected graph is the maximal type of its components.

For directed graphs the type of a graph is closely related to the notion of *tree-depth* (see Chapter 6 of [NM12] for a definition). The connection becomes evident when thinking of the root of a tree implying a tree-depth of n for a graph G as the node to be deleted to reduce the type of G. However, the notion of tree-depth has no trivial extension in the setting of hypergraphs.

In his proof Ding uses the fact that bounding the length of paths also bounds the type of a graph and then proves the statement by induction over the type. This relation was already mentioned in [RS85] and for instance proven in Proposition 6.1 of [NM12] (for tree-depth instead of types), but always for directed graphs. Thus, we need to restrict ourselves to directed graphs and will in the following use $\mathcal{D}_n$ to denote the class of *directed graphs* where the longest undirected path is bounded by n, and use $\mathcal{D}_{n,k}$ to denote the subclass of $\mathcal{D}_n$ where every two nodes are connected by at most k parallel edges with the same direction and label (bounded edge multiplicity).

Lemma 5.23 ([RS85; Din92])**.** *Every graph $G \in \mathcal{D}_n$ has at most type $n+2$.*

Note that contrary to Ding our type is bounded by $n+2$ instead of n, because we measure path lengths via the number of edges instead of nodes and Ding's class $\mathcal{P}_n$ *excludes* paths of length n.

It is unclear to us whether Lemma 5.23 also holds for general hypergraphs. At least the converse, i.e. that a bounded type implies bounded paths, does not hold for hypergraphs, although a directed graph with type at most n has no paths longer than 2^n [RS85] (our types correspond to Robertson and Seymour's B-type). Figure 5.11 illustrates this by an infinite set of graphs where we can add A-labelled edges and nodes at the dotted part analogous to the other edges. Then in every graph there is a path $v_0, e_1, v_1, \ldots, e_n, v_n$ of length n. Although the lengths of paths in this set is not bounded, the type is 2, since the deletion of the node x also deletes all edges. Moreover, there might also be a proof of Proposition 5.24 for general hypergraphs not using types at all.

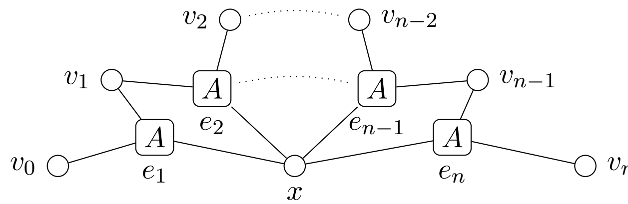


Figure 5.11.: By adding more edges and nodes at the dotted part, we can increase the longest path in this graph without increasing its type

Proposition 5.24. *Let n, k be natural numbers. The induced subgraph ordering is a wqo on the class $\mathcal{D}_{n,k}$.*

Proof. We prove this proposition by induction over the type of a graph, adapting Ding's proof in [Din92] that undirected, node-labelled graphs of bounded type are well-quasi-ordered by the induced subgraph order. From Lemma 5.23 it then follows that $\mathcal{D}_{n,k}$ is also well-quasi-ordered by induced subgraphs.

To prove this proposition we use directed graphs which are additionally node labelled, i.e. there is a second alphabet Σ of node labels and a (total) labelling function $\sigma \colon V_G \to \Sigma$. We obtain classical directed graphs if $|\Sigma| = 1$.

Let $G_1, G_2, \ldots$ be an infinite sequence of graphs of type n and with edge multiplicity bounded by k. If $n = 1$ then every G_i consists of a single node with up to $k \cdot |\Lambda|$ incident loops. Since the sets of node and edge labels are finite, there are only finitely many possibilities to attach up to $k \cdot |\Lambda|$ edges to the node, thus $G_i \trianglelefteq G_j$ for some $i < j$, i.e. $\trianglelefteq$ is a wqo on the set of all such graphs.

Now let $n > 1$. Then there is a node $v_i \in G_i$ such that the deletion of v_i (and its incident edges) splits the graph into components $G_{i,q}$ (for $1 \leq q \leq \ell_i$) of type at

most $n-1$. We define $\widetilde{G}_i$ to be the graph containing only v_i and its incident loops. Additionally we define $\widehat{G}_{i,q}$ to be $G_{i,q}$ where the label $\sigma(y)$ of every node y is changed to $\sigma'(y) = \langle f_y, \sigma(y)\rangle$, where $f_y\colon \Lambda \to \{0,1,\ldots,k\}^2$ is a function such that $f_y(\lambda) = \langle a, b\rangle$ where a is the number of incoming and b of outgoing λ-labelled edges incident to both y and v_i. Since there are only finitely many possible functions f_y (due to the multiplicity constraint), the set of labels remains finite. We extend $\trianglelefteq$ to sequences such that $\langle \widetilde{G}_i, \widehat{G}_{i,1}, \ldots, \widehat{G}_{i,\ell_i}\rangle \trianglelefteq^* \langle \widetilde{G}_j, \widehat{G}_{j,1}, \ldots, \widehat{G}_{j,\ell_j}\rangle$ if and only if $\widetilde{G}_i \trianglelefteq \widetilde{G}_j$ and there are $p_1, \ldots, p_{\ell_i}$ with $1 \leq p_1 < \ldots < p_{\ell_i} \leq \ell_j$ such that $\widehat{G}_{i,q} \trianglelefteq \widehat{G}_{j,p_q}$. As shown for the case $n = 1$, $\trianglelefteq$ is a wqo on all $\widetilde{G}_i$ and since the graphs $\widehat{G}_{i,q}$, $\widehat{G}_{j,p_q}$ are of type $n-1$, they are well-quasi-ordered by induction hypothesis. Hence, due to Lemma 2.7 (see also [Hig52]) $\trianglelefteq^*$ is also a wqo and there are indices $i < j$ such that $\langle \widetilde{G}_i, \widehat{G}_{i,1}, \ldots, \widehat{G}_{i,\ell_i}\rangle \trianglelefteq^* \langle \widetilde{G}_j, \widehat{G}_{j,1}, \ldots, \widehat{G}_{j,\ell_j}\rangle$. It remains to be shown that this implies $G_i \trianglelefteq G_j$. By Lemma 5.21 there are induced subgraph morphisms $\mu_0\colon \widetilde{G}_j \rightarrowtail\!\!\!\!\rightarrow \widetilde{G}_i$ and $\mu_q\colon \widehat{G}_{j,p_q} \rightarrowtail\!\!\!\!\rightarrow \widehat{G}_{i,q}$ for $1 \leq q \leq \ell_i$. We define the morphism $\mu\colon G_j \rightharpoonup G_i$ as

$$\mu(x) = \begin{cases} v_i & \text{if } x = v_j \\ \mu_q(x) & \text{if } x \in \widehat{G}_{j,p_q} \text{ for some } q \\ \mu_0(x) & \text{if } x \in E_{\widetilde{G}_j} \text{ and } c_{\widetilde{G}_j}(x) = v_j v_j \\ \mu^v(x) & \text{if } x \in E_{G_j} \text{ and } c_{G_j}(x) = v_j v \vee c_{G_j}(x) = v v_j \\ & \text{for } v_j \neq v \in V_{G_j} \text{ and } \mu(v) \text{ is defined} \\ \text{undefined} & \text{else} \end{cases}$$

where μ^v is any total, bijective morphism from G_j – restricted to v_j, v and the edges between them – to G_i – restricted to v_i, $\mu_q(v)$ (if $v \in \widehat{G}_{j,p_q}$) and any edges between them (both not including loops). Note that μ^v exists since v and $\mu_q(v)$ are labelled with some $\langle f, \alpha\rangle$, thus the number of edges between v_j and v is equal to the number of edges between v_i and $\mu_q(v)$ for all labels and directions.

We now show that μ is a induced subgraph morphism. First note that μ is a valid morphism since μ_q, μ_0 and μ^v are morphisms and labels of edges in G_i, G_j are the same as their representative in $\widehat{G}_{j,p_q}$, $\widehat{G}_{i,q}$ and representatives of nodes are labelled with $\langle f, \alpha\rangle$, where the original is labelled α also implying equality on labels. We then observe that μ is injective and surjective, since μ_q, μ_0 and μ^v are all injective and surjective and v_j is mapped to v_i. Assume there is an edge $e \in E_{G_j}$ for which μ is undefined. If e is contained in one of the components $\widehat{G}_{j,p_q}$ or in $\widetilde{G}_j$, at least one incident node is undefined, since μ_q and μ_0 are induced subgraph morphisms. If e connects v_j and a node v of a component $\widehat{G}_{j,z}$, then either z is not of the form p_q and μ is undefined on $\widehat{G}_{j,z}$ or $z = p_q$ and $\mu(v)$ is undefined since otherwise μ^v has a mapping for e. Since μ is an induced subgraph morphism, we obtain that $G_i \trianglelefteq G_j$. □

We can easily show that the well-structuredness of the subgraph ordering wrt. all GTS

also holds for the induced subgraph ordering.

Proposition 5.25 (WSTS wrt. the induced subgraph ordering)**.** *Let $\mathcal{T}$ be an arbitrary GTS and let $\mathcal{G}$ be any downward-closed class of graphs on which the induced subgraph ordering is a wqo. The systems $\mathcal{T}_{\mathcal{G}(\Lambda)}$, $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ are all $\mathcal{G}$-restricted WSTS wrt. the induced subgraph ordering.*

Proof. We can show that every GTS satisfies the compatibility conditions by using the proof of Proposition 5.17 and additionally showing that the inverse of ν is an induced subgraph morphism.

Assume there is an edge $e \in E_{H'}$, where all incident nodes have a preimage under ν but e has none. Since r', μ^{-1}, ν, r'' is a pushout, this can only be the case if all nodes incident to e have a preimages in G and G', and e has a preimage in G'. Because μ is an induced subgraph morphism, e has a preimage in G. Due to commutativity r' cannot be undefined on this preimage, thus, e has to have a preimage in H, violating the assumption. □

As a result of Propositions 5.24 and 5.25 we know that arbitrary GTS form $\mathcal{D}_{n,k}$-restricted WSTS wrt. the induced subgraph ordering.

In fact, we can even improve Proposition 5.25 by introducing negative application conditions (NACs). Rules with NACs are applicable if a match of the rule exists such that no match of the NACs exist, i.e. the graph to be matched must contain the rules left-hand side, but must not contain the NAC. In general NACs violate the compatibility condition wrt. any order, since rules may become inapplicable to a graph by adding nodes or edges. However, the induced subgraph ordering is fine enough such that graph transformation systems with NACs still satisfy the compatibility condition if the negative application conditions only forbid the existence of edges and not of nodes, as shown in the following example.

Example 5.26. Let the following simple rule be given, where the negative application condition is indicated by the dashed edge, i.e. the rule is applicable if and only if there is a match only for the solid part of the left-hand side and this match cannot be extended to match also the dashed part.

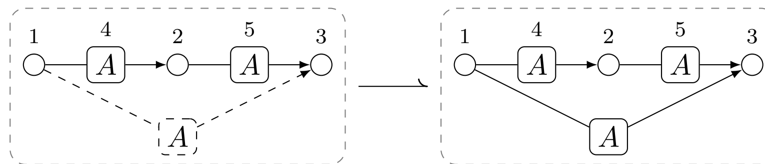


Figure 5.12.: Rule for computing the transitive closure using a negative application condition

Applied to a graph containing only A-labelled edges, this rule computes the transitive closure and will terminate at some point. This GTS satisfies the compatibility condition wrt. the induced subgraph ordering, since for instance a directed path of length two (the left-hand side) does not represent graphs where there is an edge from the first to the last node of the graph.

The principle described in Example 5.26 can be extended to all negative application conditions which forbid the existence of edges but not of nodes. This is the case, because if there is no edge between two nodes of a graph, there is also no edge between these two nodes in any larger graph. Hence, if there is no mapping from the negative application condition into the smaller graph, there can also be none into the larger graph. Graphs violating the negative application condition are simply not represented by the smaller graph. In the following we will prove this, first extending our GTS with negative application conditions.

Definition 5.27 (Rule with NACs)**.** A *rule with negative application conditions* is a pair $\langle r, \mathcal{N} \rangle$ where $r\colon L \rightharpoonup R$ is a rule and $\mathcal{N}$ is a set of negative application conditions. A *negative application condition* (NAC) is a total, injective morphism $n\colon L \rightarrowtail N$.

Let $m\colon L \rightarrowtail G$ be an injective *match* of r into G. We say that m satisfies a NAC n if there is *no* total, injective morphism $n'\colon N \rightarrowtail G$ such that $n' \circ n = m$. A rule with NACs is *applicable* to a graph G if there is a match satisfying all NACs.

A rule with NACs is applied as defined for standard rules (cf. Definition 3.17), but only using matches satisfying all NACs.

Graph transformation systems with negative application conditions are GTS (cf. Definition 3.19) which may consist of standard rules and rules with NACs. Note that the morphism n' in the above definition is usually restricted to being injective since this increases the expressiveness of NACs. We can for instance give a rule that is applicable only if there are no k parallel edges between two nodes, which is not possible if non-injective n' are allowed – the same rule with non-injective n' would forbid the existence of any edge between the two nodes. Thus, we can also restrict ourselves to injective matches, since for non-injective matches the triangle $n' \circ n = m$ does trivially not commute if both n and n' must be injective.

For injective matches we can now state a result than is even stronger than Proposition 5.25.

Proposition 5.28. *Let $\mathcal{T}$ be a GTS with NACs and let $\mathcal{G}$ be any downward-closed class of graphs on which the induced subgraph ordering is a wqo. The system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ is a $\mathcal{G}$-restricted WSTS wrt. the induced subgraph ordering if for all rules $\langle r, \mathcal{N} \rangle \in \mathcal{T}$ and all NACs $n \in \mathcal{N}$, n is bijective on nodes.*

Proof. In the proof of Proposition 5.25 we have already shown that the compatibility condition is satisfied without NACs. It remains to be shown that if a match satisfies all NACs, then it also satisfies all NACs for any larger graph.

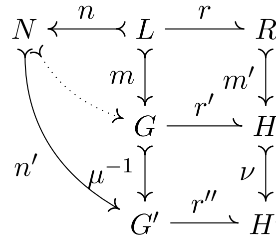


Figure 5.13.: With restricted NACs n we can show that any morphism n', violating the NAC wrt. G', can be extended to a morphism violating the NAC wrt. G

Let $\langle r\colon L \rightharpoonup R, \mathcal{N}\rangle$ be a rule and let $(n\colon L \rightarrowtail N) \in \mathcal{N}$ be a NAC, as shown in Figure 5.13. Furthermore, let $m\colon L \rightarrowtail G$ be an injective match and let $G \trianglelefteq G'$, i.e. there is a $\mu\colon G' \rightarrowtail G$. We use the fact that the inverse μ^{-1} of μ is total and injective to obtain the (injective) match $\mu^{-1} \circ m$ and assume that there is an $n'\colon N \rightarrowtail G'$ violating the NAC, i.e. $n' \circ n = \mu^{-1} \circ m$.

We will show that $\mu(n'(x))$ is defined for all $x \in N$. If x is a node, then x has (exactly) one preimage $x_L \in L$ due to the bijectivity of n. Since all involved morphisms are injective, we obtain $\mu(n'(x)) = m(x_L)$. If x is an edge, then by the previous argument for all incident nodes v, $\mu(n'(v))$ is defined. Since μ is an induced subgraph morphism, $\mu(n'(x))$ must be defined as well. Therefore, $\mu^{-1} \circ n'$ is total and injective, and computes with m in the necessary way, i.e. $\mu \circ n' \circ n = \mu \circ \mu^{-1} \circ m = m$.

Hence, if m satisfies all NACs, $\mu^{-1} \circ m$ satisfies all NACs as well, and the compatibility condition holds. □

5.4. Further Interesting Orders

In addition to the order presented in this chapter – which we will further investigate in Chapter 6 – there are other interesting orders, which are compatible with our approach. Each of these orders affects the notion of upward-closure and may well-quasi-order different sets of graphs.

Between Minors and Induced Subgraphs

Table 5.1 shows three further orders taken from [FHR09; FHR12], each finer than the minor ordering and coarser than the subgraph ordering. Note that the authors use undirected binary graphs (with only edges of arity two), which means that their results need not necessarily hold for hypergraphs. Each column of Table 5.1 represents an operation which can be used to create a smaller graph from a larger one. The operators are deleting a vertex, including all incident edges, deleting an edge, contracting an edge, i.e. deleting an edge an merging its incident nodes, and topologically contracting an edge,

i.e. same as contraction but one of the nodes needs to have a degree of less or equal than two. Clearly, allowing more operators leads to a coarser order.

	vertex deletions	edge deletions	top. contractions	contractions
ind. subgraph	✓	✗	✗	✗
subgraph	✓	✓	✗	✗
ind. top. minor	✓	✗	✓	✗
top. minor	✓	✓	✓	✗
ind. minor	✓	✗	✓	✓
minor	✓	✓	✓	✓

Table 5.1.: Shows different orders and their allowed operations (taken from [FHR12])

The authors show that the topological minor ordering is a wqo on the class of graphs with bounded minimal feedback-vertex-set, a set of vertices such that for each circle of the graph at least one vertex is in the set. They also show that the induced minor ordering is a wqo on graphs with bounded circumference, the length of the longest circle in the graph. For the induced topological minor ordering no interesting class is known, apart from the one inherited for the induced subgraph ordering.

The algorithmic complexity of checking these orders is also investigated in [FHR12] and more thoroughly for the induced minor ordering in [FK+95]. Somewhat expectable, the general problems are NP-complete but become polynomial-time solvable – some even linear-time solvable – when fixing one parameter.

Orders Respecting Directed Edges

So far all order we have seen, including those of the previous section, have ignored the direction of edges when deleting vertices, deleting edges or contracting edges. Especially a restriction of the third operation may yield interesting results in the context of directed graphs. Although some approaches exist, the results for directed graphs are scarce compared to the undirected case.

In [JR+01] the authors use *butterfly minors* in the context of directed tree-width, a generalization of tree-width for directed graphs. In fact, they do not use the term "butterfly minor" (they simply call their relation "minor"), but the name established itself later. A graph G is a butterfly minor of G' if G can be generated by taking a subgraph of G' and then contracting edges which are the only outgoing edge of their source node or the only incoming edge of their target node (or both). Clearly, this contraction preserves directed path in the sense that after the contraction a directed path from x to y exists only if there was a path prior to the contraction. Unfortunately, it is unclear on which interesting classes this order could be a well-quasi-order.

Another interesting work investigating different width measures for directed graphs is [GH+10]. There the authors use a *directed topological minor ordering*, a less restrictive

version of butterfly minors. A graph G is a directed topological minor of a graph G' if G can be generated by taking a subgraph of G' and then contracting edges, where none of the contractions may introduce a new path from some node x of degree three of higher to some node y of degree three of higher. Again, it is unclear on which class this relation could be a wqo, since this does not lie within the scope of [GH+10].

Finally, Chudnovsky and Seymour introduced weak and strong immersion on directed graphs [CS11; CS14]. A weak immersion of a graph G into G' is a mapping of nodes of G to nodes of G' and a compatible mapping (similar to the morphism property) of edges of G to directed paths of G', such that for every two distinct edges of G their corresponding paths in G' are edge disjoint (they may share nodes). A weak immersion is also a strong immersion if for all edges of G only the endpoints of their corresponding paths have preimages in G. As a result of Robertson and Seymour's work the weak immersion is a wqo on the class of undirected graphs (ignoring direction in the definition above), but this is still an open problem for the strong immersion. For the class of all directed graphs already weak immersion is no wqo, but Chudnovsky and Seymour could for instance show that strong immersion is a wqo on the class of tournaments (complete graphs with an assigned direction for each edge) [CS11] and complete directed bipartite graphs [CS14]. Note that the latter publication focuses on Rao-containment, another wqo on all graphs.

Chapter 6
Backward Analysis

In Sections 2.3 and 2.4 we introduced well-structured transition systems as well as their generalization, Q-restricted WSTS. We have then shown in Chapter 5 that graph transformation systems also form Q-restricted WSTS and investigated possible orders. However, before we can apply Theorem 2.21 for the decidability results we still need to prove the existence of an effective pred-basis or Q-pred-basis. As we will see, the existence and the computation of such bases depends on the order, the class Q and the type of matches used. We will work out the computation for our three main orders and present necessary and sufficient conditions which need to be proven for further order to fit our framework. For better readability some proofs of this chapter where moved to Appendix C.

6.1. A General Backward Procedure

We start by restating Algorithm 2.15 as a specialized version for graph transformation systems. We split this specialized version into the general backward search and the computation of a single backward step, i.e. computing the ($\mathcal{Q}$)-pred-basis, where the former uses the latter. Note that in the context of graph transformation systems we will write $\mathcal{Q}$ instead of Q to emphasize that $\mathcal{Q}$ is a class of graphs. Unfortunately, the existence of an effective pred-basis or $\mathcal{Q}$-pred-basis is not guaranteed in general. For instance, for the induced subgraph ordering an effective pred-basis does not exist, simply because the predecessor set is not finitely representable wrt. the order (a $\mathcal{D}_{n,k}$-pred-basis does however exist, i.e. for directed graphs where the length of paths and the multiplicity of edges is bounded). In a nutshell, our algorithm will perform the following steps:

1. Prepare the rules by composing them with order morphisms.
2. Compute a representative (finite) set of pushout complements for all rules and matches to graphs.

3. Minimize the currently computed set of graphs (the working set) by removing all non-minimal graphs.

4. If some pushout complements computed in step 2 were not dropped in step 3, then repeat steps 2 to 4 another time.

We will illustrate how these steps allow us to face several problems occurring when computing a ($\mathcal{Q}$)-pred-basis. The first problem is that graphs do not simply represent themselves, but also all larger graphs, i.e. their upward closure. This means that we have to apply rules backwards to an infinitely large class of graphs! To implement this we use the abstract representation of orders by a class of order morphisms (cf. Definition 5.1), which we have proven to be possible for our three main orders. By composing rules with order morphisms, as shown in Figure 6.1, we can simulate the application of r not just to G, but also to the upward closure of G. We illustrate this in the following example.

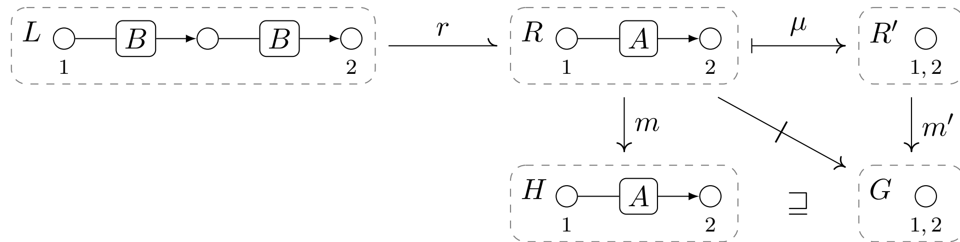


Figure 6.1.: We can simulate the application of rules to the upward closure of G by composing rules with order morphism

Example 6.1. In Figure 6.1 there is no co-match of the right-hand side R of r to the graph G, since the A-labelled edge can not be mapped. However, there is an obvious match m to the graph H which is in the upward closure of G. Instead of generating H from G and then applying r backwards using the match m, we can also apply $\mu \circ r$ backwards using the match m'. Effectively, the composition of r and μ simulates the generation of H for G and the backwards application of r at the same time. In this way we can greatly simplify the computation of predecessors.

By composing all rules with all possible order morphism, we simply need to search for all *total* co-matches – or injective matches if desired – from the new right-hand side R' to G. In this way the application of rules to the upward closure of G is done implicitly. Altogether this gives rise to the notion of a prepared GTS, generated by these compositions.

Definition 6.2 (Prepared GTS)**.** Let $\mathcal{M}_{\preceq}$ be a class of order morphisms and let $\mathcal{T}$ be a graph transformation system, i.e. a finite set of rules. The prepared graph transformation

system $\mathcal{T}'$ consists of all rules $\mu \circ r\colon L \rightharpoonup R'$, where $r\colon L \rightharpoonup R$ is a rule of $\mathcal{T}$ and $\mu\colon R \mapsto R'$ is an order morphism. Furthermore, there is a function *origin*: $\mathcal{T}' \to \mathcal{P}(\mathcal{T})$ satisfying $r \in \mathit{origin}(r')$ if and only if there is a μ with $r' = \mu \circ r$. We call the rules of a prepared GTS *prepared rules*.

Note that due to Definition 5.1 it is guaranteed that for any GTS and order, the prepared GTS has only finitely many rules. We will later need the function *origin* to determine from which original rules a prepared rule could have been generated from. The function will be non-injective if for instance the left-hand sides of different (original) rules are isomorphic.

Alternative to our approach one could also "expand" G, i.e. generate larger graphs and then search for matches to these graphs. However, this would require us to prove the existence and compute a bound on the number of expansions necessary for each order. Our approach requires an equivalent but significantly simpler condition, we need our order morphisms to be *preserved by pushouts along total morphisms*, i.e. Definition 6.3 must hold.

Definition 6.3 (Preservation by pushouts)**.** We say that a class of order morphisms $\mathcal{M}_{\preceq}$ is *preserved by pushouts along total morphisms* if the following holds: if $\mu\colon G_0 \mapsto G_1$ is an order morphism in $\mathcal{M}_{\preceq}$ and $g\colon G_0 \to G_2$ is total, then the morphism μ' in the pushout diagram in Figure 6.2 is an order morphism in $\mathcal{M}_{\preceq}$.

$$\begin{array}{ccc} G_0 & \overset{\mu}{\longmapsto} & G_1 \\ \downarrow g & & \downarrow g' \\ G_2 & \overset{\mu'}{\longmapsto} & G_3 \end{array}$$

Figure 6.2.: A class $\mathcal{M}_{\preceq}$ is preserved by pushouts along total morphisms, if $\mu \in \mathcal{M}_{\preceq}$ implies $\mu' \in \mathcal{M}_{\preceq}$

Example 6.4. Figure 6.3 illustrated why minor morphisms are preserved by total pushouts (we will prove this in Lemma 6.20). The minor morphism μ contracts the B-labelled edge with the number 5 and is the identity on all other elements. Since g is total, the contacted edge is still existent in G_2 and can be contracted by μ' to simulate the contraction of μ. Even the fact that g non-injectively maps the nodes 1 and 2 does not prevent this, since by definition the nodes 1, 2 and 3 have to be merged in the pushout. The same argument holds for node and edge deletions as well. Thus, μ' is a minor morphism if μ is a minor morphism.

Although Definition 6.3 is independent of the type of matches used, for a reasonable application we can only use conflict-free or more restrictive matches (including injective

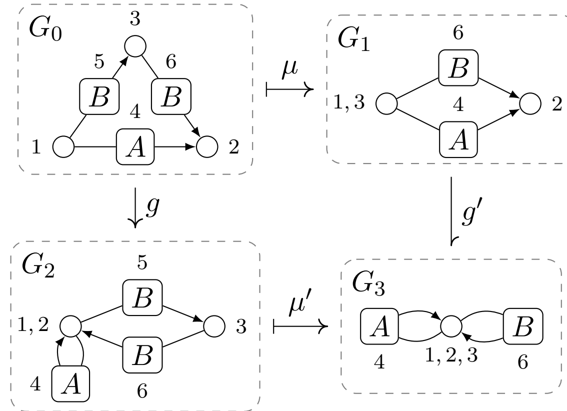


Figure 6.3.: Example of minor morphisms being preserved by pushouts along total morphisms

ones), since a conflict-free match ensures that the co-match is total. The use of more general matches would still require us to search for partial co-matches. Since we will need this property in later proofs, we prove it in the following lemma.

Lemma 6.5. *Let $r\colon L \rightharpoonup R$ be a partial and $m\colon L \rightarrow G$ a total morphism. Furthermore, let $\langle H, r', m'\rangle$ with $r'\colon G \rightharpoonup H$ and $m'\colon R \rightharpoonup H$ be the pushout of r and m. If m is injective, then m' is total and injective as well. If m is conflict-free wrt. r, then m' is total.*

Proof. See Appendix C.1. □

Another problem occurring in the computation of a ($\mathcal{Q}$)-pred-basis is finding a representative set of pushout complements. At first sight we simply need to compute pushout complements when applying rules backward, a problem for which we have given several constructions in Section 3.5, namely Propositions 3.30 and 3.33. However, these constructions can generate infinitely many pushout complements. Thus, we need to restrict ourselves only to minimal pushout complements, wrt. the order used, and be able to exhaustively enumerate these to achieve effectiveness. Note that the finiteness of the set of minimal pushout complements is only guaranteed for $\mathcal{Q}$-pred-bases if the order is a well-quasi-order on $\mathcal{Q}$. For pred-bases (or other classes $\mathcal{Q}$) we also have to prove finiteness. We will revisit this problem at a later point and will now start to successively define our main algorithm.

Main Search and Auxiliary Procedures

Before further investigating the correctness of our backward search, we state our (backward) Algorithm 6.6 for graph transformation systems. For now we assume the existence

of a $\mathcal{Q}$-pred-basis $preds_{\mathcal{Q}}(\mathcal{T}, G)$ (cf. Definition 2.18), where $\mathcal{T}$ is a prepared graph transformation system and G a graph. The $\mathcal{Q}$-pred-basis contains all graphs which can be rewritten to some graph larger that G using a rule of $\mathcal{T}$. Later we will present generic algorithms which can be used directly by some orders and build upon to define $\mathcal{Q}$-pred-basis for other orders.

In addition to the $\mathcal{Q}$-pred-basis Algorithm 6.6 uses two other simple algorithms for preparing the rules ($prepare_{\preceq}$), as introduced previously, and minimizing the working set ($minimize_{\preceq}$). Note that although the latter is a slight optimization, we will introduce more substantial optimizations in Section 6.5.

Algorithm 6.6 (General backward search for GTS)**.**
Input: A $\mathcal{Q}$-restricted well-structured graph transformation system $\mathcal{T}$, a finite set of (final) graphs $\mathcal{F}$, an order $\preceq$ represented by the class $\mathcal{M}_{\preceq}$ and a $\mathcal{Q}$-pred-basis $preds_{\mathcal{Q}}$ for the order $\preceq$.
Output: A finite basis $\mathcal{W}_{old}$ of the class of all graphs from which a graph of $\mathcal{F}$ is coverable.

1: $\mathcal{W}_{old} \leftarrow \emptyset$
2: $\mathcal{W}_{new} \leftarrow minimize_{\preceq}(\mathcal{F})$
3: $\mathcal{T}' \leftarrow prepare_{\preceq}(\mathcal{T})$
4: **while** $\uparrow\mathcal{W}_{old} \neq \uparrow\mathcal{W}_{new}$ **do**
5: $\quad \mathcal{W}_{old} \leftarrow \mathcal{W}_{new}$
6: $\quad$ **for all** $G \in \mathcal{W}_{old}$ **do**
7: $\quad\quad \mathcal{W}_{new} \leftarrow \mathcal{W}_{new} \cup preds_{\mathcal{Q}}(\mathcal{T}', G)$
8: $\quad$ **end for all**
9: $\quad \mathcal{W}_{new} \leftarrow minimize_{\preceq}(\mathcal{W}_{new})$
10: **end while**
11: **return** $\mathcal{W}_{old}$

The algorithm will terminate as soon as the condition in line 4 is met. To check whether $\uparrow\mathcal{W}_{old}$ and $\uparrow\mathcal{W}_{new}$ are equal we can exploit the minimization in line 9. Since the sets are both minimal, we can simply search for a bijective mapping $f\colon \mathcal{W}_{old} \to \mathcal{W}_{new}$ such that G is isomorphic to $f(G)$ for every $G \in \mathcal{W}_{old}$. Since the class $\uparrow\mathcal{W}_{old}$ is obviously a subclass of $\uparrow\mathcal{W}_{new}$, the condition is guaranteed to be satisfied at some point if the order is a well-quasi-order on $\mathcal{Q}$ and we use a $\mathcal{Q}$-pred-basis. However, since this algorithm can be used for all cases of Theorem 2.21, termination effectively depends on which representatives $preds_{\mathcal{Q}}$ computes. Before we proceed to the computation of the different variants of $preds_{\mathcal{Q}}$, we state the two auxiliary procedures $minimize_{\preceq}$ and $prepare_{\preceq}$.

The minimization procedure is defined in Algorithm 6.7 and simply removes all graphs for which the set contains smaller or isomorphic graphs. Note that minimization can remove elements, but will never affect the upward closure, i.e. $\uparrow\mathcal{G} = \uparrow minimize_{\preceq}(\mathcal{G})$ for all finite sets $\mathcal{G}$. Thus, minimization only affects the performance and not the correctness of Algorithm 6.6.

Algorithm 6.7 (Minimization)**.**
Input: A set of graphs $\mathcal{G}$ and a decidable order $\preceq$.
Output: The set $\mathit{minimize}_{\preceq}(\mathcal{G})$, a minimized version of $\mathcal{G}$.

1: **while** $\exists G_1, G_2 \in \mathcal{G} : (G_1 \neq G_2 \wedge G_1 \preceq G_2)$ **do**
2: $\quad\mathcal{G} \leftarrow \mathcal{G} \setminus \{G_2\}$
3: **end while**
4: **return** $\mathcal{G}$

The rule preparation is performed by Algorithm 6.8 by composing rules and order morphisms, as already motivated on page 98.

Algorithm 6.8 (Rule preparation)**.**
Input: A graph transformation system $\mathcal{T}$ and a class of order morphisms $\mathcal{M}_{\preceq}$.
Output: The prepared GTS $\mathit{prepare}_{\preceq}(\mathcal{T})$.

1: $\mathcal{T}' \leftarrow \emptyset$
2: **for all** $(r\colon L \rightharpoonup R) \in \mathcal{T}$ **do**
3: $\quad$**for all** $(\mu\colon R \rightarrowtail R') \in \mathcal{M}_{\preceq}$ **do**
4: $\quad\quad\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\mu \circ r\}$
5: $\quad$**end for all**
6: **end for all**
7: **return** $\mathcal{T}'$

Note that in line 3 of the algorithm we only need to choose one R' out of its isomorphism class and compose all order morphisms $\mu\colon R \rightarrowtail R'$ with each rule. Although each isomorphism class may be infinite, the number of isomorphisms classes is finite due to Definition 5.1.

The Conflict-Free $\mathcal{Q}$-Pred-Basis and Its Correctness

Since Algorithm 6.6 is just a restatement of Algorithm 2.15, the correctness only depends on the $\mathcal{Q}$-pred-basis used. We therefore present two very similar generic $\mathcal{Q}$-pred-bases – $\mathit{preds}^i_{\mathcal{Q}}(\mathcal{T}, G)$ for injective and $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, G)$ for conflict-free matches – which can be used in most cases. Both procedures first search for co-matches of all (prepared) rules r to G and then use the construction of Proposition 3.33 to compute all (partial) pushout complements for each co-match. Since the set of pushout complements need not be finite, we need to compute the set of minimal pushout complements of a rule r and a match m wrt. the order used, which is defined as follows.

Definition 6.9 (Set of minimal pushout complements)**.** Let $\preceq$ be an order, let $\mathcal{Q}$ be a downward-closed class of graphs, let $r\colon L \rightharpoonup R$ be a (prepared rule) and let $m\colon R \to G$ be a (total) co-match to some graph G. The *set of minimal pushout complements* is the finite set $\mathit{minpoc}^{\preceq}_{\mathcal{Q}}(r, m) \subseteq \mathcal{Q}$ where:

- every $G' \in minpoc^{\preceq}_{\mathcal{Q}}(r,m)$ is a pushout complement object of r, m, where the match $m' \colon L \rightarrow G'$ is conflict-free wrt. some element of $origin(r)$,
- for every pushout complement object $G' \in \mathcal{Q}$ of r, m, satisfying the previous condition, there is an $G'' \in minpoc^{\preceq}_{\mathcal{Q}}(r,m)$ with $G'' \preceq G'$, and
- if there are $G', G'' \in minpoc^{\preceq}_{\mathcal{Q}}(r,m)$ with $G' \preceq G''$, then $G' = G''$.

Note that the last condition only exists for efficiency reasons and we could drop it without affecting correctness. The finiteness of the set is guaranteed if $\preceq$ is a well-quasi-order on $\mathcal{Q}$. Unfortunately this means that a procedure computing this set can only be generalized to a limited extent, namely we can use Propositions 3.30 and 3.33 for the actual computation of pushout complements. We will later prove the computability for our three main orders, but for now we just assume the existence of a procedure for this set, which we will also denote by $minpoc^{\preceq}_{\mathcal{Q}}(r,m)$.

Algorithm 6.10 (Conflict-free $\mathcal{Q}$-pred-basis)**.**
Input: A prepared rule set $\mathcal{T}$, a graph $S \in \mathcal{Q}$ and a procedure for computing minimal pushout complements $minpoc^{\preceq}_{\mathcal{Q}}$.
Output: The set $preds^{c}_{\mathcal{Q}}(\mathcal{T}, S)$, an effective $\mathcal{Q}$-pred-basis for conflict-free matches.

1: $\mathcal{G} \leftarrow \emptyset$
2: **for all** $(r \colon L \rightharpoonup R) \in \mathcal{T}$ **do**
3: **for all** total co-matches $m \colon R \rightarrow S$ **do**
4: $\mathcal{G}_{poc} \leftarrow minpoc^{\preceq}_{\mathcal{Q}}(r,m)$
5: **for all** $\langle G', m' \colon L \rightarrow G', r' \colon G' \rightharpoonup S \rangle \in \mathcal{G}_{poc}$ **do**
6: $\mathcal{G} \leftarrow \mathcal{G} \cup \{G'\}$
7: **end for all**
8: **end for all**
9: **end for all**
10: **return** $\mathcal{G}$

The algorithm only considers total co-matches, since a partial co-match automatically implies a non-conflict-free match and vice versa. In fact, by requiring that order morphisms are preserved by pushouts along total morphisms, we can show that every minimal pushout complement is a valid predecessor. However, we also need to prove that every predecessor is either computed this way or represented by another graph computed this way. Thus, for $preds^{c}_{\mathcal{Q}}$ to be a $\mathcal{Q}$-pred-basis, a class of order morphisms also needs to be *pushout closed*, a property we define as follows.

Definition 6.11 (Pushout closure)**.** Let $m \colon L \rightarrow G$ be a conflict-free match wrt. $r \colon L \rightharpoonup R$. A class of order morphisms is called *pushout closed* if the following holds: if the diagram on the left of Figure 6.4 is a pushout and $\mu \colon H \rightarrowtail S$ an order morphism, then

there exist graphs R' and G' and order morphisms $\mu_R\colon R \rightarrowtail R'$, $\mu_G\colon G \rightarrowtail G'$, such that:

1. the diagram on the right of Figure 6.4 commutes and the outer square is a pushout, and
2. the morphisms $\mu_G \circ m\colon L \to G'$ and $n\colon R' \to S$ are total and $\mu_G \circ m$ is conflict-free wrt. r.

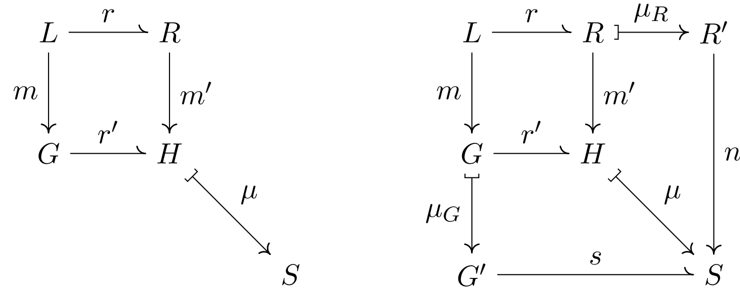


Figure 6.4.: Two diagrams illustrating Definition 6.11

Example 6.12. In a sense, pushout closure can be seen as a compositionality property: if H can be obtained by composing G and R, then every graph smaller than H can be obtained by composing graphs smaller then G and R. For the minor ordering this is illustrated in Figure 6.5 where μ contracts both edges of H. Deletions and contractions performed by μ must be simulated by μ_R or μ_G (or both). Thus, the B-labelled edge is contracted by μ_R and the C-labelled edge is contracted by μ_G. In this way, S is in fact the pushout of the diagram.

Pushout closure enables us to prove that Algorithm 6.10 is in fact a $\mathcal{Q}$-pred-basis. We do this by first proving that $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ is a subset of the predecessors of G (Lemma 6.13) and then proving that all graphs of the upward closure of the predecessors of G are represented by $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ (Lemma 6.14).

Lemma 6.13. *The set $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ computed by Algorithm 6.10 is a finite subset of $\uparrow\mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$ for any prepared rule set $\mathcal{T}$.*

Proof. By assumption the sets of minimal pushout complements $\mathit{minpoc}^{\preceq}_{\mathcal{Q}}(r, m)$ is finite for all rules and matches. Since the prepared rule set and the number of matches are both finite as well, $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ is finite.

Now let $G \in \mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ be a graph generated by Algorithm 6.10. Then there is a rule $r\colon L \rightharpoonup R$, an order morphism $\mu\colon R \rightarrowtail R'$ and a conflict-free match $m\colon L \to G$, such that the left diagram in Figure 6.6 is a pushout.

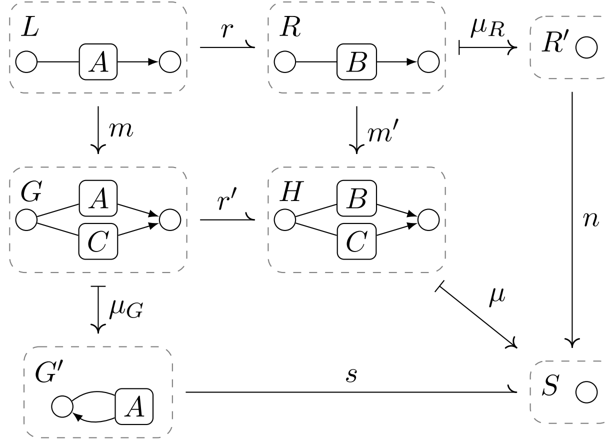


Figure 6.5.: Minor morphisms are pushout closed, i.e. μ_R and μ_G always exist

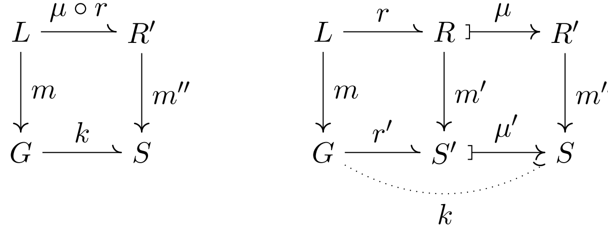


Figure 6.6.: Diagrams illustrating the proof of Lemma 6.13

Let $m' \colon R \rightarrow S'$, $r' \colon G \rightharpoonup S'$ be the pushout of m, r. Because the outer diagram on the right of Figure 6.6 commutes, there is a unique morphism $\mu' \colon S' \rightarrowtail S$. The left and the outer squares are both pushouts and therefore the right square is a pushout as well (cf. Lemma 3.7). Since m is total and conflict-free, m' is also total. By assumption $\mathcal{M}_{\preceq}$ is preserved by pushouts along total morphisms, thus μ' is in fact an order morphism. This means we can use r to rewrite G to some graph S' larger than S, hence $G \in \mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$. $\square$

Lemma 6.14. *It holds that* $\uparrow \mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S) \supseteq \uparrow \mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$ *for any prepared rule set* $\mathcal{T}$.

Proof. Let X be an element of $\uparrow \mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$. Then there is a minimal representative $G \in \mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$ with $G \preceq X$ and a rule $r \colon L \rightharpoonup R$ rewriting G with a conflict-free match m to some element H of $\uparrow\{S\}$. Note that this also implies $G \in \mathcal{Q}$. According to Definition 6.11 the left diagram in Figure 6.7 can be extended to the right diagram.

Since the outer square is a pushout, G' is a pushout complement object. Due to the downward closure of $\mathcal{Q}$ we know that $G' \in \mathcal{Q}$ and thus, a graph $G'' \preceq G'$ will be

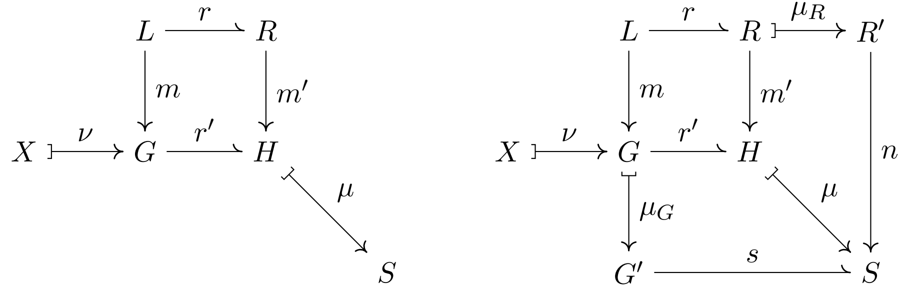


Figure 6.7.: Diagrams illustrating the proof of Lemma 6.14

obtained by Algorithm 6.10 using the rule $\mu_R \circ r$. More precisely $G'' \in \mathit{minpoc}^{\preceq}_{\mathcal{Q}}(\mu_R \circ r, n)$. Summarized, this means that $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ contains a graph G'' for every graph X such that $G'' \preceq G' \preceq G \preceq X$, i.e. every X is represented by an element of $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$. □

The previous two lemmas prove the existence of a $\mathcal{Q}$-pred-basis for all orders which are representable by morphisms, preserved by pushouts along total morphisms, pushout closed under conflict-free matches and for which the set of minimal pushout complements is finite and computable. We summarize this by the following proposition.

Proposition 6.15. *The set $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ computed by Algorithm 6.10 is an effective $\mathcal{Q}$-pred-basis for the transition system $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ and all orders satisfying Definitions 5.1, 6.3 and 6.11, and for which the set of minimal pushout complements is finite and computable.*

Proof. We have proven the correctness of the equality $\uparrow\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S) = \uparrow\mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$ by Lemmas 6.13 and 6.14. Moreover, since the prepared rule set, the number of matches and the number of pushout complements are all finite, $\mathit{preds}^c_{\mathcal{Q}}(\mathcal{T}, S)$ is finite as well. The effectiveness follows from the computability of matches and pushout complements. □

The Injective $\mathcal{Q}$-Pred-Basis and Its Correctness

With some modifications we can change Algorithm 6.10 to be an *injective Q-pred-basis* $\mathit{preds}^i_{\mathcal{Q}}(\mathcal{T}, S)$, in which case our backward algorithm will analyse the transitions system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ induced by injective matches. Unfortunately, we cannot simply compute $\mathit{minpoc}^{\preceq}_{\mathcal{Q}}$ and drop all pushout complements with non-injective matches, since these might represent pushout complements with injective matches which are not otherwise represented. This is especially the case for orders where order morphisms can be non-injective, as we illustrate for the minor ordering in the following example.

Example 6.16. Consider a rule that simply replaces an A-edge with a B-edge, as shown in Figure 6.8, and let S be a minor of H. As we will prove later, minor morphisms satisfy Definition 6.11, i.e. the graphs R', G' exist, the outer square is a pushout and the diagram

commutes. This means that G' is a pushout complement where the match $\mu_G \circ m$ is non-injective. However, it represents G, a graphs that can be rewritten to H with $S \sqsubseteq H$ using an injective match. Thus, G should be computed by our procedure or at least represented by another computed graph, but G' should not. Note that G is not a pushout complement of $\mu_R \circ r, n$ at all, since it is rewritten to a graph strictly larger than S.

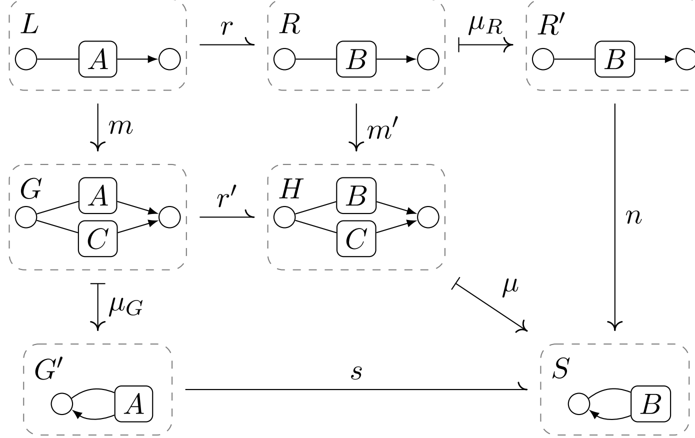


Figure 6.8.: Pushout complements with non-injective matches might represent graphs that can be rewritten to some H larger than S using an injective match

We therefore need a procedure that computes all such graphs G for every rule and pushout complement. Effectively this procedure needs to compute all possibilities to appropriately split the match into an injective match and an order morphism. In the following we will call the set of graphs obtained by such splits the *represented injective predecessors*, formally defined as follows.

Definition 6.17 (Represented injective predecessors)**.** Let $\preceq$ an order, $\mathcal{Q}$ a downward-closed class of graphs, $r\colon L \rightharpoonup R$ a prepared rule, $m\colon L \rightarrow G$ a match to some graph G and S the pushout of r, m. A *set of represented injective predecessors* is a set $\mathit{reps}^{\preceq}_{\mathcal{Q}}(r,m) \subseteq \mathcal{Q}$ such that:

- if $G' \in \mathit{reps}^{\preceq}_{\mathcal{Q}}(r,m)$ then there are $r_1\colon L \rightharpoonup R'$, $r_2\colon R' \mapsto R$, $m_1\colon L \rightarrowtail G'$, $m_2\colon G' \mapsto G$ with $r_1 \in \mathit{origin}(r)$, as shown in Figure 6.9 such that $m = m_2 \circ m_1$ and $r = r_2 \circ r_1$, and for the pushout H' of m_1, r_1 it holds that $S \preceq H'$, and
- for all graphs $G' \in \mathcal{Q}$ with $G' \notin \mathit{reps}^{\preceq}_{\mathcal{Q}}(r,m)$ satisfying the previous condition, there is a $G'' \in \mathit{reps}^{\preceq}_{\mathcal{Q}}(r,m)$ with $G'' \preceq G'$.

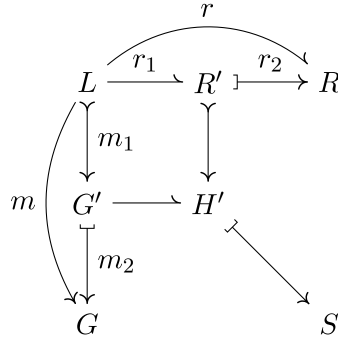


Figure 6.9.: Illustrates the morphisms of Definition 6.17

To be effective, we need a set of represented injective predecessors that is finite. In general we prefer a minimal set, i.e. if $G' \preceq G''$ holds for two elements, then $G' = G''$. Of course we will have to prove the finiteness and computability wrt. our main orders later, but at this point we assume that there is a procedure computing a finite set of represented injective predecessors and denote this procedure by $\mathit{reps}^{\preceq}_{\mathcal{Q}}(r, m)$ as well. With this assumption we can then state an algorithm for the injective $\mathcal{Q}$-pred-basis.

Algorithm 6.18 (Injective $\mathcal{Q}$-pred-basis)**.**
Input: A (prepared) rule set $\mathcal{T}$ and a graph $S \in \mathcal{Q}$.
Output: The set $\mathit{preds}^i_{\mathcal{Q}}(\mathcal{T}, S)$, an effective $\mathcal{Q}$-pred-basis for injective matches.

1: $\mathcal{G} \leftarrow \emptyset$
2: **for all** $(r\colon L \rightharpoonup R) \in \mathcal{T}$ **do**
3:     **for all** total co-matches $m\colon R \to S$ **do**
4:         $\mathcal{G}_{poc} \leftarrow \mathit{minpoc}^{\preceq}_{\mathcal{Q}}(r, m)$
5:         **for all** $\langle G', m'\colon L \to G', r'\colon G' \rightharpoonup S\rangle \in \mathcal{G}_{poc}$ **do**
6:             **if** m' is injective **then**
7:                 $\mathcal{G} \leftarrow \mathcal{G} \cup \{G'\}$
8:             **else**
9:                 $\mathcal{G} \leftarrow \mathcal{G} \cup \mathit{reps}^{\preceq}_{\mathcal{Q}}(r, m')$
10:            **end if**
11:        **end for all**
12:    **end for all**
13: **end for all**
14: **return** $\mathcal{G}$

We can state the correctness of this algorithm in a proposition similar to Proposition 6.15, and even reuse most of the proofs.

Proposition 6.19. *The set $preds^i_{\mathcal{Q}}(\mathcal{T}, S)$ computed by Algorithm 6.18 is an effective $\mathcal{Q}$-pred-basis for the transition system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ and all orders satisfying Definitions 5.1, 6.3 and 6.11, and for which the set of minimal pushout complements as well as the set of represented injective predecessors are finite and computable.*

Proof. Since $reps^{\preceq}_{\mathcal{Q}}$ is by assumption finite and computable, $preds^i_{\mathcal{Q}}$ is finite and computable as well, i.e. we inherit the effectiveness of $preds^c_{\mathcal{Q}}$. Thus, we only need to prove the equality $\uparrow preds^i_{\mathcal{Q}}(\mathcal{T}, S) = \uparrow Pred_{\mathcal{Q}}(\uparrow\{S\})$ by proving Lemmas 6.13 and 6.14 for $preds^i_{\mathcal{Q}}$.

Let $G \in preds^i_{\mathcal{Q}}(\mathcal{T}, S)$, then G is either a pushout complement itself (if the match is injective) or a represented injective predecessor. In the first case we can apply the argument of Lemma 6.13 to prove that G can be rewritten to a graph larger than S. In the second case, by definition G can be rewritten to an H' with $S \preceq H'$ using an injective match and a rule of the original GTS.

We now adapt the proof of Lemma 6.14 to respect the use of $reps^{\preceq}_{\mathcal{Q}}$. So let X be an element of $\uparrow Pred_{\mathcal{Q}}(\uparrow\{S\})$. Then there is a minimal representative $G \in Pred_{\mathcal{Q}}(\uparrow\{S\})$ with $G \preceq X$ and a rule $r\colon L \rightharpoonup R$ rewriting G with an injective match m to some element H of $\uparrow\{S\}$. Note that this also implies $G \in \mathcal{Q}$. According to Definition 6.11 the left diagram in Figure 6.10 can be extended to the right diagram.

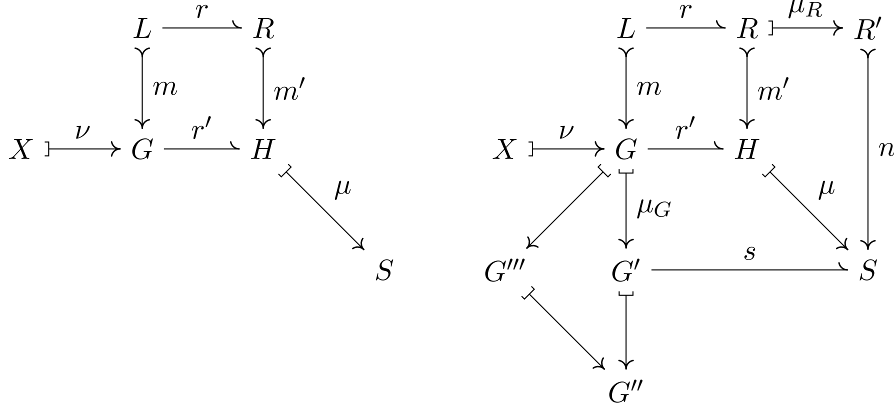


Figure 6.10.: Diagrams illustrating the proof of Proposition 6.19

Since the outer square is a pushout, G' is a pushout complement object. Due to the downward closure of $\mathcal{Q}$ we know that $G' \in \mathcal{Q}$ and thus, a graph $G'' \preceq G'$ will be obtained using the rule $\mu_R \circ r$. If the match $k\colon L \to G''$ is injective, G'' will be kept, i.e. $G'' \in preds^i_{\mathcal{Q}}(\mathcal{T}, S)$, and $G'' \preceq G' \preceq G \preceq X$ holds. If k is not injective, then by definition there is an $G''' \in reps^{\preceq}_{\mathcal{Q}}(\mu_R \circ r, k)$ with an injective match $k'\colon L \rightarrowtail G'''$ and $G'' \preceq G''' \preceq G \preceq X$. Note that r, μ_R is one of the possible decompositions of $\mu_R \circ r$ considered by $reps^{\preceq}_{\mathcal{Q}}$.

Summarized, this means that every $X \in \uparrow Pred_{\mathcal{Q}}(\uparrow\{S\})$ is represented by elements G'' or G''' of $preds^i_{\mathcal{Q}}(\mathcal{T}, S)$. $\square$

Summarizing the Requirements

To conclude this section we sum up the requirement and conditions we need to prove for any order we want to use Algorithm 6.6 with. For this let $\mathcal{Q}$ be a downward-closed class of graphs, let $\preceq$ be a quasi order on $\mathcal{G}(\Lambda)$ and let $\mathcal{T}$ be a graph transformation system. The following conditions have to be met for the algorithm to be correct and computable for conflict-free matches.

- The induced transition system $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ must be a $\mathcal{Q}$-restricted WSTS wrt. $\preceq$, i.e. $\preceq$ must be a well-quasi-order on $\mathcal{Q}$ and $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ must satisfy the compatibility condition.
- The order $\preceq$ must be decidable and satisfy Definitions 5.1 and 6.3, i.e. it must be representable by a class of morphisms $\mathcal{M}_{\preceq}$ which is preserved by pushouts along total morphisms.
- An effective $\mathcal{Q}$-pred-basis $preds^c_{\mathcal{Q}}$ must exist, which is guaranteed if the following conditions are satisfied.
  - The morphisms $\mathcal{M}_{\preceq}$ must be pushout closed (Definition 6.11).
  - The set of minimal pushout complements $minpoc^{\preceq}_{\mathcal{Q}}$ (Definition 6.9) must be computable (its finiteness is guaranteed by $\preceq$ being a well-quasi-order on $\mathcal{Q}$).

We can now use Algorithm 6.6 to partially decide general and restricted coverability according to cases (ii) and (iii) of Theorem 2.21. Furthermore, we can use the algorithm also for case (iv), i.e. deciding general coverability if the algorithm terminates, if a pred-basis $preds^c_{\mathcal{G}(\Lambda)}$ exists. To prove the existence of $preds^c_{\mathcal{G}(\Lambda)}$ we need to show finiteness of $minpoc^{\preceq}_{\mathcal{G}(\Lambda)}$ (which is no longer guaranteed) in addition to computability.

Very similar conditions can be stated for the injective case, where we additionally need computability of $reps^{\preceq}_{\mathcal{Q}}$.

- The induced transition system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ must be a $\mathcal{Q}$-restricted WSTS wrt. $\preceq$, i.e. $\preceq$ must be a well-quasi-order on $\mathcal{Q}$ and $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ must satisfy the compatibility condition.
- The order $\preceq$ must be decidable and satisfy Definitions 5.1 and 6.3.
- An effective $\mathcal{Q}$-pred-basis $preds^i_{\mathcal{Q}}$ must exist, which is guaranteed if the following conditions are satisfied.
  - The morphisms $\mathcal{M}_{\preceq}$ must be pushout closed (Definition 6.11).
  - The set of minimal pushout complements $minpoc^{\preceq}_{\mathcal{Q}}$ (Definition 6.9) must be computable (finiteness is guaranteed).

– The set of represented injective predecessors $reps^{\preceq}_{\mathcal{Q}}$ (Definition 6.17) must be computable (finiteness is guaranteed).

As in the conflict-free case this ensures decidability according to cases (ii) and (iii) of Theorem 2.21, and also for case (iv) if a pred-basis $preds^{i}_{\mathcal{G}(\Lambda)}$ exists. In the latter case we need to prove finiteness and computability for both $minpoc^{\preceq}_{\mathcal{G}(\Lambda)}$ and $reps^{\preceq}_{\mathcal{G}(\Lambda)}$.

6.2. Minor Ordering

In this section we will prove that the minor ordering satisfies all necessary conditions for applying Algorithm 6.6 both with conflict-free and injective matches. In fact, since the minor ordering is a well-quasi-order on all graphs (see Proposition 5.6), we can show that the coverability problem is decidable for all GTS satisfying the compatibility condition wrt. the minor ordering. In [KS12b; KS12a] we have also shown under what conditions the algorithms can still be applied in the presence of negative application conditions.

In Lemma 5.5 we have already proven that the minor ordering can be represented by a class of minor morphisms. Thus we only have to show that this class is preserved by pushouts along total morphisms (Lemma 6.20) and pushout closed (Lemma 6.21). These two properties where first proven by König and Joshi in [JK08], which ultimately gave rise to the framework presented in this thesis.

Lemma 6.20. *Minor morphisms are preserved by pushouts along total morphisms (cf. Definition 6.3).*

Proof. See Appendix C.2. □

Lemma 6.21. *Minor morphisms are pushout closed (cf. Definition 6.11).*

Proof. See Appendix C.2. □

Minimal pushout complements wrt. the minor ordering can be computed by the constructions given in Propositions 3.30 and 3.33. To remain finite we do not add edges in step 3 of Proposition 3.30. Since removing an edge is an allowed operation for minors, all pushout complements left out in this way are still represented by other pushout complement where no edges where added. This allows us to obtain the following general result.

Proposition 6.22. *The coverability problem wrt. the minor ordering is decidable for every transition system $\mathcal{T}^{c}_{\mathcal{G}(\Lambda)}$ which satisfies the compatibility condition wrt. the minor ordering (see of instance Proposition 5.7).*

Proof. Since the minor ordering is a well-quasi-order on all graphs (Proposition 5.6), every GTS satisfying the compatibility condition is a WSTS and therefore also a $\mathcal{Q}$-restricted WSTS, where $\mathcal{Q}$ is the class of all graphs. Furthermore, we have proven that the minor ordering is representable by morphisms (Lemma 5.5) which are preserved by pushouts along total morphisms (Lemma 6.20) and pushout closed (Lemma 6.21). Since the set of minimal pushout complements is computable, we satisfy all conditions necessary to apply Algorithm 6.6 with conflict-free matches and obtain decidability due to case (i) of Theorem 2.21. □

It is also possible to use injective matches together with the minor ordering. We can show that a procedure computing $reps_{\mathcal{Q}}^{\sqsubseteq}$ exists, although it is quite complex. Given a match $m'\colon L \rightarrow G'$, the basic idea is to use "decontractions" to split nodes which are non-injectively matched by m'. This means that we are searching for graphs G'' from which we can obtain G' by a single edge contraction and to which we can extend m' (to obtain m'') as shown in Figure 6.11. We illustrate this idea in the following example.

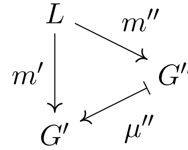


Figure 6.11.: Shows the basic idea of splitting matches used by Algorithm 6.24

Example 6.23. Let the rule and match be the same as in Example 6.16, i.e. we have a non-injective match $m'\colon L \rightarrow G'$. We can not simply drop m', since there are G'' larger than G' that can be rewritten with injective matches. Some of these G'' are shown in Figure 6.12 and each one has an additional edge which can be contracted to obtain G'. In the presence of edges of arity more than two, such a decontraction can also add new nodes. However, the D-labelled edge in this example could also be incident to one of the original nodes multiple times. To cover all represented injective predecessors every decontraction needs to be computed where m'' is less non-injective than m'. In this example decontracting once is enough.

To ensure that computing decontractions does terminate at some point, we need to require that we only compute those G'' where m'' has less pairs of non-injectively mapped nodes than m'. The other G'' need not be computed, since they are already represented by those satisfying this restriction. With this termination criteria we can now define Algorithm 6.24 to compute this set.

Algorithm 6.24 (Represented injective predecessors for the minor ordering)**.**
Input: A prepared rule $r\colon L \rightharpoonup R$ and a conflict-free match $m\colon L \rightarrow G$.
Output: The set $reps_{\mathcal{Q}}^{\sqsubseteq}(r, m)$, a finite set of represented injective predecessors.

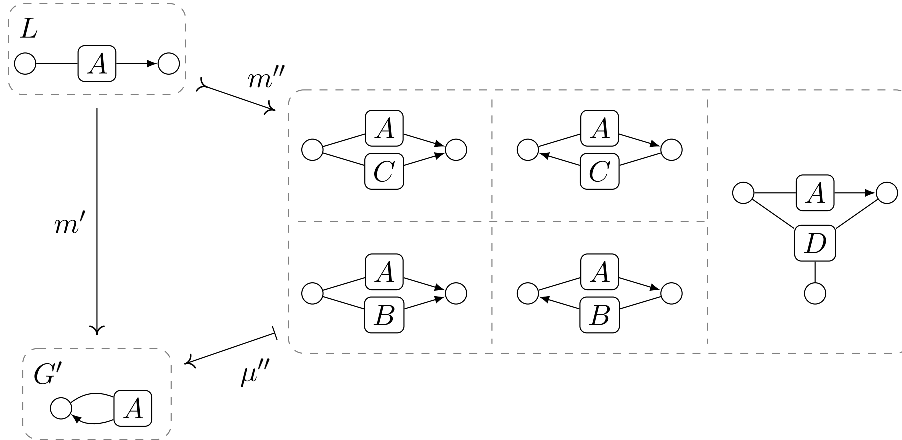


Figure 6.12.: Shows some possible decontractions, each resulting in an injective m''

1: **if** m is injective **then**
2: **return** $\{G\}$
3: **end if**
4: $\mathcal{G} \leftarrow \emptyset$
5: $\mathcal{M} \leftarrow \{m\}$
6: **while** $\mathcal{M} \neq \emptyset$ **do**
7: let $m' \colon L \rightarrow G'$ be any element of $\mathcal{M}$
8: $\mathcal{M} \leftarrow \mathcal{M} \setminus \{m'\}$
9: $\mathcal{P} \leftarrow$ all pairs $\langle G'', \mu'' \colon G'' \mapsto G' \rangle$ where μ'' is undefined on exactly one edge e and the identity on all other edges and all nodes except the nodes incident to e
10: **for all** $\langle G'', \mu'' \rangle \in \mathcal{P}$ with $G'' \in \mathcal{Q}$ **do**
11: **for all** total $m'' \colon L \rightarrow G''$ with $m' = \mu'' \circ m''$ **do**
12: **if** there are $v_1, v_2 \in V_L$ with $m'(v_1) = m'(v_2)$ and $m''(v_1) \neq m''(v_2)$ **then**
13: **if** m'' is injective **then**
14: $\mathcal{G} \leftarrow \mathcal{G} \cup \{G''\}$
15: **else**
16: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m''\}$
17: **end if**
18: **end if**
19: **end for all**
20: **end for all**
21: **end while**
22: **return** $\mathcal{G}$

The algorithm above uses $\mathcal{G}$ to store the graphs for which injective matches exist and $\mathcal{M}$ to store all matches it still has to process. In each iteration of the loop it chooses

one match for which it computes all G'' from which we obtain G' by a single contraction and for which a match m'' commuting with m' exists. However, the condition in line 12 ensures that there are less pairs of nodes matched non-injectively by m'' than by m' (if this is not the case, m'' is dropped). If m'' is injective, we have found a final graph, but if it is not, we have to perform more decontractions to become injective.

Interestingly we do not need to know the graph S or the original rule from which r was prepared. In fact we can show that the property of being rewritten to a graph larger than S is automatically fulfilled for all graphs we compute.

Lemma 6.25. *The set* $\mathit{reps}_{\mathcal{Q}}^{\sqsubseteq}(r, m)$ *computed by Algorithm 6.24 satisfies the conditions of Definition 6.17.*

Proof. See Appendix C.2. □

With the existence of an appropriate $\mathit{reps}_{\mathcal{Q}}^{\sqsubseteq}$ we have proven the necessary conditions to state a variant of Proposition 6.22 for transition systems induced by injective matches.

Proposition 6.26. *The coverability problem wrt. the minor ordering is decidable for every transition system* $\mathcal{T}^{i}_{\mathcal{G}(\Lambda)}$ *which satisfies the compatibility condition wrt. the minor ordering (see for instance Proposition 5.7).*

Proof. Algorithm 6.24 is an appropriate $\mathit{reps}_{\mathcal{Q}}^{\sqsubseteq}$ for the minor ordering (see Lemma 6.25). All other necessary conditions hold due to Proposition 6.22. □

6.3. Subgraph Ordering

The second main order we will prove compatible with our framework is the subgraph ordering. In fact a lot of the results for the minor ordering can be reused since the subgraph ordering is strictly finer, i.e. contractions are not allowed. We will also see that this significantly simplifies the pred-basis computation for injective matches.

We start by proving preservation by pushouts along total morphisms and pushout closure.

Lemma 6.27. *Subgraph morphisms are preserved by pushouts along total morphisms (cf. Definition 6.3).*

Proof. Let G_3 be the pushout of G_0 along the subgraph morphism $\mu\colon G_0 \rightarrowtail G_1$ and total morphism $g\colon G_0 \to G_2$. Since every subgraph morphism is also a minor morphism, we can use Lemma 6.20 to prove this lemma. It remains to be shown that $\mu'\colon G_2 \rightarrowtail G_3$ is injective where it is defined.

Assume there are two different elements $x_1, x_2 \in G_2$ such that $\mu'(x_1) = \mu'(x_2)$. For G_3 to be a pushout, both x_1 and x_2 have to have preimages $x_1', x_2' \in G_0$ with $g(x_1') = x_1$ and $g(x_2') = x_2$. The diagram commutes, thus μ is defined for both elements and

these elements are mapped injectively to $x_1'', x_2'' \in G_1$ respectively. Hence, there is a commuting diagram with $g'(x_1'') = \mu'(x_1) \neq g'(x_2'') = \mu'(x_2)$, for which there is no mediating morphism from G_3. Note that this last argument holds, since every possible pair of preimages x_1', x_2' is matched injectively by μ. Since this violates the pushout properties of the diagram, μ' has to be injective. □

Lemma 6.28. *Subgraph morphisms are pushout closed (cf. Definition 6.11).*

Proof. This lemma immediately follows from Lemma 6.21. Note that by construction μ_R and μ_G only contract edges if μ does, thus, μ_R and μ_G are injective and subgraph morphisms. □

As for the minor ordering, the set of *minimal pushout complements wrt. the subgraph ordering*, not just restricted to the class $\mathcal{G}_n$ of graphs with bounded paths, is always finite and can be computed by the constructions described in Propositions 3.30 and 3.33. Again we need not add edges in step 3 of Proposition 3.30, since the original graph is a subgraph of all graphs generated this way. This leaves us with the following result.

Proposition 6.29. *For every transition system $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ an effective pred-basis and $\mathcal{G}_n$-pred-basis for the subgraph ordering exists, and the decidability results of Theorem 2.21 apply.*

Proof. We have already proven in Proposition 5.17 that every transformation system $\mathcal{T}$ induces a $\mathcal{G}_n$-restricted WSTS wrt. the subgraph ordering. Furthermore, in Lemmas 5.14, 6.27 and 6.28 we have shown that the subgraph ordering satisfies the necessary conditions to apply Algorithm 6.6. The set of minimal pushout complements – not just restricted to $\mathcal{G}_n$ – can be computed in the same way as it is done for the minor ordering. Thus $preds^c_{\mathcal{G}(\Lambda)}$ is an effective pred-basis and $preds^c_{\mathcal{G}_n}$ is an effective $\mathcal{G}_n$-pred-basis. □

This means that we can apply Algorithm 6.6 using $preds^c_{\mathcal{G}_n}$ to achieve case (iii) of Theorem 2.21. On the one hand, if a graph is represented by the final working set, then it can cover a graph of the initial working set in the unrestricted transition system. On the other hand, if a graph is not represented, it can not cover a graph of the initial working set without exceeding the path bound. Furthermore, we can use $preds^c_{\mathcal{G}(\Lambda)}$ for case (iv) of Theorem 2.21, i.e. if the sequence of working sets becomes stationary – which is not guaranteed in this case – then (general) coverability is decidable for the used instance. In principle Proposition 6.29 does hold not just for $\mathcal{G}_n$, but for all downward-closed classes of graphs (wrt. the subgraph ordering). However, $\mathcal{G}_n$ is the largest class we know of on which the subgraph ordering is a well-quasi-order.

Additionally, the existence of a pred-basis (not just $\mathcal{G}_n$-pred-basis) leads to another interesting result. Given a GTS, a graph G and a constant n, the question whether from G we can reach a graph which contains a path of length at least n, is decidable. This also enables us to decide whether we can use case (ii) of Theorem 2.21 for a given GTS, for which we require that every reachable graph is in the class $\mathcal{Q}$.

Proposition 6.30. *Let $\mathcal{T}$ be a graph transformation system, let G_0 be a graph and let n be a constant. Furthermore, let $\mathcal{T}^c_{\mathcal{G}(\Lambda)} = \langle \mathcal{G}, \Rightarrow \rangle$ be the transition system on all graphs generated by $\mathcal{T}$. The following problem is decidable: is there a graph G with $G_0 \Rightarrow^* G$ and G has a path of length at least n?*

Proof. We use that paths are preserved by the subgraph ordering, i.e. if a graph G has a path of length n, then any larger graph also has a path of length n (or longer). We observe that the problem above is therefore equivalent to checking if a path of length n is coverable or not and we will now show that this is decidable.

Let $\mathcal{P}_n$ be the set of all paths of length n, i.e. the set of all connected graphs which contain a path of length n and are minimal in the sense that any deletion of a node or edge decreases every longest path of the graph by at least one. The set $\mathcal{P}_n$ is a finite basis for the (upward-closed) class of all graphs containing a path of length at least n.

Let $\mathcal{W}$ be the result of applying Algorithm 6.6 with $\mathit{preds}^c_{\mathcal{G}_n}$ to $\mathcal{P}_n$ using the given rule set and restricting the graph class to $\mathcal{G}_n$. Then $G \in \uparrow\mathcal{W}$ implies that G can cover a path of $\mathcal{P}_n$ and $G \notin \uparrow\mathcal{W}$ implies that G cannot cover a path of $\mathcal{P}_n$ in $\Rightarrow_{\mathcal{G}_n}$. However, any minimal pushout complement not in $\mathcal{G}_n$ and therefore not returned by $\mathit{minpoc}^{\subseteq}_{\mathcal{G}_n}$ would have been subsumed by $\mathit{minimize}_{\subseteq}$ (in line 9 of Algorithm 6.6), since every such pushout complement is already represented by the initial working set. Thus, the restriction to $\mathcal{G}_n$ has no effect on the result and $G \notin \uparrow\mathcal{W}$ in fact implies that G cannot cover a path of $\mathcal{P}_n$ in $\Rightarrow$. ☐

With this, if the paths of all graphs reachable from a set of initial graphs are bounded, then the restricted coverability problem is decidable (for this instance).

Checking whether a graph is reachable where a set path bound is exceeded is not the only problem where the "optimistic" variant of Algorithm 6.6, i.e. using a pred-basis not guaranteeing termination, is helpful. We can also use this variant to prove that the coverability problem is decidable for context-free graph transformation systems. In fact, the correctness is guaranteed by the compatibility condition (see case (iv) of Theorem 2.21) and we only need to prove termination. We can do this by exploiting the restricted form of context-free rules.

Proposition 6.31. *Let $\mathcal{T}$ be a context-free graph transformation system. The coverability problem (wrt. the subgraph ordering) is decidable for the transition system $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ induced by conflict-free matches.*

Proof. Let $\mathcal{T}$ be a context-free GTS and let $r \colon L \rightharpoonup R$ be a prepared rule. Due to the special nature of rules of $\mathcal{T}$, we know that L consists of a single edge including its incident nodes and R may be an arbitrary graph. Note that since r is a composition of a context-free rule and a subgraph morphism, r need not be defined for all nodes. Let $m \colon R \rightarrowtail G$ be a co-match of r to some graph G. We will show that if the sum of edges and isolated nodes of a pushout complement G' of r, m is larger than that of G, then

$G \subseteq G'$, i.e. G' is dropped since it is subsumed by G. If this is the case, the algorithm will only keep graphs with a less or equal sum of edges and isolated nodes, of which only finitely many exist for any given G (the number of edge labels and the arity of each label is finite). Thus, in the worst case the algorithm will terminate as soon as all these graphs were computed for all input graphs.

We first observe that every pushout complement G' has at most as many isolated nodes as G. An isolated node can only be added backwards if L has an isolated node (which is by definition not the case). Furthermore, the backward application of every r where R contains at least one edge, removes at least one edge and adds exactly one edge. Thus, the sum of edges and isolated nodes does not increase for such rules.

So assume that R consists only of nodes and there is one node v which has no preimage in L. The co-match m can only match v to an isolated node, since otherwise the dangling condition (see Proposition 3.21) will be violated, i.e. no pushout complement would exist. But this means that the number of isolated nodes decreases by at least one while the number of edges increases by exactly one. Thus, the sum does not increase for this kind of rule as well.

So we finally assume that R consists only of nodes and all nodes have a preimage in L. Then G' is obtained from G by adding some new (non-isolated) nodes and a new edge, i.e. $G \subseteq G'$. A more precise proof of this last statement can be found in Section 6.5, where we prove that pushout complements of any rules which are also order morphisms are immediately subsumed (cf. Proposition 6.44).

Note that all these arguments hold for conflict-free and injective matches alike, since context-free rules are injective. This implies that the co-rule is also injective, hence, two elements are matched non-injectively by the match if and only if these elements are matched non-injectively by the co-match (also implying that there is only one minimal pushout complement). Thus, the backwards application of a rule does not cause any "demergings" (splitting an existing node).

Summarized this means that the number of graphs the algorithm computes is bounded by the size of the input graphs for which the first backward steps are computed. Termination is therefore guaranteed. $\square$

Note that this result is already implied for minor morphisms by Proposition 5.11, in which we have proven that every transition system – regardless of matches – induced by a context-free GTS forms a WSTS wrt. the minor ordering. Thus we can apply Algorithm 6.6 as proven in Section 6.2.

The subgraph ordering can also be used with injective matches. The computation is significantly simpler compared to the minor ordering, since we can reduce $\mathit{reps}_{\mathcal{Q}}^{\subseteq}$ to the following simple check.

Algorithm 6.32 (Represented injective predecessors for the subgraph ordering)**.**
Input: A prepared rule $r\colon L \rightharpoonup R$ and a conflict-free match $m\colon L \rightarrow G$.
Output: The set $\mathit{reps}_{\mathcal{Q}}^{\subseteq}(r, m)$, a finite set of represented injective predecessors.

```
1: if m is injective then
2:     return {G}
3: else
4:     return ∅
5: end if
```

The reason for this is that subgraph morphisms are injective. Therefore the composition of an injective match and a subgraph morphism can never be non-injective, as shown in the following example.

Example 6.33. Let the prepared rule $\mu_R \circ r$ and the co-match n be given such that G' is a pushout complement, as shown in Figure 6.13. Any graph G consists of G' possibly extended with additional nodes and edges, in this case an A- and a B-labelled edge (the dotted parts). Since Definition 6.17 is only concerned with those splits of $\mu_G \circ m$ into μ_G, m that commute, m is non-injective if $\mu_G \circ m$ is. Thus, there are simply no represented injective predecessors and we can drop $\mu_G \circ m$. Furthermore, since injectivity is preserved by pushouts, we do not even have to consider non-injective co-matches n, since these always cause the match to be non-injective as well.

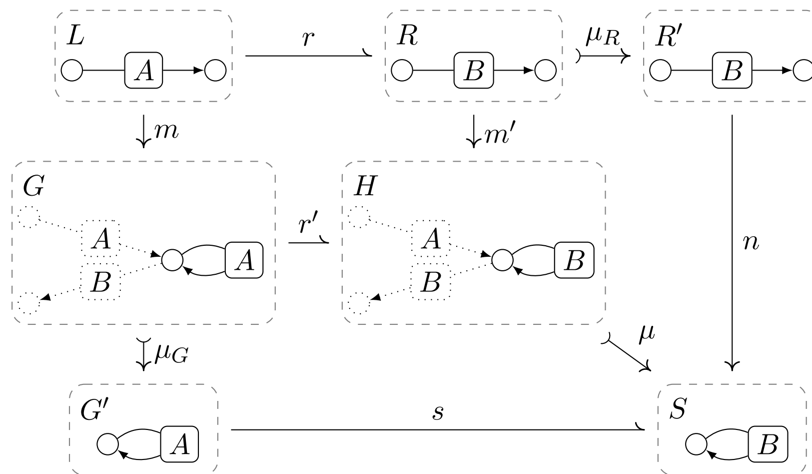


Figure 6.13.: For subgraph morphisms a non-injective match $\mu_G \circ m$ implies a non-injective m

In general, the computation of a set $reps_{\mathbb{Q}}^{\preceq}$ is only necessary if the order $\preceq$ used allows some form of contraction or merging. The proof that $reps_{\mathbb{Q}}^{\subseteq}$ is in fact sufficient for the subgraph ordering is very straight-forward.

Lemma 6.34. *The set* $reps_{\mathbb{Q}}^{\subseteq}(r, m)$ *computed by Algorithm 6.32 satisfies the conditions of Definition 6.17.*

Proof. The first condition of Definition 6.17 is obviously satisfied, since either $reps^{\subseteq}_{Q}(r, m)$ is empty or we return G and can therefore split m into $id_G \circ m$, r into the rule from which it was prepared and a subgraph morphism, and the pushout is S (which is a subgraph of S).

The second condition clearly holds if $reps^{\subseteq}_{Q}(r, m) = \{G\}$, since for every graph G' satisfying the first condition by definition $G \subseteq G'$ holds. Thus, we only need to show that no split is possible if m is non-injective. So assume that there is a split $m = m_2 \circ m_1$ where m_2 is a subgraph morphism and m_1 is injective. This implies that m is injective as well, violating the assumption. □

We can even prove that for injective matches we need not consider total co-matches in line 3 of Algorithm 6.18, but can restrict to injective co-matches. Although an injective co-match does not imply an injective match – we still need to drop some pushout complements – the other direction does hold. All pushout complements for non-injective co-matches will therefore be dropped by $reps^{\subseteq}_{Q}$.

Hence, we obtain for injective matches the same decidability result as for conflict-free matches. For subgraphs the injective case is even easier to compute than the conflict-free case.

Proposition 6.35. *For every transition system $\mathcal{T}^{i}_{\mathcal{G}(\Lambda)}$ an effective pred-basis and $\mathcal{G}_n$-pred-basis for the subgraph ordering exists, and the decidability results of Theorem 2.21 apply.*

Proof. The applicability of the backward search for conflict-free matches (Algorithm 6.6) has been stated in Proposition 6.29. The only additional condition for injective matches is the existence of a procedure $reps^{\subseteq}_{\mathcal{G}(\Lambda)}$, which we have shown in Lemma 6.34. Thus $preds^{i}_{\mathcal{G}(\Lambda)}$ is an effective pred-basis and $preds^{i}_{\mathcal{G}_n}$ is an effective $\mathcal{G}_n$-pred-basis. □

One of the implications of this result is that coverability is decidable for context-free graphs transformation systems and injective matches, i.e. Proposition 6.31 holds for injective matches as well.

6.4. Induced Subgraph Ordering

Our last main order is the induced subgraph ordering, which is strictly finer than the subgraph ordering (pure edge deletions are not allowed). We can again use the results for the subgraph ordering to prove preservation by pushouts along total morphisms and pushout closure.

Lemma 6.36. *Induced subgraph morphisms are preserved by pushouts along total morphisms (cf. Definition 6.3).*

Proof. Since every induced subgraph morphism is also a subgraph morphism, μ' is injective and surjective by Lemma 6.27. Let $e \in G_2$ be an edge on which μ' is undefined. Then e has a preimage $e' \in G_0$ since otherwise the pushout of μ and g would contain e. Since $\mu'(g(e'))$ is undefined, so is $g'(\mu(e'))$. First assume that μ is undefined for e'. Since μ is an induced subgraph morphism, at least one of the nodes v incident to e' is undefined and also μ' has to be undefined on $g(v)$ for the diagram to commute. Now assume $\mu(e')$ is defined, then $g'(\mu(e'))$ must be undefined. However, this can only be the case if there is a conflict, i.e. there is another $e'' \in G_0$ with $g(e') = g(e'') = e$ where $\mu(e'')$ is undefined. This also implies that μ' is undefined on at least one node incident to e. □

Lemma 6.37. *Induced subgraph morphisms are pushout closed (cf. Definition 6.11).*

Proof. In Lemma 6.28 we have shown that there are subgraph morphisms μ_R and μ_G if μ is a subgraph morphism. We will show that these morphisms are induced subgraph morphisms if μ is an induced subgraph morphism.

Let $e \in R$ be an edge on which μ_R is undefined. By definition $\mu(m'(e))$ is undefined and since m' is total, μ is undefined on $m'(e)$ (which is defined). Hence, at least one node v incident to $m'(e)$ has no image under μ and all its preimages under m' (which exist since e has a preimage) are undefined under μ_R. Thus, e is incident to at least one node on which μ_R is undefined on.

Let $e' \in G$ be an edge on which μ_G is undefined. By definition $\mu(r'(e'))$ is undefined and e' has no preimage under m. Because of the latter property, e' is in the pushout H and therefore defined under r'. Thus, μ is undefined on $r'(e)$ and on at least one incident node. All preimages under r' of this node are undefined under μ_G since the diagram commutes. Hence, μ_G is an induced subgraph morphism. □

Unfortunately, the computation of minimal pushout complements wrt. the induced subgraph ordering is more involved. The set of minimal pushout complements wrt. all graphs is also not guaranteed to be finite, as shown in the following example.

Example 6.38. Let a very simple rule and match be given, as shown in Figure 6.14. Both G' and G'' are pushout complements of r, m, since every edge incident to the node 1 is deleted when computing the pushout. For subgraphs we only need to compute G', since it is a subgraph of G''. However, it is not an induced subgraph. Thus, both graphs are minimal wrt. the induced subgraph ordering. In general, every addition of an edge incident to at least one deleted node leads to a new minimal pushout complement. The set is therefore infinite in this case and we need to rely on the bound on edge multiplicity – which is also necessary for the induced subgraph to be a well-quasi-order (see Proposition 5.24) – to achieve a finite set.

We therefore obtain the following algorithm to compute minimal pushout complements.

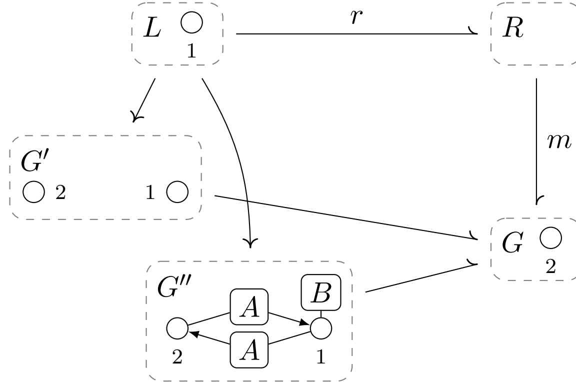


Figure 6.14.: Shows two minimal pushout complements for r, m wrt. the induced subgraph ordering

Algorithm 6.39 (Minimal pushout complements wrt. induced subgraphs)**.**
Input: A rule $r\colon L \rightharpoonup R$ and a co-match $m\colon R \to G$.
Output: The set $\mathit{minpoc}^{\trianglelefteq}_{\mathcal{G}_{n,k}}(r, m)$, the set of minimal pushout complements wrt. $\mathcal{G}_{n,k}$ and the induced subgraph ordering.

1: $\mathcal{G} \leftarrow \mathit{minpoc}^{\subseteq}_{\mathcal{G}_{n,k}}(r, m)$
2: $\mathcal{F} \leftarrow \mathcal{G}$
3: **while** $\mathcal{G} \neq \emptyset$ **do**
4: let G' be any element of $\mathcal{G}$
5: $\mathcal{G} \leftarrow \mathcal{G} \setminus \{G'\}$
6: $\mathcal{H} \leftarrow$ all graphs obtained by adding an edge of any label to G' (without adding new nodes) incident to at least one node deleted by the pushout
7: $\mathcal{F} \leftarrow \mathcal{F} \cup (\mathcal{H} \cap \mathcal{G}_{n,k})$
8: $\mathcal{G} \leftarrow \mathcal{G} \cup (\mathcal{H} \cap \mathcal{G}_{n,k})$
9: **end while**
10: **return** $\mathcal{F}$

The correctness of this algorithm follows from the correctness of our pushout complement constructions Propositions 3.30 and 3.33.

Lemma 6.40. *Algorithm 6.39 computes the set of minimal pushout complements in $\mathcal{G}_{n,k}$ wrt. the induced subgraph ordering.*

Proof. First we observe that line 6 of the algorithm will increase the number of edges between some sequence of nodes. Since no new nodes are added, at some point every sequence of nodes will exceed the multiplicity bound set by $\mathcal{G}_{n,k}$ and no more graphs will be added to $\mathcal{G}$ in line 8. Thus, the algorithm terminates.

Since line 6 corresponds to step 3 of the construction described in Proposition 3.30, it is guaranteed that every graph of $\mathcal{F}$ is a pushout complement of r, m. So we only need to prove that every pushout complement in $G' \in \mathcal{G}_{n,k}$ is represented by some element computed. Without loss of generality we can assume that G' was calculated by the constructions described in Propositions 3.30 and 3.33, i.e. we first compute a graph G'' and then add edges in step 3 of Proposition 3.30 which all satisfy the condition in line 6 of this algorithm. Due to the downward closure of $\mathcal{G}_{n,k}$ (wrt. subgraphs and induced subgraphs), $G'' \in \mathit{minpoc}^{\subseteq}_{\mathcal{G}_{n,k}}(r, m)$. Furthermore, every graph of the sequence $G'', G_1, G_2, \ldots, G_\ell, G'$ obtained by successively adding edges (one per graph) is an element of $\mathcal{G}_{n,k}$ and will not be dropped in lines 7 and 8 of our algorithm. Thus, all minimal pushout complements are contained in $\mathcal{F}$. □

This means that for the induced subgraph ordering an effective $\mathcal{G}_{n,k}$-pred-basis exists. In fact, our pred-basis does only need a multiplicity bound and no path bound to work (similar to the subgraph ordering). However, we were only able to prove that the induced subgraph ordering is a well-quasi-order on $\mathcal{D}_{n,k}$, i.e. directed graphs with bounded paths and multiplicity. We therefore obtain the following somewhat weaker result.

Proposition 6.41. *For every transition system $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ an effective $\mathcal{D}_{n,k}$-pred-basis for the induced subgraph ordering exists, and the decidability results of Theorem 2.21 apply.*

Proof. In Propositions 5.24 and 5.25 we have shown that $\mathcal{T}^c_{\mathcal{G}(\Lambda)}$ forms a $\mathcal{D}_{n,k}$-restricted WSTS wrt. the induced subgraph ordering. Then in Lemmas 5.21, 6.36 and 6.37 we have proven that a class of induced subgraph morphisms exists that satisfies the necessary conditions, i.e. is preserved by pushouts along total morphisms and pushout closed. Finally in Lemma 6.40 we have proven that the set of minimal pushout complements wrt. $\mathcal{D}_{n,k}$ is finite and computable. □

For injective matches the results for subgraphs can be transferred to induced subgraphs. We can again use the simple Algorithm 6.32 to obtain a valid procedure for $\mathit{reps}^{\trianglelefteq}_{\mathcal{Q}}$ (for any $\mathcal{Q}$), i.e. we can drop any pushout complements with non-injective matches. Hence, the decidability result is equivalent to Proposition 6.41.

Proposition 6.42. *For every transition system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ an effective $\mathcal{D}_{n,k}$-pred-basis for the induced subgraph ordering exists, and the decidability results of Theorem 2.21 apply.*

Proof. We have shown most of the necessary conditions in the proof of Proposition 6.41. Algorithm 6.32 computes a valid set $\mathit{reps}^{\trianglelefteq}_{\mathcal{Q}}$ due to Lemma 6.34. Clearly, if for non-injective matches no split is possible for subgraph morphism, then it is also not possible for induced subgraph morphism, a strict subclass of morphisms. □

6.5. Optimizations

When computing our backward analysis there are several places where combinatorial explosions can occur: in the rule preparation, the search for co-matches, the computation of minimal pushout complements and the computation of injective predecessors. Not all these problems can be avoided, but many can be countered by implementing intelligent search strategies to achieve efficiency in the average case, e.g. when searching for co-matches we can abort if we detect that the gluing condition will not be satisfied. In this section we focus not on these search optimizations, but on those that arise from our special setting. The following three optimizations are covered:

- The very simple Algorithm 6.32 for computing $reps_{\mathcal{Q}}^{\preceq}$ can be used for every order $\preceq$ for which $\mathcal{M}_{\preceq}$ contains only injective morphisms (Lemma 6.43). This also means that we only need to compute injective co-matches for those order.
- A rule that is also an order morphism can be removed from the GTS without affecting the results of the backward search (Proposition 6.44). This enables us to reduce the number of rules we need to apply backwards, which turns out to significantly improve the performance.
- Under certain conditions we can convert a rule r to a rule r' which behaves the same when applied backwards, but is undefined for less edges. This works for injective (Proposition 6.46) as well as conflict-free matches (Proposition 6.48). Replacing r with r' enables us to benefit even more from the previous optimization.

We start with the first optimization regarding $reps_{\mathcal{Q}}^{\preceq}$. For subgraphs and induced subgraphs we have shown that the simple Algorithm 6.32 is sufficient for computing $reps_{\mathcal{Q}}^{\preceq}$. We can easily translate this result to any order not allowing some form of contraction or merging, and obtain the following.

Lemma 6.43. *Let $\mathcal{M}_{\preceq}$ be a class of order morphisms for $\preceq$ which contains all isomorphisms. If every $\mu \in \mathcal{M}_{\preceq}$ is injective, then Algorithm 6.32 (which calculates $reps_{\mathcal{Q}}^{\preceq}$) correctly computes the set of represented injective predecessors, i.e. the conditions of Definition 6.17 are satisfied.*

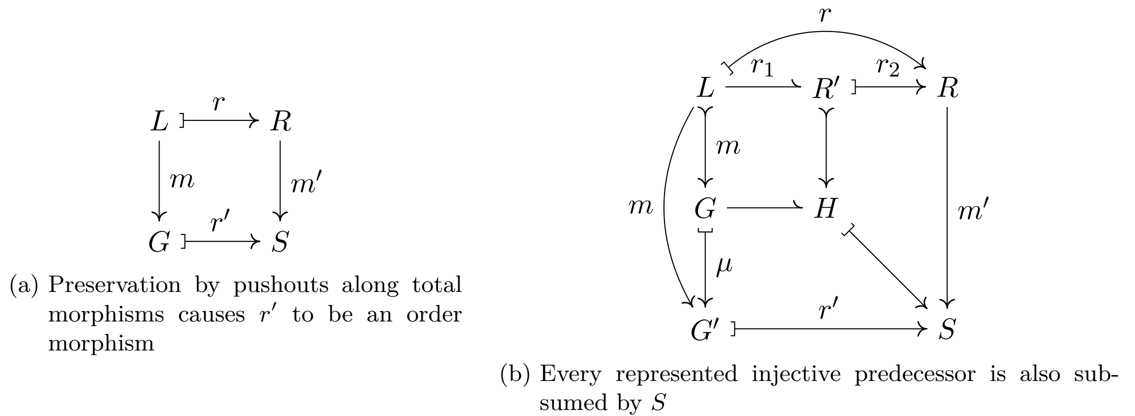
Proof. In the correctness proof for subgraphs (see Lemma 6.34) we only use two properties of subgraph morphisms: every subgraph morphisms is injective and every isomorphisms is a subgraph morphism. Both properties hold by assumption. □

For such order we only need to compute pushout complements for injective co-matches, since non-injective co-matches imply non-injective matches and pushout complement with these matches are dropped.

A rather significant performance boost can be achieved by removing rules for which we know that every pushout complement computed using this rule will be subsumed by another graph in the minimization step. An example for such a rule (for all orders presented so far) is $r\colon L \rightharpoonup R$ where R is the empty graph and L is arbitrary. Given the only possible match $m\colon R \to G$ to some G the minimal pushout complement wrt. the minor and subgraph orderings is G', the disjoint union of G and either L or a version of L where some nodes and edges are merged (if conflict-free matches are used). However, G is obviously a subgraph and minor of G'. For induced subgraph the minimal pushout complements contain not just G', but also all graphs obtained from G' by adding edges incident to at least one node of L. Again G is an induced subgraph of all such graphs. In fact, we can prove that every pushout complement computed using a rule which is also an order morphism will be subsumed, regardless of whether we use injective or conflict-free matches and also independent of how minimal pushout complements are computed.

Proposition 6.44. *Let $\mathcal{M}_{\preceq}$ be a class of order morphisms that satisfies all necessary conditions to apply the backward search (Algorithm 6.6) using either $preds^c_{\mathcal{Q}}$ (Algorithm 6.10) or $preds^i_{\mathcal{Q}}$ (Algorithm 6.18) as $\mathcal{Q}$-pred-basis. For every $G \in preds^c_{\mathcal{Q}}(\mathcal{T}, S)$ or $G \in preds^i_{\mathcal{Q}}(\mathcal{T}, S)$ computed using a rule $r \in \mathcal{T} \cap \mathcal{M}_{\preceq}$ it holds that $S \preceq G$.*

Proof. First assume that $G \in preds^c_{\mathcal{Q}}(\mathcal{T}, S)$. This means that there is a (prepared) rule $r\colon L \rightarrowtail R$ (by assumption an order morphism) and a conflict-free match $m\colon L \to G$ such that S is the pushout of r, m, as shown in Figure 6.15a. Since $\mathcal{M}_{\preceq}$ is preserved by pushouts along total morphisms (cf. Definition 6.3), the morphism r' is an order morphism as well. Thus, $S \preceq G$ holds. Note that this argument holds regardless of the computation of $minpoc^{\preceq}_{\mathcal{Q}}(r, m)$, since by definition every element of this set is a pushout complement.



(a) Preservation by pushouts along total morphisms causes r' to be an order morphism

(b) Every represented injective predecessor is also subsumed by S

Figure 6.15.: Every graph computed by $preds^c_{\mathcal{Q}}$ or $preds^i_{\mathcal{Q}}$ is subsumed by S

Now assume $G \in preds^i_Q(\mathcal{T}, S)$. Again, there is a (prepared) rule $r\colon L \mapsto R$ and an injective match $m\colon L \rightarrowtail G$, but the pushout of r, m need not be S. By definition we can split r into $r_2 \circ r_1$ and there is a $\mu\colon G \mapsto G'$ such that $\mu \circ m$ is conflict-free wrt. r and S is the pushout of r and $\mu \circ m$, as shown in Figure 6.15b. As in the conflict-free case r' is an order morphism proving $S \preceq G' \preceq G$ for all G. □

The proof of Proposition 6.44 also implies that isomorphisms can me removed from the set of rules, since isomorphisms are preserved by pushouts as well. Due to Definition 5.1 – even if r' is no order morphism – $S \preceq G$ is guaranteed. Thus, an obvious improvement is to modify $prepare_{\preceq}$ to not return prepared rules that are order morphisms or isomorphism.

Another possibility to reduce the number of rules is the deletion of "equivalent" rules. Under some circumstances rules will produce the same pushout complements, i.e. behave the same, even if the morphisms are not isomorphic.

Example 6.45. For instance, let $r\colon L \rightharpoonup R$ be a rule where L consists of a single edge together with its incident nodes, R is isomorphic to L and r is the isomorphism. This rule behaves the same as a rule $r'\colon L \rightharpoonup R$ which is the same as r, but undefined on the edge. Obviously, r does not change any graph it is applied to (forwards or backwards), but this also holds for r', since an application deletes an edge and then immediately adds an edges with the same label at the same position. We therefore only need to consider one of these rules in our analysis. In fact, we can delete r' – regardless of the order used – since it behaves like an isomorphism, although it is neither an isomorphism nor an order morphism!

We will first prove this property for injective matches and then extend it to conflict-free matches. However, this extension is only possible when the left-hand side satisfies some correctness criteria, namely it may not contain another edge with the same label. Later in Example 6.47 we will illustrate the problem occurring if this condition is violated.

Proposition 6.46. *Let $r\colon L \rightharpoonup R$ be a rule, let $e \in E_L$ be an edge where $r(e)$ has only one preimage in L (which is e) and let $r'\colon L \rightharpoonup R$ be defined as $r'(x) = r(x)$ for $x \neq e$ and $r'(e)$ undefined. For every injective match $m\colon L \rightarrowtail G$ the pushout object of r, m is isomorphic to the pushout object of r', m.*

Proof. By definition there is a $k\colon L \rightharpoonup L$ with $r' = k \circ r$ which is the identity on every $x \neq e$ and undefined on e. We can now form the pushout of k, m and then of r, m', as shown in Figure 6.16. By the properties of pushouts, G'' is the pushout of m, r'. Since k and k' are both injective (and almost the identity), the commutativity $m' \circ k = k' \circ m$ ensures that $m(x) = m'(x)$ for all $x \neq e$. This means that $k'(c_G(m(e))) = c_{G'}(m'(e))$, which – together with the fact that k' is a bijection on nodes – implies that m, m' and G, G' are isomorphic, respectively. Thus, the pushouts of r, m and r', m are isomorphic as well. □

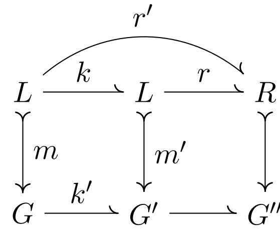


Figure 6.16.: Diagram showing the morphisms of Proposition 6.46

As a result of Proposition 6.46 we can further minimize the set of rules for the subgraph and induced subgraph orderings. Before or after rule preparation we can "extend" all prepared rules $r\colon L \rightharpoonup R$ where there are $e \in E_L$ and $e' \in E_R$ where:

- $r(e)$ is undefined, but $r(v)$ is defined for all nodes v incident to e,
- e' has no preimage in L and
- $r(c_L(e)) = c_R(e')$.

The extended rule $r'\colon L \rightharpoonup R$ with $r'(x) = r(x)$ for $x \neq e$ and $r'(e) = e'$, fully replaces r. We can repeat the extensions until there are no more pairs e, e' satisfying the necessary conditions. Since this may cause more rules to be subgraph or induced subgraph morphisms, it can further reduce the number of rules when used with the other optimizations.

Unfortunately, this optimization needs to be restricted before we can use it also for conflict-free matches. This is especially a problem for the minor ordering, since we need to compute pushout complement for conflict-free matches even if we restrict to injective ones. The problems occurring for conflict-free matches are illustrated in the following example.

Example 6.47. Let the rule r be given as shown in Figure 6.17, where m is a conflict-free match resulting in the pushout G'. Note that r is undefined on all edges. If we would extend r to match for instance the edge 3 in L to 5 in R, then m would no longer be conflict-free. In fact, G would no longer be a pushout complement of r, m' and would not be computed by our backward algorithm!

Our optimization is therefore only possible for rules where it is guaranteed that edges are matched injectively. Since nodes can also be matched non-injectively, this is only the case if edges in L have unique labels. Note that this means that r may be extended for some edge, but no for others, even if they satisfy the conditions of Proposition 6.46. This leads to a more restricted version of this proposition.

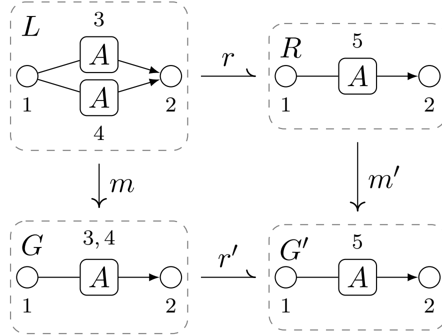


Figure 6.17.: Shows problem with extending rules when using conflict-free matches

Proposition 6.48. *Let $r\colon L \rightharpoonup R$ be a rule, let $e \in E_L$ be an edge where $r(e)$ has only one preimage in L (which is e) and where e is the only edge in L with the label $l_L(e)$, and let $r'\colon L \rightharpoonup R$ be defined as $r'(x) = r(x)$ for $x \neq e$ and $r'(e)$ undefined. For every conflict-free match $m\colon L \rightarrow G$ the pushout object of r, m is isomorphic to the pushout object of r', m.*

Proof. From the proof of Proposition 6.46 it already follows that m, m' and G, G' are isomorphic, and since m is conflict-free wrt. r' it is guaranteed that m' is total. Thus, we only need to show that m' is also conflict-free wrt. r.

So assume that there are $x_1, x_2 \in L$ with $m'(x_1) = m'(x_2)$ and $r(x_1)$ is defined while $r(x_2)$ is undefined. Neither x_1 nor x_2 can be the edge e on which r is defined but r' is not, since this would require that there is another edge with the same label in L. However, any other x_1, x_2 for which m' has a conflict would imply that m has the same conflict wrt. r', which is false by assumption. □

When using conflict-free matches we can therefore also extend rules, similar to the injective case.

6.6. Universally Quantified Rules

One drawback of the SPO approach, used in this thesis, is that rules can only change a fixed part of a graph they are applied to, i.e. the match of the left-hand side. However, if we want to model for instance broadcast protocols, we need to have rules that can match the entire neighbourhood of a node. An example of such a rule can be seen in Figure 6.18. There we have a process node, marked with an unary S_1-labelled edge indicating that it is currently in state S_1, and we want to send a message along all outgoing connections (C-labelled edges). The rule should automatically match all incident C-labelled edges and rewrite them to M-labelled edges, indicating that a message was sent along this

connection. At the same time the process should change its state from S_1 to S_2. Note that this rule is not expressible by normal SPO rules.

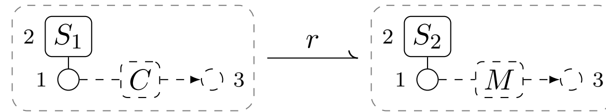


Figure 6.18.: A rule rewriting all C-labelled edges incident to the S_1-marked node to M-labelled edges

In [DSZ10] Delzanno et al. used similar but less expressive broadcast operations – in addition to local operations (i.e. standard replacement) – and the induced subgraph ordering, to analyse broadcast protocols. For this they defined their own rewriting syntax and semantic. For category-based graph transformation so-called adaptive star grammars exist [DH+06], which can clone left-hand sides to extend a rule to match an arbitrary large part of the graph. In this section we use a similar idea in combination with the theory of amalgamated graph transformations [BFH87] to define our own, more flexible, rules which we call universally quantified. We then prove the applicability of our backward search even for this more general type of rules, but only for the subgraph ordering in conjunction with injective matches. In principle the proofs also hold for the induced subgraph ordering, but not necessarily for the minor ordering or conflict-free matches, since they assume injectivity of matches as well as order morphisms. Whether such extensions are possible was not yet investigated. The theory presented in this section was first published in [DS14b; DS14a].

We start by formalizing the notion of universally quantified rules as an extension of normal rules. These new rules need to be instantiated via a sequence of recursive instantiation steps to generate a rule morphism which can then be applied as per the standard SPO approach.

Definition 6.49 (Universally quantified rules)**.** A *universally quantified rule* is a pair $\rho = \langle r, U\rangle$, where $r\colon L \rightharpoonup R$ is a partial morphism, called the *core rule*, and U is a finite set of universal quantification pairs. A *universal quantification pair (or short q-pair)* is a pair $\langle p_u, q_u\rangle = u \in U$ where $p_u\colon L \rightarrowtail L_u$ is a total injective morphism and $q_u\colon L_u \rightharpoonup R_u$ is a partial morphism satisfying the restriction that $q_u(p_u(x))$ is defined and has exactly one preimage in L_u for every $x \in L$.

With $qn(u)$ we denote the set of *quantified nodes* of u, which is the set of all $v \in V_L$ such that there is an edge incident to $p_u(v)$ which has no preimage in L. We denote the quantified nodes of a rule the same way, i.e. $qn(\rho) = \bigcup_{u\in U} qn(u)$. We require that $qn(u) \neq \emptyset$ for all $u \in U$.

A *universally quantified graph transformation system* (UGTS) is a graph transformation system containing universally quantified rules in addition to regular ones.

The idea of universally quantified rules is to take the core rule and extend it using the q-pairs to finally obtain an applicable rule morphism, a so-called instantiation. For this the core rule and q-pairs are recursively merged via amalgamation. The morphism p_u of a q-pair u defines the common interface for this amalgamation between the core rule and u, whereas q_u defines what the q-pair adds to the rule. Some consistency conditions in Definition 6.49 ensure that this amalgamation does not cause unexpected behaviour. To ensure that an instantiation cannot be applied if it can be extended further, we will state match conditions for quantified nodes, since these are the nodes of which the degree is affected by instantiation. Note that the set of q-pairs of a rule may be empty, but the semantic of such universally quantified rules differs from standard SPO rules.

Definition 6.50 (Instantiation of universally quantified rules)**.** An instantiation of a universally quantified rule $\rho = \langle r, U \rangle$ consists of a total injective morphism $\pi \colon L \rightarrowtail \overline{L}$ and a partial morphism $\gamma \colon \overline{L} \rightharpoonup \overline{R}$ and is recursively defined as follows:

- The pair $\langle id_L \colon L \rightarrowtail L, r \rangle$, where id_L is the identity on L, is an instantiation of ρ.
- Let $\langle \pi \colon L \rightarrowtail \overline{L}, \gamma \colon \overline{L} \rightharpoonup \overline{R} \rangle$ be an instantiation of ρ and let $\langle p_u \colon L \rightarrowtail L_u, q_u \colon L_u \rightharpoonup R_u \rangle = u \in U$. Furthermore, let $\overline{L}_u$ be the pushout of π, p_u and let $\overline{R}_u$ be the pushout of $\gamma \circ \pi$, $q_u \circ p_u$, as shown in the diagram in Figure 6.19. Then $p'_u \circ \pi$ and the (unique) mediating morphism η are also an instantiation of ρ. We write $\langle p'_u \circ \pi, \eta \rangle = \langle \pi, \gamma \rangle \diamond u$ to indicate that the instantiation $\langle \pi, \gamma \rangle$ was extended by using u.

We say that the length of an instantiation is the number of steps performed to generate the instantiation, where $\langle id_L, r \rangle$ has a length of 0.

$$\begin{array}{ccccc}
L & \xrightarrow{\pi} & \overline{L} & \xrightarrow{\gamma} & \overline{R} \\
{\scriptstyle p_u}\downarrow & & \downarrow{\scriptstyle p'_u} & & \downarrow \\
L_u & \xrightarrow{\pi'} & \overline{L}_u & & \\
{\scriptstyle q_u}\downarrow & & & \searrow{\scriptstyle \eta} & \\
R_u & & \longrightarrow & & \overline{R}_u
\end{array}$$

Figure 6.19.: Diagram showing an instantiation of a universally quantified rule

Example 6.51. In Figure 6.20 we illustrate the instantiation process for the rule $\rho = \langle r, \{\langle p_u, q_u \rangle\} \rangle$ given in Figure 6.18. The core rule simply replaces a unary S_1-labelled edge with a unary S_2-labelled edge and can be extended using u to also replace all outgoing

C-labelled edges with M-labelled edges. Figure 6.20 shows the trivial instantiation (with length 0) in the top row and extends it by u to obtain the instantiation $\langle p'_u \circ id_L, \eta\rangle$ (which has length 1). The common interface for the amalgamation of r and q_u consists only of the (grey) quantified node and its incident S_1-labelled edge. Note that the conditions for q-pairs forbid q_u to change anything in the common interface, thus preventing a conflict while instantiating. The rule morphisms η of the new instantiation now performs both the changes r and q_u would perform.

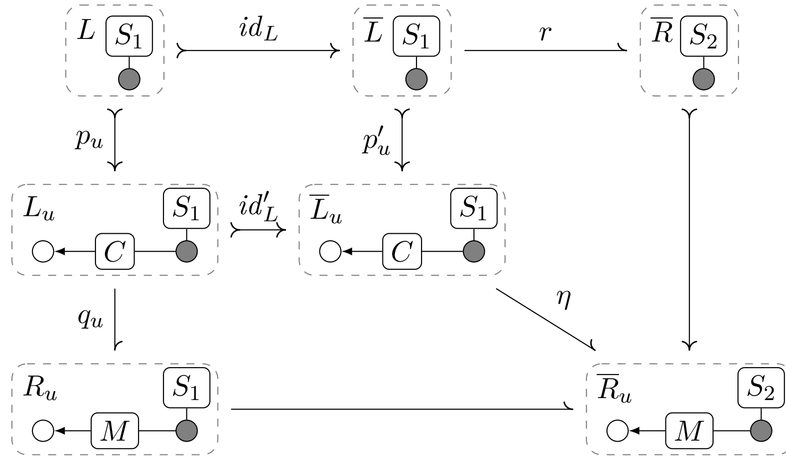


Figure 6.20.: A possible instantiation of a rule r with a quantification $\langle p_u, q_u\rangle$

Any further instantiation with u will add an additional node and C-labelled edge to $\overline{L}_u$ and an additional node and M-labelled edge to $\overline{R}_u$. As we will define later, the rule morphism η is only applicable if the grey node is matched to a node with degree (exactly) two, preventing the application of an instantiation which can still be instantiated further. The rule application is performed by calculating the pushout of η (not r) and a valid match m as normal. The match is only valid if all edges incident to the grey node have a preimage in $\overline{L}_u$, such that an application will always result in all incident C-labelled edges to be replaced by M-labelled edges. Although the number of affected edges can be arbitrary large, the quantification it bounded to the neighbourhood of the grey node and therefore the changes are still local.

From Definitions 6.49 and 6.50 it implicitly follows that parts of an instantiation which are generated in different instantiation steps are independent. In fact, we can prove that the order in which q-pairs are used to generate an instantiation can be neglected, since any two instantiation steps of a sequence can be swapped without changing the resulting instantiation. Therefore we can uniquely specify instantiations by the number of uses of each q-pair in the instantiation sequence, as shown in the following lemma.

Lemma 6.52. *Let $\rho = \langle r, U\rangle$ be a rule and let $f\colon U \to \mathbb{N}_0$ be any function assigning a quantity to each q-pair. Every instantiation of ρ which is generated by using $f(u)$ occurrences for each u respectively, yields the same morphisms (up to isomorphism).*

Proof. See Appendix C.3 □

Finally, based on instantiations and quantified nodes we can define when a universally quantified rule is applicable. This is the case if the entire neighbourhood of every quantified node is matched. Intuitively, we can search for a match of the core rule and then successively perform instantiation steps while updating the match until the entire neighbourhood is matched. If this is not possible, the rule is not applicable.

Definition 6.53 (Application of universally quantified rules)**.** Let ρ be a universally quantified rule. We say that ρ is applicable to a graph G, if there is an instantiation $\langle \pi\colon L \rightarrowtail \overline{L}, \gamma\colon \overline{L} \rightharpoonup \overline{R}\rangle$ of ρ and a total injective match $m\colon \overline{L} \rightarrowtail G$, such that for every $x \in qn(\rho)$, there is no $e \in E_G$ incident to $m(\pi(x))$ without a preimage in $\overline{L}$. The application of ρ to G via m results in the graph H, the pushout of m and γ.

We reuse the notation $G \overset{\rho,m}{\Rightarrow} H$ (or simply $G \Rightarrow H$) to denote a rewriting step from G to H using an appropriate instantiation of ρ and match m.

Note that this definition introduces a restricted form of negative application condition since the existence of an edge, which cannot be mapped, may block the application of a rule if no larger instantiation can map this edge. As we will see later, this causes the working set computed by the backward search to be an over-approximation, since the compatibility condition of WSTS is no longer satisfied. It is also possible to weaken Definition 6.53 to only require that an instantiation is only applicable if there is no larger instantiation which is also applicable by extending the match. The neighbourhood of quantified nodes need then not be matched entirely, but only those edges that can be matched by extending with some q-pair. In fact, this alternative definition coincides with the one used by Delzanno et al. [DSZ10]. The backward search would still compute an over-approximation for the subgraph ordering, but would be precise for the induced subgraph ordering.

We now need to extend Algorithm 6.6, our backward search, to handle universally quantified rules in addition to regular ones. In the following we assume that $\mathcal{T}$ is a UGTS, i.e. contains both kind of rules. Applying a universally quantified rule backward can be done quite straight-forwardly by computing all necessary instantiations and applying those backward which are applicable. However, this requires a bound on the number of instantiations (since there are infinitely many) and requires instantiations to be prepared after their computation. We can prove that we do not need to compute instantiations that have a greater length than the size of G, the graph to which we want to apply ρ backwards. Intuitively, if we perform more instantiation steps than G has nodes and edges, then the subgraph morphisms composed with the instantiation must be undefined

on all elements added in one of those steps (otherwise there can be no co-match). By not performing that step in the first place, we obtain a smaller instantiation that already represents all pushout complements that we would compute with the larger instantiation. We prove this in the following proposition.

Proposition 6.54. *Let ι be an instantiation of length k of some rule ρ. If k is larger than the number of nodes and edges of G, then every graph computed by the backward application of ι is already represented by the backward application of an instantiation of lower length.*

Proof. See Appendix C.3. □

In the following we use $bound_\rho(G)$ to denote the bound on the number of instantiations that exists due to Proposition 6.54. Note that this bound only depends on G and not on ρ. By taking the structure of ρ into account we can further reduce the bound, but this is not necessary to prove the correctness of our $\mathcal{Q}$-pred-basis.

Preparation of the universally quantified rules ρ is only possible in a very limited way. Since we need to prepare the instantiations anyway, we can define $prepare_{\preceq}(\mathcal{T})$ to preserve all universally quantified rules unchanged, without loosing correctness. With this assumption we can define the $\mathcal{Q}$-pred-basis $uq\text{-}preds^i_{\mathcal{Q}}$ as follows.

Algorithm 6.55 (Extended injective $\mathcal{Q}$-pred-basis)**.**
Input: A (prepared) rule set $\mathcal{T}$ and a graph $S \in \mathcal{Q}$. We use $\mathcal{T}_s$ to denote the set of (prepared) standard rules and $\mathcal{T}_{uq}$ to denote the set of (unprepared) universally quantified rules.
Output: The set $uq\text{-}preds^i_{\mathcal{Q}}(\mathcal{T}, S)$, an effective $\mathcal{Q}$-pred-basis for the subgraph ordering with injective matches and universally quantified rules.

1: $\mathcal{G} \leftarrow preds^i_{\mathcal{Q}}(\mathcal{T}_s, S)$
2: **for all** $\rho \in \mathcal{T}_{uq}$ **do**
3:     **for all** instantiations $\langle \pi, \gamma \colon \overline{L} \rightharpoonup \overline{R} \rangle$ of ρ with length $bound_\rho(G)$ or less **do**
4:         **for all** subgraph morphisms $\mu \colon \overline{R} \rightarrowtail \overline{R}'$ **do**
5:             **for all** total co-matches $m \colon \overline{R}' \rightarrow S$ **do**
6:                 $\mathcal{G}_{poc} \leftarrow minpoc^{\subseteq}_{\mathcal{Q}}(\mu \circ \gamma, m)$
7:                 **for all** $\langle G', m' \colon L \rightarrow G', r' \colon G' \rightharpoonup S \rangle \in \mathcal{G}_{poc}$ **do**
8:                     **if** m' is injective and satisfies Definition 6.53 **then**
9:                         $\mathcal{G} \leftarrow \mathcal{G} \cup \{G'\}$
10:                     **end if**
11:                 **end for all**
12:             **end for all**
13:         **end for all**
14:     **end for all**
15: **end for all**

16: **return** $\mathcal{G}$

In line 1 we use the standard injective $\mathcal{Q}$-pred-basis to compute predecessors of standard rules. We then need to compute instantiations for every universally quantified rule that do not exceed the bound (line 3). This instantiations are prepared in line 4. Note that, as in $\mathit{prepare}_{\preceq}(\mathcal{T})$, we only need to consider one representative for every class of isomorphic subgraph morphisms. Finally, after computing co-matches and pushout complements we need to drop all graphs where the match is not valid (line 8), i.e. where not the entire neighbourhood of all quantified nodes is matched. The latter check can be lifted to a check for the co-match, i.e. a post condition, as we will later introduce as an optimization.

To prove that Algorithm 6.55 does compute a correct $\mathcal{Q}$-pred-basis we need to show that the Lemmas 6.13 and 6.14 hold for $\mathit{uq\text{-}preds}^i_{\mathcal{Q}}$, i.e. the computed set is finite but still contains all minimal predecessors. For this we can reuse the proof of Proposition 6.19, where we have shown that $\mathit{preds}^i_{\mathcal{Q}}$ is a valid $\mathcal{Q}$-pred-basis. In fact, very property we have shown for prepared rules holds for prepared instantiations as well. Additionally we can prove that if a pushout complement satisfies the application condition, any smaller pushout complement satisfies it as well. This allows us to prove effectiveness of $\mathit{uq\text{-}preds}^i_{\mathcal{Q}}$ either for all graphs or graphs where the longest path is bounded, which we state in the following proposition.

Proposition 6.56. *Let $\mathcal{T}$ be a UGTS containing standard rules as well as universally quantified rules. The set $\mathit{uq\text{-}preds}^i_{\mathcal{G}_n}(\mathcal{T}, S)$ computed by Algorithm 6.55 is an effective $\mathcal{G}_n$-pred-basis for the transition system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ when using the subgraph ordering. Furthermore, $\mathit{uq\text{-}preds}^i_{\mathcal{G}(\Lambda)}(\mathcal{T}, S)$ is an effective pred-basis.*

Proof. See Appendix C.3. □

Unfortunately, although we have proven the existence of a $(\mathcal{G}_n)$-pred-basis for UGTS, enabling us to apply our main backward search (Algorithm 6.6), we obtain a weaker result than the one for standard GTS. The problem is that UGTS do not satisfy the compatibility condition of $\mathcal{G}_n$-restricted WSTS, since a rule may become inapplicable to some G by the addition of a single edge to G. However, our procedure exploits this property and may therefore find invalid paths in the transition system. Thus, the final working set is an over-approximation of the actual set (which need not be finitely representable).

We will now discuss some optimizations regarding rule preparation, tightening the bound and checking the application condition as a postcondition.

Optimization by Preparation

In Algorithm 6.55 we prepare instantiations rather than universally quantified rules. Preparing the rules itself is more complex, but would allow for more optimization and

eliminate the need of preparing the instantiations. To do this we would have to compose subgraph morphisms not just with the core rule, but also with each q-pair. However, composing just one subgraph morphism with each q-pair is not sufficient, since this results in an instantiation where every part generated from the same q-pair is prepared in the same way. For instance the composition of the instantiation γ and the subgraph morphism μ shown in Figure 6.21 would not be possible. To cover this case in the preparation of a universally quantified rule we would need to add one new q-pair for every possibility to compose an old q-pair with a subgraph morphism, such that the new q-pair is still injective and total on the common interface.

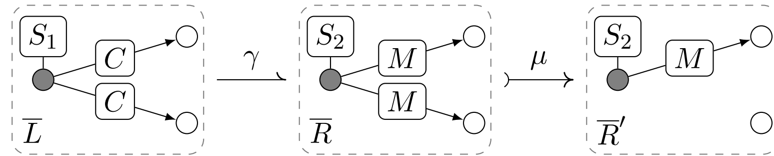


Figure 6.21.: Shows an instantiation of the rule in Figure 6.18 composed with a μ which would be covered if all q-pairs are prepared independently in the same way

In Section 6.5 we proved that rules which are also order morphisms can be removed from the rule set, since they do not affect the computation of working sets (except by worsening performance). The same can be done for universally quantified rules, but only if all instantiations of such rules are order morphisms. This is for instance the case if the core rule and all quantifications are order morphisms. Unfortunately, in general we can not simply remove a q-pair from a rule even if this q-pair is an order morphisms. It may be necessary to instantiate with this q-pair to satisfy the application condition.

Tightening the Upper Bound on Instantiations

The bound on the length of instantiations proven to exist in Proposition 6.54 can be improved depending on the rule used. For instance, let $\rho = \langle r\colon L \rightharpoonup R, U\rangle$ be a universally quantified rule. If $U = \emptyset$, then $\mathit{bound}_\rho(G) = 0$ holds obviously for all G, since there is only one possible instantiation. The same holds if instantiations only increase the left-hand side of the rule, i.e. for every $u \in U$ given the instantiation $\langle id_L, r\rangle \diamond u = \langle \pi\colon L \rightarrowtail \overline{L}_u, \gamma\colon \overline{L}_u \rightharpoonup \overline{R}_u\rangle$, the graphs $\overline{R}_u$ and R are isomorphic.

A more common situation is that q-pairs do not add edges to the right-hand side which are solely incident to nodes of the core rule r. For instance the rule in Figure 6.18 has this structure. In such a case the bound can be reduced to the number of nodes of G instead of the number of nodes and edges. The reason is that if the composed subgraph morphism removes all nodes which where added by an instantiation step using some q-pair, all edges are removed as well, effectively making the instantiation step unnecessary. Thus every meaningful instantiation step introduces at least one node. We formalize this in the following lemma.

Lemma 6.57. *Let* $\rho = \langle r\colon L \rightharpoonup R, U\rangle$ *and let* $\langle id_L, r\rangle \diamond u = \langle \pi\colon L \rightarrowtail \overline{L}_u, \gamma\colon \overline{L}_u \rightharpoonup \overline{R}_u\rangle$*. If for every* $u \in U$ *every edge* $e \in \overline{R}_u$ *without preimage in* R *is incident to a node* $v \in \overline{R}_u$ *without preimage in* R*, then* $bound_\rho(G) = |V_G|$*.*

Proof. See Appendix C.3. □

Lifting the Application Condition to a Post Condition

In Algorithm 6.55 the application condition is checked in line 8 for each pushout complement. However, we can lift the application condition over the instantiation to check beforehand whether the backward step will yield new graphs. For instance, if the co-match does not already match the entire neighbourhood of an image of a quantified node, the edge not matched will violate the application condition for every existing pushout complement. Checking a postcondition instead of a precondition can greatly reduce the number of pushout complements computed. We prove this lifting in the following lemma.

Lemma 6.58. *Let* ρ *be a rule,* $\langle \pi\colon L \rightarrowtail \overline{L}, \gamma\colon \overline{L} \rightharpoonup \overline{R}\rangle$ *an instantiation of* ρ *and* $m\colon \overline{R} \rightarrowtail G$ *a co-match of the instantiation to some graph* G*. If there is a node* $x \in qn(\rho)$ *where* $m(\gamma(\pi(x))$ *is defined and incident to an edge* e *without preimage in* $\overline{R}$*, then there is no pushout complement* H *of* γ*,* m *satisfying the condition of Definition 6.53.*

Proof. See Appendix C.3. □

6.7. Summary

In this chapter I have defined the core of a framework for verifying graph transformation systems and also introduced extensions and optimizations. I will now summarize the implied decidability results and verification procedures. Practical examples can be found in Chapter 7 where these results are illustrated by a number of case studies.

Because of the generality of our framework, there are several dimensions affecting decidability. First of all there is the order, which we made variable in our framework. We have shown the minor ordering, the subgraph ordering and the induced subgraph ordering to be compatible, but are not limited to these orders (see the end of Section 6.1 for necessary and sufficient conditions).

Another dimension is the graph class $\mathcal{Q}$ to which we restrict. This class depends on the chosen order, since it needs to be downward-closed wrt. that order. If the order is a well-quasi-order on $\mathcal{Q}$, then it is guaranteed that the set of predecessors (restricted to $\mathcal{Q}$) of any set of graphs is finitely representable and that the algorithm will terminate for every instance. If this is not the case, the algorithm is still applicable if predecessor sets are finitely representable, but termination is no longer guaranteed. In general one is interested in the largest class of graphs on which the order is a well-quasi-order. We have identified these classes to be the class of all graphs $\mathcal{G}(\Lambda)$ for the minor ordering, the

class of graphs where longest paths are bounded $\mathcal{G}_n$ for the subgraph ordering and the class of *directed* graphs where longest paths and multiplicity are bounded $\mathcal{D}_{n,k}$ for the induced subgraph ordering. For other orders there might be multiple interesting classes.

The last dimension is the class of graph transformation systems. These normally arise from the order due to the restrictions imposed by the compatibility condition (see Definition 2.16). For the minor ordering it is the class of lossy systems, for the subgraph ordering it is the class of ordinary graph transformations systems and for the induced subgraph ordering it is the class of graph transformation systems with simple negative application conditions. However, we can also use graph transformation systems that do not satisfy the compatibility condition. Results obtained for such systems are then an over-approximation. Note that the transition system induced by a graph transformation system is also affected by the type of matches used. This actually forms another dimension that we will not need to distinguish in this section, since we have proven both injective and conflict-free matches to be fully compatible with our three main orders. Therefore, the results shown in the following are for both these match types.

In Table 6.1 it is shown which decidability results our framework provides for every combination wrt. the mentioned dimensions. The possible results are briefly summarized in Table 6.2 and we will investigate them for each order in the following.

GTS	class $\mathcal{Q}$	minor	subgraph	ind. subgraph
lossy systems (cf. Proposition 5.7)	$\mathcal{G}(\Lambda)$	Dec	Dec	NoRes
	$\mathcal{G}_n$	Thm	Thm	NoRes
	$\mathcal{D}_{n,k}$	Thm	Thm	Thm
ordinary GTS (cf. Definition 3.19)	$\mathcal{G}(\Lambda)$	App	NoTerm	NoRes
	$\mathcal{G}_n$	ThmApp	Thm	NoRes
	$\mathcal{D}_{n,k}$	ThmApp	Thm	Thm
GTS with simple NACs (cf. Definition 5.27)	$\mathcal{G}(\Lambda)$	App	NoTermApp	NoRes
	$\mathcal{G}_n$	ThmApp	ThmApp	NoRes
	$\mathcal{D}_{n,k}$	ThmApp	ThmApp	Thm

Table 6.1.: Shows decidability results implied by the applicability of the backward search described in this chapter for varying combinations of orders, GTS and graph classes (see Table 6.2 for a legend)

Minors

The minor ordering is the coarsest order we presented and is in fact a well-quasi-order on all graphs. Since it satisfies the compatibility condition wrt. lossy systems, the coverability problem is decidable wrt. these systems (we obtain an ordinary well-structured transition system). If we restrict to $\mathcal{G}_n$ or $\mathcal{D}_{n,k}$, we obtain the split decidability result

DEC	Algorithm decides the problem, i.e. the returned set $\mathcal{W}_*$ can be used to check coverability for arbitrary initial graphs.
APP	The set $\mathcal{W}_*$ returned by the algorithm is an over-approximation. If $G \notin \uparrow\mathcal{W}_*$, then G can not cover a final graph, but if $G \in \uparrow\mathcal{W}_*$, then this may be because of approximation.
NOTERM	Termination is not guaranteed, but if the algorithm terminates, the problem is decidable for that instance. The returned set $\mathcal{W}_*$ can be used to check coverability for arbitrary initial graphs.
NOTERMAPP	Termination is not guaranteed, but if the algorithm terminates, then $\mathcal{W}_*$ can be used as in the case of APP.
THM	Split decidability result (case (iii) of Theorem 2.21) applies. See also Figure 6.22.
THMAPP	Case (iii) of Theorem 2.21 applies, but the result is an over-approximation. If $G \notin \uparrow\mathcal{W}_*$, then G can not cover a final graph within $\mathcal{Q}$, but if $G \in \uparrow\mathcal{W}_*$, this may be caused by approximation.
NORES	The algorithm can currently not be applied.

Table 6.2.: Summary of the different cases shown in Table 6.1

of case (iii) of Theorem 2.21. This means that our algorithm will return a final working set $\mathcal{W}_*$ and we know that: if $G \in \uparrow\mathcal{W}_*$, then G can cover a final graph within the entire set of graphs $\mathcal{S}$, and if $G \notin \uparrow\mathcal{W}_*$, then G can not cover a final graph while only reaching graphs of the chosen class $\mathcal{Q}$. This mixed decidability result is illustrated in Figure 6.22. In case G can cover a final graph within $\mathcal{S}$ but not within $\mathcal{Q}$, it is undefined whether G will be presented by the working set $\mathcal{W}_*$ computed by our algorithm or not. Nonetheless, the algorithm is applicable and guaranteed to terminate. Note that the minor ordering is a well-quasi-order on $\mathcal{G}_n$ and $\mathcal{D}_{n,k}$ since they are both subsets of $\mathcal{G}(\Lambda)$ (on which it is a well-quasi-order).

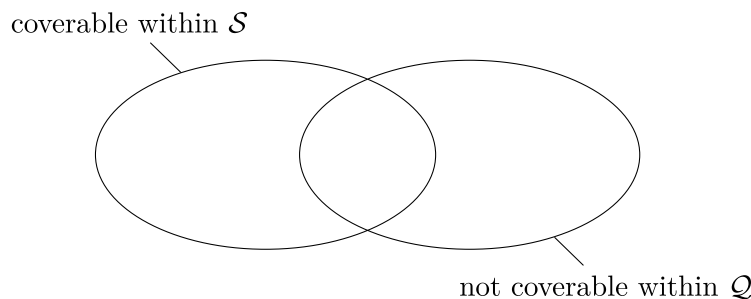


Figure 6.22.: Shows the split decidability result implied by case (iii) of Theorem 2.21

We can also use the minor ordering in conjunction with ordinary GTS or GTS with

negative application conditions, even though the compatibility condition is not satisfied. Applying our algorithm will implicitly add “lossiness” to the graph transformation system, so that it satisfies the compatibility condition. The resulting new system is an over-approximation of the original system and the decidability and split decidability results previously shown, apply to it. The final working set returned by the algorithm is exact for the new system and an over-approximation for the original system. For the latter systems this means that if $G \notin \uparrow\mathcal{W}_*$, then G can not cover a final graph within $\mathcal{Q}$ and if $G \in \uparrow\mathcal{W}_*$ then this may either be because G can cover a final graph or because of over-approximation. Which case applies can be checked by performing those steps forwardly that lead the algorithm to generate a minor of G. In general this argument also holds for GTS with NACs, but we could show in [KS12b] that some of these systems satisfy the compatibility condition wrt. the minor ordering and how our algorithm can handle these systems.

Subgraphs

An indirect consequence of the existence of our algorithm is that the coverability problem wrt. the subgraph ordering is decidable for lossy systems. This comes from the fact that a graph is coverable wrt. the subgraph ordering if and only if it is coverable wrt. the minor ordering. Obviously coverability wrt. subgraphs also implies coverability wrt. minors, since the minor ordering is coarser that the subgraph ordering. On the other hand, assume that a graph G is coverable wrt. the minor ordering, i.e. we can reach a G' with $G \sqsubseteq G'$. By definition, a lossy system contains contraction rules for every edge with an arity of at least two. By applying contraction rules to all edges of G' which are contracted to obtain G, we obtain a G'' with $G \subseteq G''$, i.e. G can also be covered wrt. the subgraph ordering. Note that we can ignore edges with arity zero or one, since a contraction of such edges is just a deletion. Thus, we inherit the decidability result of the minor ordering.

Similar to the minor ordering, if we restrict the class of graphs to $\mathcal{G}_n$ or $\mathcal{D}_{n,k}$ we obtain the split decidability result shown in Figure 6.22. Since the subgraph ordering is not as restrictive on the graph transformation systems as the minor ordering, these results are exact for lossy systems as well as ordinary GTS, but approximative for GTS with simple negative application conditions.

A special case occurs if we use the subgraph ordering in conjunction with ordinary GTS and the class of all graphs. We have shown that backward steps are computable and thus our algorithm is applicable. However, since the subgraph ordering is not a well-quasi-order on $\mathcal{G}(\Lambda)$, termination is not guaranteed. This means that we can use the algorithm “faithfully” and decide the coverability problem for every terminating instance. Remember that the algorithm does not need any initial graphs and will return a working set against which we can check any initial graph. However, if initial graphs are given, the algorithm can check whether they are represented by the working set and

terminate as soon as this is the case. With this premature termination we obtain a classical semi-decidability procedure for the coverability problem, but if the algorithm terminates without initial graphs given, we get a stronger result. If we use GTS with simple NACs instead, the results are similar, but will be an over-approximation, since the compatibility condition is no longer satisfied. The number of terminating instances can be increased by the use of graph patterns or attributed graphs, as I will investigate in the future work section (see Section 8.3).

Induced Subgraphs

For the induced subgraph ordering we can obtain the split decidability result for each of the mentioned classes of GTS if we restrict to $\mathcal{D}_{n,k}$. For all other cases our algorithm does not imply any results. The problem is that in the case of $\mathcal{G}(\Lambda)$ and $\mathcal{G}_n$ the predecessor set is not guaranteed to be finitely representable (contrary to the ordinary subgraph ordering). Our algorithm is therefore not applicable without further adaptation. Such an adaptation could for example be the use of patterns discussed in Section 8.3.

Chapter 7
Implementation and Case Studies

In this chapter we will show that the backward search presented in Chapter 6 is not just of theoretical interest, but can also be used in practice. We start by presenting the tool Uncover, which implements some of the combinations of order and match types for which we defined the algorithms in Chapter 6. This tool is specifically designed to be extendible with further orders, providing an appropriate class hierarchy. We then introduce a collection of case studies, including runtime results, which we were able to analyse using Uncover.

7.1. The Uncover Tool

Uncover is a command line tool which is written in `C++`, using the `C++11` standard, and licensed under GPLv2. It is designed to run as a background process on UNIX-like systems, e.g. a server. During the analysis the results – in addition to intermediate computations – are written to the hard drive. This allows convenient verification of case studies even if their runtimes are high. The following parts of our framework, introduced in Chapter 6, are currently implemented:

Minor ordering The minor ordering is implemented, but can only be used with conflict-free matches. Any graph transformation system can be analysed using this order. However, if the GTS is not well-structured wrt. the minor ordering (see Proposition 5.7), i.e. it does not contain the necessary edge contraction rules, then the final working set will be an over-approximation, since those rules are implicitly added to the GTS.

Subgraph ordering The subgraph ordering is implemented for the use with conflict-free as well as injective matches. For the latter matches the computation of

pushout complements is optimized by computing only injective co-matches (see Lemma 6.43). Furthermore, universally quantified rules are implemented for use with the subgraph ordering.

Induced subgraph ordering The induced subgraph ordering is not yet implemented. However, by design UNCOVER can be easily extended. In fact, we mainly need to implement the computation of minimal pushout complements, i.e. Algorithm 6.39. The check whether a given graph is an induced subgraph of another given graph is very similar to the check for subgraphs.

Optimizations UNCOVER implements several small optimizations. For every order, rules are removed that are also order morphisms (see Proposition 6.44). When adding new orders, only the check if a given morphism is an order morphism must be implemented to benefit from this optimization. Additionally, the main backward search (Algorithm 6.6) was optimized to reduce runtime and memory consumption. These optimizations include computing predecessors only for the most recent graphs, minimizing the working set as soon as possible and immediately dropping graphs which were subsumed by other graphs.

We will first show the usage and then have a brief look at the architecture of UNCOVER, where our special interest lies in extensibility. The tool itself provides a brief description of its usage which can be printed by the call `uncover -h` (see a cropped version below).

```
uncover -h

UnCoVer (Using Coverability for Verification):
---------------------------------------------
This program executes a predefined scenario. The scenario
parameters can be either given directly on the console
(--scn) or by giving a configuration file
(--scenario-file). Only one of these options may be used.
See the option list below for listing available scenarios
or their description and further configuration.

Possible options:
...
```

Usage

UNCOVER implements different functionality by defining several scenarios. A list of all available scenarios can be printed by the call `uncover -l`, as shown below.

```
uncover -l

List of available scenarios:
----------------------------
ID 100; Names: backward_analysis, backw
ID 200; Names: leq_check, leq
ID 300; Names: le_rule_creator, lerc
ID 1000; Names: gxl_pic_converter, gxl2pic
ID 1100; Names: gtxl_latex_converter, gtxl2latex
ID 10000; Names: test_xml
ID 11000; Names: test_rule_preparer, testprep
ID 12000; Names: test_backward_step, testbws
ID 13000; Names: test_matcher
```

Each scenario has a different list of parameters and may even require different external tools to run. A description for each scenario can be printed by calling `uncover -u arg`, where `arg` can be one of the synonyms or the ID. Our main backward search is implemented in `backward_analysis`. Its mandatory and optional parameters will be introduced later in this section. The scenario `leq_check` checks which graphs of a given set are in the upward closure of another given set of graphs. Given the working set computed by the backward search, this scenario will later allow us to check whether graphs are represented by the final working set, i.e. whether they are coverable. The scenario `le_rule_creator` is used to generate the graph transformation system for the leader election case study (see Section 7.3). For drawing graphs we can use the scenario `gxl_pic_converter` and for drawing graph transformation systems we can use `gtxl_latex_converter`. Both scenarios support different output formats and rely on Graphviz [Graphviz], an open source software for graph visualization, to layout the graphs. Furthermore, they use LaTeX [Latex] to layout rules and sets of graphs. The scenarios beginning with `test_` are all used to test the functionality of different parts of the implementation. In addition to scenario specific parameters, Uncover also provides general parameters such as `-o` to set the level of log messages.

To start a scenario we need to call `uncover --scn=name ...`, where `name` is the scenario name (or a synonym) and is followed by a list of parameters. The parameters may be provided as a list of key-value pairs or omitting the keys if the parameters are in the correct order (these notations may also be mixed). We illustrate this by introducing the usage of the (main) scenario `backward_analysis`. A possible invocation of this scenario can be seen in Figure 7.1. Without providing keys the first three parameters are interpreted (in order) as a GTXL-file containing the graph transformation system, a GXL-file containing the minimal (initial) error graphs and the order to be used. For storing graphs and graph transformation systems we use the Graph eXchange Language (GXL) [GXL] and Graph Transformation eXchange Language (GTXL) respectively. Both formats are

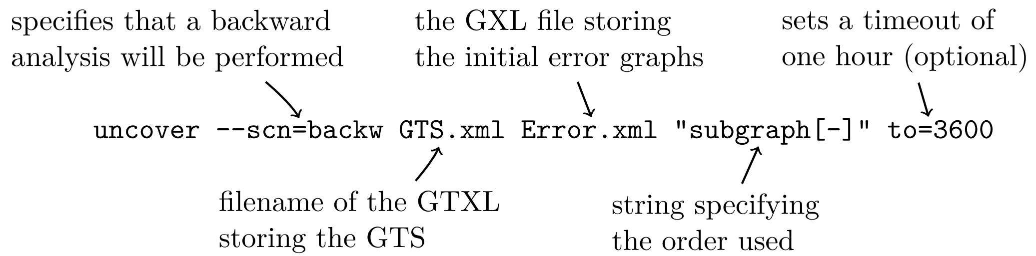


Figure 7.1.: Shows an exemplary use of the UNCOVER tool

XML-based and definition files are available with the source code [Uncover]. Note that the given GTS need not contain initial graphs, but such graphs can be provided to prematurely terminate the analysis as soon as one is represented by the working set. The third parameter may either be `minor` to specify the minor ordering or `subgraph[x]` to specify the subgraph ordering, where `x` may be a natural number setting the path bound or `-` for no bound. A detailed description of all parameters of a scenario `s` can be printed with the call `uncover -u s`. The description of the backward analysis scenario is shown below.

```
uncover -u backw
```

```
Scenario description of 'backward_analysis':
--------------------------------------------
This scenario performs a backward search algorithm. It
takes a graph transformation system and a set of error
graphs as input and computes the set of graphs, from
which the given graphs are coverable. Use the parameters
to specify the order and result storage.
Synonyms: backw

Parameter descriptions (in order):
----------------------------------
(1) gts (required):
An XML file (in GTXL) containing the GTS which will be
analyzed.

(2) error-graph, err (required):
A GXL file containing the set of error graph which should
not be coverable.

(3) order, or (required):
```

```
Specifies which wqo should be used. The given GTS has to
be well-structured with respect to the given order.
Possibilities: minor, subgraph[?] (? can be a natural
number or '-')

(4) result-folder, res:
A folder where all intermediate results will be stored.

(5) timeout, to:
If the scenario takes longer than the given timeout (in
seconds), it will be terminated (soft termination, i.e.
may run longer).

(6) check-initial, ci:
If set to true, after every backward step it will be
checked whether one of the initial graphs is represented
by the calculated graph set and the analysis will stop if
the check succeeds. Default is 'false'.

(7) matching, m:
Defines the type of matchings used. Available are:
'conflictfree' (default, alias: 'cf') and 'injective'
(alias: 'inj')
```

There are four optional parameters. If a `result-folder` is given, the scenario will store intermediate results in that folder. These results consist of graphs computed by backward steps together with the information which rule was applied to which graph to obtain them. Furthermore, it stores which graphs were removed by minimization and how the rule set changed due to preparation. For premature termination a `timeout` or initial graphs can be given. The `timeout` is enforced softly, i.e. if it is exceeded, the scenario will finish the current computations before terminating. If the `check-initial` parameter is set to `true`, the analysis will terminate as soon as one of the initial graphs – given in the GTS – is represented by the working set. In both cases, the working set computed up to that point will be stored as result. Finally, the `matching` parameter can be used to specify whether conflict-free or injective matches should be used. Note that parameters may be restricted depending on the order used and whether the GTS contains universally quantified rules.

Workflow

A brief overview over the workflow of UNCOVER is given in Figure 7.2. The GTXL and GXL files are read by the `GTXLReader` to create a `GTS` object and a collection of graphs. The string representing the order is read and instances of the `Order`, `RulePreparer` and `MinPOCEnumerator` classes are generated for the given order. These classes handle all order-specific operations, i.e. checking whether a graph is smaller than another graph (`Order`), composing rules and order morphisms (`RulePreparer`) as well as computing minimal pushout complements (`MinPOCEnumerator`). All parsed and additional optional parameters are then forwarded to the analysis engine. It implements the main backward search (Algorithm 6.6) with some slight optimizations to decrease runtime as well as memory consumption and calls other classes for preparation and minimization. After termination of the backward search the final graph set is written to a GXL file by the `GTXLWriter` class. Important information, i.e. about performed operations and progress, will be displayed as log messages. Optionally – controlled by the optional parameters – additional files will be created. This can include a prepared version of the given GTS, all graphs computed by backward steps, information about which graph was generated from which other graphs by which rule, and which graph was removed because it was subsumed by which other graph.

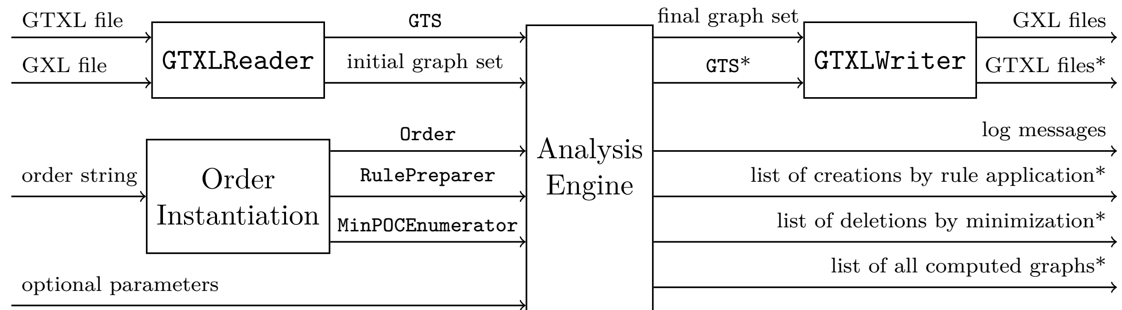


Figure 7.2.: Internal workflow of a backward analysis in UNCOVER; outputs marked with a * are optional

Improvements and Other Orders

There are several ways of improving UNCOVER, which either arise from optimizations of the underlying framework or the implementation itself. In the following we briefly address some of these improvements.

- Possible optimizations of the underlying framework were already introduced in Section 6.5. Some of these optimizations are already implemented, e.g. the removal

of rules which are also order morphisms. Especially optimizations reducing the number of rules have shown to cause great performance boosts.

- Intelligent search strategies can be used for minimizing duplicate computations as far as possible. Already changing the order on graphs for which backward steps are computed can result in a different performance.
- There is currently no tool support for modelling graphs and graph transformation systems. By allowing more input and output formats, existing graph tools could be used for modelling.
- Due to the large quantity of data, searching the optional output files, e.g. for determining how a final graph could be reached, is time-consuming. An appropriate tool support capable of handling large data is necessary.
- More orders (see Section 5.4) would strengthen the framework if one can prove the necessary conditions (see Chapters 5 and 6). UNCOVER is designed in such a way that this only requires the implementation of the classes `Order`, `RulePreparer` and `MinPOCEnumerator` for the new order. While the first two should be rather easy to implement, the `MinPOCEnumerator` class needs to compute all minimal pushout complements. However, for this UNCOVER provides classes for computing pushout complements for total rules as well as partial rules, which are also used in the implementations of `MinPOCEnumerator` for the minor and subgraph orderings.
- UNCOVER is currently single-threaded. However, backward steps for different graphs or rules can be computed in parallel quite naturally. With a good parallelisation strategy this can be used to improve performance.

Dependencies

To compile UNCOVER requires the Boost framework [Boost] in version 1.54 or higher and the XercesC++ library [XercesC++] in version 3.1 or higher. More precisely, the following Boost libraries are required:

- `boost_system`,
- `boost_filesystem`,
- `boost_program_options`,
- `boost_regex` and
- `boost_unit_test_framework`.

Note that for compilation the standard and development packages of the above libraries are necessary. The source code archive contains a CMake script (requiring CMake 2.8 or higher [CMake]) for generating the makefiles, which is capable of compiling with

GCC [GCC] and Clang [Clang]. The source code should be compilable with newer versions of the above libraries and may be compilable with older versions, but this was not tested.

Uncover is written in `C++` without using any OS specific functions and is therefore platform independent. However, due so some compatibility issues with the libraries, it does currently not compile on Windows. It is verified to compile on Linux (tested on Ubuntu 14.04, Fedora 21) and Mac OS X with MacPorts (tested on Yosemite).

To run the main backward search no other tools are required. The scenarios for drawing graphs and graph transformation systems require Graphviz [Graphviz] and LaTeX[Latex].

Similar Tools

Not many tools for verifying graph transformation systems exist. For finite state systems the Groove tool [Groove; Ren03; GM+12] exists, which supports rules with complex application conditions. It implements a controllable state space exploration and enables model checking via CTL and LTL formulas. For infinite state systems the Augur2 [Augur2; KK08] and GBT [GBT; SWJ08] tools exist. Since most problems are undecidable in the infinite case, these tools use approximations to obtain partial decidability. Augur2 uses attributed Petri nets to over-approximate attributed graph transformation systems. By using algorithms for Petri nets such as unfolding techniques and coverability checks, properties of the original GTS can be proved using the Petri net. If the abstraction is to coarse, counterexample-guided abstraction refinement can be used [KK06]. An approach similar to the one implemented in Uncover is used by GBT. There a backwards analysis is performed using graph patterns to abstractly represent sets of graphs. These graph patterns consist of a positive part which must occur as a subgraph in every represented graph and a negative part which must not occur. It is an interesting question whether this forms a well-quasi-order with some interesting class of graphs and could be integrated into our framework. In [AB+08] a prototype implementation is mentioned – including runtime results – which can analyse directed graphs with an out-degree of at most one using the minor ordering. However, the prototype is neither named nor published. Another prototype is Symgraph [Symgraph; ADR09]. There graphs are represented symbolically by so-called graph constraints and analysed by an approximative backward search.

There is also some tool support for related formalisms. For depth-bounded systems the Picasso tool [Picasso; ZWH12; BK+13] exists, which implements a forward search. It computes so-called abstract coverability trees and uses techniques such as ideal completion and widening to ensure termination. Although the results are only approximative, the achieved precision is quite good. For Petri nets the Mist2 tool [Mist2] is available. It implements the backward search as well as the EEC algorithm [GRV06] and supports transfer arcs. The tool Petruchio [Petruchio; SM12] allows a backward analysis of

Petri nets and lossy channel systems. It implements partial order reduction, pruning and backwards acceleration.

Runtimes

So far we used both the subgraph ordering and the coarser minor ordering to verify different case studies with UNCOVER. All these case studies can be downloaded from the tools main website [Uncover]. An overview of runtime results for these case studies is given in Table 7.1. These runtimes where computed on an Intel® Xeon® CPU E5-2637 v2 with 64 GB RAM using only one core (parallelisation is not yet implemented). In the following Sections 7.2 to 7.6 we will present all these case studies and give more detailed runtime results.

case study	wqo	class $\mathcal{Q}$	runtime	#EG
Termination detection (faulty)	minor	all graphs	796ms	69
Termination detection (correct)	minor	all graphs	304ms	101
Leader election ($id \leq 10$)	minor	all graphs	1m 2s	451
Leader election ($id \leq 20$)	minor	all graphs	27m 56s	2401
Rights management	subgraph	all graphs	28ms	4
Dining Philosophers	subgraph	all graphs	442ms	12
Public-private server	subgraph	path ≤ 50	14.1s	104
Public-private server	subgraph	path ≤ 100	3m 28s	204

Table 7.1.: Runtime results from UNCOVER for different case studies

7.2. Termination Detection

In the termination detection case study, we model a termination detection protocol that works on a ring structure and assumes lossiness, i.e. messages can be lost and processes can leave the ring. We first presented this modelling in [BD+12b; BD+12a] and later introduced a modified version using negative application conditions in [KS12b; KS12a]. Initially the network is a ring consisting of multiple active processes (A) and (exactly) one active detector (DA), as shown in Figure 7.3a. Independent of each other, processes and detectors can be active, indicated by the labels A and DA, or passive, indicated by the labels P and DP. The purpose of the detector process is to check whether all processes are passive and generate a termination flag (seen in Figure 7.3b) if this is the case. Thus, we can check whether the detector process works correctly by checking if a graph containing a termination flag and an active process is reachable. This can be done by checking whether the graph in Figure 7.3b is coverable.

(a) Initial ring (b) Error graph

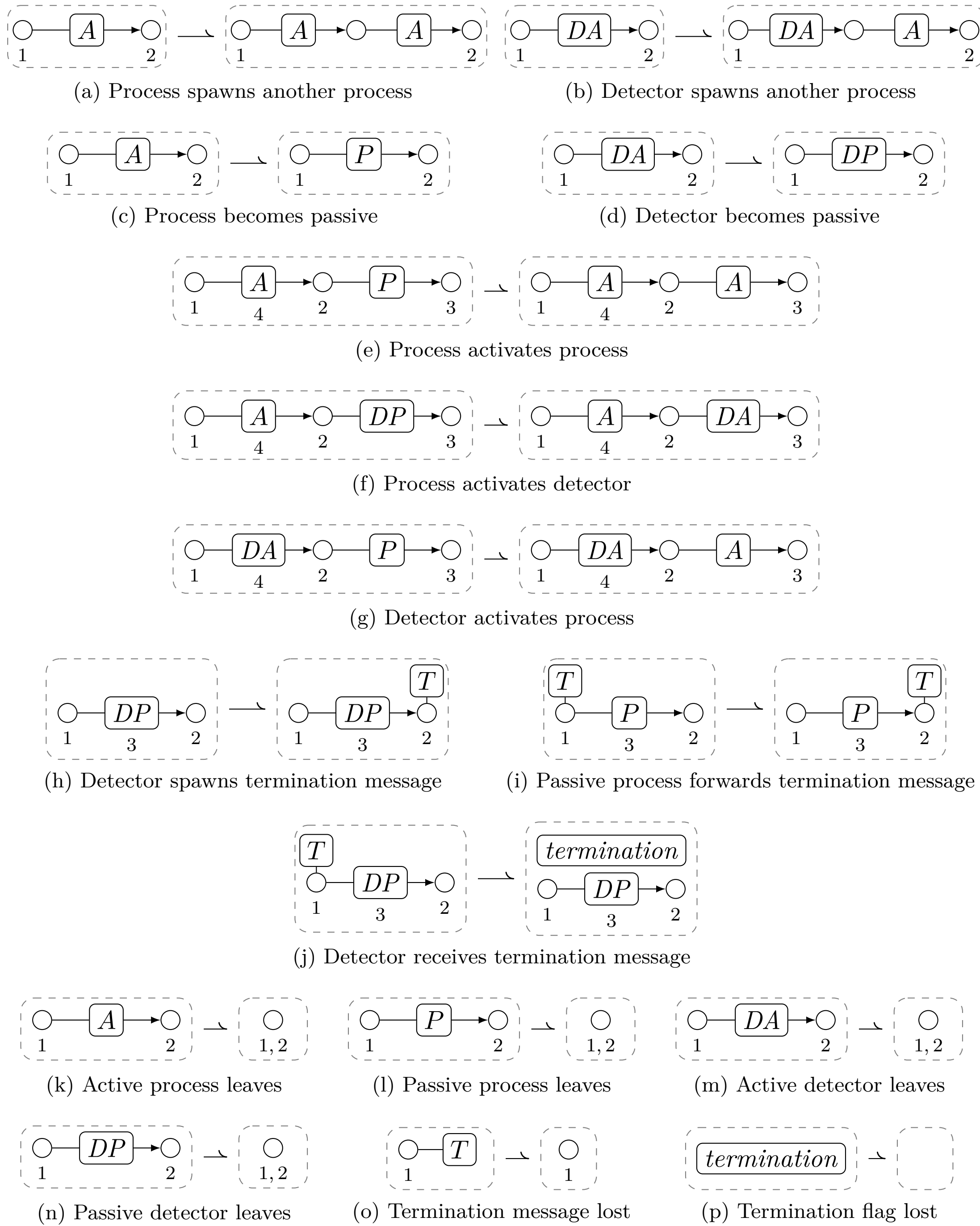
Figure 7.3.: Graphs describing the initial and erroneous configurations of the termination detection protocol

The protocol works as follows. Beginning with the initial graph (Figure 7.3a), multiple new processes (not detectors) can be spawned by an active process (Figure 7.4a) or active detector (Figure 7.4b). Note that this – as well as all other rules – changes the size of the ring, but always preserves its structure. Active processes and detectors can become passive voluntarily (Figures 7.4c and 7.4d) and may be reactivated by other processes or detectors (Figures 7.4e to 7.4g). Note that there is no rule having two detectors, since there is at most one detector in the ring. At some point the (passive) detector decides to test whether all processes have become passive. It does this by generating a termination message (Figure 7.4h) that is forwarded along the ring (Figure 7.4i) until it reaches the detector again (Figure 7.4j) at which point the termination flag is created. Since the termination message passes only passive processes, all processes of the ring are assumed to be passive as soon as the message reaches the detector again. However, as we will see, there is an error in this argument. Finally, the rules in Figures 7.4k to 7.4p cause the system to be lossy. These rules also cause the GTS to be well-structured wrt. the minor ordering, i.e. Proposition 5.7 is satisfied.

case study	wqo	class $\mathcal{Q}$	runtime	#EG
Termination detection (faulty)	minor	all graphs	796ms	69
Termination detection (correct)	minor	all graphs	304ms	101

Table 7.2.: Runtime results from Uncover for the termination detection case study

We can therefore use our backward search (with the minor ordering) to check whether the error graph is coverable. It computes a total of 69 graphs (see Table 7.2) of which the majority represents only invalid configurations, e.g. graphs which are not rings. The most interesting graph is shown in Figure 7.5 and consists of only a single *DA*-labelled loop. This graph represents all rings containing at least one detector, including the initial graph (Figure 7.3a). Thus, the protocol is erroneous! The error lies in the fact that passive processes can be reactivated as soon as they have passed the termination message. Hence, at the time the termination message reaches the detector again, it is not guaranteed that the processes have not been reactivated. And in fact, if we remove the rules in Figures 7.4e to 7.4g, we obtain a correct version of the protocol. For that



(a) Process spawns another process
(b) Detector spawns another process
(c) Process becomes passive
(d) Detector becomes passive
(e) Process activates process
(f) Process activates detector
(g) Detector activates process
(h) Detector spawns termination message
(i) Passive process forwards termination message
(j) Detector receives termination message
(k) Active process leaves
(l) Passive process leaves
(m) Active detector leaves
(n) Passive detector leaves
(o) Termination message lost
(p) Termination flag lost

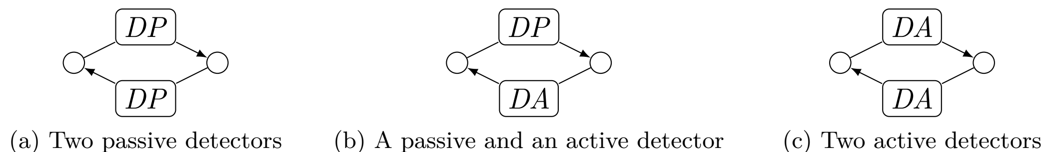
Figure 7.4.: Rule set modelling a (faulty) termination detection protocol

Figure 7.5.: One of the errors in the final working set computed by the backward search

version our analysis produces 101 graphs (see Table 7.2) of which none is isomorphic to the one in Figure 7.5. All 101 graphs represent erroneous configurations, either they are no rings or contain multiple detectors. Since none represents the initial graph, we know that the second version of the protocol is correct.

Bonus Analysis

Using the backward search we can also prove invariants to some extent. For instance, if we use the graph in Figure 7.6a to start the backward search, the final working set will contain it and the two graphs in Figures 7.6b and 7.6c. From this we know that we cannot reach rings with two detectors from rings with one detector, since otherwise the working set would represent such graphs. We therefore know that the number of detectors is invariant wrt. the rules. In general we can use such checks to prove that the modelling can only reach valid configurations.



(a) Two passive detectors (b) A passive and an active detector (c) Two active detectors

Figure 7.6.: Possible representatives of rings with two detectors

7.3. Leader Election

In the leader election case study we model a leader election protocol which works on a ring and assumes lossiness, just like the termination detection protocol. It was first used in [JK08; JK12] where the authors first proposed the idea of using the minor ordering for verification, ultimately leading to the framework presented in this paper. The protocol uses unique identifiers for each process (see Figure 7.7a) and aims to elect one as leader (shown by an L-labelled edge).

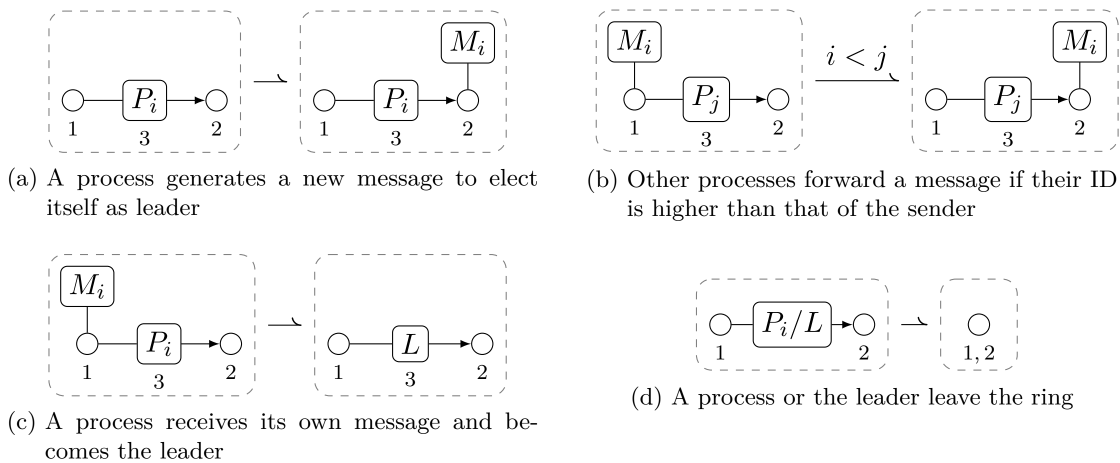
The protocol works as follows. Every process can generate a message (Figure 7.8a) to acquire leadership. These messages are passed along the ring (Figure 7.8b), but only if the passing process has a higher ID, i.e. lower priority, than the process who generated the message. If a message reaches the process who generated it (Figure 7.8c), the process is elected as leader. Processes may also decide to leave the ring (Figure 7.8d). Note that

(a) Initial ring structure

(b) Error graph

Figure 7.7.: Initial and erroneous configuration of the leader election protocol

due to these last rules the system is lossy and therefore well-structured wrt. the minor ordering (it satisfied Proposition 5.7).



(a) A process generates a new message to elect itself as leader

(b) Other processes forward a message if their ID is higher than that of the sender

(c) A process receives its own message and becomes the leader

(d) A process or the leader leave the ring

Figure 7.8.: Modelling of a leader election protocol by graph transformation rules

An error in this protocol would be the election of two leaders. Wrt. the minor ordering, every ring with two leaders is represented by the minimal graph in Figure 7.7b. Unfortunately, our formalism requires that the set of labels is finite and can thus not handle an arbitrary number of unique identifiers. We therefore need to fix the number of IDs – which leads to a graph transformation system in our classical sense – before we can use our backward search to check whether the minimal error graph is coverable. Note that this graph transformation system still induces an infinite transition system. Our analysis will then generate the final working set as shown in Figure 7.9, proving that the protocol is correct. Note that the description of those error graphs is generic, i.e. Figure 7.9 shows which set will be generated for an arbitrary high number of IDs. Our analysis is currently not capable of generating this set directly – it is only possible for fixed IDs – but runs with different numbers of IDs imply the set. To directly generate this set, our framework would need to be extended with attributed graphs. In principle

this is possible, but it significantly complicates the calculations, since it requires computing weakest preconditions of formulas on the attributes. How powerful the logic needs to be depends on the modelled system, here atoms such as $i < j$ would be sufficient.

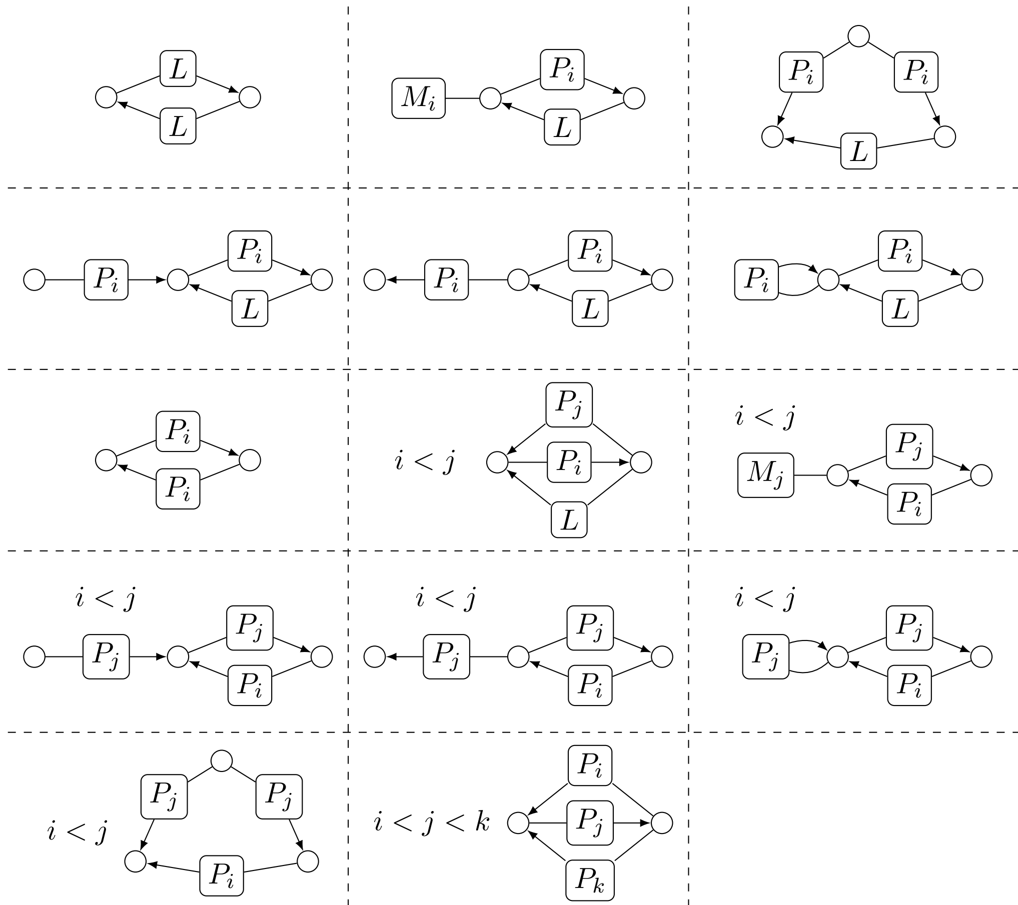
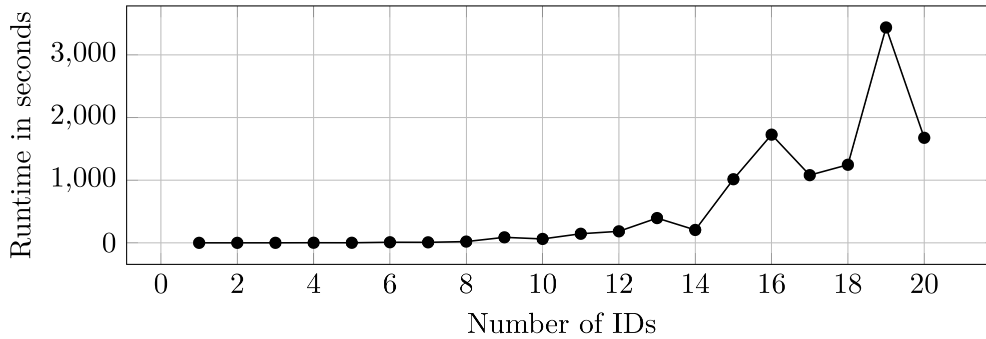


Figure 7.9.: Final working set computed by the backward search for the leader election protocol

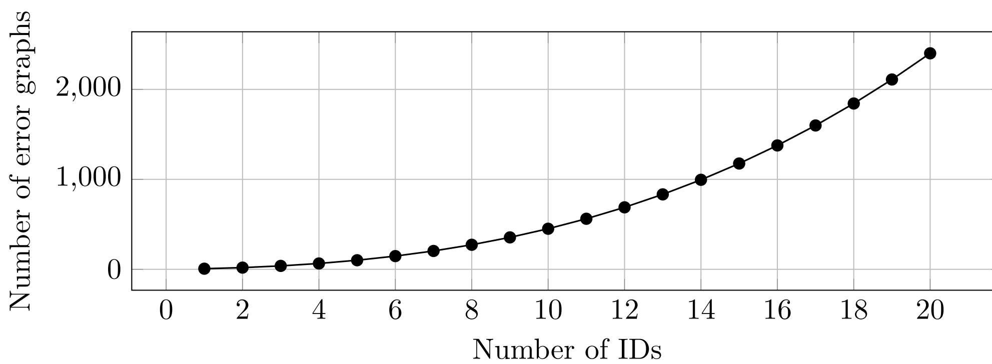
Runtime results of Uncover as well as the number of computed graphs in the final working set are shown in Figure 7.10. Although the number of rules is strictly increasing with the number of IDs, the runtime is not.

7.4. Access Rights Management

In our next case study we model a simple multi-user system as the GTS shown in Figure 7.11 inspired by [KMP02]. A graph contains user nodes, indicated by unary U-



(a) The runtime of UNCOVER depending on the number of IDs

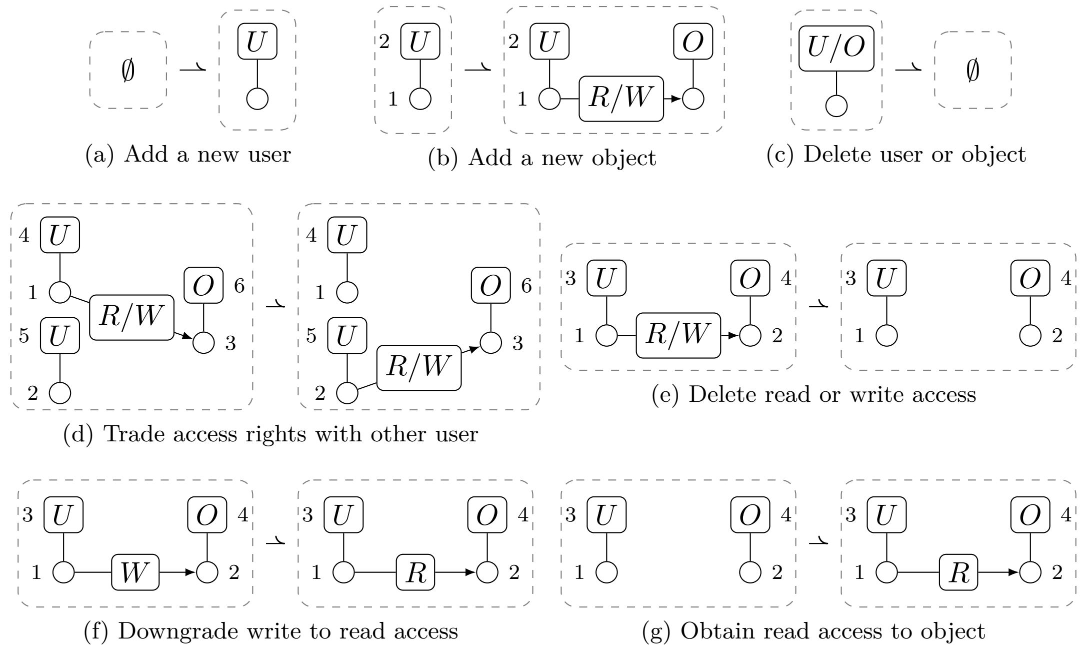


(b) Number of error graphs computed depending on the number of IDs

Figure 7.10.: Runtimes and number of error graphs for the leader election case study

labelled edges, and object nodes, indicated by unary O-labelled edges. Users can have read (R) or write (W) access rights regarding objects indicated by a (directed) edge. The system starts with no users or objects at all, i.e. the empty graph, and provides several rules to manipulate users, objects and access rights. New users can be added (Figure 7.11a), new objects can be added with read or write access associated with a user (Figure 7.11b) and users as well as object may be deleted again (Figure 7.11c). Both read and write access can be traded between users (Figure 7.11d) or dropped (Figure 7.11e). Additionally users can downgrade their write access to a read access (Figure 7.11f) and obtain read access to arbitrary objects (Figure 7.11g). Note that rules containing labels of the form R/W represent two rules, one with R-labelled edges and one with W-labelled edges.

The modelled multi-user system can generate arbitrarily many users and object, as well as arbitrarily many access rights to the same object. However, the system is only correct if there is at most one write access to each object, requiring some form of mutual exclusion. This means especially that any configuration of the system containing the



(a) Add a new user (b) Add a new object (c) Delete user or object

(d) Trade access rights with other user (e) Delete read or write access

(f) Downgrade write to read access (g) Obtain read access to object

Figure 7.11.: A GTS modelling a multi-user system

graph depicted in Figure 7.12 is considered erroneous. We can therefore check the correctness of the system by using that graph as initial working set of our backward search. Unfortunately the given system is not well-structured wrt. the minor ordering. In fact, adding contraction rules results in a too coarse over-approximation, since the contraction of read or write edges causes a node to become a user and object at the same time, i.e. the result is an invalid configuration. However, we can use the subgraph ordering instead. Although not guaranteed in general, UNCOVER terminates for this case study even without a path bound and returns four minimal graphs, consisting of the initial error graph and those shown in Figure 7.13. The two rightmost graphs represent non-valid configurations of the system and are not reachable from the initial graph. Since the empty graph is not represented, the modelled system is correct.

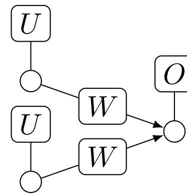


Figure 7.12.: An undesired state in the multi-user system

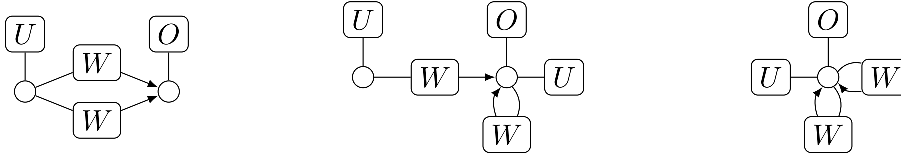
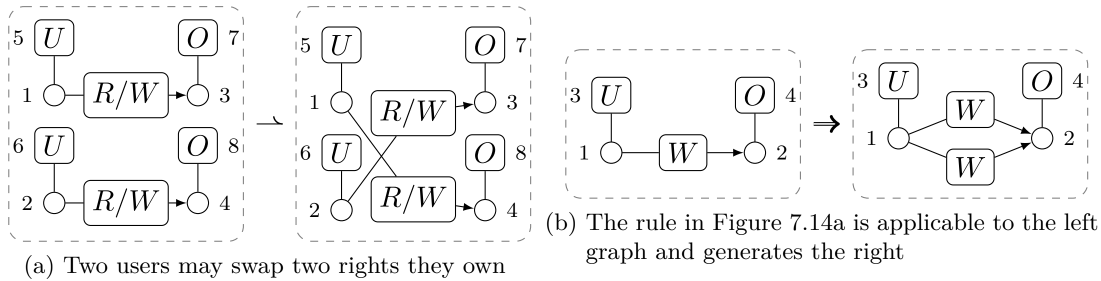


Figure 7.13.: Additional error graphs of the rights management protocol, computed by the backward search

Problems with swapping

Now we extend the rights management system with the rule in Figure 7.14a, which allows users to swap rights they own. If we now perform the analysis, UNCOVER will return just a single graph, the empty graph. This means that from the initial graph we can reach an erroneous graph. By having a look at the computed graphs we can see that the swap rule can be applied to the left graph in Figure 7.14b to get the right graph (which is one of the graphs in Figure 7.13).



(a) Two users may swap two rights they own

(b) The rule in Figure 7.14a is applicable to the left graph and generates the right

Figure 7.14.: Shows the effects of adding a swap rule to the rights management protocol

The problem is not caused by an error in the rights management system, but in the modelling itself, since it no longer appropriately reflects the system. Since the problematic match is caused by non-injectivity, we can prevent it by restricting UNCOVER to injective matches. Note that in general this will also affect the other rules, but in this case study the other rules behave the same regardless of match types. For injective matches UNCOVER will compute just two graphs, the original error and the left most graph in Figure 7.13. The runtime of all three cases – non-injective matches with and without the swap rule and injective matches with the swap rule – is shown in Table 7.3

case study	wqo	class $\mathcal{Q}$	runtime	#EG
Rights management (without swap)	subgraph	all graphs	28ms	4
Rights management (with swap)	subgraph	all graphs	95ms	1
Rights management (inj., with swap)	subgraph	all graphs	19ms	2

Table 7.3.: Runtime results from UNCOVER for different versions of the access rights management case study

7.5. Dining Philosophers

In another case study we modelled the Dining Philosophers Problem on an arbitrary graph structure. Each node represents a philosopher and has an incident unary edge specifying his state. A philosopher is either thinking (T), hungry (H) or eating (E). Between two philosophers there can be an unowned fork, indicated by an F-labelled edge, or a fork owned by one of the philosophers, indicated by an OF-labelled edge pointing to the owner. To eat a philosopher needs to own all incident forks. To model this by a rule, we need to use the universally quantified rules introduced in Section 6.6. This modelling was first published in [DS14b; DS14a].

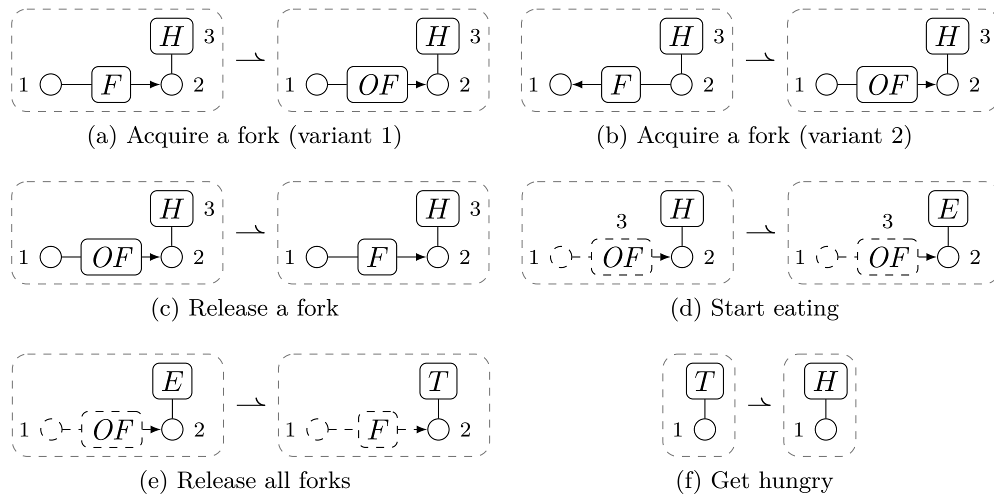


Figure 7.15.: Modelling of the Dining Philosophers Problem on an arbitrary net

The system works as follows. Hungry philosophers can take unowned forks (Figures 7.15a and 7.15b) and also release control of a previously taken fork (Figure 7.15c). If a philosopher owns all incident forks, he can start eating (Figure 7.15d). The highlighted part of the rule indicates a universal quantification, meaning that the rule can only be applied if all edges incident to the philosopher are part of the matching and in

fact forks owned by him. At some point the philosopher finishes eating and releases all forks (Figure 7.15e), going into his thinking state. When releasing all forks, all forks owned by the philosopher are converted to unowned forks. At a later point a philosopher can become hungry again (Figure 7.15f), restarting the process of acquiring forks to eat. Note that the rules in Figures 7.15d and 7.15e can not be described without universal quantifications, since we need to ensure that every incident fork is owned and later freed.

Figure 7.16.: Two minimal error graphs for the Dining Philosophers Problem

We now want to prove that two adjacent philosophers can never be eating at the same time. We can do this by checking whether one of the graphs in Figure 7.16 is coverable wrt. the subgraph ordering. Together these graphs represent all other graphs containing two eating philosopher which share a fork. UNCOVER will take 442ms to compute the 12 minimal graphs shown in Figure 7.17. Note that on the one hand the use of universally quantified rules causes the set to be an over-approximation and on the other hand we lose the guarantee of termination due to the subgraph ordering. However, in this case UNCOVER terminates and the approximation is precise enough. An initial configuration of the Dining Philosopher Problem consists only of hungry or thinking philosophers with unowned forks between them. Since every graph of Figure 7.17 contains at least one eating philosopher, none represents an initial configuration. Thus, two adjacent philosophers can never be eating at the same time.

7.6. Public-Private Server Communication

The last case study presented in this thesis describes a public-private server communication system. Every node is either a public server, marked with a unary *Spub*-labelled edge, or a private server, marked with a unary *Sprv*-labelled edge. These servers can run external (*Pext*) and internal processes (*Pint*). Connections (*C*-labelled edges) can exist between servers and processes can be transferred along these connections. Our modelling is inspired by a very similar case study in [Koz10].

Initially the system consists of a private server and a generator (see Figure 7.18a). The generator is used to control the generation of public servers (Figure 7.19j) and will at some point be removed to stop generation (Figure 7.19f). Private servers can spawn internal processes (Figure 7.19a) and public servers can spawn external processes (Figure 7.19b), which should only be processed by public servers. Arbitrary connections can be introduced between private servers (Figure 7.19c), between public servers (Figure 7.19d) and from private to public servers (Figure 7.19e). Internal and external

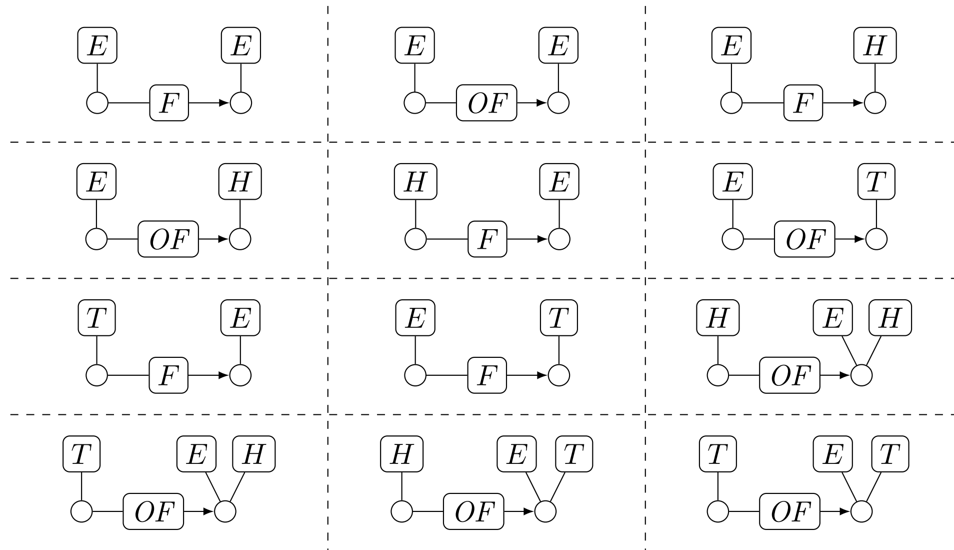
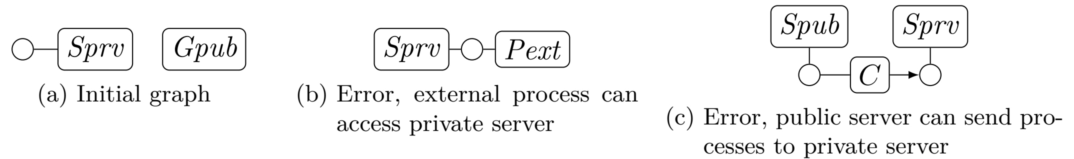


Figure 7.17.: Final working set computed by the backward search for the Dining Philosophers Problem



(a) Initial graph

(b) Error, external process can access private server

(c) Error, public server can send processes to private server

Figure 7.18.: Initial and error graphs of the public-private server communication

processes can be transferred along these connections (Figures 7.19g and 7.19h). Finally, private servers can also become public servers (Figure 7.19i).

An error occurs in this system whenever private data can be accessed by public servers or external processes. Thus, a graph is erroneous if an external process is incident to a private server (Figure 7.18b) or a public server has a connection to a private server (Figure 7.18c). If we call UNCOVER with these two minimal error graphs and use the subgraph ordering, the set of graphs shown in Figure 7.20 will be returned. However, contrary to the previous case studies, we have to fix the path bound in order to terminate. For any path length UNCOVER will compute the two fixed graphs in the left of Figure 7.20 and will compute paths of the form on the right up to the bounded path length. The runtime and number of graphs computed, depending on the path length, is shown in Figure 7.21. Note that the number of graphs computed for a path length n is always $2n + 4$, i.e. linear, as already indicated by Figure 7.20. The runtime, on the other hand,

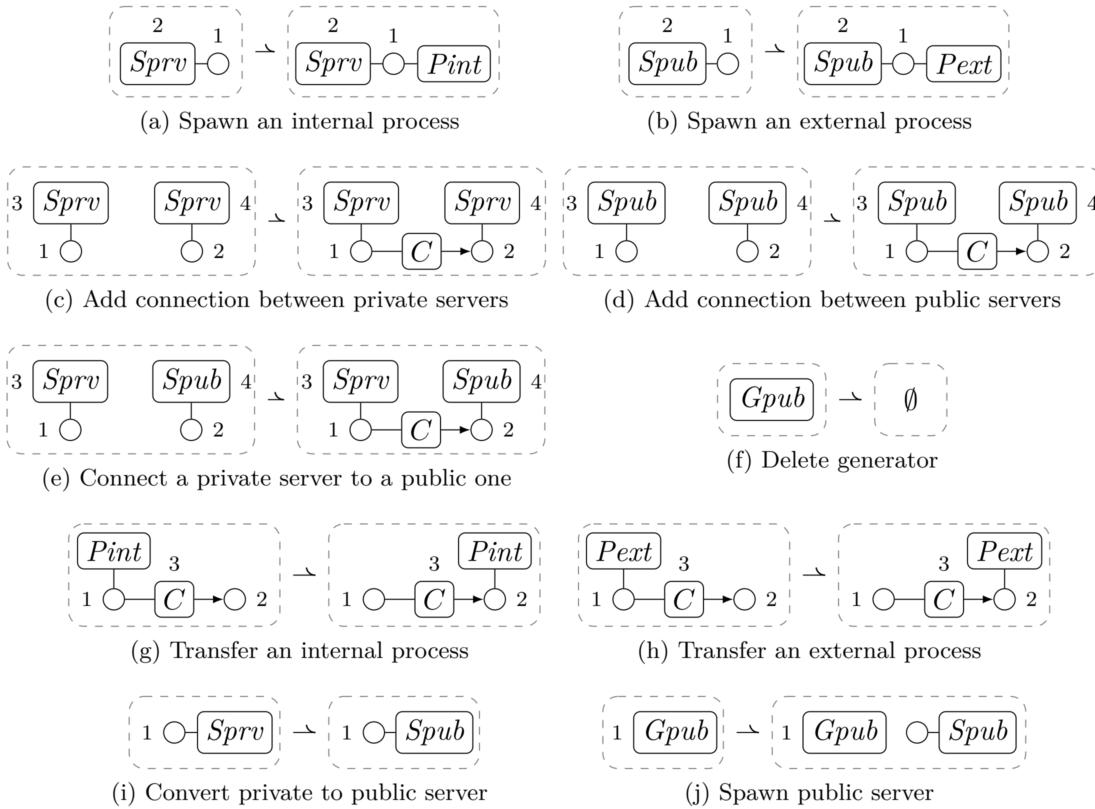


Figure 7.19.: Modelling of public-private server communication

is not linear, since the number of matches increases with increasing path lengths.

Since the initial graph is not represented by any of these graphs, no error can be reached in the modelled system. However, we can only prove this up to the set path bound, i.e. there might be a graph with a longer path that is both reachable from the initial graphs and can reach an erroneous graph. This graph would not be found by the analysis, since it (or a predecessor) would be dropped due to exceeding the path bound. To achieve termination without a path bound we would need to introduce graph patterns into our framework. In this case these patterns need to be able to finitely represent the paths shown in Figure 7.20. However, it also requires acceleration or widening techniques to reach these patterns by a backward step.

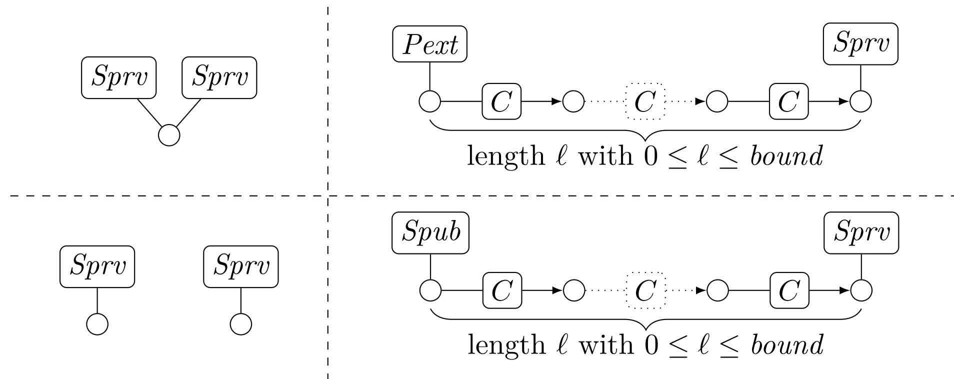
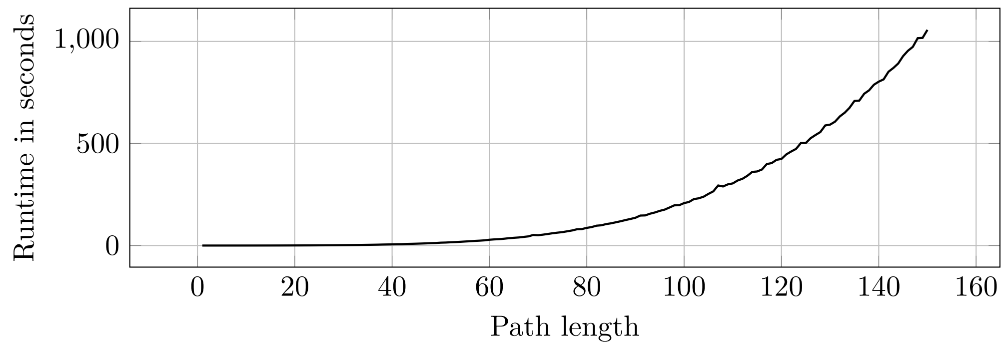
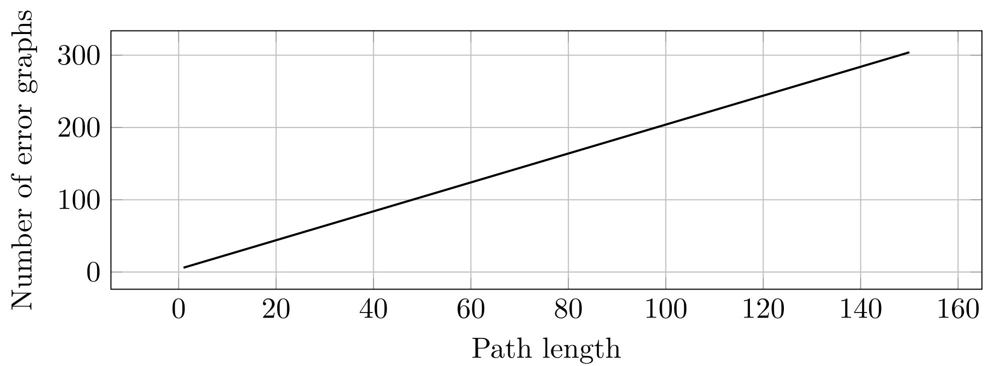


Figure 7.20.: Final working set computed by UNCOVER for the public-private server communication case study



(a) The runtime of UNCOVER depending on the bounded path length



(b) Number of error graphs computed depending on the bounded path length

Figure 7.21.: Runtimes and number of error graphs for the public-private server communication case study

Chapter 8
Conclusion

8.1. Summary

In this thesis we have presented a framework for solving the coverability problem for graph transformation systems. It is based on the theory of well-structured transition systems introduced in [AČ+96; FS01]. We require a well-quasi-order and a transition relation which is a simulation relation wrt. the order used. This can be seen as a monotonicity criteria, since we require that larger graphs (i.e. states) can mimic transitions of all smaller graphs. Of course, for coarser orderings this is a stronger condition than for finer orders. However, coarser orders are more likely to be well-quasi-orders on a larger class of graphs. This trade-off is shown for our three main orderings in Table 8.1.

ordering	wqo on	well-structured on
minor ordering	all graphs	lossy systems
subgraph ordering	bounded paths	GTS without NACs
ind. subgraph ordering	bounded paths and multip.	GTS with restricted NACs

Table 8.1.: Different orders result in a trade-off between the class of graphs and GTS

The minor ordering is a well-quasi-order on all graphs, but satisfies the compatibility condition only for a restricted class of GTS, mainly lossy systems, but also context-free GTS. On the other hand, the subgraph ordering satisfies the compatibility condition for all GTS without negative application conditions, but is no well-quasi-order on all graphs. However, it is a well-quasi-order on the class of graphs where the longest path is bounded. For the induced subgraph ordering we additionally have to restrict the edge multiplicity (i.e. the number of parallel edges with the same label between a sequence of nodes) to obtain a wqo, but we gain compatibility with GTS even in the presence of simple negative application conditions. It becomes obvious that no order is superior

to another, but each order may be suited for a specific application or modelling. For the minor and subgraph orderings we have shown this via the different case studies in Chapter 7.

We presented the backward search for solving the coverability problem in Chapter 6. However, we have to distinguish between the general coverability problem on the entire transition systems and the restricted coverability problem on the transition system restricted to the class of graphs $\mathcal{Q}$ on which the order is a wqo. In their general form both problems are undecidable, but we presented an algorithm which will terminate and either state that a graph is coverable in the unrestricted transition system or not coverable in the restricted transition system. We also showed that the restricted coverability problem becomes decidable if the minor ordering is used (in which case it coincides with the general coverability problem) or if $\mathcal{Q}$ is closed under reachability. Note that it is in general not possible to simply restrict the transition system to graphs of $\mathcal{Q}$, since the compatibility condition will likely be violated.

For the subgraph ordering we also presented so-called universally quantified rules. These rules are equipped with quantification pairs that allow them to be extended to match the entire neighbourhood of a node, which is not possible with normal SPO rules. These rules are more powerful and allow us to model for instance protocols with broadcast operations. However, since these rules impose some form of negative application condition which violates the compatibility condition, the result of our algorithm will be an over-approximation.

As proof of concept that our algorithm is not just of theoretical interest, we introduced the tool Uncover which implements several components of our framework, including the minor ordering, the subgraphs ordering and universally quantified rules.

8.2. Related Work

Beyond the approach we used in our framework, there are several other related approaches which also use the theory of well-structured transition systems to obtain verification procedures. Most of these approaches use various well-structured transition systems, solve various decision problems and apply either the forward or the backward algorithm. Depending on the setting they can be exact or approximating. We have already mentioned some of these approaches in the Chapters 2, 5 and 6, but discuss them here again for completeness.

Graphs transformation systems The approach presented in [AB+08] uses a backward algorithm very similar to the one in our framework. The authors use the minor ordering and graphs with at most one outgoing (binary) edge at each node to represent heaps. The minor ordering works especially well on this class of graphs, since it is downward-closed wrt. all three operations, i.e. node deletion, edge deletion as well as edge contraction.

In this setting the authors use the (decidable) coverability problem as basis to verify programs with dynamic heaps and also present some runtime results of an unpublished prototype implementation. This approach is somewhat contained in our approach, since we could use their class of graphs as restricted class $\mathcal{Q}$ together with the minor ordering (which we normally use in conjunction with all graphs). Since their transformation systems only reach graphs which are in this class, we obtain case (ii) of Theorem 2.21, i.e. a decidable restricted coverability problem.

In [DSZ10] the authors analyse ad hoc networks, which are modelled by a graph where every node is a process that is in turn modelled by a finite-state automaton. These networks allow local as well as broadcast operations and the authors use the induced subgraph ordering (which seems especially suited for broadcast operations). They are interested in three decision problems: control state reachability (can a configuration be reached where one of the processes is in given state), target reachability (is a configuration reachable such that every process is in some state of a given set) and repeated control state reachability (is there an infinite execution where a given state occurs infinitely often). These problems are undecidable, but become decidable if the transition system has node mobility. This approach inspired the definition and integration of universally quantified rules into our framework. Note that the application condition of universally quantified rules is slightly different than in this approach.

Another interesting approach is pursued in [SWJ08], where graph patterns containing a positive and a negative part are used to represent all graphs that contain the positive part as a subgraph, but not the negative part. This allows the use of negative application conditions, since these can be taken into account by changing the patterns negative part. The authors apply the standard backward algorithm using patterns as minimal elements of upward-closed sets to prove safety properties of the DYMO protocol. However, they do not define any well-quasi-order, which means that the approach is not guaranteed to terminate. Although they claim to over-approximate the reachability problem, they effectively solve coverability. A prototype implementation exists in the form of the GBT tool [GBT].

Depth-bounded systems In [BK+13] the authors use the subgraph ordering in conjunction with the forward algorithm. For representing downward-closed sets, they introduce abstractions by nested graphs G, where subgraphs of G can be equipped with natural numbers that indicate multiple occurrences of these subgraphs. Since these numbered subgraphs can be nested, they obtain an over-approximation. This setting is then used to solve weakly fair termination and runtime results for an extension of the Picasso tool [Picasso] are given.

Petri nets and lossy systems Another attempt to define a general framework was done in [SM12] for Petri nets (with transfer arcs) and lossy systems. The idea is to apply a

backward search for coverability and use the predecessor computation, the order and a witness function as parameters of the analysis. For search space construction, the authors integrated pruning, partial order reduction and backward acceleration. Experimental results are provided for Petri nets as well as lossy systems by the PETRUCHIO tool [Petruchio].

Approaches beyond well-structured transition systems Of course there are also approaches for analysing graph transformation systems without using well-structured transition systems. For instance, the GROOVE tool [Groove; Ren03] can be used to model graph transformation systems with very expressive application conditions and explore their state space. GROOVE can be used for model checking and verification [GM+12], but is limited to finite state spaces.

There also exists a number of approaches for infinite state systems that use abstract interpretation. The AUGUR2 tool [Augur2] implements the approach of [BCK01] where the authors use Petri graphs – which are hypergraphs extended with transitions – to abstract sets of graphs and apply techniques for Petri nets. This abstraction can also be refined via counterexample-guided abstraction refinement (CEGAR) [KK06]. A different idea is used in [HJ+15a; HJ+15b], where the authors use hyperedge replacement grammars (i.e. context-free GTS) to abstractly represent data structures such as trees, heaps or lists. A pointer-manipulating program can be analysed by modelling its heap in this ways and executing it on the abstract data structure. Another possible abstraction is represented in [Bau06] and implemented in the HIRALYSIS tool. There the author uses so-called partner abstractions where the structure of graphs is over-approximated by keeping for each node only its label and the labels of its adjacent nodes ("partners"). This approach is extended in [BR15a] to so-called cluster abstractions. Instead of only using the labels of partners, the information about possible edges between partners is also stored using three-valued logic. This extension is implemented in the ASTRA tool [Astra; BR15b].

Last but not least, there are also approaches using Hoare logic for graphs, i.e. the correctness of graph programs is checked by computing pre- and postconditions in a Hoare logic style [Pen09; HP09]. If the computations are precise enough, invariants or non-reachability of invalid configurations can be proven. The conditions used in this approach are limited to first-order structural properties, but have been extended to conditions equivalently expressive to monadic second-order logic (on graphs) in [PP14].

8.3. Future Work

There are several starting points for improving the approach presented in this thesis. In the following we will discuss some of these points and how they could be implemented.

More orders As already discussed in Section 5.4, a larger variety of order would strengthen the presented approach. Since we model properties as upward-closed sets, different orders can model different properties, i.e. a coarser order is not automatically better than a finer order (or vice versa). An interesting question is whether the induced minor ordering fits in our framework and whether the property of the induced subgraph ordering to be well-structured even in the presence of simple negative application conditions can be transferred to the induced minor ordering. If this is the case, we could analyse protocols working on ring structures (for which the subgraph ordering is not well suited) and still allow simple negative application conditions.

Especially interesting are orders which preserve directed paths in some sense, e.g. butterfly minors, where special contractions ensure that if there is no path between two nodes, there is also no path between these nodes in any butterfly minor. With such orders we could analyse protocols, where directed paths play an essential role, e.g. garbage collection algorithms, but where the subgraph ordering would not terminate.

Attributed graphs In Section 7.3 we introduced a leader election case study. We had the problem that our analysis can only analyse the system if the number of IDs used is fixed. To avoid this, we could use attributed graphs, i.e. introduce variable labels for our edges. By for instance assigning an integer variable to each process and message sent around the ring, the forwarding of a message can be modelled by only one rule, instead of one for each pair of IDs. Attributed graphs would also be helpful for modelling cryptographic protocols, since correctness of these often depends on the data send within a message. However, extending our framework in this way is not an easy task. For each graph computed by our analysis we need to symbolically represent the attributes and relations between attributes, i.e. we need to use appropriate formulae. These formulae need to be adjusted as part of each backward step by computing weakest preconditions. This requires some kind of abstraction framework (e.g. predicate abstraction) and automatic inference. It is also not clear whether our current well-quasi-orders can be extended to the pairs of graphs and formulae, such that the result is still a well-quasi-order. However, an extension to attributed graphs would be a promising enhancement of our framework.

Graph patterns We encountered a problem similar to the one mentioned in the previous point when analysing the public-private server communication case study presented in Section 7.6. There the problem was, that in each backward step the analysis computed paths of increasing length which were not in relation wrt. the subgraph ordering, i.e. the final working set is not finitely representable. We could tackle this problem by using graph patterns to describe sets that are otherwise not finitely representable. In this case a pattern for describing possibly arbitrary long paths would be sufficient. However, the problem is that the backward application of a rule becomes more difficult, since rules need now be applied (backwards) to graph patterns instead of only graphs. This

approach also requires some form of acceleration that creates graph patterns as part of a backward step. This would probably cause our approach to be approximative, but would ensure termination for some case studies (e.g. the one presented). Furthermore, we would also need to extend the orders to graph patterns. It could be helpful to investigate how abstract graph transformation [SWW11] can be used in this context. Another good starting point for this improvement would also be to check whether the graph patterns used in [SWJ08] would fit into this framework.

Forward vs. Backward Often the backward algorithm is claimed to be less efficient than the forward algorithm. This assumption is based on the fact that the backward algorithm can find states that are not reachable in the forward way, a "problem" the (non-approximating) forward algorithm does not have. General research indicates non-primitive recursive upper bounds in the worst-case and even some completeness results for some well-structured transition systems. However, these results often hold for the forward search as well. Efficiency may be better for specific WSTS and may also be improved by optimizations not part of the standard (general) backward search. The runtime results in Chapter 7 lets one assume that the average performance may be much better than expected. Furthermore, it is worth noting that the forward and backward algorithms are not fully comparable. Due to the order being a wqo, the final result of the backward search, i.e. the upward-closed set of states from which a state s is coverable, is finitely representable. This does not necessarily hold for the equivalent result set of the forward algorithm (the covering set), i.e. the downward-closed set of all states that are coverable from a state s. On the one hand, the forward algorithm relies on an adequate domain of limits to represent downward-closed sets of which the definition may be non-trivial. On the other hand, there are even WSTS (e.g. depth-bounded systems) where the covering set is not computable, even though coverability is decidable [BK+13]. All in all, a thorough comparison between the forward and backward approaches would help to understand which one is preferable for which systems.

Other decidability problems The coverability problem is not the only problem for which decidability can be achieved by using the theory of well-structured transition systems. For instance, [FS01] states four problems which are decidable by a forward search if successors can be computed effectively, the order is decidable and one of the alternative compatibility conditions in Figure 8.1 is satisfied.

The termination problem, i.e. the problem whether there is a infinite sequence of transitions from a given state, is decidable if transitive compatibility holds. Transitive compatibility (Figure 8.1a) is the same as regular compatibility, but the sequence $t_1 \Rightarrow^+ t_2$ must not be empty. As soon as the forward search finds a state that is larger than one of its predecessors, there is a sequence of transitions (from the smaller to the larger) that can be repeated arbitrarily often, i.e. the system does not terminate.

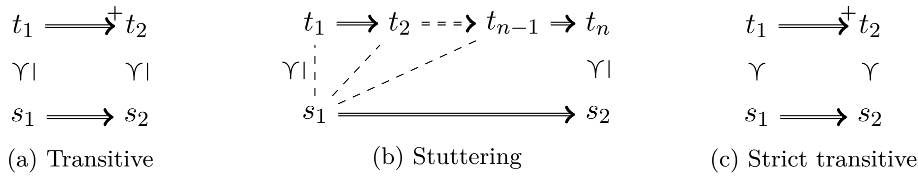


Figure 8.1.: Alternative compatibility conditions

The control-state maintainability problem, i.e. the problem whether every state reachable from a state s is larger than some set of control-states, and the inevitability problem, i.e. the problem whether every computation starting from s will reach a state not larger than a set of control-states, are both decidable if stuttering compatibility holds. Stuttering compatibility (Figure 8.1b) requires that for every t_i in the sequence $s_1 \preceq t_i$ must hold, except for the last t_n, for which $s_2 \preceq t_n$ holds.

Boundedness, i.e. the problem whether infinitely many states can be reached from some state s, is decidable if strict transitive compatibility holds. Strict transitive compatibility (Figure 8.1c) requires that if $s_1 \prec t_1$ holds, then $s_2 \prec t_2$ must hold as well (i.e. if t_1 is strictly larger, then t_2 must be strictly larger as well), and the sequence from t_1 to t_2 must not be empty.

It would be an interesting point of research how these decidability results for the mentioned problems could be translated to graph transformation systems. For the orders presented in this thesis we would need to investigate which graph transformation systems satisfy the alternative compatibility conditions.

Uncover The UNCOVER tool itself could also be further improved. For instance, most but not all parts of our framework are implemented, i.e. the induced subgraph ordering is not. Most optimizations presented in Section 6.5 are implemented as well, but additional optimizations are imaginable that improve performance and increase practical value of UNCOVER. Due to the nature of the backward search, runtimes and termination (in case of the subgraph ordering) are hard to predict. Runs with random graph transformation systems (see [Koz07]) could be helpful to benchmark the tool and analysis technique.

Appendix A
Related Formalism

In Chapter 4 we proved decidability and undecidability of reachability and coverability for several variants of graph transformation systems. These proofs use Petri nets, Turing machines or Minsky machines for the reductions. In this chapter we will briefly define these three well-known formalisms as well as the decision problems we used in the reductions.

A.1. Petri Nets

Petri nets where first introduced by Petri in [Pet62]. Intuitively a Petri net consists of places, which can contain an arbitrary number of tokens, and transitions that can fire to consume tokens from some places and add tokens to other places. The "state" of a Petri net is fully described by the number of tokens in each place, i.e. a so-called marking. Formally we define Petri nets as follows.

Definition A.1 (Petri net)**.** A *Petri net* is a tuple $\langle P, T, pre_{(\cdot)}, post_{(\cdot)}, m_0 \rangle$, where P is a set of places, T is a set of transitions, $pre_{(\cdot)}$ and $post_{(\cdot)}$ are sets of functions with $pre_t \colon P \to \mathbb{N}_0$, $post_t \colon P \to \mathbb{N}_0$ for every $t \in T$, and $m_0 \colon P \to \mathbb{N}_0$ is the initial marking.

Definition A.2 (Marking)**.** A *marking* (of a Petri net) is a function $m \colon P \to \mathbb{N}_0$. For two markings we say that $m_1 \leq m_2$ if and only if $m_1(p) \leq m_2(p)$ for all $p \in P$. Furthermore, we define $m_1 + m_2 = m'$ with $m'(p) = m_1(p) + m_2(p)$ for all $p \in P$ and $m_1 - m_2 = m''$ with $m''(p) = \max(m_1(p) - m_2(p), 0)$ for all $p \in P$.

Note that by fixing any order on the set of places, we can also write markings as tuples of natural numbers.

Definition A.3 (Firing transitions)**.** Given a marking m, we say that a transition t is enabled if $pre_t \leq m$. Any enabled transition t can be fired to obtain a new marking $m' = m - pre_t + post_t$.

Beginning with the initial marking we can fire transitions to generate new markings. We say that a marking is reachable if it can be generated by firing an arbitrary number of transitions an arbitrary number of times. Just as graph transformation rules, Petri nets can be seen as describing a transition system, where the states are markings and transitions between states correspond to firing a transition of the Petri net.

We are mainly interested in the reachability and coverability problems for Petri nets, which are both decidable [KM69; May84; May81]. In fact, coverability has been shown to be decidable even in the presence of more powerful transitions, namely transitions with reset arcs (removing all tokens of a place) and transfer arcs (moving all tokens from one place to another) [DFS98], whereas reachability is undecidable in these cases. We also use these more powerful arcs in some of our proof sketches.

A.2. Turing Machines

Turing machines are a computation model presented by Turing in [Tur37]. This simple model consists of an infinite tape which is read by a head moving along the tape. How the head moves and changes the tape depends on the current state of the machine. Formally these machines are defined as follows.

Definition A.4 (Turing machine)**.** Let Σ be an alphabet. A *(deterministic) Turing machine* is a tuple $\langle Z, \Sigma, \Gamma, \delta, z_0, \Box, E\rangle$, where Z is a set of states, $\Gamma \supset \Sigma$ is the tape alphabet, $\delta\colon Z \times \Gamma \to Z \times \Gamma \times \{R, L, N\}$ is the transition function, $z_0 \in Z$ is the start state, $\Box \in \Gamma \setminus \Sigma$ is the blank symbol and $E \subseteq Z$ is the set of final states. Additionally, we require that $\delta(z, a) = \langle z, a, N\rangle$ for all $z \in E$ and $a \in \Gamma$.

The idea of a Turing machine is that there is an infinite tape which contains a word $w \in \Sigma^*$ starting at the initial head position and $\Box$ symbols at every other position. The Turing machine then performs a step depending on the current state and the current tape symbol at the head's position. In each step, the state may be changed, the current symbol at the head's position may be changed and the head can be moved one to the left or right. We formalise this behaviour by introducing configurations and a transition relation on these configurations.

Definition A.5 (Configurations)**.** A *configuration* of a Turing machine is an element of $\Gamma^* Z \Gamma^*$. Given a word $w \in \Sigma^*$ the *initial configuration* of a Turing machine is $z_0 w \Box$.

Let $a_m \dots a_1 z b_1 \dots b_n$ be a configuration with $m \geq 0$ and $n \geq 1$. We define the transition relation $\vdash$ as follows:

- if $\delta(z, b_1) = \langle z', c, L\rangle$, then $a_m \dots a_1 z b_1 \dots b_n \vdash a_m \dots a_2 z' a_1 c b_2 \dots b_n$ if $m \geq 1$ and $z b_1 \dots b_n \vdash z' \Box c b_2 \dots b_n$ if $m = 0$,
- if $\delta(z, b_1) = \langle z', c, N\rangle$, then $a_m \dots a_1 z b_1 \dots b_n \vdash a_m \dots a_1 z' c b_2 \dots b_n$, and

- if $\delta(z, b_1) = \langle z', c, R\rangle$, then $a_m \dots a_1 z b_1 \dots b_n \vdash a_m \dots a_1 c z' b_2 \dots b_n$ if $n > 1$ and $a_m \dots a_1 z b_1 \vdash a_m \dots a_1 c z' \Box$ if $n = 1$.

We say that a Turing machine halts on a word w if from the initial configuration we can reach a configuration containing a final state, i.e. $z_0 w \Box \vdash^* \alpha z \beta$ with $z \in E$ and $\alpha, \beta \in \Gamma^*$.

Note that our Truing machines are deterministic, i.e. for every configuration ν there is exactly one configuration μ with $\nu \vdash \mu$. If the state in ν is a final state, then by definition of final state $\nu = \mu$ must hold.

Our main interest lies in the so-called halting problem, i.e. the question whether a given Turing machine will halt for a given word w. This problem is undecidable, even when the word w is fixed [Tur37].

A.3. Minsky Machines (Two-Counter Machines)

Minsky machines are a computation model which is equivalent in its expressiveness to Turing machines and was introduced by Minsky in [Min67]. A Minsky machine has a state and two counters which can be incremented and decremented independently by a set of instructions. Furthermore, instructions can also compare counters to zero, causing Minsky machines to be as powerful as Turing machines. Formally a Minsky machine is defined as follows.

Definition A.6 (Minsky machine)**.** A *Minsky machine* is a tuple $\langle Q, \Delta, \langle q_0, m, n\rangle\rangle$, where Q is the set of states, $\Delta \subseteq Q \times \mathit{Cmd} \times Q$ is the set of instructions and $\langle q_0, m, n\rangle \in Q \times \mathbb{N}_0 \times \mathbb{N}_0$ is the initial configuration of the machine. A command of Cmd can be 'c++' (an increment), 'c−−' (a decrement) or 'c=0?' (a zero-test), where c may be one of the two counters, i.e. $c \in \{c_1, c_2\}$.

Analogous to Turing machines, we can define configurations and a transition relation on configurations as follows.

Definition A.7 (Configurations)**.** A *configuration* of a Minsky machine is an element of $Q \times \mathbb{N}_0 \times \mathbb{N}_0$. Let $\langle q, n_1, n_2\rangle$ be a configuration. We name the counters c_i with $i \in \{1, 2\}$ and define the transition relation $\vdash$ as follows:

- if $\langle q, c_i{+}{+}, q'\rangle \in \Delta$, then $\langle q, n_1, n_2\rangle \vdash \langle q', n'_1, n'_2\rangle$ with $n'_i = n_i + 1$, $n'_{3-i} = n_{3-i}$,
- if $\langle q, c_i{-}{-}, q'\rangle \in \Delta$ and $n_i > 0$, then $\langle q, n_1, n_2\rangle \vdash \langle q', n'_1, n'_2\rangle$ with $n'_i = n_i - 1$, $n'_{3-i} = n_{3-i}$ and
- if $\langle q, c_i{=}0?, q'\rangle \in \Delta$ and $n_i = 0$, then $\langle q, n_1, n_2\rangle \vdash \langle q', n_1, n_2\rangle$.

Note that the increment and zero-test are blocking, i.e. an instruction containing them is only applicable if $n_i > 0$ or $n_i = 0$ respectively. Again, $\vdash$ forms a transition system where states are configurations of the Minsky machine. For Minsky machines we are interested in the reachability problem, i.e. whether a given configuration is reachable (from the initial configuration), and the control state reachability problem, i.e. whether a configuration is reachable (from the initial configuration) that contains a given state. Both problems are undecidable in general [Min67]. Note that the control state reachability problem is a variant of the coverability problem.

Appendix

B
Proofs of Chapter 3

This appendix contains proofs of Sections 3.4 and 3.5 which where moved here to ease the reading of those sections.

B.1. Proofs of Section 3.4

This section contains the correctness proofs for the constructions of pushouts in **Λ-HGt** and **Λ-HGp** respectively.

Proposition 3.22 (Pushout in Λ-HGt). *Let G, H, I be graphs with pairwise disjoint node and edge sets*[1] *and let $f\colon G \rightarrow H$, $g\colon G \rightarrow I$ be total graph morphisms. Let $\sim$ be the relation on $V_H \cup V_I \cup E_H \cup E_I$, where $f(x) \sim g(x)$ and $g(x) \sim f(x)$ for all $x \in G$ and let $\eqsim$ be the equivalence closure of $\sim$. The pushout object $J = \langle V_J, E_J, c_J, l_J \rangle$ can be constructed as follows:*

- $V_J = (V_H \cup V_I)/\eqsim$,
- $E_J = (E_H \cup E_I)/\eqsim$,
- $c_J\colon E_J \to V_J^*$ *where* $c_J([e]_\eqsim) = [v_1]_\eqsim \dots [v_k]_\eqsim$ *and* $v_1 \dots v_k = \begin{cases} c_H(e) & \text{if } e \in E_H \\ c_I(e) & \text{if } e \in E_I \end{cases}$
- $l_J\colon E_J \to \Lambda$ *where* $l_J([e]_\eqsim) = \begin{cases} l_H(e) & \text{if } e \in E_H \\ l_I(e) & \text{if } e \in E_I \end{cases}$

The resulting morphisms are $f'\colon I \rightarrow J$, $g'\colon H \rightarrow J$ with

$$f'(x) = [x]_\eqsim \qquad\qquad g'(y) = [y]_\eqsim$$

[1] Disjointness can be easily achieved by renaming.

for $x \in I$ and $y \in H$, respectively. The object J together with the morphisms f', g' is the pushout of f, g.

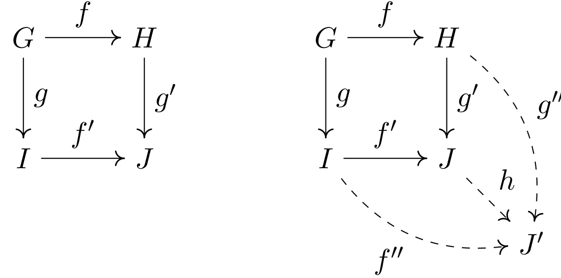


Figure B.1.: Pushout diagram for the construction in Proposition 3.22

Proof. To show that J is, in fact, the pushout, we have to prove that J is well-defined, the left diagram in Figure B.1 commutes, and the morphism h exists and is unique (up to isomorphism).

Obviously c_J and l_J assign a sequence of nodes and a label to each new edge $[e]_{\eqsim} \in E_J$. We first show that l_J is well-defined by showing that $e \eqsim e'$ implies $l(e) = l(e')$. In the following we will assume that $e \in H$ and $e' \in I$, but the proof is the same for $e, e' \in H$ and $e, e' \in I$. If $e \eqsim e'$, then there exists an alternating sequence $e = f(e_1) \sim g(e_1) = g(e_2) \sim f(e_2) = f(e_3) \sim \ldots \sim g(e_n) = e'$. Obviously the label is preserved by every equality in the sequence, and for every step $f(e_i) \sim g(e_i)$ it holds that $l_H(f(e_i)) = l_G(e_i) = l_I(g(e_i))$, thus $l(e) = l(e')$. Note that this also implies $|c(e)| = |c(e')|$.

In the same way, using the same sequence we show that c_J is well-defined by showing that $e \eqsim e'$ implies $c(e)[i] = c(e')[i]$ for $1 \leq i \leq |c(e)|$. Again, at every equality in the sequence $c_I(g(e_j))[i] = c_I(g(e_{j+1}))[i]$ or $c_H(f(e_j))[i] = c_H(f(e_{j+1}))[i]$ respectively, obviously hold. If $f(e_j) \sim g(e_j)$, then we obtain $c_H(f(e_j))[i] = f(c_G(e_j)[i]) \sim g(c_G(e_j)[i]) = c_I(g(e_j))[i]$. The fact that $e \eqsim e'$ implies $c(e) \eqsim c(e')$ is also sufficient to prove that f' and g' are morphisms.

It is easy to see that the diagram commutes, since $g'(f(x)) = [f(x)]_{\eqsim} = [g(x)]_{\eqsim} = f'(g(x))$. For two morphisms $f'' \colon I \to J'$ and $g'' \colon H \to J'$, commuting with f and g as shown in Figure B.1, the mediating morphism h is defined as

$$h([x]_{\eqsim}) = \begin{cases} g''(x) & \text{if } x \in H \\ f''(x) & \text{if } x \in I \end{cases}$$

Clearly, the diagram on the right of Figure B.1 commutes, i.e. $g'' = h \circ g'$ and $f'' = h \circ f'$ hold, also showing that h is a morphism, but we have to show that h is unique and well-defined. For the uniqueness assume that there is another morphism $h' \colon J \to J'$

commuting with f'' and g''. Then either $h'([x]_{\eqsim}) = h'(g'(x)) = g''(x) = h([x]_{\eqsim})$ or $h'([x]_{\eqsim}) = h'(f'(x)) = f''(x) = h([x]_{\eqsim})$ depending on whether $x \in H$ or $x \in I$.

Finally we show that h is well-defined. Assume there are $x \in H$ and $x' \in I$ such that $x \eqsim x'$. Note, that the proofs for $x, x' \in H$ and $x, x' \in I$ are again essentially the same as this case. Since $x \eqsim x'$, there is an alternating sequence $x = f(x_1) \sim g(x_1) = g(x_2) \sim f(x_2) = f(x_3) \sim \ldots \sim g(x_n) = x'$. Every equality preserves the image of the respective element in J', since $f''(g(x_i)) = f''(g(x_{i+1}))$ or $g''(f(x_i)) = g''(f(x_{i+1}))$ respectively hold. If $f(x_i) \sim g(x_i)$, the image is also preserved, since $g''(f(x_i)) = f''(g(x_i))$ holds by assumption that the diagram commutes. Thus, h is well-defined. □

Proposition 3.24 (Pushouts in Λ-HGp). *Let G, H, I be graphs with pairwise disjoint node and edge sets and let $f\colon G \rightharpoonup H$, $g\colon G \rightharpoonup I$ be partial graph morphisms. Let $\sim$ be the relation on $V_H \cup V_I \cup E_H \cup E_I$, where $f(x) \sim g(x)$ and $g(x) \sim f(x)$ for all $x \in G$ for which $f(x)$ and $g(x)$ are both defined and let $\eqsim$ be the equivalence closure of $\sim$.*

We say that an equivalence class on nodes is valid *if and only if it does contain no element $f(x)$ for which $g(x)$ is undefined and no element $g(x)$ for which $f(x)$ is undefined. An equivalence class on edges is* valid, *if the previous condition holds for the class and the equivalence class of every incident node is valid as well.*

We can construct the pushout J by the same means as Proposition 3.22 with the exception that V_J and E_J contain only the valid equivalence classes and $f'(x)$, $g'(x)$ are undefined if the equivalence class of x is not valid.

Proof. In the proof of Proposition 3.22 we have already shown that c_J and l_J are well-defined. This directly transfers to this setting, since c_J and l_J are only defined for valid equivalence classes and a valid equivalence class of an edge implies valid equivalence classes of its incident nodes. The same holds for the commutativity of the diagram for valid equivalence classes and the existence of the mediating morphism h (being defined only for valid equivalence classes). By definition the diagram also commutes for non-valid equivalence classes, since for any $x \in G$ the images $f(x)$ and $g(x)$ are equivalent. This means that the equivalence class of $f(x)$ is non-valid if and only if the equivalence class of $g(x)$ is non-valid. Hence, $g'(f(x)) = f'(g(x))$. Note that the diagram of this proof is shown in Figure B.1, with the exception that the involved morphisms need not be total.

Finally we have to show that for two morphisms $f''\colon I \rightharpoonup J'$ and $g''\colon H \rightharpoonup J'$ commuting with f and g, the mediating morphism $h\colon J \rightharpoonup J'$, as defined in the proof of Proposition 3.22 commutes with g'' and f''.

The morphism h is only defined on valid equivalence classes and by definition commutes with g'' on all these classes. We show that for all $x \in H$, if the equivalence class of x is not valid, then $g''(x)$ is undefined. The commutativity with f'' can be shown analogously. Assume the equivalence class of x is non-valid, then there is an alternating sequence $x = f(x_1) \sim g(x_1) = g(x_2) \sim f(x_2) = f(x_3) \sim \ldots \sim f(x_n) = x'$ where $g(x_n)$ is

undefined and $f(x_n)$ as well as every $g(x_i)$, $f(x_i)$ with $1 \leq i \leq n-1$ are defined. Note that the proof is the same if $x' \in I$ instead of $x' \in H$. Clearly, $f''(g(x_n))$ is undefined and this undefinedness is preserved by every equality, since $f''(g(x_i)) = f''(g(x_{i+1}))$ and $g''(f(x_i)) = g''(f(x_{i+1}))$ hold, respectively. So assume $f(x_i) \sim g(x_i)$, then by commutativity we know that $g''(f(x_i)) = f''(g(x_i))$ holds. Thus, h commutes with f'' and g'', i.e. $g'' = h \circ g'$ and $f'' = h \circ f'$. ☐

B.2. Proofs of Section 3.5

The following two lemmas prove correctness of the construction of pushout complements for total morphisms defined in Proposition 3.26.

Lemma 3.28. *Let $f\colon G \rightarrow H$ and $g'\colon H \rightarrow J$ be two morphisms satisfying the conditions of Proposition 3.21, i.e. at least one pushout complement exists. Then every equivalence relation $\equiv$ created by the construction in Proposition 3.26 generates a pushout complement.*

Proof. Assume that $\equiv$ is one of the equivalences of the construction of Proposition 3.26 and that I, g, f' have been obtained by factoring $G \uplus \tilde{J}$ through this equivalence (see Figure B.2).

As a first step we show that $g' \circ f = f' \circ g$, i.e. the resulting square commutes. Because $\overline{g}$ is the canonical embedding of G into $G \uplus \tilde{J}$ (and therefore injective) and $\overline{f}(x)$ is defined as $g'(f(x))$ if $x \in G$, we know that $g'(f(x)) = \overline{f}(\overline{g}(x))$ holds. Furthermore by definition of g, f' we have:

$$g'(f(x)) = \overline{f}(\overline{g}(x)) = f'([\overline{g}(x)]_{\equiv}) = f'(g(x))$$

Now we show that I is indeed a pushout complement by verifying that the second condition of Definition 3.6 is satisfied: we have to prove that for every other commuting pair of morphisms $g''\colon H \rightarrow J'$, $f''\colon I \rightarrow J'$ there is a unique morphism $h\colon J \rightarrow J'$ such that $h \circ g' = g''$ and $h \circ f' = f''$ as shown is Figure B.2. We define the required morphism h as follows:

$$h(x) = \begin{cases} f''(x') & \text{if } \exists x' \in I : f'(x') = x \\ g''(x') & \text{if } \exists x' \in H : g'(x') = x \end{cases}$$

By definition of f' every element of J has a preimage either under f' or g'. It remains to be shown that h is a well-defined morphism, and that it is the unique morphism such that the triangles commute.

Commutativity. By definition $h(g'(x)) = g''(x)$ and $h(f'(x)) = f''(x)$ hold.

Uniqueness. Let h' be another morphism with $h' \circ f' = f''$ and $h' \circ g' = g''$. Each element of J has a preimage either under f' or g':

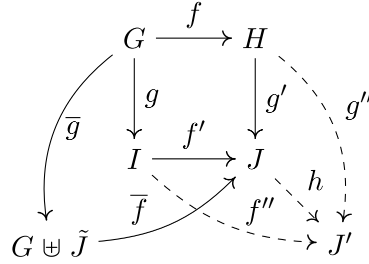


Figure B.2.: Commuting diagram showing all morphisms of Lemma 3.28

1. if $x = f'(x')$ for some $x' \in I$, then $h'(x) = h'(f'(x')) = f''(x') = h(f'(x')) = h(x)$,
2. if $x = g'(x')$ for some $x' \in H$, then $h'(x) = h'(g'(x')) = g''(x') = h(g'(x')) = h(x)$.

Well-definedness. As seen before, h is defined for all elements of J. To show well-definedness it is therefore only necessary to prove that different x' having the same image under f' or g' also have the same image under f'' or g''.

Every element of I is an equivalence class of $\equiv$. Therefore, let $x = [x']_\equiv$ and $y = [y']_\equiv$. In the following we do not strictly distinguish between an element of G and its image under $\overline{g}$ because $\overline{g}$ is a canonical embedding. Hence for $x' \in G \uplus \tilde{J}$ the property $x' \in G$ holds if and only if x' has a preimage under $\overline{g}$.

The *first property* we show is that $f'(x) = f'(y) \implies f''(x) = f''(y)$ holds for all $x, y \in I$. For $x \neq y$ there are two cases which have to be considered:

1. We assume $x', y' \in G$, i.e. the equivalence classes x, y have representatives in G (which also implies $g(x') = x$ and $g(y') = y$). We know that $x' \equiv_{\overline{f}} y'$ holds because of $\overline{f}(x') = f'([x']_\equiv) = f'(x) = f'(y) = f'([y']_\equiv) = \overline{f}(y')$. Due to this equivalence there are $x_1, y_1, \ldots, x_n, y_n \in G$ with $x' = x_1$, $y' = y_n$ such that $x_i \equiv y_i$ for $1 \leq i \leq n$ and $y_i \equiv_f x_{i+1}$ for $1 \leq i < n$. Using the definition of g and the fact that x_i and y_i are elements of G it can be shown that the equivalence $x_i \equiv y_i$ implies $g(x_i) = [\overline{g}(x_i)]_\equiv = [\overline{g}(y_i)]_\equiv = g(y_i)$. These properties lead to the equality $g''(f(x_i)) = f''(g(x_i)) = f''(g(y_i)) = g''(f(y_i)) = g''(f(x_{i+1}))$ for every i, thus $f''(x) = f''(y)$ holds.
2. Now assume x contains no elements of G (implying $x' \notin G$). Because x contains no elements of G, it also has no preimage under g. By definition $f'([x']_\equiv) = f'([y']_\equiv)$ implies $x' \equiv_{\overline{f}} y'$. Because of this equivalence there are $x_1, y_1, \ldots, x_n, y_n \in G$ with $x' = x_1$, $y' = y_n$ satisfying $x_i \equiv y_i$ for $1 \leq i \leq n$ and $y_i \equiv_f x_{i+1}$ for $1 \leq i < n$. It is easy to see that y_1 can not be an element of G, because otherwise $[x']_\equiv$ would contain elements of G. Furthermore, due to the definition of $\equiv_f$ it holds that $y_1 = x_2$ because y_1 is not in G. This property can be extended to $y_i = x_{i+1}$, which

leads to $x_i \equiv x_{i+1}$. Since this ultimately implies $x' \equiv y'$ and, thus, $x = y$, it is clear that $f''(x) = f''(y)$ holds.

The *second property* needed for well-definedness is $g'(x) = g'(y) \implies g''(x) = g''(y)$. The identification condition (see Proposition 3.21) states that because of $g'(x) = g'(y)$ there are $x', y' \in G$ such that $f(x') = x$ and $f(y') = y$. Using this and the first property the desired equality can easily be shown by:

$$\begin{array}{cccc} g'(x) = g'(y) & \implies & g'(f(x')) = g'(f(y')) & \implies \\ f'(g(x')) = f'(g'(y')) & \implies & f''(g(x')) = f''(g(y')) & \implies \\ g''(f(x')) = g''(f(y')) & \implies & g''(x) = g''(y) & \end{array}$$

The *last property* to show is $f'(x) = g'(y) \implies f''(x) = g''(y)$. We first show that $f'(x) = g'(y)$ implies that there is a $y' \in G$ with $f(y') = y$: the only items of J which are in the range of both f' and g' are the images of elements of G and nodes in the range of g' which are incident to edges which are not in the range of g'. However, due to the dangling condition (see Proposition 3.21) such nodes must have a preimage in G. Together with the first property this implies:

$$\begin{array}{cccccc} f'(x) = g'(y) & \implies & f'(x) = g'(f(y')) & \implies & f'(x) = f'(g(y')) & \implies \\ f''(x) = f''(g(y')) & \implies & f''(x) = g''(f(y')) & \implies & f''(x) = g''(y) & \end{array}$$

Morphism. Finally it is straightforward to prove that h satisfies indeed the morphism properties. For instance in order to show that $h(c_J(e)) = c_{J'}(h(e))$ for an edge $e \in J$ we have to distinguish two cases: if there exists an edge $e' \in I$ with $f'(e') = e$, then – since f' is a morphism – we have $f'(c_I(e')) = c_J(e)$. Hence $h(c_J(e)) = h(f'(c_I(e'))) = g''(c_I(e')) = c_{J'}(g''(e')) = c_{J'}(h(e))$ by definition of h. The case $e' \in H$ with $g'(e') = e$ is analogous.

This proves that every diagram formed by an equivalence generated in the given construction is a pushout diagram. □

Lemma 3.29. *Assume that $f: G \to H$ and $g': H \to J$ are given. Then every pushout complement $\langle I, g: G \to I, f': I \to J\rangle$ of f, g' can be obtained via the construction of Proposition 3.26. Furthermore two isomorphic pushout complements which commute with the isomorphism give rise to the same equivalence $\equiv$.*

Proof. Assume that I with morphisms g, f' is a pushout complement of f, g'. We will show that there is an equivalence $\equiv$, as specified by the construction of Proposition 3.26, such that I is obtained by the quotient $G \uplus \tilde{J}/\equiv$. For the given pushout of f, g' we will define a surjective morphism $k: G \uplus \tilde{J} \to I$ as seen in Figure B.3. Our next step is then to define an equivalence relation $\equiv$ where $x, y \in G \uplus \tilde{J}$ are equivalent if and only if $k(x) = k(y)$. The quotient of $G \uplus \tilde{J}$ through $\equiv$ then results in I and it has to

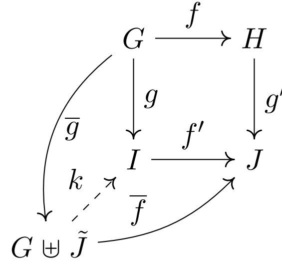


Figure B.3.: Commuting diagram showing all morphisms of Lemma 3.29

be shown that the equivalence relation $\equiv$ is one of the equivalence relations obtained by the presented construction.

Let $\eqsim$ be the equivalence closure of the relation $\sim$ where $f(x) \sim g(x)$ and $g(x) \sim f(x)$ for all $x \in G$. Due to the construction of pushouts using equivalence classes we can assume without loss of generality that $J = (I \uplus H)/\eqsim$ (see Proposition 3.22). Furthermore, for $x \in H$ we have $g'(x) = [x]_\eqsim$ and for $y \in I$ we have $f'(y) = [y]_\eqsim$. Using this, we define k as follows:

$$k(x) = \begin{cases} g(x) & \text{if } x \in G \\ y' & \text{if } x = y_c \text{ and } f'(y') = y \\ c_I(k(e_c))[i] & \text{if } x = \langle e_c, i \rangle \end{cases}$$

Well-definedness. Problems with well-definedness may arise only in the second case of the definition of k, where x is of the form y_c for some element y of J. In this case y is not in the range of g' due to the construction of $G \uplus \tilde{J}$. Therefore y as an equivalence class does not contain elements of H. Because of the definition of $\eqsim$ every equivalence class containing elements of either H or I (but not both) only contains one element, hence y contains exactly one element y' of I. Because $f'(y') = [y']_\eqsim = y$ the preimage of y under f' is unique and therefore $k(x)$ is well-defined in this case.

Morphism. Note that k is obviously a morphism on the elements of G. Furthermore, $\tilde{J}$ is a disjoint collection of nodes and edges and the third case in the definition of k ensures that it is indeed a valid morphism.

Surjectivity. We now show that k is surjective. Let therefore $x \in I$ be any element of I and we distinguish the following two cases:

1. $\exists y \in G : (g(y) = x)$: By definition $k(y) = g(y) = x$.

2. $\nexists y \in G : (g(y) = x)$: Without a preimage under g the equivalence class $[x]_\eqsim$ contains only x because x is not equivalent to any element of H according to $\eqsim$. Therefore $[x]_\eqsim$ is not in the range of g' since otherwise the equivalence class

would contain elements of H. Because of the definition of k there is a $y' \in \tilde{J}$ with $\overline{f}(y') = [x]_{\eqsim} = f'(x)$, hence $k(y') = x$.

Commutativity. We have to show that both triangles commute:

1. We first check that $k(\overline{g}(x)) = g(x)$ for any $x \in G$. As already seen, $\overline{g}(x) = x$ if $x \in G$. Using the definition of k we obtain $k(\overline{g}(x)) = k(x) = g(x)$.
2. Now we show that $f'(k(x)) = \overline{f}(x)$ for any $x \in G \uplus \tilde{J}$. There are two cases to distinguish:
   a) $x \in G$: Using $k(x) = g(x)$ if $x \in G$ and $g' \circ f = \overline{f} \circ \overline{g}$ due to the definition of $\overline{g}$ and $\overline{f}$ it can be shown that:
   $$f'(k(x)) = f'(g(x)) = g'(f(x)) = \overline{g}(\overline{f}(x)) = \overline{f}(x)$$
   b) $x \in \tilde{J}$: In this case $k(x) = y'$ and $f'(y') = \overline{f}(x)$, therefore $f'(k(x)) = f'(y') = \overline{f}(x)$.

The equivalence $\equiv$ is generated. We will now show that the equivalence $\equiv$, where two elements $x, y \in G \uplus \tilde{J}$ are equivalent if and only if $k(x) = k(y)$, is generated by the given construction. Hence we have to show that the equivalence closure of $\equiv \cup \equiv_f$ is $\equiv_{\overline{f}}$, i.e., that $\overline{\equiv \cup \equiv_f} = \equiv_{\overline{f}}$.

- $\overline{\equiv \cup \equiv_f} \subseteq \equiv_{\overline{f}}$:
  As already mentioned in Proposition 3.26, $\equiv_f$ implies $\equiv_{\overline{f}}$, i.e. $\equiv_f$ is clearly a subset of $\equiv_{\overline{f}}$. The equivalence $\equiv$ is also a subset of $\equiv_{\overline{f}}$ because of:
  $$x \equiv y \implies k(x) = k(y) \implies \overline{f}(x) = f'(k(x)) = f'(k(y)) = \overline{f}(y)$$
- $\overline{\equiv \cup \equiv_f} \supseteq \equiv_{\overline{f}}$:
  Let x, y be elements of $G \uplus \tilde{J}$ with $x \equiv_{\overline{f}} y$, hence $\overline{f}(x) = \overline{f}(y)$. As shown above the equivalence classes $\overline{f}(x)$ and $\overline{f}(y)$ of $\eqsim$ contain $k(x)$ and $k(y)$ respectively, therefore $k(x) \eqsim k(y)$. Hence there are $x_0, y_1, x_1, \ldots, y_m, x_m$ such that $x_i \sim y_i$ for $1 \leq i \leq m$ and $x_i \sim y_{i+1}$ for $0 \leq i < m$ with $k(x) = x_0$ and $k(y) = y_m$. Using the definition of $\sim$ leads to the following properties:
  $$x_i \sim y_i \quad\implies\quad \exists z_i \in G : (g(z_i) = x_i \wedge f(z_i) = y_i)$$
  $$x_i \sim y_{i+1} \quad\implies\quad \exists z_i' \in G : (g(z_i') = x_i \wedge f(z_i') = y_{i+1})$$
  This implies that z_{i+1} and z_i' have the same image under f, hence $z_{i+1} \equiv_f z_i'$, and that z_i and z_i' have the same image under g, hence $z_i \equiv z_i'$. This leads to $x \equiv z_0' \equiv_f z_1 \equiv z_1' \equiv_f \cdots \equiv z_{m-1}' \equiv_f z_m \equiv y$, hence $x \mathrel{\overline{\equiv \cup \equiv_f}} y$.

This concludes the proof that every pushout complement can be obtained by using the given construction and it remains to be shown that two isomorphic pushout complements give rise to the same equivalence.

Isomorphism of pushout complements. Let $\langle I_i, g_i \colon G \rightarrow I_i, f'_i \colon I_i \rightarrow J\rangle$ with $i = 1, 2$ be two pushout complements and an let $j \colon I_1 \rightarrow I_2$ be an isomorphism with $j \circ g_1 = g_2$ and $f'_2 \circ j = f'_1$. It is sufficient to show that j commutes with the morphisms k_1, k_2, where $k_i \colon G \uplus \tilde{J} \rightarrow I_i$ and k_1, k_2 are constructed analogously to the morphism k above. That is, we have to show that $j \circ k_1 = k_2$. Then k_1, k_2 give rise to the same equivalence $\equiv$.

We distinguish the following cases (as in the definition of k): if $x \in G$, then $j(k_1(x)) = j(g_1(x)) = g_2(x) = k_2(x)$. If x is of the form y_c for some item y of J, then we define $k_i(x) = y'_i$ for some y'_i with $f'_i(y'_i) = y$. Since $f'_2(j(y'_1)) = f'_1(y'_1) = y$ we obtain $y'_2 = j(y'_1)$. Hence $j(k_1(x)) = j(y'_1) = y'_2 = k_2(x)$. Finally, if x is of the form $\langle e_c, \ell\rangle$ for some edge e of J, then $k_i(x) = c_{I_i}(k_i(e))[\ell]$ and so $j(k_1(x)) = j(c_{I_1}(k_1(e))[\ell]) = c_{I_2}(j(k_1(e)))[\ell] = c_{I_2}(k_2(e))[\ell] = k_2(x)$. This completes the proof. □

In the following we prove soundness of the construction defined in Proposition 3.30. The completeness is proven in the Lemmas 3.31 and 3.32.

Proposition 3.30 (Pushout complements in Λ-HGp I). *Let $f \colon G \rightharpoonup H$ be a injective and surjective, partial morphism and let $g' \colon H \rightarrow J$ be a total morphism such that f and g' satisfy Proposition 3.21, i.e. a pushout complement exists. We can compute a pushout complement $\langle I, g, f'\rangle$ with $g \colon G \rightarrow I$ and $f' \colon I \rightharpoonup J$ as follows (see also Figure B.4):*

1. *Generate $\tilde{J}$ by taking a copy of J and adding a copy v_c for every $v \in V_G$ for which $f(v)$ is undefined. Then add a copy e_c for every $e \in E_G$ for which $f(e)$ is undefined with $l_{\tilde{J}}(e_c) = l_G(e)$ and*
$$c_{\tilde{J}}(e_c)[i] = \begin{cases} g'(f(c_G(e)[i])) & \text{if } f(c_G(e)[i]) \text{ is defined} \\ v_c & \text{with } v = c_G(e)[i] \text{ if } f(v) \text{ is undefined} \end{cases}$$
   for $1 \le i \le ar(l_{\tilde{J}}(e_c))$.

   We define the morphisms $\tilde{g} \colon G \rightarrow \tilde{J}$ and $\tilde{f}' \colon \tilde{J} \rightharpoonup J$ as follows:
$$\tilde{g}(x) = \begin{cases} g'(f(x)) & \text{if } f(x) \text{ is defined} \\ x_c & \text{if } f(x) \text{ is undefined} \end{cases}$$
$$\tilde{f}'(x) = \begin{cases} x & \text{if } x \in J \\ \text{undefined} & \text{else} \end{cases}$$

2. *Now let $\equiv$ be any equivalence on $\tilde{J}$, where*
   - *if $x \equiv y$, then either $x, y \in V_{\tilde{J}}$ or $x, y \in E_{\tilde{J}}$,*

- *if* $x \equiv y$ *and* $\tilde{f}'(x)$ *is defined, then* $x = y$ *holds, and*
- *if* $x \equiv y$ *for* $x, y \in E_{\tilde{J}}$*, then* $c_{\tilde{J}}(x)[i] \equiv c_{\tilde{J}}(y)[i]$ *for* $1 \leq i \leq ar(l_{\tilde{J}}(x)) = ar(l_{\tilde{J}}(y))$*.*

3. *We obtain a pushout complement* I *by taking a copy of* $\tilde{J}/\equiv$ *and adding an arbitrary (but finite) number of edges. For each such edge* e *there has to be an index* i *such that* $f'(c_I(e)[i])$ *is undefined. The morphisms* g *and* f' *are defined as follows:*

$$g(x) = [\tilde{g}(x)]_\equiv$$
$$f'(x) = \begin{cases} \tilde{f}'(x') & \text{if } x = [x']_\equiv \in \tilde{J}/\equiv \\ \text{undefined} & \text{else} \end{cases}$$

If we do not add any edges in step 3, this construction generates only finitely many pushout complements.

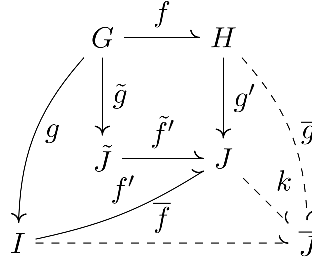


Figure B.4.: Pushout complement diagram for the construction in Proposition 3.30 and its proof (dashed morphisms)

Proof. We will show that every graph I computed by this construction is a pushout complement. First we show that all steps of the construction are well-defined.

Correctness of $\tilde{g}$ *and* $\tilde{f}'$*.* Obviously $\tilde{g}$ and $\tilde{f}'$ are well-defined and satisfy the morphism properties. The latter holds for $\tilde{g}$ and any e_c by definition of $c_{\tilde{J}}$. If $f(x)$ is defined, we obtain $\tilde{f}'(\tilde{g}(x)) = \tilde{f}'(g'(f(x))) = g'(f(x))$ and if $f(x)$ is undefined, then both $g'(f(x))$ and $\tilde{f}'(\tilde{g}(x))$, thus the morphisms commute (see Figure B.4).

Well-definedness of f *and* g*.* The last condition for $\equiv$ in step 2 of the construction ensures that g satisfies the morphism property. Obviously f' also satisfies this property, but we have to show that it is well-defined. So let $x \in \tilde{J}/\equiv$. If there is an $x' \in x$ where $\tilde{f}'(x')$ is defined, then by the second condition of step 2 x' is the only element in x. If there is no such x', then $\tilde{f}'$ is undefined for every element of x, thus f' is well-defined. Note that f' is undefined for every edge added in step 3.

Commutativity. The morphisms f', g commute with $\tilde{f}'$, $\tilde{g}$, and therefore with g', f, since by definition $f'(g(x)) = f'([\tilde{g}(x)]_\equiv) = \tilde{f}'(\tilde{g}(x))$. It remains to be shown that a unique mediating morphism exist. Let $\overline{f}\colon I \rightharpoonup \overline{J}$ and $\overline{g}\colon H \rightharpoonup \overline{J}$ be two morphisms such that $\overline{f} \circ g = \overline{g} \circ f$. Since $\tilde{f}'$ as well as f' are by definition surjective, we can define the mediating morphism $k\colon J \rightharpoonup \overline{J}$ as

$$k(x) = \overline{f}(x') \text{ for some } x' \text{ with } f'(x') = x.$$

Note that k is in fact well-defined, since $f'(x')$ is defined implying that $[x']_\equiv$ contains only x'. Obviously this also implies the commutativity $\overline{f}(x) = k(f'(x))$ for every x for which $f'(x)$ is defined or for which $\overline{f}(x)$ is undefined. It remains to be shown that $\overline{f}(x)$ is undefined if $f'(x)$ is undefined (the other way not necessarily holds). If $f'(x)$ is undefined, then x is either an equivalence class containing only x_c added in step 1 or an edge added in step 3. In the first case x has a preimage in $x' \in G$ for which $f(x')$ (and therefore also $\overline{g}(f(x'))$) is undefined. By commutativity of the diagram $\overline{f}(x')$ must be undefined as well. If x is an edge added in step 3, then by definition it is incident to a node $v \in I$ for which $f'(v)$ is undefined. By the previous argument $\overline{f}(v)$ is undefined as well and therefore $\overline{f}(x)$ must be undefined, since $\overline{f}$ is an morphism.

We now show the second commutativity $\overline{g} = k \circ g'$. Since f is injective and surjective, for every $x \in H$ there exists exactly one $x' \in G$ with $f(x') = x$, thus we can prove that the second triangle commutes by the following equalities:

$$\overline{g}(x) = \overline{g}(f(x')) = \overline{f}(g(x')) = k(f'(g(x'))) = k(g'(f(x'))) = k(g'(x))$$

Uniqueness of k. Finally it only remains to be shown that k is unique, so let $k'\colon J \rightharpoonup \overline{J}$ be another morphisms commuting with the diagram. By definition every element $x \in J$ has exactly one preimage $x' \in I$. Therefore we can use the commutativity to show $k(x) = k(f'(x')) = \overline{f}(x') = k'(f'(x')) = k'(x)$.

Finiteness. If we do not add any edges in step 3, only finitely many pushout complements are generated. This is obvious, since all involved graphs – especially $\tilde{J}$ – are finite, and thus there are only finitely many equivalences $\equiv$ satisfying the conditions in step 2. Each $\equiv$ leads to one pushout complement. □

Lemma 3.31. *Let $f\colon G \rightharpoonup H$ be a partial, injective morphism and let $g\colon G \rightharpoonup I$ be any partial morphism. The morphism f' of the pushout $\langle J, f', g'\rangle$ is injective.*

Proof. Without loss of generality we assume that $\langle J, f', g'\rangle$ was generated by the construction of Proposition 3.24, i.e. every element of J is a valid equivalence class of $\overline{\sim}$, an equivalence on the union of H and I.

Assume there are two $z, z' \in I$ with $z \neq z'$ where $f'(z) = f'(z')$, i.e. f' is non-injective. By construction $z \mathrel{\overline{\sim}} z'$, where $\overline{\sim}$ is the equivalence closure of $\sim$, which is defined as $f(x) \sim g(x)$ and $g(x) \sim f(x)$ for all $x \in G$ (where ever $f(x)$ and $g(x)$ are defined). We

can assume that z has a preimages in G, since otherwise z is not related to anything via $\sim$ and therefore $[z]_{\approx} = \{z\}$. Obviously the same holds for z'. Since $z \approx z'$, there is a sequence $g(x_1) \sim f(x_1) = f(y_1) \sim g(y_1) = g(x_2) \sim \ldots \sim g(y_n)$ where $g(x_1) = z$ and $g(y_n) = z'$. Every $f(x_i)$ and $f(y_i)$ is defined, since otherwise the equivalence class of z and z' would we invalid and their image under f' would be undefined. Furthermore, since f is injective, $f(x_i) = f(y_i)$ implies $x_i = y_i$. Thus we obtain $g(x_i) = g(y_i)$ and $g(y_i) = g(x_{i+1})$ which finally implies $z = g(x_1) = g(y_n) = z'$. Hence our assumption that $z \neq z'$ was wrong and f' must be injective. □

Lemma 3.32. *Let $f\colon G \rightharpoonup H$ be a injective and surjective, partial morphism and let $g'\colon H \to J$ be a total morphism. Every pushout complement $\langle I, g, f'\rangle$ with $g\colon G \to I$ and $f'\colon I \rightharpoonup J$ where g is conflict-free wrt. f can be obtained by the construction of Proposition 3.30.*

Proof. By Lemma 3.31 we know that f' must be injective. Furthermore, epimorphisms – partial, surjective morphisms in Λ-**HGp** – are preserved by pushouts in the same sense, thus f' is surjective as well. Effectively I contains a copy of J. We will now show that a pushout complement $\langle \hat{I}, \hat{g}, \hat{f}'\rangle$ exists which is isomorphic to I and which can be generated by the given construction. We prove this by giving an isomorphism $k\colon I \rightarrowtail \hat{I}$ such that the triangles $\hat{g} = k \circ g$ and $f' = \hat{f}' \circ k$ commute, as shown in Figure B.5.

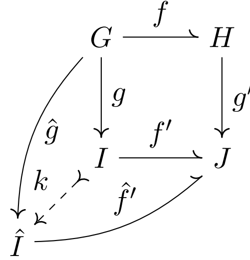


Figure B.5.: Pushout complement diagram for the proof of Lemma 3.32

Let $v \in V_I$ be some node of I. If $f'(v)$ is defined, then v belongs to the copy of J generated in step 1. If $f'(v)$ is undefined, then v must have a preimage in G, since otherwise, when computing the pushout of f and g, the equivalence class of v is valid and $f'(v)$ is defined. By commutativity and since g is conflict-free wrt. f, every preimage of v must be undefined under f. Thus we can obtain v by taking copies of element of G for which f is undefined and then merge these copies appropriately. The same argument also holds for edges, with the exception that edges need not have a preimage in G if they have no image under f'. However, according to Proposition 3.24 such an edge e is deleted only if at least one of its incident nodes is deleted. Thus, such an edge will be

added in step 3 of the construction. Based on this we can define k as follows:

$$k(x) = \begin{cases} \hat{g}(x') & \text{if } x = g(x') \text{ for some } x' \in G \\ x' & \text{if } \hat{f}'(x') = f'(x) \text{ and } f'(x) \text{ is defined} \\ x_{cp} & \text{if } f'(x) \text{ is undefined and there is no } x' \text{ with } g(x') = x \end{cases}$$

where x_{cp} denotes an edge added in step 3.

Well-definedness. Let $\equiv$ be the equivalence on $\tilde{J}$ used in step 2 of the construction. To obtain $\hat{I}$ we use the equivalence where $x \equiv y$ if $x = y$ or $x = x'_c$, $y = y'_c$, i.e. x and y are copies of elements x' and y' of G, and $g(x') = g(y')$. Clearly this equivalence satisfies the necessary conditions, since $\tilde{f}'(x)$ and $\tilde{f}'(y)$ are undefined if x and y are copies of elements of G. This equivalence can now be used to show the well-definedness of k in the first case. Since g is conflict-free wrt. f, either all preimages x', x'' of x are undefined or all are defined under f. If all are undefined, then the images of x' and x'' belong to the same equivalence class $\hat{g}(x') = \hat{g}(x'')$. If f is defined for all preimages, then $g' \circ f$ is defined as well, and by commutativity $f' \circ g$ as well. By definition of $\hat{g}$ we obtain:

$$\begin{aligned} g(x') = g(x'') \quad &\Longrightarrow \quad f'(g(x')) = f'(g(x'')) \quad \Longrightarrow \quad g'(f(x')) = g'(f(x'')) \\ &\Longrightarrow \quad \hat{f}'(\hat{g}(x')) = \hat{f}'(\hat{g}(x'')) \quad \Longrightarrow \quad \hat{g}(x') = \hat{g}(x'') \end{aligned}$$

The well-definedness of the second case of k follows form the fact, that $\hat{f}'$ is injective and surjective, i.e. there is exactly one x' with $\hat{f}'(x') = f'(x)$ for every x for which $f'(x)$ is defined. In the third case of k, x is an edge connected to at least one node v for which $f'(v)$ is undefined. Thus, we can assume that some x_{cp} was added in step 3 with $l_{\hat{I}}(x_{cp}) = l_I(x)$ and $c_{\hat{I}}(x_{cp}) = k(c_I(x))$. Finally, if an $x \in I$ has a preimage in $x' \in G$ and $f'(x)$ is defined (i.e. the first and second case both apply), then by commutativity $f'(g(x')) = g'(f(x')) = \hat{f}'(\hat{g}(x'))$ holds. Because of injectivity, $\hat{g}(x')$ is the only preimage of $f'(x)$ under $\hat{f}'$. Obviously k satisfies the morphism property (since it is defined using other morphisms) and thus k is well-defined.

Isomorphism. By definition k is total, and we can assume that k is surjective, since we need not add more edges in step 3 than necessary. Furthermore, k is obviously injective in case 2 and 3 of its definition, since f' and $\hat{f}'$ are injective. In the first case $k(x) = k(y)$ implies $\hat{g}(x') = \hat{g}(y')$ for some $g(x') = x$ and $g(y') = y$. Since $\hat{g}(x') = \hat{g}(y')$ can only be the case if $g(x') = g(y')$ (regardless of whether x' and y' have images under f), we have $x = g(x') = g(y') = y$.

Commutativity. The commutativity $\hat{g}(x) = k(g(x))$ holds by definition, since $g(x)$ obviously has a preimage in G (case one). It remains to be shown that $\hat{f}' \circ k = f'$. Let $x \in I$ such that $f'(x)$ is defined, then by definition $\hat{f}'(k(x)) = f'(x)$ holds (case two). If $x \in I$ and $f'(x)$ is undefined, then either $k(x) = x_{cp}$ and $\hat{f}'(x_{cp})$ is undefined by definition (case three), or x has a preimage $x' \in G$. Due to the commutativity

$f'(g(x')) = g'(f(x')) = \hat{f}'(\hat{g}(x'))$, the morphism $\hat{f}'$ must be undefined on $\hat{g}(x') = k(x)$ (since $\hat{g}$ is total).

Since $\hat{I}$ and I are isomorphic, I is in fact a pushout complement computed by the construction. □

The construction of pushout complements for more general (partial) morphisms is proven in the following.

Proposition 3.33 (Pushout complements in Λ-HGp II). *Let $f\colon G \rightharpoonup H$ be a partial morphism and let $g'\colon H \rightarrow J$ be a total morphisms, as shown in Figure 3.15. We can construct every pushout complement I' with morphisms $k\colon G \rightharpoonup I$ and $f'\colon I \rightharpoonup J$ where k is conflict-free wrt. f as follows:*

1. *Split f into two morphisms $f_1\colon G \rightharpoonup G'$ and $f_2\colon G' \rightarrow H$ with $f = f_2 \circ f_1$ where f_1 is injective and surjective, and f_2 is total.*
2. *Use the construction of Proposition 3.26 to compute $\langle I', g, f_2' \rangle$, a pushout complement of f_2, g' with $g\colon G' \rightarrow I'$ and $f_2'\colon I' \rightarrow J$.*
3. *Use the construction of Proposition 3.30 to compute $\langle I, k, f_1' \rangle$, a pushout complement of f_1, g with $k\colon G \rightarrow I$ and $f_1'\colon I \rightharpoonup I'$.*
4. *We define f' as the composition $f' = f_2' \circ f_1'$.*

This construction will generate finitely many pushout complements if and only if the construction of Proposition 3.30, will compute finitely many pushout complements.

Proof. First, since the left and the right square are both pushouts, by Lemma 3.7 the outer rectangle is a pushout as well. Thus $\langle I, k, f' \rangle$ is in fact a pushout complement of $f_2 \circ f_1$ and g'. Note that g is conflict-free wrt. f_2 and k is conflict-free wrt. $f_2 \circ f_1$, since otherwise g' would be partial.

It remains to be shown that every pushout complement of f and g' can be obtained in this way, so let $\langle I, k, f' \rangle$ be such a pushout complement. Every f can be split into f_1 and f_2 satisfying the necessary restrictions. We generate G' by taking a copy of G and deleting every element which has no image under f. The morphism f_1 is the identity for every element for which f is defined, and undefined else. We define f_2 as $f_2(x) = f(x)$ for all $x \in G'$. Obviously f_1 is injective and surjective, and f_2 is total. According to Lemma 3.8 there is a unique way to split f' into morphisms f_1' and f_2' such that $\langle I', f_1', g \rangle$ is the pushout of k, f_1 and $\langle J, f_2', g' \rangle$ is the pushout of g, f_2. We observe that k is conflict-free wrt. f_1, since otherwise k could not be conflict-free wrt. f, and thus g is total. The pushout complements I' and I will be generated by the constructions of Propositions 3.26 and 3.30 respectively, as proven by the Lemmas 3.29 and 3.32, and therefore $\langle I, k, f' \rangle$ is generated by this construction.

The split in step 1 is unique and the number of pushout complements for two total morphisms (computed in step 2) is always finite. Thus, we will only obtain infinitely many pushout complements if step 3 generates infinitely many pushout complements, i.e. the construction of Proposition 3.30. □

Appendix C
Proofs of Chapter 6

This appendix contains proofs of Sections 6.1, 6.2 and 6.6 which where moved here to ease the reading of those sections.

C.1. Proofs of Section 6.1

Here we prove the general property that conflict-free matches ensure total co-matches. In fact, a match is conflict-free if and only if the co-match is total. The fact that injective matches ensure injective co-matches already follows from the categorical framework we use.

Lemma 6.5. *Let $r\colon L \rightharpoonup R$ be a partial and $m\colon L \rightarrow G$ a total morphism. Furthermore, let $\langle H, r', m'\rangle$ with $r'\colon G \rightharpoonup H$ and $m'\colon R \rightharpoonup H$ be the pushout of r and m. If m is injective, then m' is total and injective as well. If m is conflict-free wrt. r, then m' is total.*

Proof. In any category monomorphisms, the generalised notion of injectivity, are preserved by pushouts in the sense that m' is a monomorphism if m is a monomorphism [LS04]. Note that in Λ-**HGp** a morphism is a monomorphism if and only if it is total and injective. Since injectivity implies conflict-freeness, we just have to show that m' is total if m is conflict-free.

Assume there is an element $x \in R$ for which $m'(x)$ is undefined. Without loss of generality we can assume that H was constructed according to Proposition 3.24 using the relation $\sim$ (where $r(x) \sim m(x)$ and vice versa for all $x \in L$). By definition $m'(x)$ is only undefined if the equivalence class of x is not valid, i.e there is an y with $x \overline{\approx} m(y)$ and $r(y)$ is undefined (note that m is total). Furthermore, there has to be an x' with $r(x') = x$ since elements in R without preimages in L have valid equivalence classes. Hence there

is a sequence $r(x_1) \sim m(x_1) = m(x_2) \sim r(x_2) = r(x_3) \sim \ldots \sim m(x_{n-1}) = m(x_n)$ with $x_1 = x'$ and $x_n = y$, where all $r(x_i)$ for $i \neq n$ are defined (otherwise we could shorten this sequence to satisfy this condition). This means that there are x_{n-1}, x_n with $m(x_{n-1}) = m(x_n)$ where $r(x_{n-1})$ is defined, but $r(x_n)$ is not. Thus, m is not conflict-free. □

C.2. Proofs of Section 6.2

In this section we prove that minor morphisms are preserved by pushouts along total morphisms and pushout closed. Furthermore, we prove that our algorithm for computing represented injective predecessors is correct.

Lemma 6.20. *Minor morphisms are preserved by pushouts along total morphisms (cf. Definition 6.3).*

Proof. Let G_3 be the pushout of G_0 along $\mu\colon G_0 \mapsto G_1$ and $g\colon G_0 \rightarrow G_2$, as shown in Figure C.1. Without loss of generality we can assume that G_3 was computed by the the construction we gave in Proposition 3.24. Let $v, w \in V_{G_2}$ be two nodes that are mapped to the same node $z \in V_{G_3}$ via the morphism $\mu'\colon G_2 \rightharpoonup G_3$. This means that v and w are in the same equivalence class, and thus necessarily have preimages v' and w' in G_0.

$$\begin{array}{ccc} G_0 & \overset{\mu}{\longmapsto} & G_1 \\ \downarrow g & & \downarrow g' \\ G_2 & \overset{\mu'}{\longmapsto} & G_3 \end{array}$$

Figure C.1.: Diagrams illustrating the proof of Lemma 6.20

Now, since they are in the same equivalence class, there exists a sequence of nodes $y_1, y_2, \ldots, y_n \in V_{G_0}$ with $y_1 = v'$ and $y_n = w'$ such that $\mu(y_i) = \mu(y_{i+1})$ for every odd i and $g(y_j) = g(y_{j+1})$ for every even j. Since μ is a minor morphism there exists a path (in G_0) from y_i to y_{i+1} for every odd i such that all nodes on the path are mapped to $\mu(y_i)$. Since g is total, there also exists a path in G_2 from $g(y_i)$ to $g(y_{i+1})$ for every odd i. Now, due to commutativity, all nodes on such a path (in G_2) will be mapped to the same node in G_3. Also due to commutativity, the images of all the edges in this path are undefined, since the equivalence class is not valid (due to μ being a minor morphism). Furthermore, since $g(y_j) = g(y_{j+1})$ for even j, there is a path from $g(y_1) = v$ to $g(y_n) = w$ such that all nodes on that path are mapped to the same node $z \in V_{G_3}$, and none of the edges in this path lie in the domain of μ'. Also, surjectivity (on nodes and edges) and injectivity (on edges) is preserved by the pushout construction.

Thus, μ' is a minor morphism. □

Lemma 6.21. *Minor morphisms are pushout closed (cf. Definition 6.11).*

Proof. We will prove that minor morphisms are pushout closed by first constructing R', G' together with minor morphisms $\mu_R\colon R \mapsto R'$ and $\mu_G\colon G \mapsto G'$. We then prove that S is the pushout of $\mu_R \circ r$, $\mu_G \circ m$ and that $\mu_G \circ m$ is conflict-free wrt. r. So first, let the morphisms r, r', m, m', μ be given as shown in Figure C.2 such that the square is a pushout.

Construction of R' and μ_R. From R, construct a minor R' (and simultaneously a minor morphism μ_R) as follows: first, let R' be simply a copy of R. For $e \in E_R$, if the image of e in H under m' is contracted to construct S, then contract the corresponding edge in R' in the same way, i.e. according to the same partition. In this case e is undefined under μ_R and its incident nodes are mapped to merged nodes in R'. If e is deleted (without contracting the nodes), delete it in R' as well, and leave e undefined under μ_R. Now, let $v \in V_R$ be such that its image in H is deleted in constructing S. This implies that the image of v in H is either an isolated node, or all its incident edges were deleted. So since we deleted corresponding edges in R', we can safely delete v in R', and leave it undefined under μ_R. (Note that it is not possible for R to have an edge that is not mapped to an edge in H, since m is total and conflict-free and hence m' must be total).

Now, R' is a minor of R, because the construction involved only the allowed operations. Further, due to its construction, μ_R is a minor morphism.

Construction of G' and μ_G. Perform a similar construction for G', with one difference: For $x \in G$ such that x has a preimage in L, do not contract/delete it in G', even if x had an image in H that was contracted/deleted in constructing S. (The intuition for this is that it is enough for an item to be contracted/deleted by one of the minor morphisms for it to be contracted/deleted in the pushout graph and a second deletion/contraction would cause the match to be non-conflict-free.) The rest of the construction is as before. Again, G' is a minor of G and $\mu_G\colon G \mapsto G'$ is a minor morphism. Further $\mu_G \circ m$ is total since m is total and μ_G is defined for all elements with a preimage in L.

Conflict-freeness of $\mu_G \circ m$. Also, $\mu_G \circ m$ is conflict-free with respect to r. To see this, suppose there exist nodes $v_1, v_2 \in L$ such that $(\mu_G \circ m)(x_1) = (\mu_G \circ m)(x_2)$. Whenever x_1, x_2 are edges, then $m(x_1) = m(x_2)$, since μ_G does not merge edges. In this case by assumption r is either undefined on both or defined on both. A similar argument applies whenever x_1, x_2 are nodes and $m(x_1) = m(x_2)$. So now assume that x_1, x_2 are nodes and $y_1 = m(x_1) \neq m(x_2) = y_2$. Then, $\mu_G(y_1) = \mu_G(y_2)$ implies that y_1 and y_2 are nodes and have distinct images in H with a path connecting them which is contracted while constructing S. Hence, $r(x_1)$ and $r(x_2)$ are both defined and distinct. Thus there cannot be a deletion/preservation conflict.

Construction of n. Now, we construct the morphisms $n\colon R' \to S$ and $s\colon G' \rightharpoonup S$ (see Figure C.2). For any $x \in R$ we define n as $n(\mu_R(x)) = \mu(m'(x))$. To see that this is

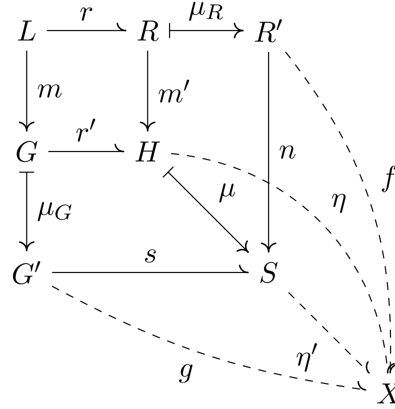


Figure C.2.: Shows how the small pushout square can be extended to the large pushout square; morphisms used in the correctness proofs are dashed

valid, first note that if $\mu_R(x)$ is undefined, then $\mu(m'(x))$ will also be undefined because of the construction of μ_R. Then, if there exist x_1, x_2 such that $\mu_R(x_1) = \mu_R(x_2)$, then x_1 and x_2 must be nodes and not edges, because μ_R is injective on edges. And we must have $\mu(m'(x_1)) = \mu(m'(x_2))$. Thus the above definition is valid.

Further, m' is total (since m is total and conflict-free), $\mu(m'(x))$ is undefined if and only if μ is undefined at $m'(x)$, and in that case $\mu_R(x)$ will also be undefined. This, combined with the fact that μ_R is surjective, implies that n is total. Also, the relevant part of the diagram commutes, due to the definition of n.

Construction of s. Similarly, s is defined as $s(\mu_G(x)) = \mu(r'(x))$. By essentially the same argument as in the case of n, we can show that this definition is valid and the relevant part of the above diagram commutes. However, in this case s may be partial, because for a node x with a preimage in L, $r'(\mu(x))$ may be undefined but $\mu_G(x)$ will still be defined. For such a node, s will be left undefined.

Commutativity and correctness of the outer square. Furthermore it can be straightforwardly shown that n, s satisfy all properties of a morphism. And finally, since each of the parts of the diagram in Figure C.2 commute, the diagram as a whole also commutes.

Existence of the mediating morphism η'. Now, let there be some graph X and two morphisms $f\colon R' \rightharpoonup X$ and $g\colon G' \rightharpoonup X$, such that $(f \circ \mu_R) \circ r = (g \circ \mu_G) \circ m$, then, since H is a pushout, there exists $\eta\colon H \rightharpoonup X$ such that η is the unique morphism with $f \circ \mu_R = \eta \circ m'$ and $g \circ \mu_G = \eta \circ r'$. For $x \in H$, define a morphism $\eta'\colon S \rightharpoonup X$ as follows $\eta'(\mu(x)) = \eta(x)$. Since μ is surjective, every element $y \in S$ has a preimage $x \in H$, hence η' can in principle be defined for every such y by the above definition. The current situation is depicted in Figure C.2.

Correctness and uniqueness of η'. It is left to show that η' is well-defined and unique. Now, if there exist x_1, x_2 in H such that $\mu(x_1) = \mu(x_2)$, then x_1 and x_2 must be nodes (since μ is a minor morphism), and further, there must be a path connecting them, such that all nodes on this path are also mapped to the same node. Then, if $\eta(x_1) \neq \eta(x_2)$, there exist y_1, y_2 which lie on this path from x_1 to x_2, such that they are adjacent, and $\mu(y_1) = \mu(y_2)$ but $\eta(y_1) \neq \eta(y_2)$. Let e be the edge connecting them. It must have a preimage in either R or G (or both). Suppose e has a preimage e' in R with the preimages of y_1 and y_2 being y_1' and y_2' respectively. Then, if e is contracted in S it holds that e' is contracted in R', and hence $\mu_R(y_1') = \mu_R(y_2')$. But then, $f \circ \mu_R = \eta \circ m'$ implies that $\eta(y_1) = \eta(y_2)$, which leads us to a contradiction. Now, suppose e has a preimage e'' in G instead of R. If e'' does not have a preimage in L, then we arrive at a contradiction by a similar argument as before. On the other hand, if e'' has a preimage in L, then e must have a preimage in R (since H is a pushout), hence the previous argument applies. Hence, such x_1, x_2 cannot exist, and η' is well-defined.

$$\begin{array}{ccc} L & \xrightarrow{\mu_R \circ r} & R' \\ {\scriptstyle \mu_G \circ m}\downarrow & & \downarrow{\scriptstyle n} \\ G' & \xrightarrow[s]{} & S \end{array}$$

Figure C.3.: The outer square of Figure C.2 is a pushout

This gives us an $\eta' \colon S \rightharpoonup X$ such that $\eta' \circ \mu = \eta$. This implies $f \circ \mu_R = (\eta \circ \mu) \circ m'$ and $(g \circ \mu_G) = (\eta \circ \mu) \circ r'$. Now η and therefore $\eta' \circ \mu$ is the *unique* morphism with this property. Since μ is fixed and surjective, this means that η' is the unique morphism such that $f = n \circ \eta'$ and $g = s \circ \eta'$. Thus, the diagram in Figure C.3 is a pushout.

As shown before, n and $\mu_G \circ m$ are both total, and $\mu_G \circ m$ is conflict-free with respect to r. □

Lemma 6.25. *The set $reps_{\mathcal{Q}}^{\sqsubseteq}(r, m)$ computed by Algorithm 6.24 satisfies the conditions of Definition 6.17.*

Proof. Let $r \colon L \rightharpoonup R$ be a prepared rule, let $m \colon L \to G$ be a match and let S be the pushout of r, m. First we observe that the parameters of Algorithm 6.24 match the requirements of Definition 6.17 and that $reps_{\mathcal{Q}}^{\sqsubseteq}(r, m) \subseteq \mathcal{Q}$ obviously holds due to line 10. Moreover, termination is guaranteed mainly by the condition in lines 9 and 12. On the one hand, there are only finitely many graphs G'' and morphisms μ''. On the other hand, line 12 ensures that only those m'' are kept that have less pairs of non-injectively matched nodes than m'. This means that at some point any m'' will be injective and no more matches will be added in line 16. Note that for any pair $\langle G'', \mu'' \rangle$ the number of possible m'' is finite.

First condition. We now prove the first condition of Definition 6.17, i.e. that for every $G' \in reps_{\mathcal{Q}}^{\sqsubseteq}(r,m)$ there are $r_1\colon L \rightharpoonup R'$, $r_2\colon R' \mapsto R$, $m'\colon L \rightarrowtail G'$, $\mu\colon G' \mapsto G$ with $r_1 \in origin(r)$ such that $m = \mu \circ m'$ and $r = r_2 \circ r_1$, and for the pushout H' of m', r_1 it holds that $S \preceq H'$. The existence and equality $r = r_2 \circ r_1$ is obvious, since every prepared rule is generated by composing an original rule with an order morphism. By definition every G' computed by the procedure is calculated in n steps by splitting the match into a match and a minor morphism as shown in Figure C.4. Due to the condition in line 11 $m_i = \mu_{i+1} \circ m_{i+1}$ holds for $0 \leq i \leq n-1$ where $m_0 = m$. Thus, we can split m into $m' = m_n$ and $\mu = \mu_1 \circ \ldots \circ \mu_n$.

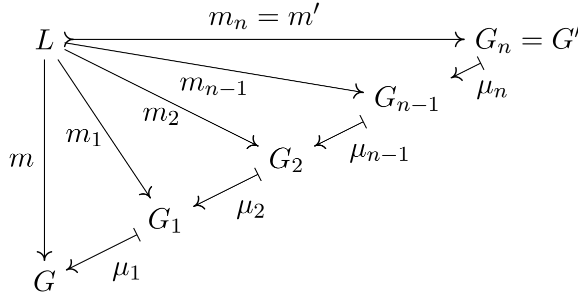


Figure C.4.: Diagram illustrating the computation of m' and μ by Algorithm 6.24

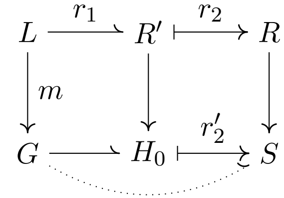
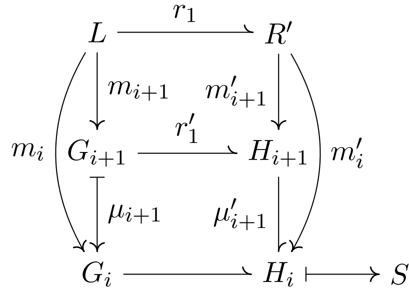


Figure C.5.: Diagram proving $S \sqsubseteq H_0$

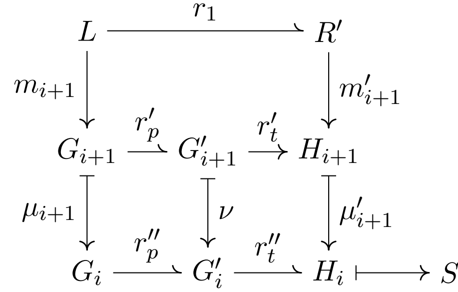
We now show that S is a minor of the pushout of $m' = m_n$ and r_1 by induction over n. By an argument we already used in the proof of Lemma 6.13, S is a minor of the pushout H_0 of m, r_1, as shown in Figure C.5. Due to pushout properties the morphism $r_2'\colon H_0 \mapsto S$ exists and is a minor morphism, since the right square is a pushout and all matches are total.

Now let H_i be the (outer) pushout of r_1, m_i and let H_{i+1} be the (upper) pushout of r_1, m_{i+1} with $m_i = \mu_{i+1} \circ m_{i+1}$, as shown in Figure C.6a. Due to commutativity a unique morphism $\mu'_{i+1}\colon H_{i+1} \rightharpoonup H_i$ exists and due to pushout properties (see Lemma 3.7) the lower square is also a pushout. By proving that μ'_{i+1} is a minor morphisms, we will show that $S \sqsubseteq H_{i+1}$ holds ($S \sqsubseteq H_i$ holds by induction hypothesis). As shown in Figure C.6b we split r_1' into $r_p'\colon G_{i+1} \rightharpoonup G'_{i+1}$ and $r_t'\colon G'_{i+1} \rightarrow H_{i+1}$ where r_p' is injective, surjective and defined for an x if an only if $r_1'(x)$ is defined. Note that this guarantees that r_t' is total. We then form the pushout G_i' from which due to commutativity a morphisms to H_i exists such that the right lower square is a pushout.

We now show that ν is a minor morphism. First assume there is an element $x \in G_i'$ without preimage under ν. Due to pushout properties x must have a preimage in G_i. Since μ_{i+1} is surjective, x has a preimage under $r_p'' \circ \mu_{i+1}$ and due to commutativity also needs to have a preimage in G'_{i+1}. Now assume there are two edges $e_1, e_2 \in G'_{i+1}$ with $\nu(e_1) = \nu(e_2)$. Then e_1, e_2 have to have preimages $e_1', e_2' \in G_{i+1}$, otherwise the



(a) Commuting diagram existing due the defintion of Algorithm 6.24



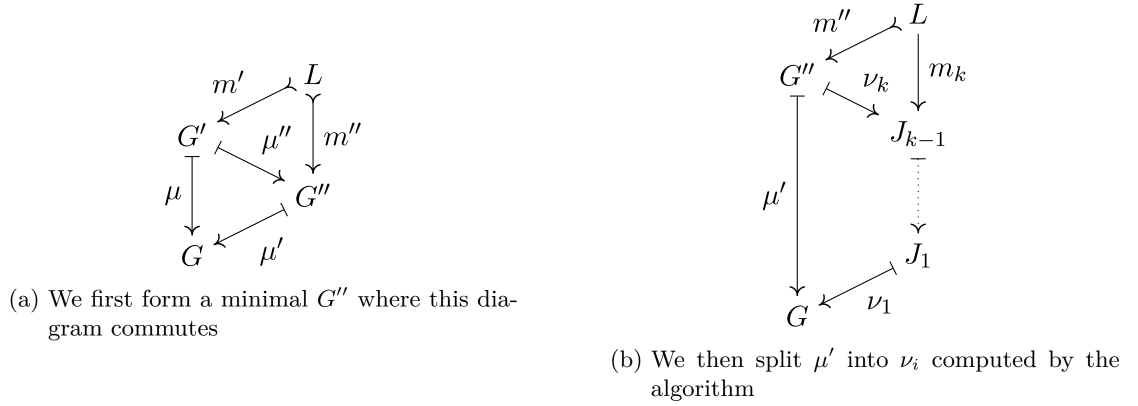
(b) Spliting r'_1 into a partial r'_p and a total r'_t splits the lower pushout into two

Figure C.6.: Diagram proving $S \sqsubseteq H_{i+1}$ if $S \sqsubseteq H_i$

diagram would not be a pushout. Since the injectivity of r'_p implies injectivity of r''_p (see Lemma 3.31), this means that $\mu_{i+1}(e'_1) = \mu_{i+1}(e'_2)$ would have to hold, which leads to a contradiction. Finally let $x_1, x_2 \in G'_{i+1}$ be nodes with $\nu(x_1) = \nu(x_2)$. Again this can only be the case if these nodes have preimages $x'_1, x'_2 \in G_{i+1}$ under r'_p and $\mu_{i+1}(x'_1) = \mu_{i+1}(x'_2)$. This means that there is a path $v_0, e_1, \ldots, e_k, v_k$ in G_{i+1} with $v_0 = x'_1$ and $v_k = x'_2$. All $r'_p(e_j)$ (for $1 \leq j \leq k$) are defined, since any e_j without image under r'_p must have a preimage in L due to the upper rectangle being a pushout. But this would mean that there is an element in L for which $\mu_{i+1} \circ m_{i+1} = m_i$ is undefined! Thus the path in G_{i+1} used to contract x'_1, x'_2 also exists in G'_{i+1} and can be used to contract x_1, x_2. Thus, ν is a minor morphism and, since r'_t is total, μ'_{i+1} is a minor morphism as well due to Lemma 6.20. Hence we obtain $S \sqsubseteq H_{i+1}$ if $S \sqsubseteq H_i$.

Second condition. We now prove the second condition of Definition 6.17, i.e. that every graph satisfying the first condition is represented by an element of $reps^{\sqsubseteq}_{\mathcal{Q}}(r, m)$. So let G' be such a graph with an injective match $m' \colon L \rightarrowtail G'$ and a minor morphism $\mu \colon G' \mapsto G$. We first compute a minimal graph G'' with morphisms as shown in Figure C.7a such that the diagram commutes. With minimal we mean that any further split of μ' into $\nu \circ \nu' = \mu'$ will cause $\nu' \circ m''$ to be non-injective. We now prove that G'' is computed by Algorithm 6.24 by splitting μ' into minor morphisms that satisfy the conditions of lines 9 and 12.

So first split μ' into morphisms $\mu_i \colon G_{i-1} \mapsto G_i$ (for $1 \leq i \leq n$) with $G = G_0$, $G'' = G_n$ and where every μ_i deletes a single node or contracts a single edge. No μ_i can simply delete an edge, since this edge can not have a preimage in L ($\mu' \circ m''$ would not be total) and can thus already be deleted by μ'', contradicting the minimality of G''. By the same argument every node deleted by an μ_i must be incident (in G'') to an edge contracted by some μ_j, since it would be isolated otherwise. Hence we can form minor morphism $\nu_i \colon J_{i-1} \mapsto J_i$ (for $1 \leq i \leq k$) with $J_0 = G$, $J_k = G''$ as shown in Figure C.7b where



(a) We first form a minimal G'' where this diagram commutes

(b) We then split μ' into ν_i computed by the algorithm

Figure C.7.: Shows the basic diagrams for proving the second condition of Definition 6.17

we compose each μ_i which deletes a node with the first μ_j introducing an edge incident to the deleted node. It is obvious that every ν' satisfies the conditions of line 9, so we only need to show the existence of nodes $v_3, v_4 \in G''$ with preimages $v_1, v_2 \in L$ and $\nu_k(v_3) = \nu_k(v_4)$. This is straight-forward, since $\nu_k \circ m'' = m_k$ would be injective if there are no such nodes, a contradiction to G'' being minimal. This argument also holds for all other ν_i, since the order on edge contractions is arbitrary, i.e. if the property does not hold for some ν_i, we can give a different decomposition of μ' where it does not hold for ν_k, doing the problematic edge contraction in the first step.

Thus for every G' Algorithm 6.24 will successively compute, beginning with G, every ν_i until finally obtaining and storing a G'' with $G'' \sqsubseteq G'$. ☐

C.3. Proofs of Section 6.6

This section contains all proofs regarding universally quantified rules. This includes the proofs that instantiation steps can be swapped, $bound_\rho$ exists, our $(\mathcal{G}_n)$-pred-basis is correct and the miscellaneous proofs about optimizations.

Lemma 6.52. *Let $\rho = \langle r, U\rangle$ be a rule and let $f\colon U \to \mathbb{N}_0$ be any function assigning a quantity to each q-pair. Every instantiation of ρ which is generated by using $f(u)$ occurrences for each u respectively, yields the same morphisms (up to isomorphism).*

Proof. We prove this property by showing that we can swap each two instantiation steps without changing the instantiation containing both steps. Let $\iota = \langle \pi\colon L \rightarrowtail \overline{L}, \gamma\colon \overline{L} \rightharpoonup \overline{R}\rangle$ be an instantiation of some rule $\rho = \langle r\colon L \rightharpoonup R, U\rangle$ and let $u = \langle p_u\colon L \rightarrowtail L_u, q_u\colon L_u \rightharpoonup R_u\rangle, v = \langle p_v\colon L \rightarrowtail L_v, q_v\colon L_v \rightharpoonup R_v\rangle \in U$ be two q-pairs as shown in

Figure C.8. There the upper part of the diagram is the instantiation $\iota_v = \iota \diamond v$, while the front part of the diagram is the instantiation $\iota_u = \iota \diamond u$.

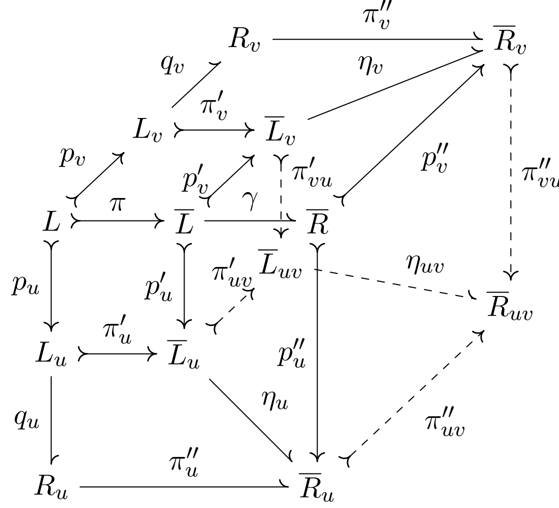


Figure C.8.: Extending an instantiation with $\langle p_v, q_v\rangle$ and $\langle p_u, q_u\rangle$ leads to the same final instantiation, regardless of order

Let $\overline{L}_{uv}$ be the pushout of p'_u, p'_v and let $\overline{R}_{uv}$ be the pushout of p''_u, p''_v. By the properties of pushouts a unique η_{uv} exists and we will show that $\iota_u \diamond v = \langle \pi'_{vu} \circ \pi'_v \circ p_v, \eta_{uv}\rangle = \iota_v \diamond u$.

By construction all small squares $p'_u \circ \pi = \pi'_u \circ p_u$, $p'_v \circ \pi = \pi'_v \circ p_v$ and $\pi'_{vu} \circ p'_v = \pi'_{uv} \circ p'_u$ are pushouts. Therefore, the squares $\pi'_{vu} \circ \pi'_v \circ p_v = \pi'_{uv} \circ p'_u \circ \pi$ and $\pi'_{vu} \circ p'_v \circ \pi = \pi'_{uv} \circ \pi'_u \circ p_u$ are pushouts as well. Thus, $\overline{L}_{uv}$ is the pushout of p_v, $p'_u \circ \pi$ computed in the construction of $\iota_u \diamond v$ as well as the pushout of p_u, $p'_v \circ \pi$ computed in the construction of $\iota_v \diamond u$. The same property holds for $\overline{R}_{uv}$ using the three large outer squares. Since η_{uv} is unique, both sequences of the instantiation steps give rise to the same morphisms. This means that every instantiation can be uniquely characterized only by the number on instantiation steps for each $u \in U$. □

To prove that *bound*$_\rho$ exists, we first prove the two auxiliary Lemmas C.1 and C.2. These lemmas prove that instantiations of greater length will produce larger pushout complements. In the proof of Proposition 6.54 this is then used to show that pushout complements of sufficiently large instantiations are already represented by smaller instantiations.

Lemma C.1. *Let $\rho = \langle r, U\rangle$ be a rule and let $\langle \pi : L \rightarrowtail \overline{L}, \gamma : \overline{L} \rightharpoonup \overline{R}\rangle$ be an instantiation of ρ. For every further instantiation $\langle \pi, \gamma\rangle \diamond u$ using some $u \in U$, there are two subgraph morphisms $\mu'_u : \overline{L}_u \rightarrowtail \overline{L}$ and $\mu''_u : \overline{R}_u \rightarrowtail \overline{R}$ such that $\gamma \circ \mu'_u = \mu''_u \circ \eta$.*

Proof. By definition p_u and $q_u \circ p_u$ are total and injective, thus, p'_u and $q''_u \circ p''_u$ are total and injective as well (see Lemma 6.5). Hence, the reverse morphisms μ'_u and μ''_u are partial, injective and surjective, i.e. subgraph morphisms. By using Lemma 3.7 it can be shown that by forming the pushout $\overline{R}'$ of p'_u and γ, the pushout R'_u of q_u and π' and then the pushout $\overline{R}_u$ of q'_u and γ', we obtain the same graph $\overline{R}_u$ as by forming the pushout of $q_u \circ p_u$ and $\gamma \circ \pi$ directly. Furthermore, the diagram in Figure C.9 commutes with the exception of μ'_u and μ''_u, for which we still have to show the commutativity with η and γ.

$$\begin{array}{ccccc}
L & \xrightarrow{\pi} & \overline{L} & \xrightarrow{\gamma} & \overline{R} \\
\downarrow p_u & & \downarrow p'_u \uparrow \mu'_u & & \downarrow p''_u \uparrow \mu''_u \\
L_u & \xrightarrow{\pi'} & \overline{L}_u & \xrightarrow{\gamma'} & \overline{R}' \\
\downarrow q_u & & \downarrow q'_u & \searrow \eta & \downarrow q''_u \\
R_u & \xrightarrow{\pi''} & R'_u & \xrightarrow{\gamma''} & \overline{R}_u
\end{array}$$

Figure C.9.: When extending γ to η, there are commuting subgraph morphisms μ'_u, μ''_u

Let $x \in \overline{L}_u$ and assume $\gamma(\mu'_u(x))$ is defined. This means that there is exactly one $x' \in \overline{L}$ with $p'_u(x') = x$ and $\gamma(x')$ is defined. Since $q''_u \circ p''_u$ is total and injective, there is an $x'' \in \overline{R}_u$ with $q''_u(p''_u(\gamma(x'))) = x''$. Due to commutativity of the diagram we obtain $\eta(x) = x''$. Hence, we know that $\mu''_u(\eta(x))$ is defined and $\gamma(\mu'_u(x)) = \mu''_u(\eta(x))$.

Now assume $\mu'_u(x)$ is defined, but $\gamma(\mu'_u(x))$ is undefined. Because of commutativity, $\gamma'(x)$ is undefined as well and therefore also $\eta(x) = q''_u(\gamma'(x))$ is undefined. Thus, $\gamma(\mu'_u(x)) = \mu''_u(\eta(x))$ are both undefined.

Now assume $\mu'_u(x)$ is undefined. If $\eta(x)$ is undefined, $\gamma(\mu'_u(x)) = \mu''_u(\eta(x))$ are obviously both undefined, so we assume that $\eta(x)$ is defined and show that $\eta(x)$ has no preimages under $q''_u \circ p''_u$. For this we only have to consider elements in $\overline{R}'$ which have preimages in $\overline{L}_u$, since an element without a preimage and mapped to $\eta(x)$ would violate the pushout property of the lower right square. We observe that $\gamma'(x)$ has no preimage in $\overline{R}$, since the top right square would not be a pushout. In fact this holds for every $x' \in \overline{L}_u$ with $\eta(x') = \eta(x)$ if x' has no preimage in $\overline{L}$. By the same argument we also know that $\gamma'(x')$ has exactly one preimage in $\overline{L}_u$. This means that two x' with and without preimage in $\overline{L}$ are not merged by γ'. By showing that these x' are also not merged by q'_u, we know that their image in the pushout $\overline{R}_u$ would not be equal and prove that there are in fact no x' with preimage in $\overline{L}$.

If x' has a preimage in $\overline{L}$ but not in L_u, then x' is not merged with any other element by q'_u, since the left lower square is a pushout. If x' has a preimage in $\overline{L}$ and L_u, it also has (exactly) one preimage in L, because of the top left square being a pushout. Thus,

by Definition 6.49 q_u may not merge the preimage of x' with anything else, especially not with the preimage of x. Since neither π' nor q_u merge the preimage of x' with anything, x is not merged with anything via q'_u as well. Thus, $\eta(x') = \eta(x)$ cannot hold and $\eta(x)$ has no preimage in $\overline{R}$.

We have shown that $\gamma(\mu'_u(x))$ is undefined if and only if $\mu''_u(\eta(x))$ is undefined, thus the commutativity $\gamma \circ \mu'_u = \mu''_u \circ \eta$ follows from μ'_u being the reverse of p'_u and μ''_u being the reverse of $q''_u \circ p''_u$. □

Lemma C.2. *Let $\rho = \langle r, U \rangle$ be a rule and let $\langle \pi_i : L \rightarrowtail L_i, \gamma_i : L_i \rightharpoonup R_i \rangle$ for $i \in \{1, 2\}$ be two instantiations of ρ with $\langle \pi_2, \gamma_2 \rangle = \langle \pi_1, \gamma_1 \rangle \diamond u$ for some $u \in U$. Furthermore, let $\mu_L : L_2 \rightarrowtail L_1$, $\mu_R : R_2 \rightarrowtail R_1$, $\mu'_R : R_1 \rightarrowtail R$ be subgraph morphisms with $\gamma_1 \circ \mu_L = \mu_R \circ \gamma_2$ and let $m : R \rightarrowtail G$ be a match. For every pushout complement H_2 of $\mu'_R \circ \mu_R \circ \gamma_2$ and m where $m'_2 : L_2 \rightarrowtail H_2$ is total and injective, there is a pushout complement H_1 of $\mu'_R \circ \gamma_1$ and m with $H_1 \subseteq H_2$.*

Proof. We will show this by using the fact, that subgraph morphisms are preserved by total pushouts and successively building the commuting diagram in Figure C.10.

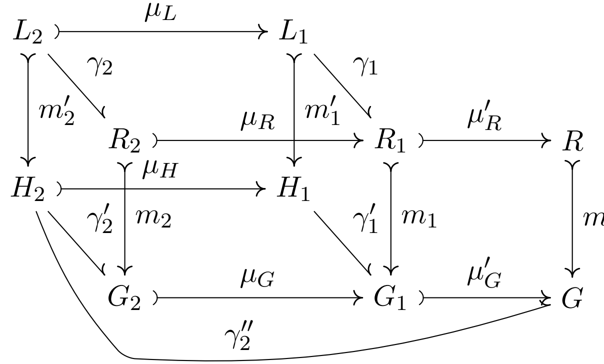


Figure C.10.: Application of γ_2 results in a G_2 with $G_1 \subseteq G_2$ if $\gamma_1 \circ \mu_L = \mu_R \circ \gamma_2$

Let H_2 be a pushout complement of $\mu'_R \circ \mu_R \circ \gamma_2$ and m, where m'_2 is total and injective. We compute the pushout G_2 of γ_2 and m'_2 and then the pushout G_1 of μ_R and m_2. Due to Lemma 3.7, G_1 is also the pushout of $\mu_R \circ \gamma_2$ and m'_2 and therefore there is a unique $\mu'_G : G_1 \rightarrowtail G$ such that the diagram commutes. Since G is the pushout of $\mu'_R \circ \mu_R \circ \gamma_2$ and m'_2, the rightmost square is in fact a pushout as well. Now form the pushout H_1 of μ_L and m'_2. Again the existence of γ'_1 follows from the pushout properties. Since the diagram $m_1 \circ \gamma_1 \circ \mu_L = \gamma'_1 \circ \mu_H \circ m'_2$ commutes with the pushout $m_1 \circ \mu_R \circ \gamma_2 = \mu_G \circ \gamma'_2 \circ m'_2$, it is also a pushout and hence, G_1 is a pushout of γ_1 and m'_1. This means that H_1 is in fact a pushout complement of $\mu'_R \circ \gamma_1$ and m. Since subgraph morphisms are preserved by total pushouts, μ_H is a subgraph morphism. Thus, $H_1 \subseteq H_2$. □

Proposition 6.54. *Let ι be an instantiation of length k of some rule ρ. If k is larger than the number of nodes and edges of G, then every graph computed by the backward application of ι is already represented by the backward application of an instantiation of lower length.*

Proof. Let $\iota_{k-1} = \langle \pi : L \rightarrowtail \overline{L}, \gamma : \overline{L} \rightharpoonup \overline{R} \rangle$ be a rule instantiation of length $k-1$ of $\langle r, U \rangle$ such that $\iota_k = \langle \pi, \gamma \rangle \diamond u$ for some $u \in U$, let $\nu : \overline{R}_u \rightarrowtail R$ be a subgraph morphism and let $m : R \rightarrowtail G$ be a co-match as shown in Figure C.11.

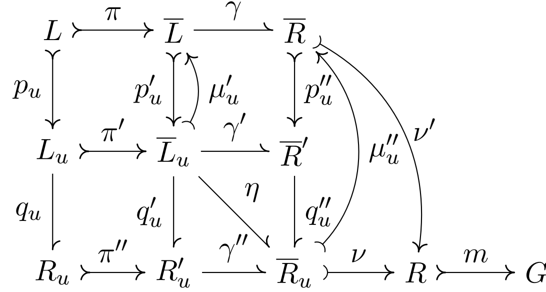


Figure C.11.: If η exceeds the instantiation bound, then a ν' exists, making the backward application of η superfluous

From Lemma C.1 we know that μ'_u and μ''_u exist and the diagram commutes. We will show the existence of a subgraph morphism $\nu' : \overline{R} \rightarrowtail R$ satisfying $\nu = \nu' \circ \mu''_u$. Then from Lemma C.2 it follows that every graph computed by a backward step of $\nu \circ \eta$, the instantiation ι_k, is already represented by a backward step of $\nu' \circ \gamma$, the instantiation ι_{k-1}.

First assume that $\gamma''(\pi''(x_u))$ is undefined for every $x_u \in R_u$ which has no preimage under $q_u \circ p_u$. We can show that $q''_u \circ p''_u$ is a subgraph morphism by showing that it is surjective. Assume there is an $\overline{x}_u \in \overline{R}_u$ without preimage under $q''_u \circ p''_u$. Since the large square is a pushout, there is an $x'_u \in R_u$ with $\gamma''(\pi''(x'_u)) = \overline{x}_u$. By the first assumption x'_u must have a preimage $x \in L$ under $q_u \circ p_u$ for $\gamma''(\pi''(x'_u))$ to be defined. Due to the commutativity, $\gamma(\pi(x))$ is defined and there is a preimage of $\overline{x}_u$ in $\overline{R}$, violating the second assumption. Hence, $q''_u \circ p''_u$ is a subgraph morphism commuting with μ''_u (in fact $\overline{R}$ and $\overline{R}_u$ are isomorphic). The morphism $\nu' = \nu \circ q''_u \circ p''_u$ satisfies the necessary properties.

If at least one q-pair within ι_k satisfies the previous restriction, by Lemma 6.52 we can assume w.l.o.g. that it is the last instantiation step. So assume that for every instantiation step there is at least one $x_u \in R_u$ without preimage under $q_u \circ p_u$ such that $\gamma''(\pi''(x_u))$ is defined. Since $\gamma''(\pi''(x_u))$ has no preimage under $q''_u \circ p''_u$ (otherwise it would have a preimage in L), the graph $\overline{R}_u$ has at least k nodes and edges. Thus, since R has less than k nodes and edges, for at least one instantiation step within ι_k for every $x'_u \in R_u$ without a preimage under $q_u \circ p_u$, the image $\nu(\gamma''(\pi''(x'_u)))$ is undefined. Again

by Lemma 6.52 we can assume w.l.o.g. that it is the last instantiation step of ι_k.

In this case $\nu' = \nu \circ q''_u \circ p''_u$ satisfies the necessary conditions. Obviously ν' is injective and $\nu = \nu' \circ \mu''_u$ holds, so it remains to be shown that it is surjective. Assume there is an $y \in R$ without a preimage under ν'. Since ν is injective and surjective, there is exactly one $\overline{y}_u \in \overline{R}_u$ with $\nu(\overline{y}_u) = y$. Because of commutativity, $\overline{x}_u$ cannot have a preimage under $q''_u \circ p''_u$. Since the outer square is a pushout, there has to be an $y_u \in R_u$ with $\gamma''(\pi''(y_u)) = \overline{y}_u$. By assumption this y_u has a preimage under $q_u \circ p_u$ (otherwise $\nu(\gamma''(\pi''(y_u)))$ would be undefined), which in turn has an image in $\overline{R}$. By commutativity y must have a preimage under ν'. Thus, ν' is surjective and a subgraph morphism. □

Proposition 6.56. *Let $\mathcal{T}$ be a UGTS containing standard rules as well as universally quantified rules. The set uq-preds$^i_{\mathcal{G}_n}(\mathcal{T}, S)$ computed by Algorithm 6.55 is an effective $\mathcal{G}_n$-pred-basis for the transition system $\mathcal{T}^i_{\mathcal{G}(\Lambda)}$ when using the subgraph ordering. Furthermore, uq-preds$^i_{\mathcal{G}(\Lambda)}(\mathcal{T}, S)$ is an effective pred-basis.*

Proof. Let $\mathcal{Q}$ be any downward-closed class of graphs where membership is decidable. We prove this proposition by showing that the Lemmas 6.13 and 6.14 hold for *uq-preds*$^i_{\mathcal{Q}}$ and use the proof of Proposition 6.19. In that proof we have shown that *preds*$^i_{\mathcal{Q}}$ is a correct $\mathcal{Q}$-pred-basis. Since both $\mathcal{G}(\Lambda)$ and $\mathcal{G}_n$ are downward-closed, this proves both statements of this proposition.

We first show that *uq-preds*$^i_{\mathcal{Q}}(\mathcal{T}, S)$ is a finite subset of $\mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$. In fact, in the proof of Proposition 6.19 we have already shown that every pushout complement of a prepared rule is an element of $\mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$, which also holds for prepared instantiations. Furthermore, we drop every pushout complement where the instantiation is not applicable and thus only use valid matches for the original rule ρ. Note that the application condition is independent of the order morphisms composed with instantiations. Since the number of instantiations is bounded by $\mathit{bound}_\rho(S)$, only finitely many rules are applied backwards and only finitely many pushout complements are added to $\mathcal{G}$, i.e. *uq-preds*$^i_{\mathcal{Q}}(\mathcal{T}, S)$ is finite.

We now prove that $\uparrow$*uq-preds*$^i_{\mathcal{Q}}(\mathcal{T}, S) \supseteq \uparrow\mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$. Let G_0 be an element of $\uparrow\mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$. Then there is a minimal representative $G_1 \in \mathit{Pred}_{\mathcal{Q}}(\uparrow\{S\})$ with $G_1 \subseteq G_0$ via some morphism $\nu\colon G_0 \rightarrowtail G_1$ and an instantiation $\langle \pi\colon L \rightarrowtail \overline{L}, \gamma\colon \overline{L} \rightharpoonup \overline{R}\rangle$ of some rule ρ rewriting G_1 with a injective match m satisfying the application conditions of Definition 6.53 to some element G_2 of $\uparrow\{S\}$. In Lemma 6.28 we have shown that subgraph morphisms are pushout closed. Since m is injective and therefore conflict-free, the left diagram in Figure C.12 can be extended to the right diagram, where the inner and outer squares are pushouts.

Since m and μ_G are injective, $\mu_G \circ m$ is injective as well and due to the properties of pushouts we know that n is also injective. Furthermore the pushout closure guarantees that $\mu_G \circ m$ is total. Since m satisfied the application condition, every edge in G_1 incident to a universally quantified node has a preimage in $\overline{L}$ and therefore also an image in G_3.

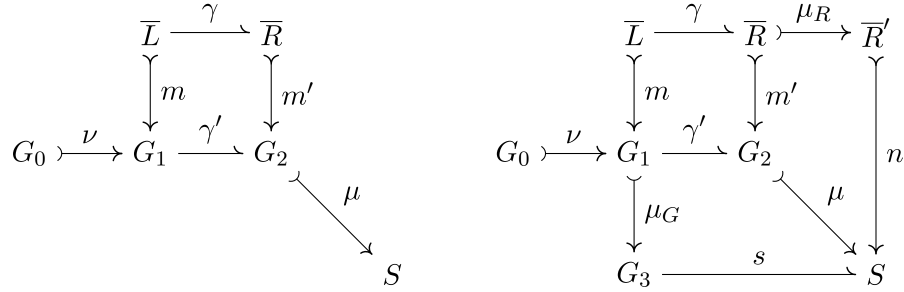


Figure C.12.: Subgraph morphisms are pushout closed not just for ordinary rules, but also for instantiations

The surjectivity of μ_G ensures that the application condition is also satisfied by $\mu_G \circ m$. Note that since G_1 is an element of $\mathcal{Q}$ and $\mathcal{Q}$ is downward-closed, G_3 is also in $\mathcal{Q}$.

Since the outer square is a pushout, G_3 is a pushout complement object. Thus, a graph G_4 with $\mu'_G \colon G_3 \rightarrowtail G_4$ will be obtained by $\mathit{uq\text{-}preds}^i_{\mathcal{Q}}(\mathcal{T}, S)$ at some point using the (prepared) instantiation $\mu_R \circ \gamma$. By the same argument as above $\mu'_G \circ \mu_G \circ m$ satisfies the application condition and is an element of $\mathcal{Q}$, thus G_4 will not be dropped by the procedure. Summarized, this means that $\mathit{uq\text{-}preds}^i_{\mathcal{Q}}$ computes a graph G_4 for every graph G_0 such that $G_4 \subseteq G_3 \subseteq G_1 \subseteq G_0$, i.e. every G_0 is represented by an element of $\mathit{uq\text{-}preds}^i_{\mathcal{Q}}(\mathcal{T}, S)$. ☐

Lemma 6.57. *Let $\rho = \langle r \colon L \rightharpoonup R, U\rangle$ and let $\langle id_L, r\rangle \diamond u = \langle \pi \colon L \rightarrowtail \overline{L}_u, \gamma \colon \overline{L}_u \rightharpoonup \overline{R}_u\rangle$. If for every $u \in U$ every edge $e \in \overline{R}_u$ without preimage in R is incident to a node $v \in \overline{R}_u$ without preimage in R, then $\mathit{bound}_\rho(G) = |V_G|$.*

Proof. This can be shown by using the proof ideas of Proposition 6.54. If ν does not delete all elements of $\overline{R}_u$ which where created in the instantiation step, $\overline{R}_u$ contains at least one node more than R. If the created element not deleted by ν is an edge, by conditions of this lemma, it is incident to a created node not deleted by ν. Thus, every non-negligible instantiation step increases the number of nodes of the right-hand side by at least one. No matches can exist if the number of instantiation steps is larger than the number of nodes in G. ☐

Lemma 6.58. *Let ρ be a rule, $\langle \pi \colon L \rightarrowtail \overline{L}, \gamma \colon \overline{L} \rightharpoonup \overline{R}\rangle$ an instantiation of ρ and $m \colon \overline{R} \rightarrowtail G$ a co-match of the instantiation to some graph G. If there is a node $x \in qn(\rho)$ where $m(\gamma(\pi(x))$ is defined and incident to an edge e without preimage in $\overline{R}$, then there is no pushout complement H of γ, m satisfying the condition of Definition 6.53.*

Proof. Assume there is a $x \in qn(u)$ where $x' = m(\gamma(\pi(x)))$ is defined and there is an edge e incident to x' without preimage in $\overline{R}$. Furthermore, assume H with $m' \colon \overline{L} \rightarrow H$

and $\gamma' \colon H \rightharpoonup G$ is a pushout complement of γ, m. Since the diagram is a pushout, there is an $e' \in H$ with $\gamma'(e') = e$, otherwise the mediating morphism does not exist or is not unique. By commutativity of the diagram, e' is incident to $m'(\pi(x))$ and there cannot be an $e'' \in \overline{L}$ with $m'(e'') = e'$. Since $x \in qn(u)$, this violates the condition of Definition 6.53. $\square$

Bibliography

[AB+05] Parosh Aziz Abdulla, Nathalie Bertrand, Alexander Rabinovich, and Philippe Schnoebelen. “Verification of probabilistic systems with faulty communication”. In: *Information and Computation* 202.2 (2005), pp. 141–165.

[AB+08] Parosh Aziz Abdulla, Ahmed Bouajjani, Jonathan Cederberg, Frédéric Haziza, and Ahmed Rezine. “Monotonic Abstraction for Programs with Dynamic Memory Heaps”. In: *Proceedings of CAV ’08*. Vol. 5123. LNCS. 2008, pp. 341–354.

[AC+04] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. “Using Forward Reachability Analysis for Verification of Lossy Channel Systems”. In: *Formal Methods in System Design* 25 (1 July 2004), pp. 39–65.

[AČ+96] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. “General Decidability Theorems for Infinite-State Systems”. In: *Proceedings of LICS ’96*. IEEE, 1996, pp. 313–321.

[ADR09] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. “Automatic Verification of Directory-Based Consistency Protocols”. In: *Proceedings of RP ’09*. Ed. by Olivier Bournez and Igor Potapov. Vol. 5797. LNCS. Springer Berlin Heidelberg, 2009, pp. 36–50.

[AE+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. “Graph transformation for specification and programming”. In: *Science of Computer Programming* 34.1 (Apr. 1999), pp. 1–54.

[Astra] Astra *Website*. url: `http://www.rw.cdl.uni-saarland.de/~rtc/astra/`.

[Augur2] AUGUR2 *Website*. URL: http://www.ti.inf.uni-due.de/research/tools/augur2/.

[AWK02] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. “Scalable, graph-based network vulnerability analysis”. In: *Proceedings of CCS ’02*. New York, NY, USA: ACM, 2002, pp. 217–224.

[Bau06] Jörg Bauer. “Analysis of Communication Topologies by Partner Abstraction”. PhD thesis. Universität des Saarlandes, 2006.

[BC+10] Paolo Baldan, Andrea Corradini, Fabio Gadducci, and Ugo Montanari. “Frm Ptr Nts t Grph Trnsfmtn Sstms”. In: *Manipulation of Graphs, Algebras and Pictures. Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday*. Ed. by Berthold Hoffmann. Hohnholt Reprografischer Betrieb, 2010.

[BC87] Michel Bauderon and Bruno Courcelle. “Graph expressions and graph rewritings”. In: *Mathematical Systems Theory* 20.1 (1987), pp. 83–127.

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. “A Static Analysis Technique for Graph Transformation Systems”. In: *Proceedings of CONCUR ’01*. Vol. 2154. LNCS. Springer-Verlag, 2001, pp. 381–395.

[BCM05] Paolo Baldan, Andrea Corradini, and Ugo Montanari. “Relating SPO and DPO graph rewriting with Petri nets having read, inhibitor and reset arcs”. In: *Proceedings of PNGT ’05*. Vol. 127.2. ENTCS. 2005, pp. 5–28.

[BD+12a] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. *On the Decidability Status of Reachability and Coverability in Graph Transformation Systems*. Tech. rep. DISI-TR-11-04. Dipartimento di Informatica e Scienze dell’Informazione, Università di Genova, 2012.

[BD+12b] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. “On the Decidability Status of Reachability and Coverability in Graph Transformation Systems”. In: *Proceedings of RTA ’12*. Vol. 15. LIPIcs. Schloss Dagstuhl – Leibniz Center for Informatics, 2012, pp. 101–116.

[BFH87] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. “Amalgamation of graph transformations: A synchronization mechanism”. In: *Journal of Computer and System Sciences* 34.2-3 (1987), pp. 377–408.

[BG11] Laura Bozzelli and Pierre Ganty. “Complexity Analysis of the Backward Coverability Algorithm for VASs”. In: *Proceedings of RP ’11*. Ed. by Giorgio Delzanno and Igor Potapov. Vol. 6945. LNCS. Springer Berlin Heidelberg, 2011, pp. 96–109.

[BK+13] Kshitij Bansal, Eric Koskinen, Thomas Wies, and Damien Zufferey. “Structural counter abstraction”. In: *Proceedings of TACAS ’13*. Vol. 7795. LNCS. Springer-Verlag, 2013, pp. 62–77.

[BM99] Ahmed Bouajjani and Richard Mayr. “Model Checking Lossy Vector Addition Systems”. In: *Proceedings of STACS ’99*. Ed. by Christoph Meinel and Sophie Tison. Vol. 1563. LNCS. Springer Berlin Heidelberg, 1999, pp. 323–333.

[Boost] *Boost Website*. URL: `http://www.boost.org/`.

[BR15a] Peter Backes and Jan Reineke. “Analysis of Infinite-State Graph Transformation Systems by Cluster Abstraction”. In: *Proceedings of VMCAI ’15*. Ed. by Deepak D’Souza, Akash Lal, and Kim Guldstrand Larsen. Vol. 8931. LNCS. Springer Berlin Heidelberg, 2015, pp. 135–152.

[BR15b] Peter Backes and Jan Reineke. “ASTRA: A Tool for Abstract Interpretation of Graph Transformation Systems”. In: *Proceedings of SPIN ’15*. Ed. by Bernd Fischer and Jaco Geldenhuys. Vol. 9232. LNCS. Springer International Publishing, 2015, pp. 13–19.

[BS03] Nathalie Bertrand and Philippe Schnoebelen. “Model Checking Lossy Channels Systems Is Probably Decidable”. In: *Proceedings of FoSSaCS ’03*. Ed. by Andrew D. Gordon. Vol. 2620. LNCS. Springer, Apr. 2003, pp. 120–135.

[Clang] *Clang Website*. URL: `http://clang.llvm.org/`.

[CM+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. “Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. World Scientific, 1997. Chap. 3.

[CMake] *CMake Website*. URL: `http://www.cmake.org/`.

[CMZ04] Jérémie Chalopin, Yves Métivier, and Wieslaw Zielonka. “Election, Naming and Cellular Edge Local Computations”. In: *Proceedings of ICGT ’04*. Vol. 3256. LNCS. Springer, 2004, pp. 242–256.

[Cou90] Bruno Courcelle. “The Monadic Second-order Logic of Graphs. I. Recognizable Sets of Finite Graphs”. In: *Information and Computation* 85.1 (Mar. 1990), pp. 12–75.

[CS11] Maria Chudnovsky and Paul Seymour. “A well-quasi-order for tournaments”. In: *Journal of Combinatorial Theory, Series B* 101 (1 Jan. 2011), pp. 47–53.

[CS14] Maria Chudnovsky and Paul Seymour. “Rao’s Degree Sequence Conjecture”. In: *Journal of Combinatorial Theory, Series B* 105 (Mar. 2014), pp. 44–92.

[Dam90] Peter Damaschke. “Induced subgraphs and well-quasi-ordering”. In: *Journal of Graph Theory* 14.4 (July 1990), pp. 427–435.

[DFS98] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. “Reset nets between decidability and undecidability”. In: *Proceedings of ICALP ’98*. Vol. 1443. LNCS. Springer, 1998, pp. 103–115.

[DH+06] Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. “Adaptive Star Grammars”. In: *Proceedings of ICGT ’06*. Vol. 4178. LNCS. Springer, 2006, pp. 77–91.

[Dic13] Leonard Eugene Dickson. “Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors”. In: *American Journal of Mathematics* 35.4 (1913), pp. 413–422.

[Din92] Guoli Ding. “Subgraphs and well-quasi-ordering”. In: *Journal of Graph Theory* 16 (5 Nov. 1992), pp. 489–502.

[DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. “Hyperedge Replacement Graph Grammars”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. World Scientific, 1997. Chap. 2.

[DS14a] Giorgio Delzanno and Jan Stückrath. *Parameterized Verification of Graph Transformation Systems with Whole Neighbourhood Operations*. arXiv:1407.4394. 2014.

[DS14b] Giorgio Delzanno and Jan Stückrath. “Parameterized Verification of Graph Transformation Systems with Whole Neighbourhood Operations”. In: *Proceedings of RP ’14*. Ed. by Joël Ouaknine, Igor Potapov, and James Worrell. Vol. 8762. LNCS. Springer, 2014, pp. 72–84.

[DSZ10] Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. “Parameterized Verification of Ad Hoc Networks”. In: *Proceedings of CONCUR ’10*. Vol. 6269. LNCS. Springer, 2010, pp. 313–327.

[DSZ11] Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. “On the power of cliques in the parameterized verification of Ad Hoc networks”. In: *Proceedings of FoSSaCS ’11*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 441–455.

[EE+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag New York, Inc., 2006.

[EE+99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages, and Tools*. World Scientific Publishing, 1999.

[EH+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. “Algebraic Approaches to Graph Transformation, Part II: Single Pushout Approach and Comparison with Double Pushout Approach”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. World Scientific, 1997. Chap. 4.

[EK+99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific Publishing, 1999.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. “Graph grammars: An algebraic approach”. In: *Proceedings of Switching and Automata Theory ’73*. 1973, pp. 167–180.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. “Node Replacement Graph Grammars”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997, pp. 1–94.

[FG09a] Alain Finkel and Jean Goubault-Larrecq. “Forward Analysis for WSTS, Part I: Completions”. In: *Proceedings of STACS ’09*. 2009, pp. 433–444.

[FG09b] Alain Finkel and Jean Goubault-Larrecq. “Forward Analysis for WSTS, Part II: Complete WSTS”. In: *Proceedings of ICALP ’09*. 2009, pp. 188–199.

[FHR09] Michael R. Fellows, Danny Hermelin, and Frances A. Rosamond. “Well-Quasi-Orders in Subclasses of Bounded Treewidth Graphs”. In: *Proceedings of IWPEC ’09*. Vol. 5917. LNCS. Springer, 2009, pp. 149–160.

[FHR12] Michael R. Fellows, Danny Hermelin, and Frances A. Rosamond. “Well-Quasi-Orders in Subclasses of Bounded Treewidth Graphs and Their Algorithmic Applications”. In: *Algorithmica* 64.1 (2012), pp. 3–18.

[FK+95] Micheal Fellows, Jan Kratochvíl, Matthias Middendorf, and Frank Pfeiffer. “The complexity of induced minors and related problems”. In: *Algorithmica* 13.3 (1995), pp. 266–282.

[FS01] Alain Finkel and Philippe Schnoebelen. “Well-Structured Transition Systems Everywhere!” In: *Theoretical Computer Science* 256.1-2 (Apr. 2001), pp. 63–92.

[GBT] *Graph Backwards Tool (*GBT*) Website*. URL: `http://www.it.uu.se/research/group/mobility/adhoc/gbt`.

[GCC] *GCC Website*. URL: `https://gcc.gnu.org/`.

[GH+10] Robert Ganian, Petr Hlinený, Joachim Kneis, Daniel Meister, Jan Obdrzálek, Peter Rossmanith, and Somnath Sikdar. “Are There Any Good Digraph Width Measures?” In: *Proceedings of IPEC ’10*. 2010, pp. 135–146.

[GM+12] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. “Modelling and analysis using GROOVE”. In: *International Journal on Software Tools for Technology Transfer* 14.1 (Feb. 2012), pp. 15–40.

[Göt88] Herbert Göttler. *Graphgrammatiken in der Softwaretechnik: Theorie und Anwendungen*. Vol. 178. Informatik-Fachberichte. Springer, 1988.

[Graphviz] GRAPHVIZ *Website*. URL: `http://www.graphviz.org/`.

[Groove] GROOVE *Website*. URL: `http://groove.cs.utwente.nl/`.

[GRV06] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. “Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS”. In: *Journal of Computer and System Sciences* 72.1 (Feb. 2006), pp. 180–203.

[GXL] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. *GXL Website*. URL: `http://www.gupro.de/GXL/`.

[Hab89] Annegret Habel. “Hyperedge Replacement: Grammars and Languages”. PhD thesis. Universität Bremen, 1989.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Vol. 643. LNCS. Springer-Verlag, 1992.

[Hig52] Graham Higman. “Ordering by Divisibility in Abstract Algebras”. In: *Proceedings of the London Mathematical Society* s3-2.1 (Jan. 1952), pp. 326–336.

[HJ+10] Marvin Heumüller, Salil Joshi, Barbara König, and Jan Stückrath. “Construction of Pushout Complements in the Category of Hypergraphs”. In: *Proceedings of GCM ’10*. 2010.

[HJ+11] Marvin Heumüller, Salil Joshi, Barbara König, and Jan Stückrath. “Construction of Pushout Complements in the Category of Hypergraphs”. In: *Selected Revised Papers from the Workshop on Graph Computation Models (GCM 2010)*. Vol. 39. Electronic Communications of the EASST. 2011.

[HJ+15a] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. “Juggrnaut: using graph grammars for abstracting unbounded heap structures”. In: *Formal Methods in System Design* 47.2 (2015), pp. 159–203.

[HJ+15b] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. “Verifying pointer programs using graph grammars”. In: *Science of Computer Programming* 97, Part 1 (2015). Special Issue on New Ideas and Emerging Results in Understanding Software, pp. 157–162.

[HP01] Annegret Habel and Detlef Plump. “Computational Completeness of Programming Languages Based on Graph Transformation”. In: *Proceedings of FoSSaCS '11*. Vol. 2030. LNCS. Springer, 2001, pp. 230–245.

[HP02] Annegret Habel and Detlef Plump. “Relabelling in Graph Transformation”. In: *Proceedings of ICGT '02*. Vol. 2505. LNCS. Springer, 2002, pp. 135–147.

[HP09] Annegret Habel and Karl-Heinz Pennemann. “Correctness of High-level Transformation Systems Relative to Nested Conditions”. In: *Mathematical Structures in Computer Science* 19.2 (Apr. 2009), pp. 245–296.

[JK08] Salil Joshi and Barbara König. “Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems”. In: *Proceedings of CAV '08*. Vol. 5123. LNCS. Springer, 2008, pp. 214–226.

[JK12] Salil Joshi and Barbara König. *Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems*. Tech. rep. 2012-01. Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, 2012.

[JR+01] Thor Johnson, Neil Robertson, Paul Seymour, and Robin Thomas. “Directed Tree-Width”. In: *Journal of Combinatorial Theory, Series B* 82.1 (2001), pp. 138–154.

[KK06] Barbara König and Vitali Kozioura. “Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems”. In: *Proceedings of TACAS '06*. Vol. 3920. LNCS. Springer, 2006, pp. 197–211.

[KK08] Barbara König and Vitali Kozioura. “Augur 2—A New Version of a Tool for the Analysis of Graph Transformation Systems”. In: *Proceedings of GT-VMT ’06*. Vol. 211. ENTCS. Elsevier, 2008, pp. 201–210.

[KKR12] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. “The disjoint paths problem in quadratic time”. In: *Journal of Combinatorial Theory, Series B* 102.2 (2012), pp. 424–435.

[KM69] Richard M. Karp and Raymond E. Miller. “Parallel Program Schemata”. In: *Journal of Computer and System Sciences* 3.2 (May 1969), pp. 147–195.

[KMP00] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. “A Formal Model for Role-Based Access Control Using Graph Transformation”. In: *Proceedings of ESORICS ’00*. Vol. 1895. LNCS. Springer, 2000, pp. 122–139.

[KMP02] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. “Decidability of Safety in Graph-Based Models for Access Control”. In: *Proceedings of ESORICS ’02*. London, UK: Springer-Verlag, 2002, pp. 229–243.

[KMP05] Manuel Koch, L. V. Mancini, and Francesco Parisi-Presicce. “Graph-based specification of access control policies”. In: *Journal of Computer and System Sciences* 71.1 (July 2005), pp. 1–33.

[Koz07] Vitali Kozioura. “Verification of Random Graph Transformation Systems”. In: *Electronic Notes in Theoretical Computer Science* 175.4 (2007), pp. 63–72.

[Koz10] Vitaly Kozioura. “Abstraction and abstraction refinement in the verification of graph transformation systems”. PhD thesis. Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften, 2010.

[KS12a] Barbara König and Jan Stückrath. *Well-Structured Graph Transformation Systems with Negative Application Conditions*. Tech. rep. 2012-03. Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, 2012.

[KS12b] Barbara König and Jan Stückrath. “Well-Structured Graph Transformation Systems with Negative Application Conditions”. In: *Proceedings of ICGT ’12*. Vol. 7562. LNCS. Springer, 2012, pp. 89–95.

[KS14a] Barbara König and Jan Stückrath. *A General Framework for Well-Structured Graph Transformation Systems*. arXiv:1406.4782. 2014.

[KS14b] Barbara König and Jan Stückrath. “A General Framework for Well-Structured Graph Transformation Systems”. In: *Proceedings of CONCUR ’14*. Ed. by Paolo Baldan and Daniele Gorla. Vol. 8704. LNCS. Springer, 2014, pp. 467–481.

[KS16] Barbara König and Jan Stückrath. “Well-Structured Graph Transformation Systems”. In: *Information and Computation* (2016). Accepted for publication.

[Latex] *LaTeX*. URL: `http://www.latex-project.org/`.

[Ler11] Jérôme Leroux. “Vector addition system reachability problem: a short self-contained proof”. In: *Proceedings of POPL ’11*. 2011, pp. 307–316.

[LR80] Andrea S. Lapaugh and Ronald L. Rivest. “The subgraph homeomorphism problem”. In: *Journal of Computer and System Sciences* 20.2 (1980), pp. 133–149.

[LS04] Stephen Lack and Pawel Sobocinski. “Adhesive Categories”. In: *Proceedings of FoSSaCS ’04*. 2004, pp. 273–288.

[Mac78] Saunders Mac Lane. *Categories for the Working Mathematician*. 2nd. Graduate Texts in Mathematics, Vol. 5. Springer, 1978.

[May03] Richard Mayr. “Undecidable Problems in Unreliable Computations”. In: *Theoretical Computer Science* 297.1-3 (Mar. 2003), pp. 337–354.

[May81] Ernst W. Mayr. “An algorithm for the general Petri net reachability problem”. In: *Proceedings of STOC ’81*. STOC ’81. New York, NY, USA: ACM, 1981, pp. 238–246.

[May84] Ernst W. Mayr. “An Algorithm for the General Petri Net Reachability Problem”. In: *SIAM Journal on Computing* 13.3 (1984), pp. 441–460.

[May98] Richard Mayr. *Lossy Counter Machines*. Tech. rep. TUM-19827. Technische Universität München, Oct. 1998.

[Mey09] Roland Meyer. “Structural Stationarity in the π-Calculus”. PhD thesis. Carl-von-Ossietzky-Universität Oldenburg, 2009.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1967.

[Mist2] Pierre Ganty. MIST2 *GitHub Repository*. URL: `https://github.com/pierreganty/mist/wiki`.

[Mur89] Tadao Murata. “Petri Nets: Properties, Analysis and Applications”. In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580.

[Nag79] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979.

[NM12] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity*. Springer Berlin Heidelberg, 2012.

[OBM06] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. "A scalable approach to attack graph generation". In: *Proceedings of CCS '06*. New York, NY, USA: ACM, 2006, pp. 336–345.

[PE02] Julia Padberg and Bettina E. Enders. "Rule Invariants in Graph Transformation Systems for Analyzing Safety-Critical Systems". In: *Proceedings of ICGT '02*. Ed. by Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 2505. LNCS. Springer Berlin Heidelberg, 2002, pp. 334–350.

[PEM87] Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. "Graph rewriting with unification and composition". In: *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*. Springer, 1987, pp. 496–514.

[Pen09] Karl-Heinz Pennemann. "Development of Correct Graph Transformation Systems". PhD thesis. Department für Informatik, Universität Oldenburg, 2009.

[Pet62] Carl Adam Petri. "Kommunikation mit Automaten". PhD thesis. Universität Bonn, 1962.

[Petruchio] PETRUCHIO *Website*. URL: `http://csd.informatik.uni-oldenburg.de/~critter/petruchio/`.

[Picasso] PICASSO *Website*. URL: `http://pub.ist.ac.at/~zufferey/picasso/`.

[Pie91] Benjamin C. Pierce. *Basic category theory for computer scientists*. Foundations of computing series. MIT Press, 1991.

[Plu12] Detlef Plump. "The Design of GP 2". In: *Proceedings of WRS '11*. Ed. by Santiago Escobar. Vol. 82. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2012, pp. 1–16.

[PP14] Christopher M. Poskitt and Detlef Plump. "Verifying Monadic Second-Order Properties of Graph Programs". In: *Proceedings of ICGT '14*. Ed. by Holger Giese and Barbara König. Vol. 8571. LNCS. Springer International Publishing, 2014, pp. 33–48.

[PS98] Cynthia Phillips and Laura Painton Swiler. "A graph-based system for network-vulnerability analysis". In: *Proceedings of NSPW '98*. New York, NY, USA: ACM, 1998, pp. 71–79.

[Rac78] Charles Rackoff. “The covering and boundedness problems for vector addition systems”. In: *Theoretical Computer Science* 6.2 (1978), pp. 223–231.

[Ren03] Arend Rensink. “The GROOVE Simulator: A Tool for State Space Generation”. In: *Proceedings of AGTIVE ’03*. Vol. 3062. LNCS. Springer, 2003, pp. 479–485.

[Ros75] Barry K. Rosen. “Deriving Graphs from Graphs by Applying a Production”. In: *Acta Informatica* 4 (1975), pp. 337–357.

[Roz97] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. World Scientific Publishing, 1997.

[RS04] Neil Robertson and Paul Seymour. “Graph Minors XX. Wagner’s conjecture”. In: *Journal of Combinatorial Theory, Series B* 92 (2 Nov. 2004), pp. 325–357.

[RS10] Neil Robertson and Paul Seymour. “Graph Minors XXIII. Nash-Williams’ immersion conjecture”. In: *Journal of Combinatorial Theory, Series B* 100 (2 Mar. 2010), pp. 181–205.

[RS85] Neil Robertson and Paul Seymour. “Graph Minors – A Survey”. In: *Surveys in Combinatorics*. Vol. 103. London Mathematics Society Lecture Notes Series. Cambridge University Press, 1985, pp. 153–171.

[RS95] Neil Robertson and Paul Seymour. “Graph Minors. XIII. The Disjoint Paths Problem”. In: *Journal of Combinatorial Theory, Series B* 63.1 (1995), pp. 65–110.

[Sch04] Philippe Schnoebelen. “The Verification of Probabilistic Lossy Channel Systems”. In: *Validation of Stochastic Systems. A Guide to Current Research*. Ed. by Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, Markus Siegle, and Frits Vaandrager. Vol. 2925. LNCS. Springer, 2004, pp. 445–465.

[Sch10] Philippe Schnoebelen. “Lossy Counter Machines Decidability Cheat Sheet”. In: *Proceedings of RP ’10*. Ed. by Antonín Kučera and Igor Potapov. Vol. 6227. LNCS. Springer, Aug. 2010, pp. 51–75.

[SM12] Tim Strazny and Roland Meyer. “An Algorithmic Framework for Coverability in Well-Structured Systems”. In: *Proceedings of ACSD ’12*. June 2012, pp. 173–182.

[SS12] Sylvain Schmitz and Philippe Schnoebelen. “Algorithmic Aspects of WQO Theory”. lecture notes. Aug. 2012.

[SS13] Sylvain Schmitz and Philippe Schnoebelen. "The Power of Well-Structured Systems". In: *Proceedings of CONCUR '13*. Ed. by Pedro R. D'Argenio and Hernán) Melgratti. Vol. 8052. LNCS. Springer, Aug. 2013, pp. 5–24.

[Ste07] Sandra Steinert. "The Graph Programming Language GP". PhD thesis. The University of York, 2007.

[Stü15] Jan Stückrath. "Uncover: Using Coverability Analysis for Verifying Graph Transformation Systems". In: *Proceedings of ICGT '15*. Ed. by Francesco Parisi-Presicce and Bernhard Westfechtel. Vol. 9151. LNCS. Springer, 2015, pp. 266–274.

[SW01] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. New York, NY, USA: Cambridge University Press, 2001.

[SW14a] Jan Stückrath and Benjamin Weyers. "Lattice-extended Coloured Petri Net Rewriting for Adaptable User Interface Models". In: *Proceedings of GT-VMT '14*. 2014.

[SW14b] Jan Stückrath and Benjamin Weyers. *Lattice-extended Coloured Petri Net Rewriting for Adaptable User Interface Models*. Tech. rep. 2014-01. Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, 2014.

[SWJ08] Mayank Saksena, Oskar Wibling, and Bengt Jonsson. "Graph grammar modeling and verification of ad hoc routing protocols". In: *Proceedings of TACAS '08*. LNCS. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 18–32.

[SWW11] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. "Sound and complete abstract graph transformation". In: *Proceedings of SBMF '11*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 92–107.

[Symgraph] SYMGRAPH *Website*. URL: `http://www.disi.unige.it/person/DelzannoG/Symgraph/`.

[Tur37] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265.

[Uncover] Jan Stückrath. UNCOVER *Website*. URL: `http://www.ti.inf.uni-due.de/research/tools/uncover/`.

[WZH10] Thomas Wies, Damien Zufferey, and Thomas A. Henzinger. "Forward Analysis of Depth-Bounded Processes". In: *Proceedings of FoSSaCS '10*. 2010, pp. 94–108.

[XercesC++] *XercesC++ Website*. URL: `http://xerces.apache.org/xerces-c/`.

[ZWH12] Damien Zufferey, Thomas Wies, and Thomas A. Henzinger. “Ideal Abstractions for Well-Structured Transition Systems”. In: *Proceedings of VMCAI ’12*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. LNCS. Springer, 2012, pp. 445–460.

List of Symbols

Basic Notation

$\mathbb{N}_0$, $\mathbb{N}$	Natural numbers with ($\mathbb{N}_0$) and without ($\mathbb{N}$) zero, page 9
$\subseteq$, $\subset$	Subset ($\subseteq$) and strict subset ($\subset$) relation, page 9
$A \times \ldots \times A$ or A^n	Cartesian product of length n over the set A, page 9
$\langle a_1, \ldots, a_n \rangle$	Element of a Cartesian product, page 9
A^*	Set of all sequences over the set A, page 10
$\mathbf{s}[i]$	The i-th element of the sequence $\mathbf{s}$, page 10
$\lvert\mathbf{s}\rvert$	Length of the sequence $\mathbf{s}$, page 10
$[a]_\equiv$	Equivalence class of a wrt. the equivalence $\equiv$, page 9
$A/\equiv$	Quotient set, i.e. set of all equivalence classes of A wrt. the equivalence $\equiv$, page 9
$\overline{R}$	Equivalence closure of a binary relation R, i.e. the smallest equivalence containing R, page 10

General Transition Systems

$\Rightarrow$	One step transition relation, page 12
$\Rightarrow_Q$	One step transition relation within Q, page 17
$\Rightarrow^*$	Multi step transition relation, page 12
$\mathit{Succ}(I)$	Set of direct successors of I, page 12
$\mathit{Succ}_Q(I)$	Set of direct successors of I within Q, page 18
$\mathit{Succ}^*(I)$	Set of indirect successors of I, page 12

$Succ^*_Q(I)$	Set of indirect successors of I within Q, page 18
$Pred(I)$	Set of direct predecessors of I, page 12
$Pred_Q(I)$	Set of direct predecessors of I within Q, page 18
$Pred^*(I)$	Set of indirect predecessors of I, page 12
$s_1 \rightarrowtriangle s_2$	Backward step from s_1 to s_2, page 16

Graphs

$\langle V_G, E_G, c_G, l_G \rangle$	Notation of a graph G, page 27
V_G	Set of nodes of G, page 27
E_G	Set of edges of G, page 27
$c_G \colon E_G \to V_G^*$	Connection function of G, page 27
$l_G \colon E_G \to \Lambda$	Labelling function of G, page 27
$ar \colon \Lambda \to \mathbb{N}_0$	Function assigning an arity to each label, page 27
Λ-**HGt**	Category of Λ-hypergraphs and total morphisms, page 29
Λ-**HGp**	Category of Λ-hypergraphs and partial morphisms, page 29
$\langle \Lambda_V, \Lambda_E \rangle$-**HGtr**	Category of $\langle \Lambda_V, \Lambda_E \rangle$-hypergraphs and relabelling morphisms, page 63

Graph Morphisms

$\rightharpoonup$	Partial morphism, page 28
$\rightarrow$	Total morphism, page 28
$\rightarrowtail$	Total and injective morphism, page 28
$\mapsto$	Generic order morphism, page 72
$\mapsto$	Minor morphism, page 73
$\rightarrowtail$	Subgraph morphism, page 82
$\rhd\!\!\rightarrow$	Induced subgraph morphism, page 88

Graph Classes and Transition Systems

$\mathcal{G}(\Lambda)$	Class of all graphs over the alphabet Λ, page 27
$\mathcal{G}_n$	Class of all graphs where the longest undirected path has length n, page 83

$\mathcal{D}_n$	Class of directed graphs where the longest undirected path has length n, page 89
$\mathcal{D}_{n,k}$	Subclass of $\mathcal{D}_n$ where every two nodes are connected by at most k parallel edges with the same direction and label, page 89
$\mathcal{T}_{\mathcal{G}}$	Transition system on $\mathcal{G}$ induced by the GTS $\mathcal{T}$ and general matches, page 31
$\mathcal{T}^c_{\mathcal{G}}$	Transition system on $\mathcal{G}$ induced by the GTS $\mathcal{T}$ and conflict-free matches, page 31
$\mathcal{T}^i_{\mathcal{G}}$	Transition system on $\mathcal{G}$ induced by the GTS $\mathcal{T}$ and injective matches, page 31

Quasi-Orders and Well-Quasi-Orders

$\preceq$	Generic quasi-order, page 10
$\sqsubseteq$	The minor ordering, page 72
$\subseteq$	The subgraph ordering, page 82
$\trianglelefteq$	The induced subgraph ordering, page 88
$\uparrow B$	Upward closure of a set B, page 11
$\downarrow B$	Downward closure of a set B, page 11

Procedures, Functions and Sets of the Backward Search

$\mathit{minimize}_{\preceq}(\mathcal{G})$	Procedure minimizing (wrt. $\preceq$) the set of graphs $\mathcal{G}$, page 102
$\mathit{prepare}_{\preceq}(\mathcal{T})$	Procedure preparing (wrt. $\preceq$) the GTS $\mathcal{T}$, page 102
$\mathit{minpoc}^{\preceq}_{\mathcal{Q}}(r,m)$	Procedure computing the set of minimal pushout complements in $\mathcal{Q}$ (wrt. $\preceq$) for the rule r and match m, page 102 (definition), page 111 (for $\sqsubseteq$), page 115 (for $\subseteq$), page 121 (for $\trianglelefteq$)
$\mathit{origin}(r)$	Function returning the set of original rules from which a prepared rule r could have been generated from, page 99
$\mathit{reps}^{\preceq}_{\mathcal{Q}}(r,m)$	Procedure computing the set of represented injective predecessors in $\mathcal{Q}$ (wrt. $\preceq$) for a prepared rule r and match m, page 107 (definition), page 112 (for $\sqsubseteq$), page 117 (for $\subseteq$ and $\trianglelefteq$)
$\mathit{preds}^{c}_{\mathcal{Q}}(\mathcal{T},S)$	The conflict-free $\mathcal{Q}$-pred-basis for a prepared GTS $\mathcal{T}$ and a graph $S \in \mathcal{Q}$, page 103
$\mathit{preds}^{i}_{\mathcal{Q}}(\mathcal{T},S)$	The injective $\mathcal{Q}$-pred-basis for a prepared GTS $\mathcal{T}$ and a graph $S \in \mathcal{Q}$, page 108

$\mathit{uq\text{-}preds}^{i}_{\mathcal{Q}}(\mathcal{T}, S)$	The injective $\mathcal{Q}$-pred-basis for the subgraph ordering, a prepared UGTS $\mathcal{T}$ and a graph $S \in \mathcal{Q}$, page 132
$qn(u)$ or $qn(\rho)$	Set of quantified nodes of a q-pair u or universally quantified rule ρ, page 128
$\mathit{bound}_{\rho}(G)$	Function bounding the number of instantiations of ρ required to be applied backwards to G, page 132

Index

adjacent nodes or edges, 27
antisymmetric up to equivalence, 72
arity of an edge, 27
arrow (of a category **C**), 24

backward search for GTS, 101
  conflict-free Q-pred-basis, 103
  inj. Q-pred-basis, 108
  inj. Q-pred-basis for UGTS, 132
  minimization procedure, 102
  rule preparation procedure, 102
backward search for WSTS, 15
basis
  see upward-closed set
  see downward-closed set
butterfly minor, 95

category, 24
  Set, 24
  diagram, 25
  Λ-**HGp**, 29
  Λ-**HGt**, 29
  $\langle \Lambda_V, \Lambda_E \rangle$-**HGtr**, 63
circumference, 95
co-match, 31
codomain, 24
commuting diagram, 25
composition, 24, 28, 29
conflict-free Q-pred-basis, 103
conflict-free match, 30
context-free GTS, 79
coverability, *see* coverability problem
coverability problem, 13, 18
  backward search for GTS, 101
  backward search for WSTS, 15
  context-free GTS, 116
  existential, 64
  forward search for WSTS, 16
  GTS with constant nodes, 46, 48
  ind. subg. and cf. matches, 122
  ind. subg. and inj. matches, 122
  minors and cf. matches, 111
  minors and inj. matches, 114
  node- and edge-deleting GTS, 62
  non-deleting GTS, 51
  Q-restricted WSTS, 20
  restricted coverability problem, 18
  subgraphs and cf. matches, 115
  subgraphs and inj. matches, 119
  WSTS, 15

dangling condition, 32
diagram, 25
directed graph, **27**, 89
directed topological minor, 95
domain, 24
doubly-labelled hypergraph, 62

downward closure, 11
downward-closed set, 11

edge multiplicity, 89
effective Q-pred-basis, **18**, 101
  $preds^c_Q$ for GTS, 103
  $preds^i_Q$ for GTS, 108
  $uq\text{-}preds^i_Q$ for UGTS, 132
effective pred-basis, **14**, 101
equivalence closure, 10
existential coverability problem, 64
  edge relabelling GTS, 65
  node and edge relabelling GTS, 67
  node relabelling GTS, 66

feedback-vertex-set, 95
finite basis property, 11
forward search for WSTS, 16

general coverability problem, 18
  see coverability problem
gluing condition, 32
graph, *see* hypergraph
graph morphism, 28
  label-preserving, 63
  relabelling, 63
graph transformation system, 31
  constant number of nodes, 48
  context-freeness, 79
  edge-contracting, 57
  edge-deleting, 57
  negative application conditions, 93
  no deletion and creation of nodes, 46
  node-deleting, 57
  non-deleting, 50
graph transition system, 31
GTS, *see* graph transformation system

hyperedge replacement systems, 79
hypergraph, 27
  doubly-labelled, 62
  visualisation, 28

identification condition, 32
incident, 27
induced minor, 95
induced subgraph, 88
induced subgraph morphism, 88
induced topological minor, 95
injective Q-pred-basis, 108
  for UGTS, 132
isomorphism, 24

Λ-**HGp**, 29
Λ-**HGt**, 29
$\langle\Lambda_V, \Lambda_E\rangle$-**HGtr**, 63
label-preserving graph morphism, 63

marking (of a Petri net), 171
match, 30, 63
minimal pushout complements, 102
  induced subgraph ordering, 121
  minor ordering, 111
  subgraph ordering, 115
minimization procedure, 102
minor, 72
minor morphism, 73
minor rules, 56
Minsky machine, 173
morphism, 24
multipath, 54

negative application condition, 93

object (of a category **C**), 24
order morphism, 72
  induced subgraph morphism, 88
  minor morphism, 73
  subgraph morphism, 82

path, *see* undirected path
Petri net, 171
predecessor set, 12

prepared GTS, 98
prepared rule, 99
preservation by pushouts, 99
induced subgraph ordering, 119
minor ordering, 111
subgraph ordering, 114
pushout, 25
construction in Λ-**HGp**, 35
construction in Λ-**HGt**, 34
existence in Λ-**HGp** and Λ-**HGt**, 30
special properties, 25, 26
pushout closure, 103
induced subgraph ordering, 120
minor ordering, 111
subgraph ordering, 115
pushout complement, 27
construction in Λ-**HGp**, 40, 42
construction in Λ-**HGt**, 36
existence in Λ-**HGp** and Λ-**HGt**, 32

q-pair, *see* universal quantification pair
Q-predecessor set, 18
Q-restricted WSTS, 17
effective Q-pred-basis, 18
Q-predecessor set, 18
Q-successor set, 18
Q-successor set, 18
quantified nodes, 128
quasi-order, 10
antisymmetric up to equivalence, 72
butterfly minor, 95
directed topological minor, 95
induced minor, 95
induced subgraph, 88
induced topological minor, 95
minor, 72
representable by morphisms, 72
strong immersion, 96
subgraph, 82
topological minor, 95
weak immersion, 96
quotient set, 9

reachability, *see* reachability problem
reachability problem, 12
for context-free GTS, 81
for deleting and contracting GTS, 58
GTS with constant nodes, 46, 48
non-deleting GTS, 51
relabelling
graph morphism, 63
rule, 63
representable by morphisms, 72
represented injective predecessors, 107
minor ordering, 112, 114
subgraph ordering, 117, 118
restricted coverability problem, 18
ind. subg. and cf. matches, 122
ind. subg. and inj. matches, 122
minors and cf. matches, 111
minors and inj. matches, 114
Q-restricted WSTS, 20
subgraphs and cf. matches, 115
subgraphs and inj. matches, 119
rewriting rule, 30
rewriting step, 30
rule, 30
context-free, 79
edge contraction, 57
edge deletion, 57
negative application condition, 93
node deletion, 57
relabelling, 63
rule preparation procedure, 102

strong immersion, 96
subgraph, 82
subgraph morphism, 82
successor set, 12

topological minor, 95
transition system, 12
  predecessor set, 12
  successor set, 12
tree-depth, 89
Turing machine, 172
type of a graph, 89

undirected path, 82
universal quantification pair, 128
universally quantified GTS (UGTS), 128
universally quantified rules, 128
  application, 131
  instantiation, 129
upward closure, 11
upward-closed set, 11

weak immersion, 96
well-quasi-order, 10
  alternative definitions, 11
well-structured transition system, 14
  backward search, 15
  effective pred-basis, 14
  forward search, 16
  Q-restricted WSTS, 17
wqo, *see* well-quasi-order
WSTS, *see* well-structured TS