Graph Automata and Their Application to the Verification of Dynamic 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

Christoph Blume
aus
Duisburg

1. Gutachter: Prof. Dr. Barbara König
2. Gutachter: Prof. Dr. Annegret Habel
Datum der mündlichen Prüfung: 12. September 2014

To my family

Abstract

The aim of this thesis is to provide verification techniques based on formal language theory and graph theory for graph transformation systems. Graph transformation systems can be used in a large number of application areas. In many cases it is very natural to model concurrent and distributed systems or other systems, where evolving graph-like structures play a role, by means of graph transformation systems. However, systems which are modelled by graph-like structures usually have an additional level of complexity compared to rule-based systems where states have either a word or tree structure. But since these latter structures have a well-established theory which can be used for several verification techniques, it is natural to ask for an adaption to the setting of graph-like structures. Especially the regular word and tree languages have been studied with great success.

In this work we will study regular graph languages – often called recognizable graph languages – as introduced by Courcelle. For this purpose, we will generalize finite automata and tree automata to obtain automata which accept graphs. But similar to decompositions of words into letters in the case of finite automata, these graph-accepting automata depend on so-called cospan decompositions of graphs. Hence, we investigate the connection between cospan decompositions and the well-known notions of path and tree decompositions.

Subsequently, we introduce different notions of graph-accepting automata: the categorial notion of automaton functors presented by Bruggink and König, consistent tree automata as generalization of tree automata (accepting tree-like decompositions of graphs) and graph automata as a more automaton-theoretic view on automaton functors (accepting path-like decompositions of graphs). Due to the acceptance of path-like decompositions graph automata are of special interest to us, since this automaton model is highly comparable to finite automata. Therefore, many techniques for finite automata can be adapted to the setting of graph automata. Precisely, we study how recent automaton-based techniques solving the language inclusion problem for finite automata can be adapted for graph automata and can be used for invariant checking in graph transformation systems. But since graph automata suffer from a combinatorial explosion in the size of the maximally permitted pathwidth, we develop techniques to symbollically represent graph automata by means of binary decision diagrams.

Finally, we introduce the prototype implementation of a tool suite, called Raven, for creating and manipulating graph automata. We use this tool for a number of experiments and discuss the runtime results.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Dr. Barbara König for her ongoing support and encouragement on this research project. I also appreciate and admire the patience she showed in answering the many questions I had. I was very fortunate to have a supervisor who gave me the freedom to explore my own ideas and at the same time the guidance I needed not to veer off course. Furthermore, I am deeply grateful to Prof. Dr. Annegret Habel from the University of Oldenburg for her interest in my work. I feel very honoured that she agreed to act as a referee for my thesis.

Special thanks go to my former and present colleagues Sander Bruggink, Tobias Heindel, Mathias Hülsbusch, Henning Kerstan, Sebastian Küpper and Jan Stückrath for their help and many interesting discussions on the topics of this thesis as well as the good working atmosphere and the many interesting discussions during lunch.

Furthermore, I would like to thank the students who have been involved in the development of the software tool Raven: Dominik Engelke, Martin Friedrich, Weixiang Guan, Dennis Nolte and Laura Steinert. It was a very interesting and pleasant team work experience.

I further owe a debt of gratitude to Daniel Deja and to others (which I have already mentioned) who were so kind as to read this thesis and to provide me with valuable comments.

I also would like to thank Susanne Kleine-Pralat, Christian Knieper and Michael Wennersheide for the opportunity to broaden my horizon by gaining new insights into various disciplines and methodologies.

Last but not least, none of this would have been possible without the assistance and patience of my family and friends. In the end, I am what they give me, more than anything else, for which I will be forever grateful.

Contents

"The belief in a certain idea gives to the researcher the support for his work. Without this belief he would be lost in a sea of doubts and insufficiently verified proofs."

Konrad Zuse (1910 – 1995)

1 Introduction

1.1. Context

Formal language theory has a long history in computer science in which it has been extensively studied. Beside other classes of languages in the Chomsky hierarchy the class of regular languages has a well-established theory with a great number of applications, which range from the design of electronic circuits [67] and the design of communication protocols [16] over language parsing [87] to different analysis techniques, e. g. regular model checking [26], termination analysis [76] and reachability analysis [71]. From Büchi, Elgot and Trakhtenbrot [38, 65, 125] it is well-known that there is a strong connection between monadic second-order logic and regular languages. Monadic second-order logic is an extension of first-order logic which allows quantification over both objects and sets of objects.

In addition to the classical work on regularity for word languages a lot of effort has been spent on lifting the theory from words to other, more general kinds of structures. Therefore, the notion of regularity has been studied not only for the case of words, but also for the more general case of trees. This has opened the possibility to define regular tree languages and tree automata – similar to regular word languages and finite automata [39]. Straightforwardly, many results of regular word languages have been lifted to regular tree languages, which includes closure properties and constructions such as minimization and determinization of tree automata. Similar to the case of word languages, this gives rise to different applications, e. g. verification techniques [25, 104].

Due to these successful and promising results, it is a natural question to ask for an extension of the notion of regularity to the setting of graphs – since graphs generalize trees just as trees generalize words. In the last 30 years several notions of regular graph languages (also called recognizable graph languages) have been introduced [28, 44, 48, 80, 110, 127, 130], but in each case a slightly different notion of regularity (respectively recognizability) is studied. The most established theory for recognizable

graph languages is the one of Courcelle, who has made many important contributions to this topic in recent years, e. g. different algebras to compose graphs from a finite number of graph operations, an investigation of the connection between the monadic second-order logic of graphs and recognizable graph languages, or a special kind of graph transformation, which is called monadic second-order transduction [7, 42, 44, 48].

One of his most important individual results is the well-known *Courcelle's Theorem* [44], which states that every problem definable in monadic second-order logic of graphs is solvable in linear time for graphs of bounded treewidth. This is especially important, because many NP-hard problems on graphs can be defined in monadic second-order graph logic. For the class of trees it is long-known [38, 65, 123, 124] that there exist efficient algorithms to solve problems that are NP-hard in the general case. But due to Courcelle's Theorem one can solve these problems efficiently even on graphs that are not trees, but are only "tree-like" [4, 5, 18, 24, 44]. The "tree-likeliness" of a graph is measured by the *treewidth* of the graph, which is a parameter introduced by Robertson and Seymour in their seminal work on graph minors [113, 114, 116], e. g. trees and forests have a treewidth of 1 and the n-clique or the $(n \times n)$-grid have a treewidth of $n - 1$. The definition of treewidth is based on the notion of *tree decompositions* of a graph G. The idea is that a tree decomposition consists of a tree in which each tree node is annotated with a *bag*, i. e. a set of vertices of G, such that certain properties hold. To each tree decomposition a *width* is assigned by taking the size of the biggest bag. The connection between treewidth and tree decomposition is as follows: If some graph G has a treewidth of at most k, then there exists some tree decomposition of G with width at most k.

Decomposing graphs into fragments connected via small interfaces is the basis of many efficient dynamic programming methods for graphs [19] and a prerequisite for graphs to be used as input for an automaton. There are several heuristics for obtaining tree decompositions that work well in practice [20]. By Courcelle's results one can obtain an automaton-theoretic approach solving the aforementioned monadic second-order definable problems on graphs with small treewidth. However, there are also some other approaches, e. g. [4, 41, 50, 51, 56, 72, 103], which are less common. The original idea is of the following form: Let a monadic-second order formula φ which describes the problem to be solved and a tree decomposition of a graph G of width at most k be given. First a finite tree automaton depending on the formula φ and the width k is generated. Then the membership problem is solved for the tree decomposition of G using the tree automaton computed before. However, even if this approach yields a positive theoretical result, it is hard to exploit in practice, due to the huge (exponential) number of states in the resulting automata. Initial work has been presented by Klarlund et al. and Soguet [89, 122], where Klarlund et al. developed a tool, named MONA[1], that encodes monadic second-order logic on trees into tree automata. In [77] Gottlob et al. showed how to reduce different problems which originate from the area of artificial intelligence to the model checking problem of monadic second-order graph logic. To solve these model checking problems they used MONA. However, this thesis does not focus on efficient solutions to the membership problem, i. e. answering the question whether a graph is accepted by a given automaton, but on the use of graph-accepting automata for verification problems.

[1] Available at http://www.brics.dk/mona/

1.2. Publications

In this section the publications of the author are summarized. The thesis is based on the following papers:

- Christoph Blume, H. J. Sander Bruggink, and Barbara König. "Recognizable Graph Languages for Checking Invariants". In: *Proceedings of GT-VMT 2010 (Workshop on Graph Transformation and Visual Modeling Techniques)*. Vol. 29. Electronic Communications of the EASST. 2010.
- Christoph Blume. "Recognizable Graph Languages for the Verification of Dynamic Systems". In: *Proceedings of ICGT '10 (International Conference on Graph Transformation), Proceedings of the Doctoral Symposium*. Ed. by Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr. Vol. 6372. Lecture Notes in Computer Science. Springer, 2010, pp. 384–387.
- Christoph Blume. "Efficient Implementation of Automaton Functors for the Verification of Graph Transformation Systems". In: *Proceedings of ICGT '10 (International Conference on Graph Transformation), Proceedings of the Doctoral Symposium*. Ed. by Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr. Vol. 38. Electronic Communications of the EASST. 2010.
- Christoph Blume, H. J. Sander Bruggink, Martin Friedrich, and Barbara König. "Treewidth, Pathwidth and Cospan Decompositions". In: *Proceedings of GT-VMT 2011 (Workshop on Graph Transformation and Visual Modeling Techniques)*. Vol. 41. Electronic Communications of the EASST. 2011.
- Christoph Blume, H. J. Sander Bruggink, Dominik Engelke, and Barbara König. "Efficient Symbolic Implementation of Graph Automata with Applications to Invariant Checking". In: *Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 7562. Lecture Notes in Computer Science. Springer, 2012, pp. 264–278.
- Christoph Blume, H. J. Sander Bruggink, Martin Friedrich, and Barbara König. "Treewidth, Pathwidth and Cospan Decompositions with Applications to Graph-accepting Tree Automata". In: *Journal of Visual Languages & Computing* 24.3 (2013), pp. 192–206.

1.3. Contributions

The thesis can be split in two parts. The goal of the first part is to provide an automaton-theoretic view on automaton functors, introduced by H. J. Sander Bruggink and Barbara König [33], and to present how classical automaton-based techniques can be used for invariant checking in graph transformation systems (see Figure 1.1). The second part aims at the development of a (prototype) implementation of the techniques that emerged in the first part, which make use of symbolic Bdd-based techniques to handle the state space explosion problem.

In paper [15] the author of this thesis in cooperation with H. J. Sander Bruggink and Barbara König proposed an approximation-based approach for solving the invariant checking problem. This first approach is based on the Myhill-Nerode quasi-order [60] lifted to the setting of recognizable graph languages. But it suffers from the fact that

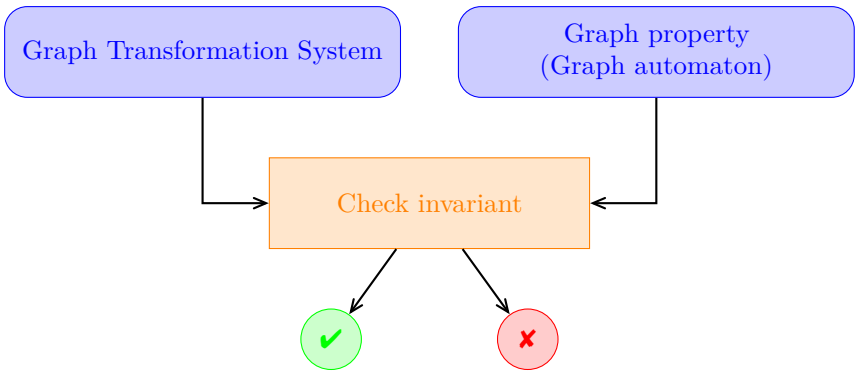


Figure 1.1.: The invariant checking problem

the computation of the Myhill-Nerode quasi-order requires the determinization of the underlying automata. To overcome this state space explosion, the Myhill-Nerode quasi-order has been approximated by a simulation-based approach. Since this modified approach still suffers from the huge size of the involved automata and due to one-sided errors which were caused by the approximation, the main part of this thesis takes a different approach.

The papers [11] and [9] summarize the research project of this thesis and give a further outlook of possible approaches which are realized in this thesis.

In the papers [13] and [14] H. J. Sander Bruggink, Martin Friedrich, Barbara König and the author have presented several categorial notions of so-called cospan decompositions of graphs. More specifically, we have defined several types of cospan decompositions with appropriate width measures and have shown that these width measures coincide with the well-established notions of pathwidth and treewidth. Such cospan decompositions of small width play an important role to efficiently decide graph properties specified by a graph-accepting automaton. Furthermore the authors have shown that the same notion of recognizability is obtained regardless of whether path-like or tree-like cospan decompositions are considered.

Finally, in paper [12] graph automata have been introduced as a more automaton-theoretic view on automaton functors by the author together with H. J. Sander Bruggink, Dominik Engelke and Barbara König. This new approach has been used to establish different techniques from classical automaton theory to the setting of graph automata. Since earlier related work, see paper [15], suffered from the explosion of the size of the automata and the need of approximations due to the non-determinism of the automata, we have employed symbolic Bdd-based techniques and recent antichain algorithms for language inclusion to overcome the previous issues. Furthermore, the presented techniques have been implemented in a software-tool which is used to generate, manipulate and analyze graph automata and to perform experimental evalutions called Raven.

1.4. Structure of this thesis

The structure of this thesis is as follows (see also the dependency graph in Figure 1.2 on page 7):

Chapter 2 – Foundations
In this chapter we fix notations and recall some of the basic notions of regular language theory. Later on, in Chapter 6, we adapt and generalize most of the results for regular languages to the setting of recognizable graph languages. Furthermore, we give a brief introduction of terms and tree automata, which will play a role in Chapters 5 and 6. In order to establish the theory needed behind these concepts, we also give a short introduction to the basic structure of category theory at the end of this chapter.

Chapter 3 – Graphs and Graph Transformation Systems
This chapter introduces the several kinds of graphs which will be used throughout this thesis. Furthermore, we recall the algebraic approach to graph transformation and show the connection to reactive systems.

Chapter 4 – Boolean Functions and Binary Decision Diagrams
The fourth chapter gives a short introduction to Boolean functions which will be needed later on in Chapter 7 in order to represent the huge graph automata. In addition, a short overview over binary decision diagrams is given which are an efficient data structure for representing (large) Boolean functions. Binary decision diagrams are therefore one of the keys to the efficient implementation of Raven.

Chapter 5 – Tree, Path and Cospan Decompositions
In this chapter we generalize the notions of path and tree decompositions of graphs to so-called (path-like and tree-like) cospan decompositions. We will see that, in contrast to the setting of path and tree decompositions, we obtain different variants (of each type) of cospan decompositions. But the main results of this chapter show that all these different variants of cospan decompositions coincide with the well-established notions of path and tree decompositions.

Chapter 6 – Recognizable Graph Languages
This chapter presents recognizable graph languages, which were first investigated by Courcelle and are introduced here in terms of automaton functors: a categorical notion introduced by Bruggink and König to represent recognizable graph languages (and more general languages). We will see that many properties of regular word languages can be generalized to recognizable graph languages. Next, we study a more automaton-theoretic view of automaton functors called graph automata, which are also used in the tool suite Raven (see Chapter 8). Graph automata cannot read all graphs but only graphs up to a certain width (in our case: pathwidth). The second part of this chapter deals with the connection between the well-known monadic second-order logic on graphs and recognizable graph languages. Furthermore, a new logic, called linear cospan logic, is introduced, which is used to automatize the creation of new graph automata.

Chapter 7 – Symbolically Represented Graph Automata
In the seventh chapter we explain how graph automata, whose state spaces

typically grow exponentially in the size of the maximum permitted width, can be efficiently encoded in a symbolic fashion using binary decision diagrams. Furthermore, we investigate different variants of symbolic encodings for the various types of graph automata, since this is the keystone to develop an efficient tool like Raven. Afterwards we present three different recent approaches, which can be used to solve the language inclusion problem for non-deterministic finite automata, and explain how these approaches can be generalized in order to work with graph automata.

Chapter 8 – Raven – A Verification Tool Suite Based on Recognizability

This chapter gives an overview over the tool suite Raven which is developed during and based on the theory of this thesis. The main goal of Raven is to provide a tool suite to handle and manipulate graph automata (introduced in Chapter 6) up to a bounded interface size (respectively up to a bounded pathwidth). After an explanation of the underlying system structure of Raven, we give a short introduction into the use of the software. Subsequently, we compare Raven to other tools used to verify dynamic systems.

Chapter 9 – Experimental Results

In this chapter we present different case studies and give benchmark results which we have obtained for these examples by the tool suite Raven. The list of case studies includes a simple multi-user file system, different languages which are invariants for several graph transformation rules and some statements from graph theory. Besides the comparison of the different algorithms which are presented in Chapter 7, we also give counterexamples which are computed by Raven in cases the result of the invariant or language inclusion check respectively is negative.

Chapter 10 – Conclusion

In the last chapter we discuss related work and draw some conclusions. In the end we provide thoughts on potential connection points for future work.

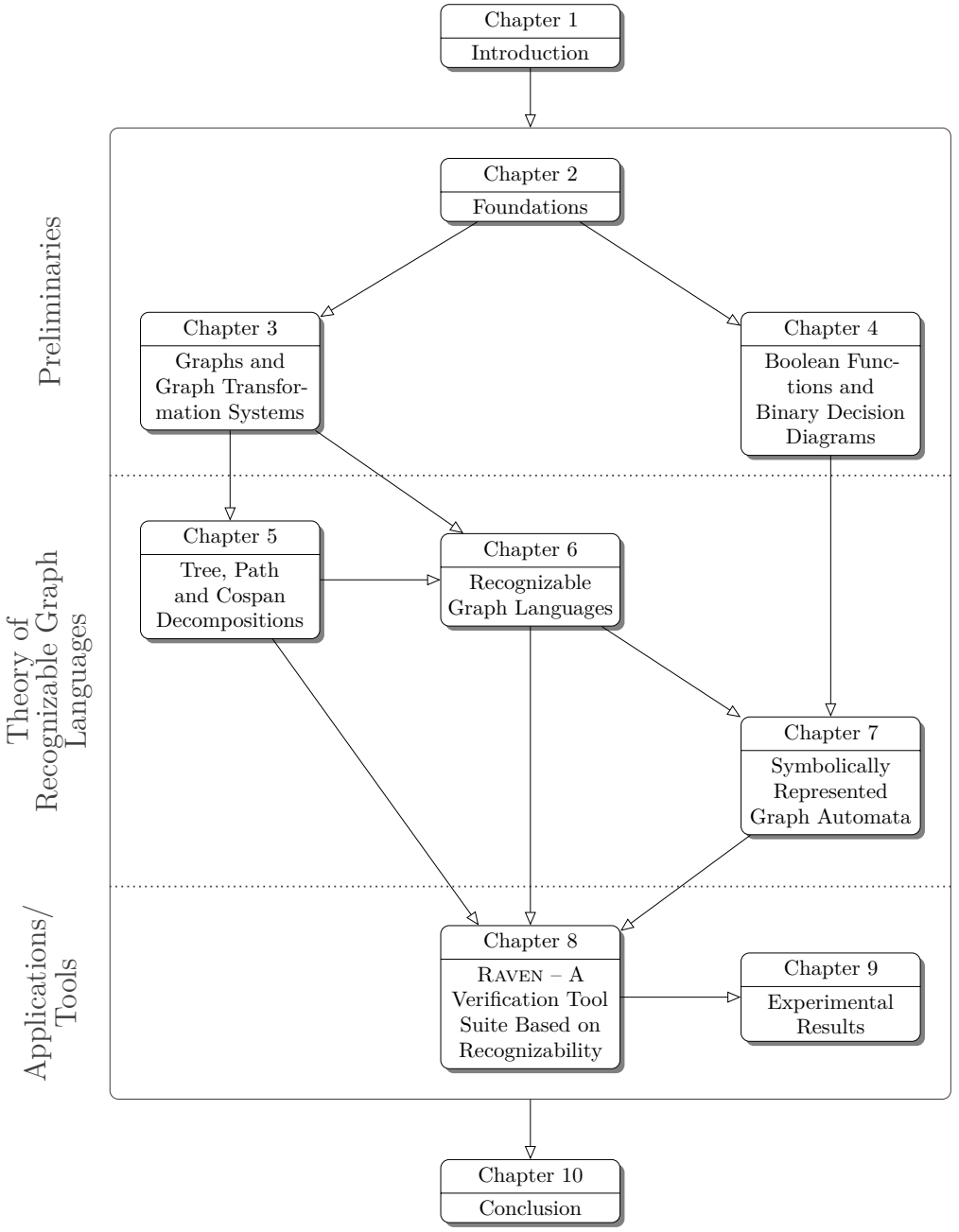


Figure 1.2.: Dependency graph of this thesis

Part I.

Preliminaries

"If I have seen further it is by standing on the shoulders of giants."

Isaac Newton (1642 – 1726)

2 Foundations

In this chapter, we introduce some basic notations which will be used throughout the whole thesis. In the first section we fix and repeat definitions about sets, relations and functions. In the second section we give a short review on finite automata and regular languages. The third section deals with terms and tree automata and in the fourth section a brief introduction to category theory is given. Especially, the notions of pushouts, cospans and other categorial notions which play a fundamental role in the presented theory are introduced.

2.1. Sets, Relations, Orders

By $\mathbb{N}$ we denote the set of *natural numbers*, , i. e. $\{0, 1, 2, \ldots\}$ and by $\mathbb{N}_k$ we denote the set $\{0, 1, 2, \ldots, k-1\}$. The set of *(finite) sequences* over a set A is denoted by A^*. Let $\vec{a} \in A^*$ be a sequence. By $|\vec{a}|$ we denote the *length* of $\vec{a}$ and by $\vec{a}[i]$ the i-th element in the sequence $\vec{a}$. The *powerset* of A is denoted by $\wp(A)$ and, for $n \in \mathbb{N}$, $A^n = A \times \cdots \times A$ denotes the *n-ary cartesian product* of A.

Let A be an arbitrary set and $R \subseteq A \times A$ be a (binary) relation on A. The *inverse relation of R* is denoted by R^{-1}. The relation R is said to be *reflexive* if and only if for all $a \in A$ we have that $\langle a, a\rangle \in R$; it is *symmetric* if and only if for all $\langle a, b\rangle \in R$ it holds that $\langle b, a\rangle \in R$; it is *antisymmetric* if and only if for all $\langle a, b\rangle \in R$ such that $\langle b, a\rangle \in R$ it holds $a = b$; it is *transitive* if and only if for all $\langle a, b\rangle, \langle b, c\rangle \in R$ it also holds that $\langle a, c\rangle \in R$.

Let an arbitrary set A, a relation $R \subseteq A \times A$ and $n \in \mathbb{N}$ be given, we define the relation R^n by

$$R^n = \begin{cases} \{\langle a, a\rangle \mid a \in A\}, & \text{if } n = 0 \\ R \,;\, R^{n-1}, & \text{otherwise} \end{cases}$$

where ; denotes the composition of relations. The *reflexive and transitive closure* of

R is the relation $R^* = \bigcup_{i\in\mathbb{N}} R^i$. The *transitive closure* of R can then be defined as $R^+ = \bigcup_{i\in\mathbb{N}\setminus\{0\}} R^i$.

A *function* f from A to B is written as $f\colon A \to B$ and we implicitly extend it to subsets and sequences as follows: for $A' \subseteq A$ and $\vec{a} = a_1 \dots a_n \in A^*$ we set $f(A') = \{f(a) \mid a \in A'\}$ and $f(\vec{a}) = f(a_1) \dots f(a_n)$.

Definition 2.1 (Quasi-Order, Equivalence, Partial Order)**.** Let A be an arbitrary set. A *quasi-order* $\sqsubseteq$ *on* A is a binary relation on A which is reflexive and transitive. A quasi-order $\sqsubseteq$ is *total* if for all $a, b \in A$ either $a \sqsubseteq b$ or $b \sqsubseteq a$ (or both) holds.
If $\sqsubseteq$ is symmetric it is called an *equivalence*.
If $\sqsubseteq$ is antisymmetric it is called a *(partial) order*.

Let $\equiv$ be an equivalence on an arbitrary set A, then $[\![a]\!]_\equiv$ denotes the *equivalence class* of a (w. r. t. $\equiv$), i. e.

$$[\![a]\!]_\equiv = \{a' \mid a \equiv a'\}.$$

and we call a a *representative* of this equivalence class. By $A/\equiv$ we denote the *quotient set* of all equivalence classes of A by $\equiv$. For convenience, if the equivalence $\equiv$ is clear from the context, we usually write $[\![a]\!]$ instead of $[\![a]\!]_\equiv$. The equivalence has a *finite index* if and only if the quotient set $A/\equiv$ contains only finitely many equivalence classes.

Let $\sqsubseteq$ be a quasi-order on A and let $B \subseteq A$. An *upper bound* (or *lower bound* respectively) of B is an element $a \in A$ such that for all $b \in B$ it holds that $b \sqsubseteq a$ (or $a \sqsubseteq b$ respectively). The *least upper bound* (or *greatest lower bound* respectively) of B is the upper bound (lower bound) a such that for every other upper bound (lower bound) a' it holds that $a \sqsubseteq a'$ ($a' \sqsubseteq a$). We denote the *least upper bound* (*greatest lower bound*) of B by $\bigsqcup B$ (or $\bigsqcap B$ respectively). But note that the least upper bound (greatest lower bound) need not always exist. An element $a \in B$ is *minimal* (or *maximal* respectively) in B with respect to $\sqsubseteq$ if for all $b \in B$ the condition $a \sqsubseteq b$ (or $b \sqsubseteq a$ respectively) holds. The set of all minimal (maximal) elements of B with respect to $\sqsubseteq$ is denoted by $\lfloor B \rfloor$ ($\lceil B \rceil$).

B is called a *chain* if and only if $\sqsubseteq$ limited to B is total. The dual notion is an *antichain*, i. e. for all $a, b \in B$ neither $a \sqsubseteq b$ nor $b \sqsubseteq a$ holds. The subset B is *upward-closed*, or simply *closed*, with respect to $\sqsubseteq$, if the following condition holds for all $a, b \in A$:

$$\big(a \in B \text{ and } a \sqsubseteq b\big) \implies b \in B.$$

A (possibly infinite) sequence $a_0, a_1, a_2, \dots \in A$ is *ascending* if and only if for all $i \in \mathbb{N}$ it holds that $a_i \sqsubseteq a_{i+1}$. The quasi-order $\sqsubseteq$ satisfies the *ascending chain condition* if and only if for all ascending sequences $a_0, a_1, a_2, \dots \in A$ there exists an index $m \in \mathbb{N}$ such that $a_i = a_{i+1}$ holds for all $i \geq m$. The notions of descending sequences and descending chain condition are defined symmetrically.

Finally, we define the notion of well-quasi-orders which are quasi-orders with an additional property:

Definition 2.2 (Well-Quasi-Order)**.** A quasi-order $\sqsubseteq$ on an arbitrary set A is a *well-quasi-order* if $a_0, a_1, a_2, \dots$ is an infinite sequence in A, then there exist

indices $0 < i < j$ such that $a_i \sqsubseteq a_j$.

Besides the definition above, there are many other equivalent definitions of well quasi-order which states the following proposition due to [83].

Proposition 2.3. Let $\sqsubseteq$ be a quasi-order on an arbitrary set A. The following conditions are equivalent:

(i) $\sqsubseteq$ is a well-quasi-order,

(ii) the ascending chain condition holds for the closed subsets of A (ordered by inclusion),

(iii) every infinite sequence of elements of A has an infinite ascending subsequence,

(iv) every non-empty subset $B \subseteq A$ has at least one minimal element but not more than a finite number of (non-equivalent) minimal elements.

(v) there exists neither an infinite strictly descending sequence in A, nor an infinite number of pairwise incomparable elements of A

Finally, we give two properties of well-quasi-orders which will come in handy for some proofs in the following chapters.

Lemma 2.4. Let $\sqsubseteq_1 \subseteq \sqsubseteq_2$ be quasi-orders on a set A. If $\sqsubseteq_1$ is a well-quasi-order on A, then $\sqsubseteq_2$ is as well.

Proof. By Proposition 2.3, every infinite sequence $a_1, a_2, \ldots$ in A contains an infinite ascending subsequence $a'_1, a'_2, \ldots$, i. e. $a_i \sqsubseteq_1 a_j$ for $i < j$. Since $\sqsubseteq_1 \subseteq \sqsubseteq_2$, it holds that $a_i \sqsubseteq_2 a_j$ for $i < j$. Again by Proposition 2.3, we obtain that $\sqsubseteq_2$ is a well-quasi-order. □

Lemma 2.5. Let $\equiv$ be an equivalence on some set A. The equivalence $\equiv$ has a finite index (i. e. it has finitely many equivalence classes), if and only if $\equiv$ is a well-quasi-order on A.

Proof. Trivial. □

2.2. Formal Languages and Finite Automata

In this section, we review the formalism of finite automata and regular languages. A detailed introduction to the theory of regular languages and finite automata can be found in [86].

An *alphabet* Σ is a non-empty finite set. The elements of $\sigma \in \Sigma$ are called *letters*. A *word* w (over Σ) is a sequence $w = \sigma_0 \ldots \sigma_{n-1}$, where each σ_i is a letter from Σ. By $|w|$ we denote the *length* of w, i. e. $|w| = n$. The *empty word*, i. e. the word of length 0, is denoted by ε. The set of all words over Σ is denoted by Σ^*. A *language* $\mathcal{L}$ *(over* Σ*)*

is a (possibly infinite) set of words over Σ, i. e. $\mathcal{L} \subseteq \Sigma^*$. The *empty language* is denoted by $\emptyset$.

One class of word languages with nice properties is the class of *regular languages*. In this thesis we will define this class as the languages accepted by finite automata:

Definition 2.6 (Non-deterministic finite automaton)**.** A *non-deterministic finite automaton (*NFA*)* is a 5-tuple $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ consisting of

- a finite set of *states* Q,
- a finite *input alphabet* Σ,
- a *transition function* $\delta \colon Q \times \Sigma \to \wp(Q)$ which maps a state q for some letter σ to the set of *successor states*,
- a set $\mathrm{I} \subseteq Q$ of *initial states*, and
- a set $\mathrm{F} \subseteq Q$ of *final states*.

Note that for an automaton $\mathcal{A}$ the successor state of some state $q \in Q$ according to some letter $\sigma \in \Sigma$ must not be unique. Due to the non-determinism of the automaton q may have one, two, or more successors (or even no successor) according to σ.

Let $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ be an NFA, $X \subseteq Q$ be a set of states. By $\overline{X}$ we denote the complement of the state set X, i. e. $\overline{X} = Q \setminus X$. Furthermore, let $w \in \Sigma^*$ be a word. We extend the transition function to X and w in the following way:

$$\hat{\delta} \colon \wp(Q) \times \Sigma^* \to \wp(Q),$$
$$\hat{\delta}(X, w) = \begin{cases} X, & \text{if } w = \varepsilon \\ \hat{\delta}(\delta(X, \sigma), w'), & \text{if } w = \sigma w',\ \sigma \in \Sigma \end{cases}$$

We say that the set X is *accepting* if it contains at least one final state, i. e. $X \cap \mathrm{F} \neq \emptyset$. The *language accepted by some state* $q \in Q$, denoted by $\mathcal{L}(q)$, is defined as

$$\mathcal{L}(q) = \{w \in \Sigma^* \mid \hat{\delta}(q, w) \cap \mathrm{F} \neq \emptyset\}.$$

The *language accepted by a state set* $X \subseteq Q$, denoted by $\mathcal{L}(X)$, is defined as $\mathcal{L}(X) = \bigcup_{q \in X} \mathcal{L}(q)$. The language which is accepted by some automaton is the set of words for which the automaton, starting in some initial state, can reach a final state.

Definition 2.7 (Accepted language, Regular language)**.** The language *accepted* by a non-deterministic finite automaton $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ is

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathrm{I}) = \{w \in \Sigma^* \mid \hat{\delta}(\mathrm{I}, w) \cap \mathrm{F} \neq \emptyset\}.$$

A language $\mathcal{L}$ is *regular* if and only if it is accepted by some non-deterministic finite automaton $\mathcal{A}$, i. e. $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

If we replace the transition function in Definition 2.6 by $\delta\colon Q \times \Sigma \to Q$ and allow only a single initial state, we obtain another type of finite automata: *deterministic finite automata (*Dfa*)*. The language accepted by a Dfa is defined in the obvious way.

A well-known fact is that non-deterministic finite automata and deterministic finite automata accept the same class of languages. To build a Dfa which accepts the same language as a given Nfa we use the so-called *subset construction* which we introduce here:

Definition 2.8 (Subset Construction)**.** Let a non-deterministic finite automaton $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, \mathrm{I}_\mathcal{A}, \mathrm{F}_\mathcal{A}\rangle$ be given. We define a deterministic finite automaton $\mathcal{B} = \langle Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, i_\mathcal{B}, \mathrm{F}_\mathcal{B}\rangle$ as follows:

- $Q_\mathcal{B} = \wp(Q_\mathcal{A})$,
- $\delta_\mathcal{B}\colon Q_\mathcal{B} \times \Sigma \to Q_\mathcal{B}$ with $\delta_\mathcal{B}(S, \sigma) = \bigcup_{s\in S} \delta_\mathcal{A}(s, \sigma)$,
- $i_\mathcal{B} = \mathrm{I}_\mathcal{A}$,
- $\mathrm{F}_\mathcal{B} = \{S \in Q_\mathcal{B} \mid S \cap \mathrm{F}_\mathcal{A} \neq \emptyset\}$

Proposition 2.9. Let $\mathcal{L}$ be an arbitrary language. $\mathcal{L}$ can be accepted by some Nfa if and only if it can be accepted by some Dfa.

Proof. See for example [86]. □

In the rest of this section we give some very important properties of regular languages. One of them is the theorem of Myhill-Nerode which gives an alternative characterization of regular languages in terms of the union of equivalence classes of a monotone (or right-monotone) congruence of finite index. In [60, 102], a generalized order-theoretic variant of the theorem is given which states that a language is regular if and only if it is upward-closed with respect to a monotone well-quasi-order.

But before we go to this important theorem, we will introduce two special relations on arbitrary (word) languages which play an important role for regular languages:

Definition 2.10 (Myhill-Nerode Quasi-Order, Syntactical Congruence)**.** Let $\mathcal{L} \subseteq \Sigma^*$ be a language. The *Myhill-Nerode quasi-order $\leq_\mathcal{L}$ (relative to $\mathcal{L}$)* is defined as

$$x \leq_\mathcal{L} y \iff \Big(\forall u, v \in \Sigma^*\colon uxv \in \mathcal{L} \Longrightarrow uyv \in \mathcal{L}\Big).$$

The *syntactical congruence $\approx_\mathcal{L}$ (relative to $\mathcal{L}$)* is defined as

$$x \approx_\mathcal{L} y \iff \Big(\forall u, v \in \Sigma^*\colon uxv \in \mathcal{L} \Longleftrightarrow uyv \in \mathcal{L}\Big).$$

As one can easily verify, both the Myhill-Nerode quasi-order and the syntactical congruence (with respect to $\mathcal{L}$) are monotone and $\mathcal{L}$ is upward-closed w. r. t. the Myhill-Nerode quasi-order and the syntactical congruence.

As stated above, besides the characterization of a regular language in terms of finite automata, we can give another, equivalent definition of regular languages due to Ehrenfeucht et al. [60]:

Proposition 2.11 (Generalized Myhill-Nerode Theorem)**.** Let $\mathcal{L} \subseteq \Sigma^*$ be a language. The following statements are equivalent:

(i) $\mathcal{L}$ is regular,

(ii) $\mathcal{L}$ is upward-closed with respect to some monotone well-quasi-order $\sqsubseteq$ on Σ^*,

(iii) the Myhill-Nerode quasi-order $\leq_{\mathcal{L}}$ is a well-quasi-order,

(iv) the syntactical congruence $\approx_{\mathcal{L}}$ has only a finite index.

In Chapter 6 we will generalize this result to recognizable graph languages.

Finally, we give some of the closure properties which hold for regular languages (the proofs can be found for example in [86]).

Proposition 2.12 (Closure under Boolean Operators)**.** If $\mathcal{L}_1$ and $\mathcal{L}_2$ are two regular languages, then $\overline{\mathcal{L}_1}$ (the complement of $\mathcal{L}_1$), $\mathcal{L}_1 \cap \mathcal{L}_2$ (the intersection of $\mathcal{L}_1$ and $\mathcal{L}_2$) and $\mathcal{L}_1 \cup \mathcal{L}_2$ (the union of $\mathcal{L}_1$ and $\mathcal{L}_2$) are also regular languages.

Proposition 2.13 (Closure under Concatenation)**.** If $\mathcal{L}_1$ and $\mathcal{L}_2$ are two regular languages, then $\mathcal{L}_1 \,;\, \mathcal{L}_2$ (the concatenation of $\mathcal{L}_1$ and $\mathcal{L}_2$) is also a regular language.

These closure properties will be extended to recognizable graph languages in Chapter 6.

2.3. Many-Sorted Terms and Tree Automata

Tree automata are a generalization of finite automata from strings to first-order terms. They are often defined in terms of algebraic structures, e. g. by Gécseg and Steinby [75], or term rewrite systems of a certain kind, e. g. by Comon et al. [39]. Here, we give a more automata-theoretic definition. Additionally, we extend the formalism to many-sorted terms. Hence, we consider sorted tree-automata here. The necessity for this extension will become clear in Chapter 6.

Let S be a set of *sorts*. An *S-type* is a pair $\langle \vec{s}, s_0 \rangle$, where $\vec{s} \in S^*$ is a sequence of *input sorts* and $s_0 \in S$ is the *output sort*. We will usually denote a type $\langle \vec{s}, s_0 \rangle$, where $\vec{s} = s_1 \dots s_n$, by $\vec{s} \to s_0$ or $\langle s_1, \dots, s_n \rangle \to s_0$, or simply by s_0 if $n = 0$.

An *S-typed set* M is a set M_0 together with a map $\mathit{type}\colon M_0 \to S^* \times S$ which assigns a type τ to each element of M. We will write $(f\colon \tau) \in M$ (or simply $f\colon \tau$ if M is clear from the context) to denote the facts that $f \in M_0$ and $\mathit{type}(f) = \tau$.

The terms we need are supposed to be *linear*, that is, every variable occurs at most once. In order to make linearity an inherent part of the definition, we use *holes*, denoted

by $\Box_s$, where s is the sort of the hole, instead of named variables. For an S-typed set Σ of *function symbols* (we call Σ the *signature*) we inductively define an S-typed set $\mathcal{T}(\Sigma)$ of terms over Σ as follows:

- for $s \in S$ it holds that $(\Box_s : s \to s) \in \mathcal{T}(\Sigma)$;
- if $(f\colon \langle s_1, \ldots, s_n\rangle \to s_0) \in \Sigma$ and $(t_1\colon \vec{r}_1 \to s_1), \ldots, (t_n\colon \vec{r}_n \to s_n) \in \mathcal{T}(\Sigma)$, then $(f(t_1, \ldots, t_n)\colon \vec{r}_1 \cdots \vec{r}_n \to s_0) \in \mathcal{T}(\Sigma)$.

Note that the second of the above inductive constructions acts as base case if $n = 0$.

If t is a term of type $\langle s_1, \ldots, s_n\rangle \to s_0$ and $t_1, \ldots, t_n$ are terms of types $\vec{r}_1 \to s_1, \ldots, \vec{r}_n \to s_n$, respectively, then $t(t_1, \ldots, t_n)$ is a term of type $\vec{r}_1 \cdots \vec{r}_n \to s_0$ which is constructed by replacing the left-most hole $\Box_{s_1}$ by t_1, the second left-most hole $\Box_{s_2}$ by t_2, etc.

Definition 2.14 (Tree automaton)**.** An S-sorted (non-deterministic, bottom-up) *tree automaton* is a tuple $\mathcal{M} = \langle Q, \Sigma, \Delta, \mathrm{I}, \mathrm{F}\rangle$, where

- $Q = (Q_\sigma)_{\sigma\in S}$ is a family of finite sets of states indexed by S,
- Σ is an S-signature,
- $\Delta = (\Delta_f)_{f\in\Sigma}$ is a family of transition functions indexed by function symbols, where, for $f\colon \langle\sigma_1, \ldots, \sigma_n\rangle \to \tau$, $\Delta_f\colon Q_{\sigma_1} \times \cdots \times Q_{\sigma_n} \to \wp(Q_\tau)$,
- $\mathrm{I} = (\mathrm{I}_\sigma)_{\sigma\in S}$ is a family of sets of initial states, such that $\mathrm{I}_\sigma \subseteq Q_\sigma$ for all $\sigma \in S$,
- $\mathrm{F} = (\mathrm{F}_\sigma)_{\sigma\in S}$ is a family of sets of accepting states, such that $\mathrm{F}_\sigma \subseteq Q_\sigma$ for all $\sigma \in S$.

We define $\widehat{\Delta} = (\widehat{\Delta}_t)_{t\in\mathcal{T}(\Sigma)}$ as a family of transition functions indexed by terms, such that, for $t\colon \langle s_1, \ldots, s_m\rangle \to s_0$,

$$\widehat{\Delta}_t\colon \wp(Q_{s_1}) \times \cdots \times \wp(Q_{s_m}) \to \wp(Q_{s_0})$$

Let $\overline{\Delta}$ be defined as $\overline{\Delta}_f(S_1, \ldots, S_n) = \bigcup \{\Delta_f(q_1, \ldots, q_n) \mid q_1 \in S_1, \ldots, q_n \in S_n\}$. Then $\widehat{\Delta}$ is defined as follows:

$$\widehat{\Delta}_{\Box_s}(S) = S$$
$$\widehat{\Delta}_{f(t_1,\ldots,t_n)}(S_1, \ldots, S_m) = \overline{\Delta}_f(\widehat{\Delta}_{t_1}(\vec{U}_1), \ldots, \widehat{\Delta}_{t_n}(\vec{U}_n)),$$

where $\vec{U}_1 \ldots \vec{U}_n = S_1 \ldots S_m$ and the length of each $\vec{U}_i$ is the same as the number of arguments required by $\widehat{\Delta}_{t_i}$.

A term $t\colon \langle s_1, \ldots, s_m\rangle \to s_0$ is *accepted* by $\mathcal{M}$, if $\widehat{\Delta}_t(\mathrm{I}_{s_1}, \ldots, \mathrm{I}_{s_m}) \cap \mathrm{F}_{s_0} \neq \emptyset$.

The intuition of tree automata is as follows: Given a term t which can be seen as tree, the automaton starts at the leaves of the tree and moves upwards to the root of the tree. On the run of the automaton to each node (representing a sub-term of the input

term) a state is assigned – in case of the leaves these states are all initial. The state of the current term now depends on the states of all sub-terms of the current term and on the current term itself. The input term is accepted if the state assigned to the root node is final.

Definition 2.15 (Accepted Language, Regular Tree Language)**.** Let an S-sorted tree automaton $\mathcal{M}$ be given. The *language* of $\mathcal{M}$, denoted $\mathcal{L}(\mathcal{M})$, is the set of all terms accepted by $\mathcal{M}$.

A language $\mathcal{L}$ is a *regular tree language* if it is the language of some tree automaton $\mathcal{M}$, i. e. $\mathcal{L} = \mathcal{L}(\mathcal{M})$.

The notion of tree automata will play a role in Chapter 6.

2.4. Basic Category Theory

In this section, we will give a short introduction to the basics of category theory which is needed in this thesis. One of the most fundamental notions of this thesis are cospans and cospan categories which we will study throughout the rest of this thesis. For a more detailed introduction to category theory one can take a look at [96] or [109].

Definition 2.16 (Category)**.** A *category* is a 6-tuple

$$\mathbf{C} = \langle \mathcal{O}, \mathcal{M}, dom, cod, id, ; \rangle$$

consisting of

- a class $\mathcal{O}$ of $\mathbf{C}$-*objects*,
- a class $\mathcal{M}(A, B)$ of $\mathbf{C}$-*morphisms*, $dom(f) = A$ and *codomain* (also called $\mathbf{C}$-arrows) for each pair of $\mathbf{C}$-objects $\langle A, B \rangle$ called *hom-class*,
- two functions *dom* and *cod* assigning to each morphism $A \xrightarrow{f} B \in \mathcal{M}$ the *domain* $dom(f) = A$ and *codomain* $cod(f) = B$ respectively,
- for every $\mathbf{C}$-object A the *identity (morphism)* $A \xrightarrow{id_A} A$,
- the *composition* associating to each two $\mathbf{C}$-morphisms $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$ a morphism $A \xrightarrow{f;g} C$, in this case f and g are said to be composable,

which satisfies the following conditions:

- the composition of three $\mathbf{C}$-morphisms $A \xrightarrow{f} B$, $B \xrightarrow{g} C$ and $C \xrightarrow{h} D$ is associative, i. e. it holds $(f \,;\, g) \,;\, h = f \,;\, (g \,;\, h)$,
- the identity is the neutral element with respect to composition, i. e. for each $\mathbf{C}$-morphism $A \xrightarrow{f} B$ it holds that $id_A \,;\, f = f$ and $f \,;\, id_B = f$.

Note that in the above definition the morphisms of some category $\mathbf{C}$ need not be functions, but can be more general associations of elements. But in most of our cases,

the morphisms of the studied categories will be (structure preserving) functions.

Example 2.17. *The category* **Rel** *has sets as objects and relations as morphisms. The composition of relations defined in Section 2 is the composition operator of this category. Its subcategory* **Set** *has only the functional relations (functions) as morphisms and the composition of functions as composition operator.*

Any monoid $\langle M, e, \circ \rangle$[1] *forms a category with a single object* X*. The morphisms from* $X \xrightarrow{m} X$ *are precisely the elements* $m \in M$*, the identity morphism of* X *is the identity* e *of the monoid, and the composition of morphisms is given by the monoid operation* $\circ$*.*

Next, we overload the definition of quasi-orders and equivalences given above by extending the notion of equivalences to categories. On this we consider only *locally small* categories, i. e. categories such that for every two objects the class of corresponding morphisms between them is a set:

Definition 2.18 (Quasi-Order, Well-Quasi-Order)**.** Let $\mathbf{C}$ be a category. A family of relations

$$\sqsubseteq_R = \{R_{C,D} \mid C, D \text{ are objects in } \mathbf{C}\}$$

is called *quasi-order (on* $\mathbf{C}$*)*, if every component $R_{C,D}$ is a quasi-order on $\mathbf{C}$-morphisms from C to D.

A quasi-order on $\mathbf{C}$ is called *(right-)monotone* if and only if for all $\mathbf{C}$-morphisms $C \xrightarrow{f} D$, $C \xrightarrow{g} D$ and $D \xrightarrow{h} E$ it holds that $f \sqsubseteq_R g$ implies $(f \,;\, h) \sqsubseteq_R (g \,;\, h)$.

A subclass $M \subseteq \mathcal{M}$ is called $\sqsubseteq_R$*-upward closed* if and only if for all $\mathbf{C}$-morphisms $C \xrightarrow{f} D$, $C \xrightarrow{g} D$ it holds that $f \in M$ and $f \sqsubseteq_R g$ implies $g \in M$.

A quasi-order $\sqsubseteq_R$ is called a *well-quasi-order* if every relation $R_{C,D} \in \sqsubseteq_R$ is a well-quasi-order.

Note that orders in categories are also considered in enriched categories [74, 88]. But in difference to enriched categories, we do not require that the order is preserved by composition, i. e. $f \sqsubseteq f'$ and $g \sqsubseteq g'$ implies $f \,;\, g \sqsubseteq f' \,;\, g'$, since we will only require right-monotonicity as defined above.

Analogously, we define equivalences and congruences on categories.

Definition 2.19 (Equivalence, Congruence)**.** Let $\mathbf{C}$ be a category. A family of relations

$$\equiv_R = \{R_{C,D} \mid C, D \text{ are objects in } \mathbf{C}\}$$

is called *equivalence (on* $\mathbf{C}$*)*, if every component $R_{C,D}$ is an equivalence relation on $\mathbf{C}$-morphisms from C to D. An equivalence $\equiv_R$ is *locally finite* if each $R_{C,D} \in \equiv_R$ is an equivalence relation of finite index.

An equivalence $\equiv_R$ is called a *(right) congruence* if the following holds for all morphisms $c, c' \colon C \to D$, $d \colon D \to E$:

$$\text{If } c \equiv_R c', \text{ then } (c \,;\, d) \equiv_R c' \,;\, d.$$

[1] A *monoid* $\langle M, e, \circ \rangle$ is a set M together with an identity e and an inner operation $\cdot$ which is associative

Since categories can be considered as a kind of (structured) objects, we can define the category of categories (which has categories as its objects). The morphisms of this category of categories, called functors, are structure-preserving morphisms between these (object-)categories.

Definition 2.20 (Functor)**.** Let $\mathbf{C}$ and $\mathbf{D}$ be two categories. A *functor* $\mathcal{F}$ from $\mathbf{C}$ to $\mathbf{D}$ assigns to each $\mathbf{C}$-object A a $\mathbf{D}$-object $\mathcal{F}(A)$ and to each $\mathbf{C}$-morphism $A \xrightarrow{f} B$ a $\mathbf{D}$-morphism $\mathcal{F}(A) \xrightarrow{\mathcal{F}(f)} \mathcal{F}(B)$ and satisfies the following conditions:

(i) $\mathcal{F}$ preserves composition, i. e. for each pair of composable morphisms f and g it holds that $\mathcal{F}(f \,;\, g) = \mathcal{F}(f) \,;\, \mathcal{F}(g)$, and

(ii) $\mathcal{F}$ preserves identities, i. e. for each $\mathbf{C}$-object it holds that $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$.

We will now define some special types of morphisms, namely monomorphisms, epimorphisms and isomorphisms, which will play important roles in the further chapters.

Definition 2.21 (Monomorphism, Epimorphism, Isomorphism)**.** Let $\mathbf{C}$ be a category and $A \xrightarrow{f} B$ be a morphism.

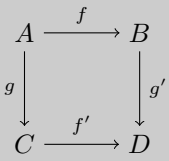
- f is called *monomorphism*, or *mono* for short, if for all morphisms $A' \xrightarrow{g_1} A$ and $A' \xrightarrow{g_2} A$ such that $g_1 \,;\, f = g_2 \,;\, f$ it follows that $g_1 = g_2$.
- f is called *epimorphism*, or *epi* for short, if for all morphisms $B \xrightarrow{h_1} B'$ and $B \xrightarrow{h_2} B'$ such that $f \,;\, h_1 = f \,;\, h_2$ it follows that $h_1 = h_2$.
- f is called *isomorphism* if there exists a morphism $B \xrightarrow{g} A$ such that $f \,;\, g = id_A$ and $g \,;\, f = id_B$.

In the following we will denote monomorphisms by $A \rightarrowtail B$, epimorphisms by $A \twoheadrightarrow B$ and isomorphisms by $A \xrightarrow{\sim} B$.

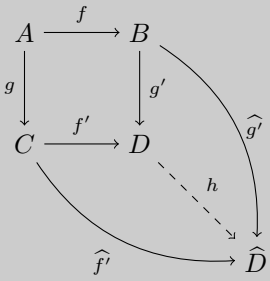
Note that in many cases, for example for the category **Set**, shown above in Example 2.17, the monomorphisms (epimorphisms) are exactly the injective (surjective) functions.

Next, we will give the notion of pushouts which will play a central role in some "gluing constructions" by which we will obtain new objects and morphisms out of known objects and morphisms.

Definition 2.22 (Pushout)**.** Let $\mathbf{C}$ be a category and $A \xrightarrow{f} B$, $A \xrightarrow{g} C$ be two morphisms. A *pushout (of f along g)* $\langle D, f', g' \rangle$ consists of a *pushout object* D and two morphisms $C \xrightarrow{f'} D$ and $B \xrightarrow{g'} D$ such that the following diagram commutes:



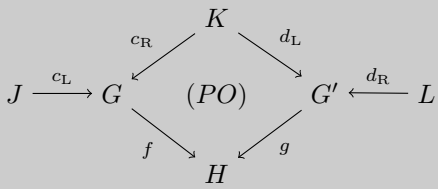
and for every two morphisms $C \xrightarrow{\widehat{f'}} \widehat{D}$ and $B \xrightarrow{\widehat{g'}} \widehat{D}$ such that $f \,;\, \widehat{g'} = g \,;\, \widehat{f'}$ there exists a unique morphism $D \xrightarrow{h} \widehat{D}$ such that the following triangles commute:



We can "generalize" pushouts to the notion of *colimits*. Given a collection (diagram) D of objects $\{A_1, \ldots, A_n\}$ and morphisms between them, the *colimit of* D is an object B together with morphisms $A_i \xrightarrow{\mu_i} B$ such that the diagram commutes, and for each object B' and morphism $A_i \xrightarrow{\mu'_i} B'$ where the diagram commutes, it holds that there exists a unique $h\colon B \to B'$ such that everything commutes. We will write $\mathit{Colim}(D) = B$ in this case.

Next, we define a special type of category: the cospan category. This type of category will be very important for the theory of recognizable graph languages.

Definition 2.23 (Cospan, Cospan Category)**.** Let $\mathbf{C}$ be a category in which all pushouts exist. A *concrete cospan* in $\mathbf{C}$ is a pair $\langle c_\mathrm{L}, c_\mathrm{R} \rangle$ of $\mathbf{C}$-morphisms with the same codomain: $J \xrightarrow{c_\mathrm{L}} G \xleftarrow{c_\mathrm{R}} K$. The composition of two cospans $\langle c_\mathrm{L}, c_\mathrm{R} \rangle, \langle d_\mathrm{L}, d_\mathrm{R} \rangle$ is computed by taking the pushout of the arrows c_R and d_L.



Two concrete cospans are isomorphic if their middle objects are isomorphic such that the isomorphism commutes with the component morphisms of the concrete cospan.

A *cospan* is an isomorphism class of concrete cospans. The *cospan category* $\mathit{Cospan}(\mathbf{C})$ has the same objects as $\mathbf{C}$ and $\mathbf{C}$-cospans as morphisms, i. e. the isomorphism class of a concrete cospan $c\colon J \xrightarrow{c_\mathrm{L}} G \xleftarrow{c_\mathrm{R}} K$ in $\mathbf{C}$ is a morphism from J to K in $\mathit{Cospan}(\mathbf{C})$ and it will be denoted by $c\colon J \nrightarrow K$.

In the following we will confuse cospans and concrete cospans, in the sense that we represent cospans by giving a representative of the isomorphism class. Furthermore, we

introduce the notion of *spans*. Spans are the dual notion of cospans, that is, they are (equivalence classes of) pairs of morphisms with the same domain.

"In mathematics the art of asking questions is more valuable than solving problems."

Georg Cantor (1845 – 1918)

3

Graphs and Graph Transformation Systems

In the first section of this chapter we introduce different types of graphs. On the one hand we have simple graphs and digraphs, which will be used in Chapters 4 and 5 to define some essential notions. On the other hand we define hypergraphs which are the objects we want to decompose in Chapter 5 and which are processed by graph automata introduced in Chapter 6. Finally, we introduce some sort of graph operations which will be used to build all graphs together with disjoint union.

3.1. Simple Graphs

In this section we introduce simple graphs and simple digraphs. Note that these kinds of graphs will only be used to give the notions of path and tree decompositions (defined below) as well as the notion of binary decision diagrams (defined in Chapter 4).

Definition 3.1 (Simple Graph, Simple Digraph)**.**

- A *simple graph* is a pair $\langle V, E\rangle$ where V is a set of *nodes* and $E \subseteq \{\{t_1, t_2\} \mid t_1, t_2 \in V, t_1 \neq t_2\}$ is a finite set of *edges*.
- A *simple digraph* is a pair $\langle V, E\rangle$ where V is a set of *nodes* and $E \subseteq \{\langle t_1, t_2\rangle \mid t_1, t_2 \in V, t_1 \neq t_2\}$ is a finite set of *edges*.

By definition, simple graphs and digraphs do not contain loops and have at most one edge beetween each pair of nodes.

A *rooted* simple graph is a simple graph $\langle V, E\rangle$ with a distinguished node $r \in V$, called the *root* of the simple graph. A *cycle* in a simple graph $\langle V, E\rangle$ is a set of nodes

$\{v_1, \ldots, v_n\} \subseteq V$, such that $\{v_i, v_{i+1}\} \in E$ and $\{v_n, v_1\} \in E$ for $1 \leq i < n$. A simple graph is called *acyclic* if it contains no cycle. These notions are also defined for simple digraphs in the obvious manner.

A *tree* is a simple graph in which for each pair of nodes exactly one path between them exists. A *path graph* is a tree in which each node is connected to either one or two other nodes.

Simple graphs, and in particular trees and path graphs, are only used to define tree and path decompositions in Chapter 5, whereas simple digraphs are only used to define binary decision diagrams in Chapter 4.

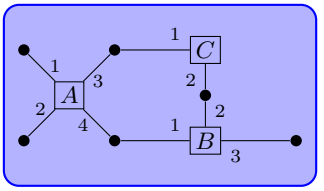
3.2. Hypergraphs and Cospans of Hypergraphs

In this section another kind of graphs is introduced, so-called hypergraphs, which are the type of structures we will investigate further in the next chapters.

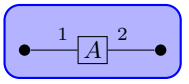
Definition 3.2 (Hypergraph)**.** A *hypergraph* over a set of labels Σ is a structure $G = \langle V, E, att, lab \rangle$, where V is a finite set of nodes, E is a finite set of edges, $att\colon E \to V^*$ maps each edge to a finite sequence of nodes attached to it, and $lab\colon E \to \Sigma$ assigns a label to each edge.

In the following a hypergraph will be simply called *graph*. The *size* of the graph G, denoted $|G|$, is defined to be the cardinality of the sum of its node and edge set, that is $|G| = |V| + |E|$. A *discrete graph* is a graph without edges; the discrete graph with node set $\mathbb{N}_n$ is denoted by D_n. We denote the *empty graph* by $\emptyset$ instead of D_0.

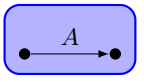
In this thesis, we use the following graphical notation for graphs: Nodes are depicted as small filled circles, hyperedges are depicted as rectangles with their label in the middle of the rectangle and lines between the rectangle and the incident nodes with small numbers attached to indicate in which order the nodes are attached to the edge. An example for this graphical notation is given below:



Furthermore, if the graph contains only edges of degree exactly two, we use the usual notation for directed graphs, where the source node of the edge is the first node and the target of the edge is the second node of the edge. For example



and



denote the same graph.

Definition 3.3 (Hypergraph Morphism)**.** A hypergraph morphism, or graph morphism, $f\colon G \to H$ from a graph $G = \langle V_G, E_G, att_G, lab_G\rangle$ to a graph $H = \langle V_H, E_H, att_H, lab_H\rangle$ is a pair of maps $f = \langle f_V, f_E\rangle$, with $f_V\colon V_G \to V_H$ and $f_E\colon E_G \to E_H$, such that for all $e \in E_G$ it holds that $lab_G(e) = lab_H(f_E(e))$ and $f_V(att_G(e)) = att_H(f_E(e))$.

The set of all graphs together with the set of all graph morphisms establish the *category of graphs and graph morphisms*, denoted by **Graph**. Note that the *monomorphisms* and *epimorphisms* of the category **Graph** are the injective and surjective graph morphisms, respectively.

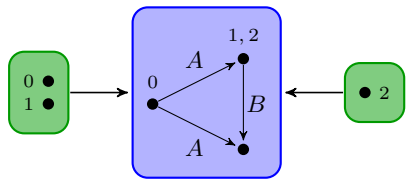
Definition 3.4 (Jointly Node-surjective)**.** Let G, G' and H be graphs. Two morphisms $f\colon G \to H$ and $g\colon G' \to H$ are *jointly node-surjective*, if each node of H has a pre-image in G or G' (along f or g, respectively).

Next, we define the category of cospans of graphs as well as the category of output-linear cospans of graphs.

A cospan $c\colon I \xrightarrow{c_L} G \xleftarrow{c_R} J$ in **Graph** can be viewed as a graph G with two interfaces I and J, called the *inner interface* and *outer interface* respectively. Informally said, only elements of G which are in the image of one of the interfaces can be "touched". Remember that the middle graph G is only fixed up-to isomorphism, as defined in Definition 2.23. An *output-linear* cospan c is a cospan whose right leg, i. e. the morphism c_R, is a monomorphism. By $[G]$ we denote the trivial cospan $\emptyset \to G \leftarrow \emptyset$, the graph G with two empty interfaces. Remember that we will also write $c\colon I \nrightarrow J$ if we are not interested in the middle graph of c.

Now, the *category of cospans of graphs* denoted by *Cospan*(**Graph**) is the category which has graphs as objects and cospans of graphs as morphisms. The *category of output-linear cospans of graphs* **OLCG** (**OLCG**$_n$) is the category which has discrete graphs (of size at most n) as objects and output-linear cospans of graphs (with interfaces of size at most n) as morphisms.

Example 3.5. *Below, a cospan is depicted, which has as inner interface the graph D_2, as outer interface the graph D_1 and as middle graph a 3-clique with binary edges labeled with A and B, respectively. The edges are depicted as arrows from the first to the second node of the edge. The nodes of the two interfaces are mapped to the middle graph as indicated by the numbers* 0, 1, 2*. That is, the nodes of the inner interface are mapped to the left and upper right node, respectively, and the single node of the outer interface is also mapped to the upper right node.*



As a convention, we depict the interfaces of a cospan as rounded green-filled rectangles and the middle graphs as rounded blue-filled rectangles.

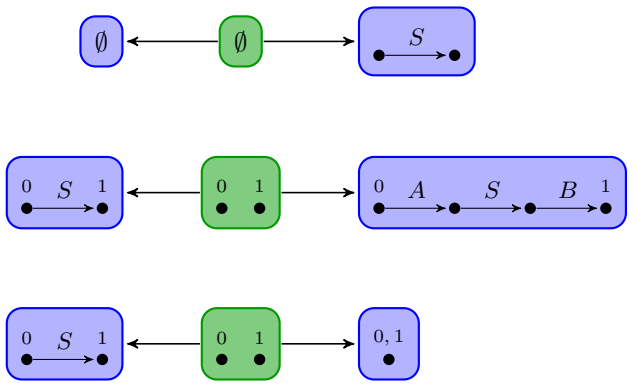
3.3. Graph Transformation Systems

In this section we will give a short introduction into the theory of graph transformation systems, which was introduced by Ehrig, Pfender and Schneider in [64] to establish a generalization from string grammars to graph grammars. The basic idea is to formulate the rewriting step of one graph to another by some kind of "gluing construction", which is done by two pushouts. Therefore, this approach is called *Double Pushout Approach* (DPO) to graph transformation [40, 61–63].

Definition 3.6 (Graph Transformation Rule, Graph Transformation System)**.** A *(graph) transformation rule* $\rho\colon L \leftarrow\ell- I -r\rightarrow R$ is a span consisting of three graphs L, I and R, called the *left-hand side*, the *interface* and the *right-hand side* respectively, and two monomorphisms ℓ and r which specify how the interface is mapped to the left- and right-side respectively.

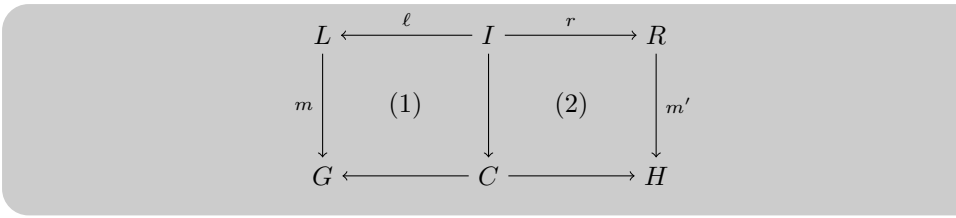
A *graph transformation system* (GTS) is a finite set $\mathcal{R}$ of transformation rules.

Example 3.7. *As an example, we take the following graph transformation system with three different transformation rules:*



The application of a transformation rule to some graph is defined as "gluing construction".

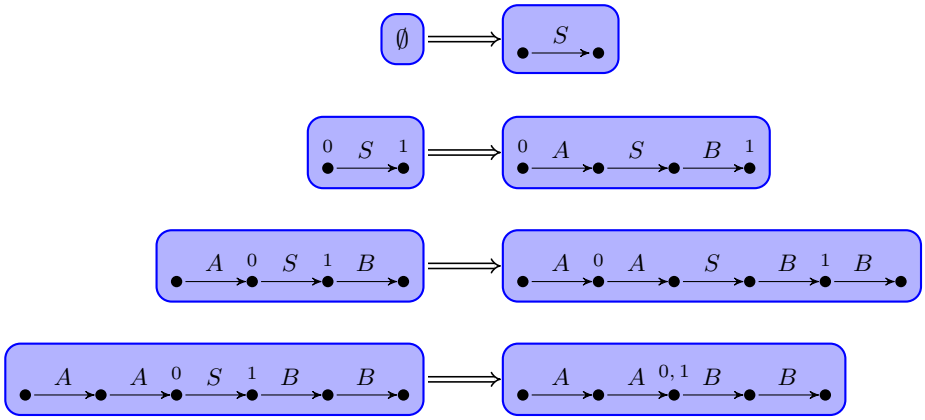
Definition 3.8 (Rule Application)**.** Let $\rho\colon L \leftarrow\ell- I -r\rightarrow R$ be a transformation rule such that ℓ and r are monomorphisms. The rule ρ is *applicable* to some graph G and *rewrites* it to a graph H if and only if there exist two morphisms m and m', called the *match* and *co-match* respectively, and an object C, such that (1) and (2) in the following diagram are both pushouts.



The main idea of the rule application is that the parts of the left-hand side L (of some transformation rule), which are not in the image $\ell(I)$ of the interface I, are removed from the graph G which yields the context graph C. Afterwards, the right-hand side R is "glued" over the image $r(I)$ of the interface into the context graph, yielding the resulting graph H.

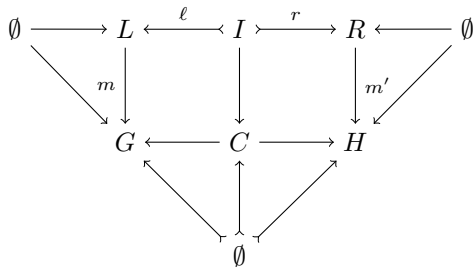
If a transformation rule ρ is applicable to a graph G by some match m, we denote the direct derivation of G to H by $G \overset{\rho,m}{\Longrightarrow} H$. If ρ and m are clear from the context we will usually write $G \Longrightarrow H$.

Example 3.9. *As an example we take a further look at the graph transformation system depicted above. If we start with the empty graph, we can derive the following graphs:*



Note that there is a close connection between the DPO-approach and cospans (of graphs) over reactive systems introduced by Leifner and Milner [98, 120]. Let $c_\ell \colon \emptyset \nrightarrow I$ and $c_r \colon \emptyset \nrightarrow I$ be two output-linear cospans (called *left-hand* and *right-hand side*). The pair $\rho = \langle c_\ell, c_r \rangle$ is called a *reaction rule*. A *reactive system* (over cospans of graphs) consists of a set of reaction rules and a collection of so-called *contexts*, i. e. cospans of the form $d \colon I \nrightarrow \emptyset$. Let $\rho = \langle c_\ell, c_r \rangle$ be a reaction rule. The rule ρ is *applicable* to a graph G if and only if $[G] = c_\ell \,;\, d$ for some context $d \colon I \to C \leftarrow \emptyset$. In this case we write $G \overset{\rho,c_\ell}{\Longrightarrow} H$, where H is the graph obtained from $[H] = c_r \,;\, d$.

The correspondence between graph transformation systems and reactive systems over cospans can now be pointed out by the diagram depicted below:



We observe that the two inner squares which consist of the morphisms $I \xrightarrow{\ell} L \xrightarrow{m} G$ and $I \to C \to G$ – in case of the left square – and the morphisms $I \xrightarrow{r} R \xrightarrow{m'} H$ and $I \to C \to H$ – in case of the second square – correspond to the double pushout diagram of Definition 3.8. Hence, we can obtain the double pushout diagram of Definition 3.8 by omitting the empty graphs and the corresponding morphisms in the diagram given above. On the other hand, we know that there is a unique morphism from the empty graph $\emptyset$ to any other graph. Therefore, we can obtain the diagram shown above from the diagram of Definition 3.8 by adding the missing unique morphisms. Altogether we can conclude: the graph G can be rewritten to the graph H by the application of the (DPO) transformation rule $\rho\colon L \xleftarrow{\ell} I \xrightarrow{r} R$ if and only if there exists a context $d\colon I \to C \leftarrow \emptyset$ such that $[G] = c_\ell \,;\, d$ and $[H] = c_r \,;\, d$ holds for the reaction rule $\sigma = (c_\ell\colon \emptyset \to L \xleftarrow{\ell} I, c_r\colon \emptyset \to R \xleftarrow{r} I)$.

3.4. Atomic Cospans

In this section, we will introduce atomic cospans, which will play the role of "atomic building blocks" for graph automata (see Chapter 6) similar to letters in the case of word automata.

In the rest of this thesis we assume that the set of nodes of each discrete graph D_n is $V_{D_n} = \{v_0, \dots v_{n-1}\}$. We denote the *disjoint union of two graphs* G_1 *and* G_2 by $G_1 \oplus G_2$. We assume that G_1 and G_2 are disjoint. Furthermore, we define the *disjoint union* $G_1 \oplus G_2 \xrightarrow{f \oplus g} H_1 \oplus H_2$ *of two graph morphisms* $G_1 \xrightarrow{f} H_1$ *and* $G_2 \xrightarrow{g} H_2$ where H_1 and H_2 are disjoint as follows:

$$(f \oplus g)(v) = \begin{cases} f(v), & \text{if } v \in V_{G_1} \\ g(v), & \text{if } v \in V_{G_2} \end{cases} \quad \text{and} \quad (f \oplus g)(e) = \begin{cases} f(e), & \text{if } e \in E_{G_1} \\ g(e), & \text{if } e \in E_{G_2} \end{cases}.$$

Now, we fix the following five (families of) atomic cospans, which are inspired by the graph operations introduced by Bauderon and Courcelle in [7]:

Definition 3.10 (Atomic cospans)**.**

Connection of a single hyperedge: Let an edge label $A \in \Sigma$ with $ar(A) \leq n$ and a function $\theta\colon \mathbb{N}_{ar(N)} \to \mathbb{N}_n$ be given. The function θ defines how the new edge

is connected to the nodes in the interface. We define the cospan as follows:

$$connect^n_{A,\theta} \colon D_n \xrightarrow{id'} G \xleftarrow{id'} D_n,$$

where $id'(x) = x$ for $x < n$ and $G = \langle \mathbb{N}_n, \{e\}, att, lab \rangle$ with $att(e) = \theta(1) \dots \theta(ar(A))$ and $lab(e) = A$.

Fusion of two nodes: Let $1 \leq i, j \leq n$ with $i \neq j$. We define the cospan as follows:

$$fuse^n_{i,j} = D_n \xrightarrow{\phi} D_{n-1} \xleftarrow{id} D_{n-1},$$

where $\phi(x) \colon D_n \to D_{n-1}$ is defined as

$$\phi(x) = \begin{cases} x & \text{if } x < j \\ i & \text{if } x = j \\ x - 1 & \text{if } x > j. \end{cases}$$

Permutation of the outer interface: We define the cospan as follows:

$$perm^n_\pi \colon D_n \xrightarrow{id} D_n \xleftarrow{\pi} D_n,$$

where $\pi \colon D_n \to D_n$ is a permutation.

Restriction of the outer interface: Let $k \in \mathbb{N}_{n+1}$ be given. We define the cospan as follows:

$$res^n_k \colon D_n \xrightarrow{id} D_n \xleftarrow{\rho} D_{n-1},$$

where $\rho \colon D_{n-1} \to D_n$ is defined as

$$\rho(x) = \begin{cases} x, & \text{if } x < k \\ x + 1, & \text{if } x \geq k \end{cases}.$$

Disjoint union with a single node: Let $k \in \mathbb{N}_{n+1}$ be given. We define the cospan as follows:

$$vertex^n_k \colon D_n \xrightarrow{\alpha} D_{n+1} \xleftarrow{id} D_{n+1},$$

where $\alpha \colon D_n \to D_{n+1}$ is defined as

$$\alpha(x) = \begin{cases} x, & \text{if } x < k \\ x + 1, & \text{if } x \geq k \end{cases}.$$

In Table 3.1 the five atomic cospans are depicted graphically.

Example 3.11. *We can use atomic cospans in order to build more complex cospans. For example the cospan depicted below (where all edges are labeled with some default label $\diamond$)*

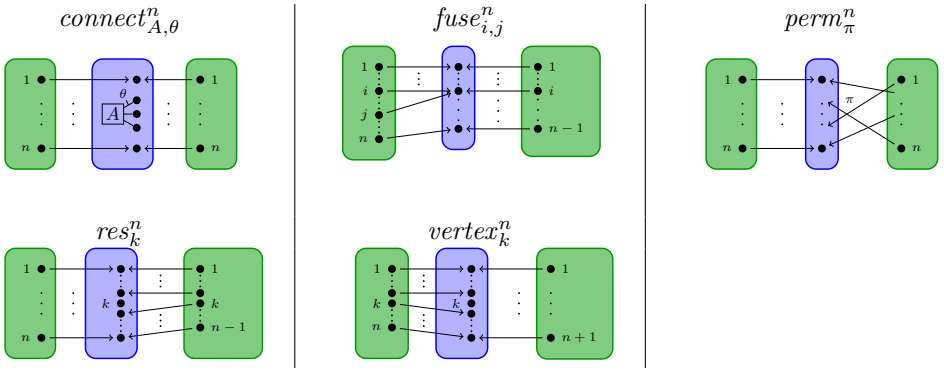
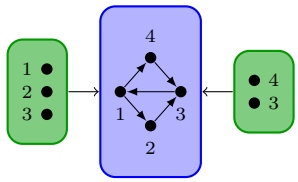


Table 3.1.: Atomic Cospans



can be build by the following sequence of atomic cospans:

$$connect^3_{12} \,;\, connect^3_{23} \,;\, connect^3_{31} \,;\, res^3_2 \,;\, vertex^2_3 \,;\, connect^3_{13} \,;\, res^3_1 \,;\, connect^2_{21} \,;\, perm^2_{21}$$

where we write $connect^n_{ij}$ *for* $connect^n_{\diamond,\theta}$ *with* $\theta(1) = i$, $\theta(2) = j$ *and* $perm^n_{i_1 \ldots i_n}$ *for* $perm^n_\pi$ *with* $\theta(1) = i_1$, $\theta(n) = i_n$.

"All men are mortal. Socrates was mortal. Therefore, all men are Socrates."

Woody Allen (1935 – present)

4

Boolean Functions and Binary Decision Diagrams

In this chapter we review the notion of Boolean functions and we introduce binary decision diagrams as compact and symbolical graphical representation of Boolean functions. We will see that there exists a special kind of binary decision diagrams which can be used to implement both an equivalence check and the usual operations on Boolean functions very efficiently.

4.1. Boolean Functions

In this section we will introduce Boolean functions, which will be used in Chapter 7 in order to define and represent the graph automata introduced in Chapter 6. A further introduction into the theory of Boolean logic can be found in the book of Crama and Hammer [52].

Let $\mathbb{B}$ be the set of *truth values*, i. e. $\mathbb{B} = \{0, 1\}$, where 1 denotes the value *true* and 0 denotes the value *false* respectively. By $\mathbb{B}^n$ we denote the set of all *bit vectors* of length n.

Definition 4.1 (Boolean Variables, Boolean Formula)**.** Let $X = \{x_1, \ldots, x_n\}$ be given. The elements of X are called *(Boolean) variables*. A *Boolean formula (over X)* is defined inductively:

(i) The constants $\top$, $\bot$ and the variables $x_1, \ldots, x_n$ are Boolean formulas,

(ii) if φ is a Boolean formula, then $\neg(\varphi)$ is a Boolean formula,

(iii) if φ and ψ are Boolean formulas, then $(\varphi \wedge \psi)$ is a Boolean formula.

Usually, we write $\varphi(x_1, \ldots, x_n)$ to express that φ is a Boolean formula in the variables $x_1, \ldots, x_n$. By $\Phi(X)$ we denote the set of all Boolean formulas over the set X.

Beside the operations introduced above, we define the following binary operations as abbreviations: *or* ($\vee$), *implication* ($\rightarrow$), *biconditional* ($\leftrightarrow$). These operations are defined in the usual way.

Let ψ be an arbitrary Boolean formula, in which the Boolean variable x_i does not occur. We define the *substitution* of every occurrence of x_i in a formula φ with ψ by $\varphi[x_i/\psi]$. Furthermore, we introduce the *existential quantification* ($\exists\, x_i(\varphi)$) and the *universal quantification* ($\forall\, x_i(\varphi)$) of Boolean variable x_i as the following formulas using substitution:

$$\exists\, x_i(\varphi) = \varphi[x_i/\top] \vee \varphi[x_i/\bot] \quad \text{and} \quad \forall\, x_i(\varphi) = \varphi[x_i/\top] \wedge \varphi[x_i/\bot],.$$

After introducing the syntax, we give now the semantics of Boolean formulas.

Definition 4.2 (Valuation)**.** Let $X = \{x_1, \ldots, x_n\}$. A *valuation (over X)* $\eta\colon X \to \mathbb{B}$ assigns to each variable $x_i \in X$ a truth value $\eta(x_i)$. The (extended) valuation $[\![\cdot]\!]_\eta\colon \Phi(X) \to \mathbb{B}$ assigns to each Boolean formula (over X) a truth value by

$$[\![\top]\!]_\eta = 1 \qquad [\![x_i]\!]_\eta = \eta(x_i) \qquad [\![\varphi \wedge \psi]\!]_\eta = \min\{[\![\varphi]\!]_\eta, [\![\psi]\!]_\eta\}$$
$$[\![\bot]\!]_\eta = 0 \qquad [\![\neg\varphi]\!]_\eta = 1 - [\![\varphi]\!]_\eta.$$

Let η be a valuation over X, $x \in X$ be a variable and $t \in \mathbb{B}$ be a truth value, by $\eta[x \mapsto t]$ we denote the valuation defined as

$$\eta[x \mapsto t]\colon X \to \mathbb{B}, \quad y \mapsto \begin{cases} t, & \text{if } y = x \\ \eta(y), & \text{else} \end{cases}.$$

This can be generalized to an arbitrary number of substitutions in the obvious manner. Now we can define Boolean functions.

Definition 4.3 (Boolean Function)**.** Let $n \geq 0$. A *Boolean function* of n variables is a function $f\colon \mathbb{B}^n \to \mathbb{B}$. The Boolean function f_φ represented by a Boolean formula φ over $\{x_1, \ldots, x_n\}$ is the unique Boolean function on $\mathbb{B}^n$ defined as follows:

$$\forall \langle t_1, \ldots, t_n \rangle \in \mathbb{B}^n\colon f_\varphi(t_1, \ldots, t_n) = [\![\varphi]\!]_{\eta[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]}.$$

Example 4.4. *Let the following Boolean formula*

$$\varphi = (x_1 \rightarrow \neg x_2) \wedge (x_2 \vee \neg x_3)$$

and the following valution

$$\eta\colon \{x_1, x_2, x_3\} \to \mathbb{B}, \quad x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1.$$

be given. The formula can be evaluated as follows:

$$\begin{aligned}
[\![\varphi]\!] &= [\![(x_1 \to \neg x_2) \wedge (x_2 \vee \neg x_3)]\!] \\
&= [\![(\neg(x_1 \wedge x_2) \wedge \neg(\neg x_2 \wedge x_3)]\!] \\
&= \min\{[\![\neg(x_1 \wedge x_2)]\!], [\![\neg(\neg x_2 \wedge x_3)]\!]\} \\
&= \min\{1 - [\![x_1 \wedge x_2]\!], 1 - [\![\neg x_2 \wedge x_3]\!]\} \\
&= \min\{1 - \min\{[\![x_1]\!], [\![x_2]\!]\}, 1 - \min\{[\![\neg x_2]\!], [\![x_3]\!]\}\} \\
&= \min\{1 - \min\{[\![x_1]\!], [\![x_2]\!]\}, 1 - \min\{1 - [\![x_2]\!], [\![x_3]\!]\}\} \\
&= \min\{1 - \min\{\eta(x_1), \eta(x_2)\}, 1 - \min\{1 - \eta(x_2), \eta(x_3)\}\} \\
&= \min\{1 - \min\{0, 1\}, 1 - \min\{0, 1\}\} \\
&= \min\{1, 1\} \\
&= 1
\end{aligned}$$

4.2. Binary Decision Diagrams: A Graphical Data Structure for Boolean Functions

Binary decision diagrams were first studied by Lee [97] and later improved by Akers [3]. In this section we will introduce binary decision diagrams as presented by Bryant [36, 37] which are an efficient data structure for the representation of Boolean functions.

Binary Decision Diagrams

Definition 4.5 (Binary Decision Diagram)**.** Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. A *binary decision diagram*, or BDD for short, (over X) is a rooted, acyclic simple digraph B such that the following conditions are satisfied:

- B has exactly two distinguished leaves, called *terminal nodes*, which are labeled with 0 and 1,
- every inner node of B is labeled with a variable in X,
- every inner node of B has exactly two distinguished outgoing edges, called *high* edge and *low* edge,
- on every path in B from the root node to a terminal node every variable in X occurs at most once.

By $var(n)$ we denote the label of the node n. By $hi(n)$ and $lo(n)$ we denote the children reachable by the outgoing *high* and *low* edges starting at the node n respectively. Furthermore we evaluate a valuation for some formula given as BDD by walking along a path of the BDD in the following way:

(1) Make the root of the BDD the current node.

(2) If the current node is a terminal node go to step (4).

(3) Let n be the current node. If the valuation assigns the value 0 to $var(n)$, make the low child $lo(n)$ the current node, otherwise make the *high* child $hi(n)$ the current node. Then go to step (2).

(4) If the current node is the terminal labeled with 0 the given formula evaluates to *false*, otherwise (if the terminal is labeled with 1) the given formula evaluates to *true*.

As an example, we consider the following set $M = \{0000, 0011, 1100, 1111\}$ of 4-bit vectors. We assume that the bits of the bit vectors are numbered from b_0 to b_3 with b_0 the least significant bit. The BDD representing this set of bit vectors is shown in Figure 4.1. Variables are depicted as rounded gray filled nodes, terminals as blue and red filled rectangular nodes. The *high* and *low* edges are depicted as blue solid and red dashed lines respectively. The root of the BDD is the node labeled with b_0. For the sake of clarity the node identities are omitted. Note that a bit vector $b_0\, b_1\, b_2\, b_3$ is contained in the set M if and only if there exists a path (according to the bit vector) in the BDD shown in Figure 4.1 that leads to the terminal labeled with 1. Hence, every bit vector whose appropiate path leads to the terminal labeled with 0 is not contained in M.

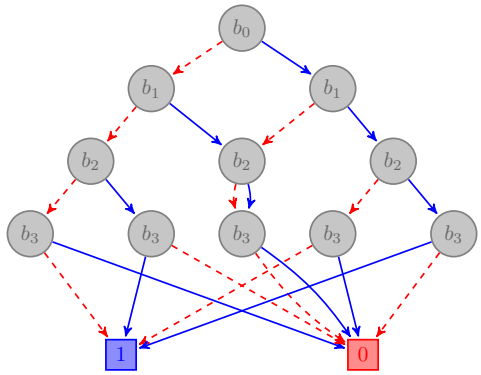


Figure 4.1.: BDD for the set $\{0000, 0011, 1100, 1111\}$

In the further chapters we will use a special class of BDDs, but before we define this special class we will have a look at the relationship between BDDs and Boolean formulas.

Definition 4.6. Let B be a binary decision diagram over $\{x_1, \ldots, x_n\}$. To every node n in B we assign a Boolean formula φ_n inductively:

1. If n is a terminal node then $\varphi_n = \begin{cases} \bot, & \text{if } n \text{ is labeled with 0} \\ \top, & \text{otherwise} \end{cases}$

2. If n is an inner node labeled with $x \in X$, then

$$\varphi_n = (x \to \varphi_{hi(n)}) \wedge (\neg x \to \varphi_{lo(n)})$$

Note that the last identity is also called *Shannon Expansion with respect to x* in the literature [121].

Now, we can identify a formula with a BDD in the following way: Let B be a binary decision diagram with root r. The Boolean formula *represented by B* is then the formula φ_r. An (arbitrary) Boolean formula is *represented by a binary decision diagram B rooted at r* if and only if it is equivalent to φ_r.

If we again take at look at the BDD depicted in Figure 4.1, we can obtain the represented formula (where unsatisfiable subformulas are omitted)

$$\varphi_r = \Big(b_0 \to \big(b_1 \to ((b_2 \to b_3) \wedge (\neg b_2 \to \neg b_3))\big)\Big) \\ \wedge \Big(\neg b_0 \to \big(\neg b_1 \to ((b_2 \to b_3) \wedge (\neg b_2 \to \neg b_3))\big)\Big).$$

Reduced and Ordered Binary Decision Diagrams

In the following we will define so-called *reduced and ordered binary decision diagrams* (ROBDDs). The difference of this class of BDDs to the general class of BDDs is that on ROBDDs an ordering on the variables (represented by the nodes of the BDD) is imposed and redundancy is eliminated.

Definition 4.7 (Reduced and Ordered Binary Decision Diagram)**.** Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables and $\sqsubseteq$ a total order on X. A *reduced and ordered binary decision diagram*, or ROBDD for short, (over X) is a binary decision diagram B which satisfies the following additional conditions:

- for every inner node n the two successor nodes of n are distinct, i. e. $lo(n) \neq hi(n)$,
- for every pair of inner nodes n, m the sub-BDDs rooted at n and m are not isomorphic,
- for every inner node n of B labeled with x_i and for every node m, labeled with x_j, in the subBDDs rooted at n it holds $x_i \sqsubseteq x_j$.

To find a good ordering of the variables of the ROBDD is essential. For two ROBDDs representing the same Boolean formula, but with two different variable orderings, the difference of the size of these two ROBDDs can be exponential in the number of BDD nodes.

For example we take a look at the formula: $\psi = (x_0 \wedge y_0) \vee (x_1 \wedge y_1) \vee (x_2 \wedge y_2)$, which is taken from Bryant [37]. In Figure 4.2 two ROBDDs for this formula are depicted based on two different variable orderings. The variable orderings imposed on the left ROBDD, shown in Figure 4.2a, is $x_0 < y_0 < x_1 < y_1 < x_2 < y_2$. For the right ROBDD the variable ordering is $x_0 < x_1 < x_2 < y_0 < y_1 < y_2$.

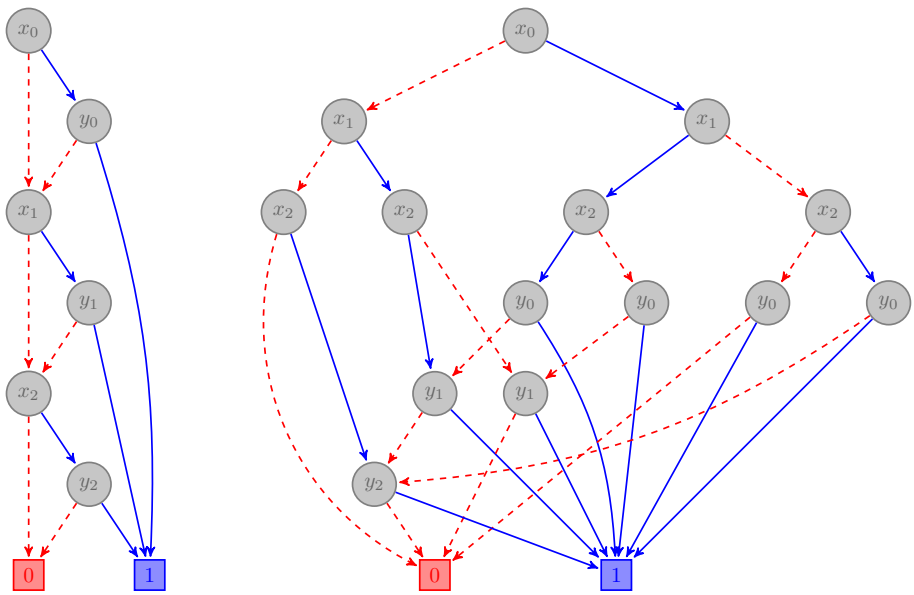


Figure 4.2.: Two ROBDDs with different variable orderings for the formula $\psi = (x_0 \wedge y_0) \vee (x_1 \wedge y_1) \vee (x_2 \wedge y_2)$.

As we can easily observe the variable y_i is strongly connected to the variable x_i (for $i \in \{1, 2, 3\}$) (and vice versa, since $\wedge$ is commutative). This is due to the fact that the formula ψ can be evaluated to *true* if and only if for at least one pair x_i and y_i ($i \in \{1, 2, 3\}$) both variables are evaluated to *true*. Therefore, we can make the following observations:

- If some variable y depends on a variable x, then we should choose a variable ordering such that $x < y$.
- If a variable y depends on a variable x, but the variable y does *not* depend on a variable z, then we should avoid a variable ordering such that $x < z < y$ if possible.

In Chapter 7 we will discuss the different variable orderings for the ROBDDs used for the implementation of the graph automata presented in this thesis. In that chapter we will see, that a good variable ordering, i. e. a variable ordering such that the resulting ROBDDs are relatively small, is essential to be able to use our graph automata.

The great advantage of ROBDDs over BDDs is, that, for a fixed variable ordering, there exists only one ROBDD representing a given Boolean formula (up to isomorphism). This property can be used for very efficient equivalence checks, since two Boolean formulas, represented as ROBDDs are equivalent if and only if the two ROBDDs are equal (up to isomorphism).

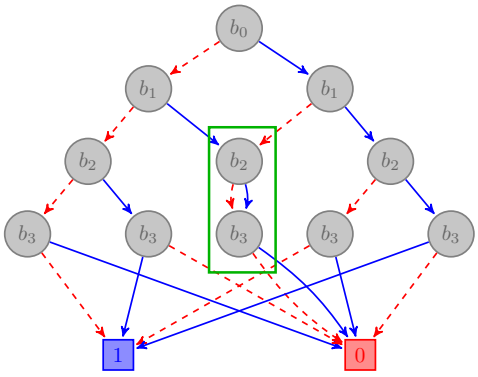
In the remaining parts of this thesis, we assume that the nodes of each BDD are ordered regarding to some ordering imposed on the variables of the BDD. In order

to transform an arbitrary BDD B to an equivalent ROBDD we repeatedly apply the following rules until the desired ROBDD is obtained:

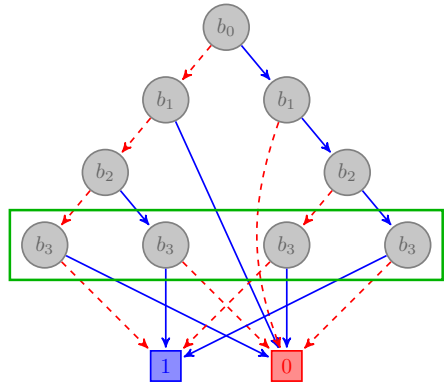
Remove redundant nodes: If an inner node $n \in B$ exists such that $lo(n) = hi(n)$, then remove the node n and redirect all incoming edges of n to $lo(n)$.

Remove duplicate nodes: If two inner nodes $m, n \in B$ exist such that $var(m) = var(n)$, $lo(m) = lo(n)$ and $hi(m) = hi(n)$, then remove the node m and redirect all incoming edges of m to the node n.

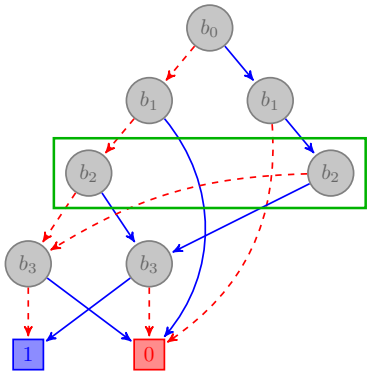
Example 4.8. *As an example we take the* BDD *which is depicted in Figure 4.1 and we use the following variable ordering:* $b_0 < b_1 < b_2 < b_3$.



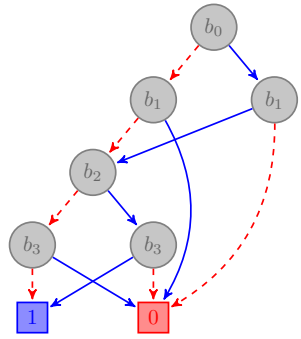
(a) Initial BDD, Next Step: Remove redundant nodes



(b) BDD after removing redundant tests, Next Step: Remove duplicate nodes



(c) BDD after first removal of duplicate inner nodes, Next Step: Remove duplicate nodes



(d) Resulting ROBDD

Figure 4.3.: The different BDDs which are obtained by the transformation of the BDD shown in Figure 4.1

In the first step we will remove inner nodes which are redundant, i. e. nodes n *such that* $hi(n) = lo(n)$. *The two nodes within the rectangle depicted in Figure 4.3a are*

redundant and can be removed, resulting in the BDD *shown in Figure 4.3b. Now, we will remove duplicate inner nodes, more precisely we eliminate a node n as long as there exists a node m such that n and m have the same label and it holds $hi(n) = hi(m)$ and $lo(n) = lo(m)$. All incoming edges of n will be redirected to be incoming edges of m. The third and fourth node within the rectangle depicted in Figure 4.3b are duplicates of the first and second node. Therefore, these two nodes can be removed, which will yield the* BDD *depicted in Figure 4.3c. In the third (and last step) another duplicate node can be removed. The right node within the rectangle shown in Figure 4.3c is a duplicate of the left node and can therefore also be eliminated. The resulting* BDD*, shown in Figure 4.3d, is the desired* ROBDD *for the set M and the ordering given above.*

Note that we do not mention here how to obtain a BDD from a given Boolean function. For detailed algorithms which can be used to directly transform a Boolean function into an equivalent ROBDD we refer to [37].

Relations And Binary Decision Diagrams

For the rest of this chapter, we assume that all sets and relations are finite. Let M be an arbitrary set. We can identify the elements of M with bit vectors (of length $\lceil \log_2 |M| \rceil$[1]) in the obvious way. Therefore, we can restrict our attention to subsets of and relations on $\mathbb{B}^n$. Let R be an arbitrary binary relation on $\mathbb{B}^n$. We represent the relation R as a subset R' of $\mathbb{B}^{2n}$ in the following way:

$$\langle b_1 \dots b_n, c_1 \dots c_n \rangle \in R \iff b_1 c_1 \dots b_n c_n \in R'$$

Note that the bit vectors are interleaved in the set R'. This is a rather "technical" step to obtain smaller BDDs due to the fact that bits which store the same "piece of information" (i. e. the bits b_i and c_i respectively, for $1 \leq i \leq n$) are mapped to consecutive nodes in the BDD.

4.3. Conclusion

In this chapter we have introduced Boolean functions and binary decision diagrams. We have seen that binary decision diagrams can be used to represent Boolean functions in a very compact way. In Chapter 7 we will use Boolean functions, represented by binary decision diagrams, as one of the keystones to efficiently represent and manipulate graph automata, which we will introduce in Chapter 6.

[1] By $\lceil \cdot \rceil$ we denote the usual ceiling function.

Part II.

Theory of Recognizable Graph Languages

"You need to study other people's work. Their approaches to problem solving and the tools they use give you a fresh way to look at your own work."

Gary Kildall (1942 – 1994)

5

Tree, Path and Cospan Decompositions

In this chapter we will take a deeper look at cospan decompositions of (cospans of) graphs and the relationship between this type of decompositions and the well-known tree and path decompositions introduced by Robertson and Seymour in their seminal work about graph minors [113, 114, 116]. The research on this topic in terms of cospan decompositions has started as diploma thesis [73].

The results of this chapter are very important for the further chapters. Especially the decompositions of (cospans of) graphs into atomic cospans play an important role, since these atomic cospan decompositions are the equivalent concept for automaton functors to input letters for word automata.

All objects we are decomposing will be hypergraphs whereby edges connect to each node at most once. Note that this is the "hypergraph counterpart" to loop-free simple graphs. Hence, we will slighty change the $connect_\theta$-cospan defined in Chapter 3, in a way that θ must be injective. Furthermore, we abstain from the *fuse*-cospan. This is due to the fact that we assume that there exists no node which is connected to some edge more than once. Hence, we consider only cospans which are both input- and output-linear[1] in this chapter and therefore we do not need to fuse nodes. The other atomic cospans introduced in Chapter 3 are used without any modification.

The contents of this chapter have been published in [13, 14].

5.1. Tree and Path Decompositions

In this section we give a short introduction to the notions of tree and path decompositions. Tree and path decompositions play an important role in many different areas of computer science. For example Courcelle's Theorem [44, 91] states that every property definable in monadic second-order graph logic can be decided in linear time for graphs

[1] A cospan $c\colon I \xrightarrow{c_\mathrm{L}} G \xleftarrow{c_\mathrm{R}} J$ is *input- and output-linear* (or *linear* for short) if both c_L and c_R are injective morphisms.

of bounded treewidth (pathwidth) (see also Section 6.2). Many algorithms use dynamic programming to solve the desired problems on tree decompositions of the input graphs [19, 22]. Furthermore, tree decompositions are used in the optimization of database queries [68]. Other fields in which tree and path decompositions are involved include the development of fast routing protocols [70] and the design of (potentially large) circuits [106].

Definition 5.1 (Tree decomposition)**.** Let $G = \langle V, E, att, lab \rangle$ be a graph. A *tree decomposition* of G is a pair $\mathcal{T} = \langle T, X \rangle$, where T is a tree and $X = \{X_{t_1}, \ldots, X_{t_n}\}$ is a family of subsets of V indexed by the nodes of T, such that:

- for each node $v \in V$, there exists a node t of T such that $v \in X_t$;
- for each edge $e \in E$, there is a node t of T such that all nodes v attached to e are in X_t;
- for each node $v \in V$, the simple graph induced by the nodes $\{t \mid v \in X_t\}$ is a subtree of T.

A tree decomposition $\mathcal{T} = \langle T, X \rangle$ is a *path decomposition* if T is in fact a path graph.

The subsets $X_{t_1}, \ldots, X_{t_n}$ are called *bags*. Now we can define the treewidth and pathwidth of tree (path) decomposition respectively. Intuitively, the treewidth and pathwidth measure how similar a graph is to a tree or to a path.

Definition 5.2 (Treewidth, Pathwidth)**.** The *width* of a tree decomposition $\mathcal{T} = \langle T, X \rangle$ is $wd(\mathcal{T}) = \big(\max_{t \in T} |X_t|\big) - 1$.

Now, the *pathwidth* $pwd(G)$ and the *treewidth* $twd(G)$ of a graph G are defined as follows:

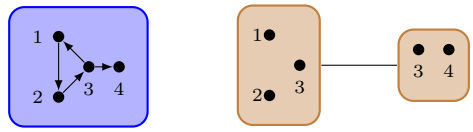
- $pwd(G) = \min\{wd(\mathcal{P}) \mid \mathcal{P} \text{ is a path decomposition of } G\}$,
- $twd(G) = \min\{wd(\mathcal{T}) \mid \mathcal{T} \text{ is a tree decomposition of } G\}$.

Note that the decrement in the definition of $wd(\mathcal{T})$ above is chosen so that discrete graphs have pathwidth and treewidth 0, trees have treewidth 1 and paths have pathwidth 1. Furthermore, an n-clique has both pathwidth and treewidth $n - 1$ [115].

Naturally it holds that $twd(G) \leq pwd(G)$ for all graphs G, where the pathwidth might be substantially larger than the treewidth. For instance, trees can have arbitrarily large pathwidth [7].

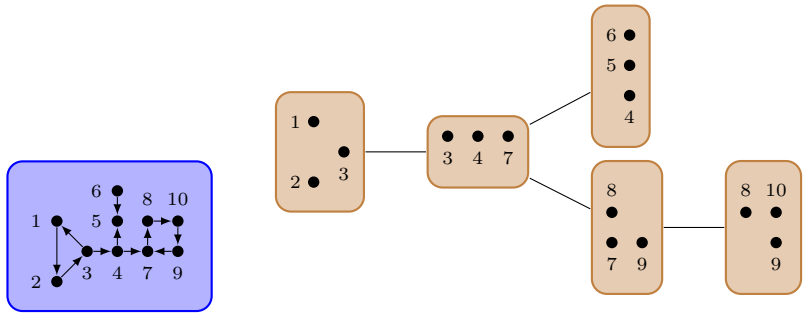
Example 5.3. *As examples we consider only unlabeled directed graphs, that is we take* $\Sigma = \{\diamond\}$ *as alphabet and* $|att(e)| = 2$ *for every edge* e*. Let* G_P *be the graph shown in Figure 5.1a. Obviously, the pathwidth of this graph is* 2 *since it contains a 3-clique (all nodes of which have to be together in at least one bag) and we have a path decomposition* $\mathcal{P}$ *of width* 2 *which is shown in Figure 5.1b.*

As an example for a tree decomposition we consider the unlabeled graph G_T *of Figure 5.2a. The treewidth of this graph is* 2 *due to the fact that it contains a* 3*-clique*



(a) Graph G_P **(b)** Path Decomposition of G_P

Figure 5.1.: The graph G_P and one of its path decompositions



(a) Graph G_T **(b)** Tree Decomposition of G_T

Figure 5.2.: The graph G_T and one of its tree decompositions

and that the tree decomposition $\mathcal{T}$ shown in Figure 5.2b has width 2.

5.2. Path-like Cospan Decompositions

In this section we explore "path-like" cospan decompositions of graphs which are the equivalent concept of path decomposition in the category of graphs. The "tree-like" equivalent to tree decompositions will be discussed in Section 5.3.

Cospan decompositions are naturally defined as sequences of cospans, which are composed to a graph by taking the colimit of the emerging diagram. Equivalently, the cospans can be iteratively composed into a single cospan, where finally the interfaces are ignored.

Definition 5.4 (Cospan decomposition)**.** Let G be a graph and $\vec{c} = c_1, \ldots, c_n$ be a sequence of composable cospans in the category **Graph**. The sequence $\vec{c}$ is a *cospan decomposition* of G, if $Colim(\vec{c}) = G$.

Now we have three different notions which are closely related: cospan decompositions, which are just sequences of cospans; by composing these cospans we obtain the single

cospan ("graph with interfaces"); and if we "forget" the two interfaces we get the center graph of this cospan which is just the colimit of the cospan decomposition.

In the following we are especially interested in cospan decompositions depending on atomic cospans. Remember that we consider only a specific subset of the atomic cospans defined in Chapter 3 due to the fact that we are only interested in linear cospans here:

Lemma 5.5. Let $c = J \xrightarrow{c_L} G \xleftarrow{c_R} K$ be a linear cospan, i. e. c_L, c_R are monos. Then there exist atomic cospans $a_1, \ldots, a_n$ such that $c = a_1 ; \cdots ; a_n$.

In fact, there exist such atomic cospans $a_1, \ldots, a_n$ such that the following condition holds: Let $a_i = D_{n_{i-1}} \to H_i \leftarrow D_{n_i}$, for $1 \leq i \leq n$. It holds that $|D_{n_i}| \leq |G|$ for all $0 \leq i \leq n$ and $|H_i| \leq |G|$ for all $1 \leq i \leq n$.

Proof. See Appendix A.1. □

Note that, for each atomic cospan $c = D_n \xrightarrow{c_L} G \xleftarrow{c_R} D_m$, the cospan $c' = D_m \xrightarrow{c_R} G \xleftarrow{c_L} D_n$, which is obtained by "flipping" c, is also an atomic cospan: the flipped version of $vertex_k^n$ is res_k^{n+1} and vice versa, flipping $connect_{A,\theta}^n$ has no effect and flipping $perm_\pi^n$ results in $perm_{\pi^{-1}}^n$.

Due to the fact that cospans (of graphs) are graphs with two equipped interfaces, we have the free choice which part of a cospan corresponds to a single bag in a path decomposition. Therefore, we consider the following types of cospan decompositions.

Definition 5.6 (Type of cospan decomposition)**.** Let $\vec{c}$ be a cospan decomposition of the graph G.

1. $\vec{c}$ is a *graph-bag decomposition*, if all cospans have discrete interfaces and consist of injective morphisms.
2. $\vec{c}$ is an *interface-bag decomposition*, if it is a graph-bag decomposition, consists of pairs of jointly node-surjective morphisms and it holds for all edges e of G, with $att(e) = v_1 \ldots v_m$, that $v_1, \ldots, v_m$ occur together in some interface[2].
3. $\vec{c}$ is an *atomic cospan decomposition*, if it consists only of atomic cospans[3].

The first two correspond to path decompositions in two different ways: in graph-bag decompositions the center graphs in the cospans correspond to the bags of definition 5.2, whereas in interface-bag decompositions the interfaces play the role of bags. In order to make the relation between path and cospan decompositions clearer, we will only consider decompositions into cospans of injective morphisms in this thesis.

It is clear that the various types of cospan decomposition are strictly contained in one another, that is:

$$\text{Atomic} \subset \text{Interface-bag} \subset \text{Graph-bag} \subset \text{All}.$$

[2] More formally: let $I_1, \ldots, I_n$ be the interfaces of the cospans in $\vec{c}$ and let $f_j \colon I_j \to G$ be the morphisms generated by the colimit. Then there exist an index j and nodes $w_1, \ldots, w_m$ in I_j such that $f_j(w_i) = v_i$ for $i \in \{1, \ldots, m\}$.

[3] Remember that only the atomic cospans $connect_A$, $perm$, res and $vertex$ are permitted in this chapter.

Definition 5.7 (Graph-bag size, interface-bag size)**.** Let $c\colon J \to G \leftarrow K$ be a cospan. We define the *graph-bag size* and *interface-bag size* of c as follows:

$$|c|_{\mathrm{gb}} := |V_G|$$
$$|c|_{\mathrm{ib}} := \max\{|V_J|, |V_K|\}$$

Observe, that for all atomic cospans c it holds that $|c|_{\mathrm{gb}} = |c|_{\mathrm{ib}}$. For convenience later on, we define $|c|_{\mathrm{at}} := |c|_{\mathrm{gb}}$ $(= |c|_{\mathrm{ib}})$.

Now we are ready to define, for all three types of cospan decomposition, a *width*:

Definition 5.8 (Width of cospan decomposition)**.**

- Let $\vec{c} = c_1, \ldots, c_n$ be a cospan decomposition. We define the *graph-bag* and *interface-bag width* of $\vec{c}$ as follows:

$$wd_{\mathrm{gb}}(\vec{c}) := \max\{|c_i|_{\mathrm{gb}} \mid 1 \leq i \leq n\} - 1$$
$$wd_{\mathrm{ib}}(\vec{c}) := \max\{|c_i|_{\mathrm{ib}} \mid 1 \leq i \leq n\} - 1$$

- Let G be a graph. The graph-bag ($cpwd_{\mathrm{gb}}(\vec{c})$), interface-bag ($cpwd_{\mathrm{ib}}(\vec{c})$) and atomic cospan width ($cpwd_{\mathrm{at}}(\vec{c})$) of G are defined as:

$$cpwd_{\mathrm{gb}}(G) := \min\{wd_{\mathrm{gb}}(\vec{c}) \mid \vec{c} \text{ is a graph-bag decomposition of } G\}$$
$$cpwd_{\mathrm{ib}}(G) := \min\{wd_{\mathrm{ib}}(\vec{c}) \mid \vec{c} \text{ is an interface-bag decomposition of } G\}$$
$$cpwd_{\mathrm{at}}(G) := \min\{wd_{\mathrm{ib}}(\vec{c}) \mid \vec{c} \text{ is an atomic cospan decomposition of } G\}$$

Similar to the case of path (and tree) width we decrement the width by one in order to guarantee that discrete graphs have width 0 and paths have width 1.

The main theorem of this section is that, for a given graph, the three notions of cospan pathwidth are the same, and moreover are the same as the pathwidth of the graph. First, we show how to transform (cospan) path decompositions into each other:

Lemma 5.9.

1. Let $\mathcal{P}$ be a path decomposition of a graph G. There exists a graph-bag decomposition $\vec{c}$ of G such that $wd_{\mathrm{gb}}(\vec{c}) = wd(\mathcal{P})$.
2. Let $\vec{c}$ be a graph-bag decomposition of G. There exists an interface-bag decomposition $\vec{d}$ of G such that $wd_{\mathrm{ib}}(\vec{d}) = wd_{\mathrm{gb}}(\vec{c})$.
3. Let $\vec{c}$ be a graph-bag decomposition of G. There exists an atomic cospan decomposition $\vec{d}$ of G such that $wd_{\mathrm{at}}(\vec{d}) \leq wd_{\mathrm{gb}}(\vec{c})$.
4. Let $\vec{c}$ be an interface-bag decomposition of G. There exists a path decomposition $\mathcal{P}$ of G such that $wd(\mathcal{P}) = wd_{\mathrm{ib}}(\vec{c})$.

Proof. See Appendix A.1. □

Theorem 5.10. For every graph G,

$$pwd(G) = cpwd_{\mathrm{gb}}(G) = cpwd_{\mathrm{ib}}(G) = cpwd_{\mathrm{at}}(G).$$

Proof. First of all, because atomic cospan decompositions are also graph-bag and interface-bag cospan decompositions, and it is easy to check that for an atomic cospan decomposition $\vec{c}$, $wd_{\mathrm{gb}}(\vec{c}) = wd_{\mathrm{ib}}(\vec{c})$, it follows for all graphs G that

$$cpwd_{\mathrm{gb}}(G) \leq cpwd_{\mathrm{at}}(G) \text{ and } cpwd_{\mathrm{ib}}(G) \leq cpwd_{\mathrm{at}}(G).$$

Together with Lemma 5.9 (iii) it follows that

$$cpwd_{\mathrm{gb}}(G) \geq cpwd_{\mathrm{at}}(G) \geq cpwd_{\mathrm{gb}}(G). \tag{5.1}$$

From Lemma 5.9 (i), (ii) and (iv) it follows that

$$pwd(G) \geq cpwd_{\mathrm{gb}}(G) \geq cpwd_{\mathrm{ib}}(G) \geq pwd(G). \tag{5.2}$$

The theorem follows directly from equations (5.1) and (5.2). □

Example 5.11. *As an example we take the graph G_P and the corresponding path decomposition $\mathcal{P}$ of Example 5.3. We use the path decomposition to construct a graph-bag decomposition of G_P. For each of the two bags in $\mathcal{P}$ we take a cospan where the center graph of the first cospan is the 3-clique and the center graph of the second cospan contains the edge from the third to the fourth node. The inner interface of the first cospan and the outer interfaces of the second cospan are both empty graphs, while the outer interface of the first cospan (which is the inner interface of the second cospan) contains the third node which is the intersection of both subgraphs. The resulting graph-bag decomposition is depicted in Figure 5.3a. The graph-bag width of G_P is 2, since the resulting graph-bag decomposition has graph-bag size 2, and the graph-bag size of every other graph-bag decomposition must have at least size 2 due to the 3-clique which has to be contained in at least one center graph.*

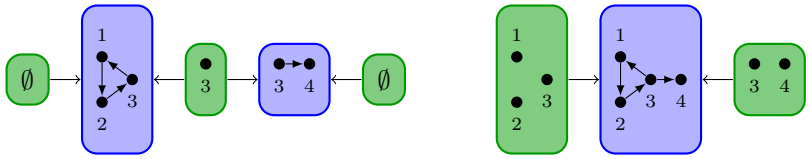
An interface-bag decomposition for the same graph is shown in Figure 5.3b. Note that it indeed satisfies the conditions of definition 5.6: specifically each cospan is jointly node-surjective and all nodes attached to an edge live together in at least one bag. The interface-bag width of G_P is 2, due to the fact that the given interface-bag decomposition has interface-bag width 2 and any other interface-bag decomposition has to contain the nodes of the 3-clique in at least one interface.

Note that in both cases, the graph-bag and the interface-bag decomposition, the bags of each decomposition (the center graphs in the first case and the interfaces in the second case) correspond to the bags of the path decomposition of G_P.

To construct the atomic decomposition we decompose the cospans of the graph-bag decomposition into atomic cospans. This is possible due to Lemma 5.5:

$$vertex_1^0 \,;\, vertex_2^1 \,;\, vertex_3^2 \,;\, connect_{12}^3 \,;\, connect_{31}^3 \,;\, connect_{23}^3 \,;\, res_0^3 \,;\, res_0^2 \,;\, vertex_2^1 \,;\, connect_{12}^2 \,;\, res_0^2 \,;\, res_0^1,$$

where we write $connect_{ij}^n$ for $connect_{\diamond,\theta}^n$ with $\theta(1) = i, \theta(2) = j$.



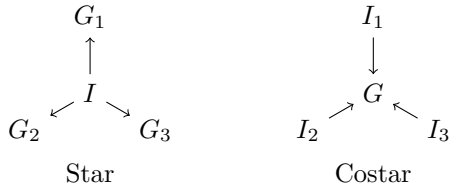
(a) Graph-bag decomposition of G_P

(b) Interface-bag decomposition of G_P

Figure 5.3.: Graph-bag and interface-bag decomposition of G_P

5.3. Tree-like Star and Costar Decompositions

In this section we repeat the work of Section 5.2 for "tree-like" cospan decompositions. We define *stars* and *costars* as generalizations of spans and cospans, respectively. A star $S = (\!|f_1, \ldots, f_n|\!)$ is a finite sequence[4] of morphisms with the same domain, while a costar $C = (\!|f_1, \ldots, f_n|\!)$ is a finite sequence of morphisms with the same codomain. We will consider a cospan $c\colon J \xrightarrow{c_L} G \xleftarrow{c_R} K$ as a special case of a costar, with $c = (\!|c_L, c_R|\!)$.



Similar to cospans, costars can be seen as graphs with interfaces, of which in the case of costars there can be arbitrarily many. Intuitively, two costars can be composed over specific tentacles i and j which have the same interface K, by gluing their center graphs together at K and removing the tentacles i and j. Formally:

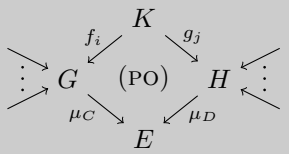
Definition 5.12 (Costar composition)**.** Let $C = (\!|f_1, \ldots, f_n|\!)$ be a costar with center graph $G = cod(f_1)$ and $D = (\!|g_1, \ldots, g_m|\!)$ a costar with center graph $H = cod(g_1)$. Furthermore, let $1 \leq i \leq n$ and $1 \leq j \leq m$ be given such that $dom(f_i) = dom(g_j)$. The composition of C and D over tentacles i and j, denoted by $C \,;_{i,j}\, D$, is defined as:

$$C \,;_{i,j}\, D = (\!|f_1 \,;\, \mu_C, \ldots, f_{i-1} \,;\, \mu_C, f_{i+1} \,;\, \mu_C, \ldots, f_n \,;\, \mu_C,\\ g_1 \,;\, \mu_D, \ldots, g_{j-1} \,;\, \mu_D, g_{j+1} \,;\, \mu_D, \ldots, g_m \,;\, \mu_D|\!),$$

where $\mu_C\colon G \to E$ and $\mu_D\colon H \to E$ are obtained by taking the pushout of f_i and

[4] We define stars and costars as *sequences* of morphisms so that a) one morphism can occur more than once in the same star; and b) we can uniquely identify the tentacles of the star or costar by specifying its index.

g_j, as shown in the following diagram:



where $K = dom(f_i) = dom(g_j)$.

Note that the tentacles over which two costars are composed are hidden by the composition operation. This has the effect that the indices of the other tentacles may change. For example, let $C = (\!|f_1, f_2, f_3|\!)$ and $D = (\!|g_1, g_2|\!)$. Then $C \;;_{2,1}\; D = (\!|f_1 \;;\; \mu_C, f_3 \;;\; \mu_C, g_2 \;;\; \mu_D|\!)$. Here, the *second* tentacle corresponds to the *third* tentacle of C, and the *third* tentacle corresponds to the *second* of D.

We define three types of tree-like decompositions: *costar decompositions*, *star decompositions* and *atomic star decompositions*. The names of the first two relate to the form of the stars (joins) in the tree; the third one is a special case of the second. A costar decomposition is a decomposition into costars, where costars are connected via the interfaces in such a way that they form a tree. Note that the edges of this tree are spans. On the other hand, a star decomposition is a decomposition into stars, where the edges of the corresponding tree-like structure correspond to cospans (see also Figure 5.4). As in the case of cospan decompositions, we restrict our attention to injective morphisms.

Definition 5.13 (Costar decomposition, star decomposition)**.**

1. A *costar decomposition* is a tuple $\mathcal{C} = \langle T, \tau \rangle$, where T is a tree and τ is function which maps each node t of T to a graph and each edge $b = \{t_1, t_2\}$ of T to a span of injective morphisms

   $$\tau(b) = \tau(t_1) \xleftarrow{\varphi_{b,t_1}} J_b \xrightarrow{\varphi_{b,t_2}} \tau(t_2).$$

   A costar decomposition $\mathcal{C}$ is a costar decomposition *of* G if $Colim(\mathcal{C}) = G$.

2. A *star decomposition* is a tuple $\mathcal{S} = \langle T, \tau \rangle$, where T is a tree and τ is function which maps each node t of T to a discrete graph J and each edge $b = \{t_1, t_2\}$ to a cospan

   $$\tau(b) = \tau(t_1) \to G_b \leftarrow \tau(t_2),$$

   which consists of a pair of jointly node-surjective, injective morphisms.

   A star decomposition $\mathcal{C}$ is a star decomposition *of* G if $Colim(\mathcal{C}) = G$ and additionally it holds for all edges e of G, with $att(e) = v_1 \dots v_m$, that $v_1, \dots, v_m$ occur together in $\tau(t)$ for some $t \in V_T$.

3. An *atomic star decomposition* is a star decomposition $\langle T, \tau \rangle$ such that $\tau(b)$ is an atomic cospan for all edges b of T.

In the case of cospan decompositions we had a clear hierarchy of the various decomposition types. In the case of tree-like decompositions, however, this is not the case:

the sets of star and costar decompositions are not related with respect to inclusion. However, by definition, each atomic star decomposition is also a star decomposition.

Definition 5.14 (Width of (co)star decomposition)**.** Let $\mathcal{S} = \langle T, \tau \rangle$ be a costar decomposition or a star decomposition. The width of $\mathcal{S}$ is defined as

$$wd_\star(\mathcal{S}) = \max_{v \in V_T} |\tau(v)| - 1.$$

Note that definition 5.14 bases the width of costar decompositions on the (non-interface) graphs they contain, while it bases the width of star decompositions on the interfaces. In both cases, however, the width of a decomposition depends on the size of the graphs that are in the image of the nodes of the tree T.

Definition 5.15 (Costar width, star width)**.** Let G be a graph. The *costar width* ($ctwd_{\mathrm{co}\star}(G)$), *star width* ($ctwd_\star(G)$) and *atomic star width* ($ctwd_{\mathrm{at}\star}(G)$) of G are defined as follows:

$$\begin{aligned} ctwd_{\mathrm{co}\star}(G) &= \min\{wd_\star(\mathcal{C}) \mid \mathcal{C} \text{ is a costar decomposition of } G\} \\ ctwd_\star(G) &= \min\{wd_\star(\mathcal{S}) \mid \mathcal{S} \text{ is a star decomposition of } G\} \\ ctwd_{\mathrm{at}\star}(G) &= \min\{wd_\star(\mathcal{S}) \mid \mathcal{S} \text{ is an atomic star decomposition of } G\} \end{aligned}$$

A fourth possibility would be to define a kind of star decomposition, which lacks the requirement that the cospans on the edges are jointly node-surjective, but whose width is measured by the sizes of the middle graphs of the cospans instead of the middle graphs of the stars. This would result in the same notion of width. However, since this would not result in a nice direct correspondence to tree decompositions, we have left it out.

As in the previous section, the various notions defined in this section are equivalent to the notion of treewidth.

Lemma 5.16.

1. Let $\mathcal{T}$ be a tree decomposition of G. There exists a star decomposition $\mathcal{S}$ of G such that $wd_\star(\mathcal{S}) = wd(\mathcal{T})$.
2. Let $\mathcal{S}$ be a star decomposition of G. There exists a costar decomposition $\mathcal{C}$ of G such that $wd_\star(\mathcal{C}) = wd_\star(\mathcal{S})$.
3. Let $\mathcal{C}$ be a costar decomposition of G. There exists a tree decomposition $\mathcal{T}$ of G such that $wd(\mathcal{T}) = wd_\star(\mathcal{C})$.
4. Let $\mathcal{S}$ be a star decomposition of G. There exists an atomic star decomposition $\mathcal{S}'$ of G such that $wd_\star(\mathcal{S}') = wd_\star(\mathcal{S})$.

Proof. See Appendix A.1. □

Theorem 5.17. For every graph G,

$$twd(G) = ctwd_{\mathrm{co}\star}(G) = ctwd_{\star}(G) = ctwd_{\mathrm{at}\star}(G).$$

Proof. It follows from Lemma 5.16 (4) and the fact that every atomic star decomposition is a star decomposition, that

$$ctwd_{\star}(G) = ctwd_{\mathrm{at}\star}(G). \tag{5.3}$$

Furthermore, from Lemma 5.16 (1)–(3), the following inequalities follow:

$$twd(G) \geq ctwd_{\star}(G) \geq ctwd_{\mathrm{co}\star}(G) \geq twd(G). \tag{5.4}$$

The derived result follows directly from (5.3) and (5.4). □

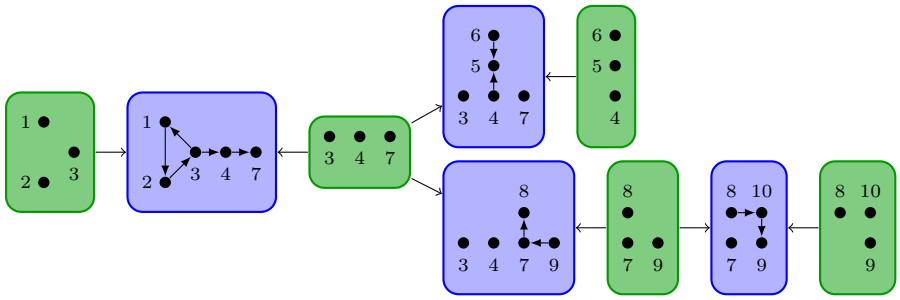
Example 5.18. *We consider the graph G_T and the tree decomposition $\mathcal{T}$ of Example 5.3. In order to construct a star decomposition of G_T, we take a cospan for each of the four edges (of the tree) of $\mathcal{T}$. The interfaces of these four cospans are the discrete graphs corresponding to the bags. The center graph of each cospan is the subgraph containing the nodes of both the inner and the outer interface of the cospan and (possibly) edges connecting these nodes. It has to be ensured that each edge occurs exactly once. This leads to the star decomposition shown in Figure 5.4a. Since the width of the given star decomposition has size 2 and the nodes of the 3-clique has to be contained together in at least one interface of any star decomposition, the star width of G_T is 2.*

The costar decomposition can be obtained from the star decomposition. Each of the four cospans of the star decomposition is converted into a span. The inner and the outer graph of each span contain the nodes of the corresponding cospan interfaces plus additional edges. (Note that due to the conditions on star decompositions, each edge can be "shifted" into at least one interface.) The center graph of the span is then the discrete graph obtained by the intersection of the inner and the outer graphs of the span. The resulting costar decomposition is shown in Figure 5.4b. The costar width of G_T is 2 due to the fact that the given costar decomposition has size 2 and that any costar decomposition must contain the 3-clique in some graph of at least one span.

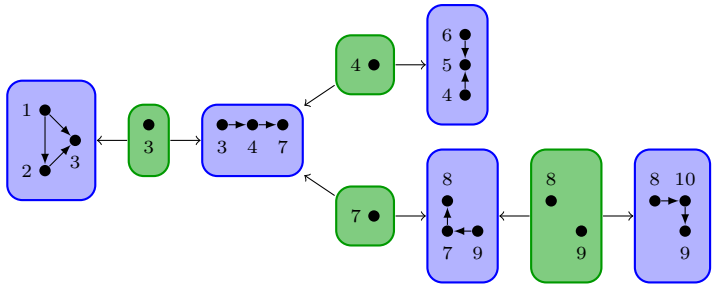
More details concerning the conversion of the various tree and star decompositions into each other can be found in the proof of Lemma 5.16.

5.4. Term Decompositions

Our aim in the next chapter will be to define a special kind of tree automata that operate on tree-like decompositions. On the one hand we have the tree-like decompositions of Section 5.3. On the other hand, however, we have tree automata which operate on *terms* rather than *trees*. Although there is a clear correspondence between trees and terms, some gaps have to be filled. That is what we do in this section, by defining *term decompositions* of graphs.



(a) Star decomposition of G_T



(b) Costar decomposition of G_T

Figure 5.4.: Star and costar decomposition of G_T

Definition 5.19 (Graph term)**.** The set of sorts we employ is the set of natural numbers including zero, that is $\mathbb{N} = \{0, 1, 2, \ldots\}$. A *graph term* is a many-sorted term over the signature $\mathcal{O}ps$, which contains (with some overloaded notation) the following function symbols:

- $vertex^n_k \colon n \to n+1$, for each $n \in \mathbb{N}$ and $1 \leq k \leq n+1$;
- $res^n_k \colon n \to n-1$, for each $n \geq 1 \in \mathbb{N}$ and $1 \leq k \leq n$;
- $connect^n_{A,\theta} \colon n \to n$, for each label $A \in \Sigma$ and each $n \geq ar(A)$ and function $\theta \colon \mathbb{N}_{ar(A)} \to \mathbb{N}_n$;
- $perm^n_\pi \colon n \to n$, for each $n \in \mathbb{N}$ and permutation $\pi \colon \mathbb{N}_n \to \mathbb{N}_n$;
- $join^n \colon \langle n, n \rangle \to n$, for each $n \in \mathbb{N}$.

The first four of the function symbols in Definition 5.19 correspond to the atomic cospans; in the following we will implicitly convert between the cospans and the functions symbols. The last one plays the role of stars in star decompositions: it allows branching. Note that holes ($\square_n$) can also occur in graph terms.

We will now define how to translate graph terms into costars. Note that for a term $t\colon \langle m_1, \ldots, m_n\rangle \to m_0$ the first morphism of the costar will correspond to the root of the term (and has domain D_{m_0}), whereas the remaining n morphisms will correspond to the holes (and have domains D_{m_i}).

Definition 5.20 (Term decomposition of a graph)**.**

(i) Let t be a graph term. The costar of t, denoted $\mathit{Costar}(t)$, is recursively constructed as follows:

- If $t = \square_n$, then $\mathit{Costar}(t) = (\!|\mathit{id}_{D_n}, \mathit{id}_{D_n}|\!)$.
- If $t = f(t')$, where $f\colon m \to n$ and $t'\colon \vec{q} \to m$, then

$$\mathit{Costar}(t) = \overline{f} \;;_{2,1}\; \mathit{Costar}(t')$$

  where $\overline{f}$ is the atomic cospan (see Section 3.4) corresponding to the function symbol f (viewed as a costar).
- If $t = \mathit{join}^n(t_1, t_2)$, then

$$\mathit{Costar}(t) = \big((\!|\mathit{id}_{D_n}, \mathit{id}_{D_n}, \mathit{id}_{D_n}|\!) \;;_{3,1}\; \mathit{Costar}(t_1)\big) \;;_{2,1}\; \mathit{Costar}(t_2).$$

(ii) A graph term t is a *term decomposition of a graph* G when G is the center graph of $\mathit{Costar}(t)$.

Similar to other types of decomposition we define the *width* of a term decomposition and the *term width* of a graph as follows:

Definition 5.21.

(i) The *width of a term decomposition* t, denoted by $\mathit{wd}(t)$, is the highest type which occurs in it minus 1; formally $\mathit{wd}(t) = \mathit{hi}(t) - 1$, where $\mathit{hi}(t)$ is inductively defined by:

$$\begin{aligned} \mathit{hi}(\square_n) &= n & \mathit{hi}(\mathit{res}^n_k(t)) &= \max\{n, \mathit{hi}(t)\} \\ \mathit{hi}(\mathit{join}^n(t_1, t_2)) &= \max\{n, \mathit{hi}(t_1), \mathit{hi}(t_2)\} & \mathit{hi}(\mathit{connect}^n_{A,\theta}(t)) &= \max\{n, \mathit{hi}(t)\} \\ \mathit{hi}(\mathit{vertex}^n_k(t)) &= \max\{n{+}1, \mathit{hi}(t)\} & \mathit{hi}(\mathit{perm}^n_\pi(t)) &= \max\{n, \mathit{hi}(t)\} \end{aligned}$$

(ii) The *term width* of a graph G is defined as:

$$\mathit{tmwd}(G) = \min\{\mathit{wd}(t) \mid t \text{ is a term decomposition of } G\}.$$

Example 5.22. *A term decomposition of the graph G_T of Figure 5.2 is the following:*

$$\begin{aligned} &\mathit{res}^1_1(\mathit{res}^2_2(\mathit{res}^3_3(\mathit{connect}^3_{12}(\mathit{connect}^3_{23}(\mathit{connect}^3_{31}(\mathit{vertex}^2_3(\mathit{vertex}^1_2(\\ &\quad \mathit{res}^2_1(\mathit{connect}^2_{12}(\mathit{vertex}^1_2(\\ &\qquad \mathit{join}^1(\\ &\qquad\quad \mathit{res}^2_2(\mathit{connect}^2_{12}(\mathit{vertex}^1_1(\mathit{res}^2_1(\mathit{connect}^2_{12}(\mathit{vertex}^1_1(\mathit{vertex}^0_1(\square_0)))))))) \end{aligned}$$

$$
\begin{array}{l}
\quad\quad ,\\
\quad\quad res^2_2(connect^2_{12}(vertex^1_1(res^2_1(\\
\quad\quad\quad join^2(\\
\quad\quad\quad\quad res^3_3(connect^3_{13}(connect^3_{32}(vertex^2_1(vertex^1_1(\\
\quad\quad\quad\quad\quad vertex^0_1(\square_0))))))\\
\quad\quad\quad\quad ,\\
\quad\quad\quad\quad res^3_3(connect^3_{23}(connect^3_{31}(vertex^2_1(vertex^1_1(\\
\quad\quad\quad\quad\quad vertex^0_1(\square_0))))))\\
\quad\quad\quad)\\
\quad\quad))))\\
\quad)\\
)))\\
)))))))
\end{array}
$$

The width of this term decomposition is 2*. As in earlier examples, we write* $connect^n_{ij}$ *for* $connect^n_{\diamond,\theta}$*, where* $\theta(1) = i$ *and* $\theta(2) = j$*.*

Because of the natural correspondence between trees and terms, it is no surprise that the notion of term width and the notion of atomic costar width introduced in Section 5.3 are equivalent.

Proposition 5.23. For every graph G, $tmwd(G) = ctwd_{\mathrm{at}\star}(G)$.

Sketch. We prove the proposition by translating every term decomposition of size k into an atomic star decomposition of size k and vice versa.

($\Rightarrow$): To translate a term decomposition into an atomic star decomposition we can proceed recursively and compose the results of the recursive calls similar to Definition 5.20.

($\Leftarrow$): Given an atomic star decomposition we can non-deterministically pick a root, then turn every cospan in the tree so that its outer interface is pointed towards the root (we can do this because, as observed on page 44, the inverse of each atomic cospan is also an atomic cospan) and then recursively transforming all the subtrees into a term. ☐

Furthermore, we have that every graph has a term decomposition. By the idea given in the proof sketch above we can transform a star decomposition, which we can obtain for every graph, into an equivalent term decomposition. However, a graph may have term decompositions of different types, for example one of type $\langle 0, 0\rangle \to 0$ and one of type $\langle 0, 0, 0\rangle \to 0$.

5.5. Conclusion

In this chapter we have shown how to convert the graph-theoretical notion of path decompositions into cospan decompositions, and tree decompositions into star or costar decompositions. As we have seen there are indeed several possible choices, mainly depending on whether we identify bags with interfaces or with the center graph in a cospan. Furthermore, there is the notion of decompositions into atomic cospans, which can be viewed as atomic building blocks.

In the next chapter, we will use decompositions for recognizable graph languages or – more specifically – for different automaton models acting as acceptors of graph languages. Especially the decompositions into atomic cospans will play an important role in this setting. But let us remark that we have not treated the question of *how* to obtain such decomposition given a single graph. This is an NP-hard problem that has been extensively studied by Bodlaender et al. [17–22]. Therefore, it is important to find good heuristics to obtain atomic cospan decomposition of a given graph in practice. We supervised a student to work on this in his master's thesis [79]. The objective of that thesis has been to develop and compare different algorithms used to compute atomic cospan decompositions. These algorithms are based on both newly developed heuristics and established heuristics for tree decompositions [20]. Furthermore, to perform the comparison of the different algorithms a prototype tool has been implemented.

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Mathison Turing (1912 – 1954)

6 Recognizable Graph Languages

Regular (word) languages have many applications in theory as well as in practice. The applications in which regular languages are used range from model-checking [26] and termination analysis [76] to parsers and text editors [86]. The notion of regularity can be generalized straightforwardly to languages of trees. Hence, it is natural to ask for the notion of regular graph languages, or in this case called recognizable graph languages.

There have been several attempts to give a notion of recognizable graph languages [44, 48, 110, 127, 130], of which the notion of Courcelle is widely accepted.

In this chapter we will introduce three different automaton models: automaton functors, consistent tree automata and graph automata. Although the concept behind these three automaton types is varying, we will show that they all accept the same class of recognizable graph languages in the sense of Courcelle.

In addition, we will take a look at two different graph logics in the second section of this chapter. The first logic is the well-known monadic second-order logic on graphs. In this context of particular importance to us is the well-known Courcelle's Theorem. The second logic we will investigate is a new logic, called linear cospan logic, which is introduced here for the first time. The scope of this logic is to enhance the construction as well as the computation of graph automata and to automatically compute invariants.

Parts of this chapter have been published in [15], [14] and [12].

6.1. Automaton Models for Recognizable Languages

In this section we first define recognizable graph languages by using automaton functors, as introduced by Bruggink and König [33], on the category of cospans of graphs. Secondly, we present consistent tree automata which are introduced by Blume, Bruggink, Friedrich and König in [14]. These automata can be seen as generalizations of tree automata, introduced in Chapter 2, which are used to process term decompositions of graphs introduced in Chapter 5. We show that the class of languages accepted by consistent

tree automata coincides with the class of languages acceptable by automaton functors. Furthermore, we give an overview over the properties of recognizable graph languages and subsequently present another view on recognizable graph languages in terms of so-called graph automata. This kind of automata can be seen as generalization of finite automata used to accept "path-like" cospan decompositions of graphs.

6.1.1. Automaton Functor – A Categorical Automaton Model

Bruggink and König have introduced automaton functors as a (categorical) automaton model for so-called recognizable arrow languages of an arbitrary (locally small[1]) category $\mathbf{C}$.

We give here the general definition for the sake of completeness, even if we are only interested in recognizable graph languages.

Definition 6.1 (Automaton Functor, [33])**.** Let J and K be two arbitrary $\mathbf{C}$-objects. A $\langle J, K\rangle$-*automaton functor* is a structure $\mathcal{A} = \langle \mathcal{A}_0, \mathrm{I}, \mathrm{F}\rangle$, where

- $\mathcal{A}_0 \colon \mathbf{C} \to \mathbf{Rel}$ is a functor, which maps every object X of $\mathbf{C}$ to a finite set $\mathcal{A}_0(X)$ (the set of *states* of X) and every arrow $X \xrightarrow{f} Y$ to a relation $\mathcal{A}_0(f) \subseteq \mathcal{A}_0(X) \times \mathcal{A}_0(Y)$ (the *transition relation* of f)
- $\mathrm{I} \subseteq \mathcal{A}_0(J)$ is the set of *initial states*
- $\mathrm{F} \subseteq \mathcal{A}_0(K)$ is the set of *final states*

An automaton functor is *deterministic* if every relation $\mathcal{A}_0(f)$ is a function and I contains exactly one element.

The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ (which contains arrows from J to K) is defined as:

> $J \xrightarrow{f} K$ is contained in $\mathcal{L}(\mathcal{A})$ if and only if states $q \in \mathrm{I}, q' \in \mathrm{F}$ exist, which are related by $\mathcal{A}_0(f)$.

A language $\mathcal{L}$ of arrows from J to K is a *recognizable language* if $\mathcal{L} = \mathcal{L}(\mathcal{A})$, for some automaton functor $\mathcal{A}$.

For an object X or a cospan c we will, in the following, usually write $\mathcal{A}(X)$ and $\mathcal{A}(c)$ instead of $\mathcal{A}_0(X)$ and $\mathcal{A}_0(c)$, respectively.

The intuition behind the definition is to have a morphism into a (locally) finite domain which guarantees that the acceptance is not affected by different decompositions of a morphism in any way. For instance, as seen in Chapter 5, the decomposition of a cospan of graphs is not unique, but the automaton functor has to ensure that if one decomposition of a cospan of graphs is accepted, all other decompositions of the same cospan must also be accepted. This is different from word languages, where there is essentially one way to decompose an object into smaller objects.

Note that the notion of automaton functors can be seen as a generalization of finite automata in the following way: Let Σ be the alphabet of an arbitrary non-deterministic finite automaton $\mathcal{M}$ and let Σ^* be the free monoid of Σ (see also Example 2.17). Then

[1] See definition on page 19

the automaton $\mathcal{M}$ is isomorphic to the automaton functor which maps the object X to the state set of $\mathcal{M}$ and every morphism, i. e. every word, to its respective transition function in $\mathcal{M}$.

Since this thesis is about recognizable graph languages we fix the category $\mathbf{C}$ for the rest of this thesis to be **OLCG** (or one of its subcategories), as defined in Chapter 3. Remember that **OLCG** is the category of (atomic) cospans which have a discrete inner interface and an injective discrete outer interface. Furthermore, we will usually write $\langle s, t\rangle$-automaton functor instead of $\langle D_s, D_t\rangle$-automaton functor or, if the (size of the) interfaces D_s and D_t are obvious from the context, we simply write automaton functor. Then we define the class of recognizable graph languages as follows:

Definition 6.2 (Recognizable Language)**.** A graph language $\mathcal{G}$ is *recognizable* if the language

$$\mathcal{L} = \{[G] : \emptyset \to G \leftarrow \emptyset \mid G \in \mathcal{G}\}$$

is the language of a $\langle 0, 0\rangle$-automaton functor $\mathcal{A}$.

This notion of recognizability coincides with the notion of Courcelle. For a detailed comparison between the algebraic view of Courcelle and the categorial view presented here (and for a proof of this statement), we refer to [33].

The restriction to output-linear cospans instead of arbitrary cospans of graphs does not affect the expressiveness of the formalism, which is due to the following robustness result inspired by results of Bruggink and König [33]:

Proposition 6.3 (Robustness)**.** Let a class $\mathcal{L}$ of output-linear cospans with discrete interfaces D_s and D_t be called *output-linear-recognizable* whenever $\mathcal{L}$ is recognizable in **OLCG**. Then $\mathcal{L}$ is recognizable in *Cospan*(**Graph**) if and only if it is output-linear-recognizable.

Proof. See Appendix A.2. □

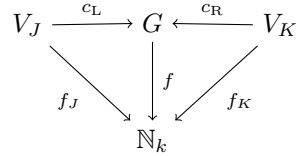
Before we go on to introduce more notions we take a look at some examples of recognizable graph languages:

Example 6.4 (k-Colorability, [33])**.** *Let G be a graph. A k-coloring of G is a function $f\colon V_G \to \mathbb{N}_k$ such that for all $e \in E_G$ and for all $v, w \in att_G(e)$ it holds that $f(v) \neq f(w)$ if $v \neq w$. We show that the language $C_{(k)}$ of all k-colorable graphs is recognizable, by considering the automaton functor $\mathcal{A}_{C_{(k)}}\colon$ **OLCG** $\to$ **Rel**:*

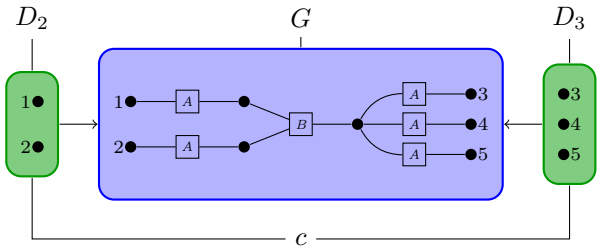
- *Every graph J is mapped to $\mathcal{A}_{C_{(k)}}(J)$, the set of all valid k-colorings of J, i. e.*

$$\mathcal{A}_{C_{(k)}}(J) = \{f\colon V_J \to \mathbb{N}_k \mid f \text{ is a valid } k\text{-coloring of } J\}.$$

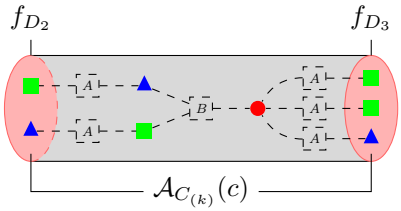
- *For a cospan $c\colon J \to{}_{c_\mathrm{L}} G \leftarrow{}_{c_\mathrm{R}} K$ the relation $\mathcal{A}_{C_{(k)}}(c)$ relates two colorings f_J and f_K if and only if there exists a coloring f for G such that the following diagram depicted on the right commutes, i. e. it holds that $f(c_\mathrm{L}(v)) = f_J(v)$ for every node $v \in V_J$ and $f(c_\mathrm{R}(v)) = f_K(v)$ for every node $v \in V_K$.*



As an example we take a look at the cospan[2] $c\colon D_2 \xrightarrow{c_L} G \xleftarrow{c_R} D_3$ *which is depicted in the figure below.*



The interfaces D_2 and D_3 could, for instance, be mapped to the colorings depicted by the following figure.



The coloring for the inner interface assigns to the nodes of the inner interface the colors ■ and ▲, respectively. The coloring for the outer interface assigns the colors ■, ■ and ▲, respectively, to the outer interface nodes. Since there exists a coloring for the middle graph G (also depicted in the figure above) which is compatible with the colorings f_{D_2} and f_{D_3}, the colorings f_{D_2} and f_{D_3} are related by $\mathcal{A}_{C_{(k)}}(c)$.

- *The set of initial and final states is the set $\mathrm{I}_\emptyset = \mathrm{F}_\emptyset = \{\emptyset\}$, where $\emptyset$ denotes the empty coloring.*

Note that a graph (seen as a cospan with empty interfaces) is accepted whenever the two empty colorings are related.

Example 6.5 (Vertex Cover)**.** *Let G be a graph. A vertex cover of G is a set $C \subseteq V_G$ such that for every edge $e \in E_G$ it holds that there exists an index $i_e \in \{1, \ldots, |att(e)|\}$ with $att(e)[i_e] \in C$. A vertex cover C has a size of at most k if $|C| \leq k$. We show that the language $V_{(k)}$ of all graphs with a vertex cover of a size of at most k is recognizable, by considering the automaton functor $\mathcal{A}_{V_{(k)}} : \mathbf{OLCG} \to \mathbf{Rel}$:*

- *Every graph J is mapped to $\mathcal{A}_{V_{(k)}}(J)$, the set of all pairs containing a subset of V_J (of arbitrary size) and an integer $i \leq k$, i. e.*

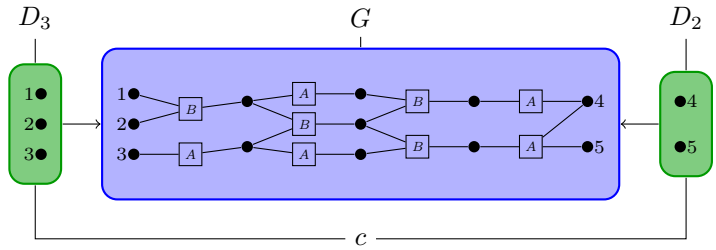
$$\mathcal{A}_{V_{(k)}}(J) = \{\langle C_J, i\rangle \mid C_J \subseteq V_J \text{ and } i \leq k\}.$$

[2] For convenience the integers indicating in which order the nodes are attached to the several edges are omitted.

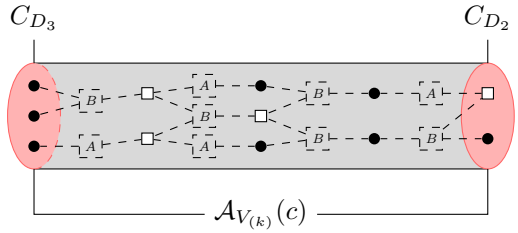
The set C_J indicates which nodes of the graph J are mapped to the vertex cover. The integer i indicates the number of nodes contained in the vertex cover which have already been seen.

- *For a cospan $c\colon J \xrightarrow{c_L} G \xleftarrow{c_R} K$ the relation $\mathcal{A}_{V_{(k)}}(c)$ relates two pairs $\langle C_J, i_J\rangle$ and $\langle C_K, i_K\rangle$ if and only if the following conditions hold:*
  - $c_L(C_J) = c_L(V_J) \cap C_G$,
  - $c_R(C_K) = c_R(V_K) \cap C_G$ *and*
  - $i_K = i_J + |C_G \setminus c_R(C_K)|$,

  where C_G is a vertex cover of G. In the figure below an example of a cospan[3] $c\colon D_3 \xrightarrow{c_L} G \xleftarrow{c_R} D_2$ is given.



A possible state to which the inner and the outer interface of the cospan c could be mapped by the automaton functor $\mathcal{A}_{V_{(k)}}$ is depicted by the figure below. The corresponding integers i_{D_3} and i_{D_2} are not shown.



The white-filled rectangular nodes indicate the nodes which belong to the vertex cover of G. Since all the vertex cover nodes belong to $V_G \setminus c_L(C_{D_3})$ we have that $c_L(C_{D_3}) = \emptyset$ must hold, which is obviously true. Furthermore, we have that the vertex cover C_{D_2} contains only the right-most white-filled rectangular node depicted in the figure above. Hence, we have $i_{D_2} = i_{D_3} + 3$.

- *The set of initial states is the set $\mathrm{I}_\emptyset = \{\langle\emptyset, 0\rangle\}$ and the set of final states is the set $\mathrm{F}_\emptyset = \{\langle\emptyset, i\rangle \mid i \leq k\}$.*

[3] For convenience the integers indicating in which order the nodes are attached to the several edges are omitted.

6.1.2. Consistent Tree Automata – An Automaton Model Processing Tree Decompositions

In this subsection we define a notion of automata which accept graph languages via their term decompositions. Hence, we adapt the notion of tree automata which has been introduced in Chapter 2 and fix the input alphabet to be $\mathcal{O}ps$, the set containing the function symbols $vertex_k^n$, res_k^n, $connect_{A,\theta}^n$, $perm_\pi^n$ and $join^n$ (see also Definition 5.19). The idea of using tree automata to accept graphs and graph languages has also been considered by Courcelle and Durand [45–47]. We also show that the class of languages accepted by the consistent tree automata presented below is exactly the class of recognizable graph languages.

Definition 6.6 (Consistent Tree Automaton)**.** A *consistent tree automaton* is a structure $\mathcal{M} = \langle Q, \Delta, \mathrm{I}, \mathrm{F}\rangle$, such that $\langle Q, \mathcal{O}ps, \Delta, \mathrm{I}, \mathrm{F}\rangle$ is an $\mathbb{N}$-sorted tree automaton and the following conditions apply:

- for all terms $t_1, t_2 \in \mathcal{T}(\mathcal{O}ps)$ of the same type it holds that $\widehat{\Delta}_{t_1} = \widehat{\Delta}_{t_2}$ if $\mathit{Costar}(t_1) = \mathit{Costar}(t_2)$ (*structural consistency*),
- for all term decompositions[4] $t_1 \colon \langle 0, \ldots, 0\rangle \to 0$ and $t_2 \colon \langle 0, \ldots, 0\rangle \to 0$ such that $\mathit{Colim}(\mathit{Costar}(t_1)) = \mathit{Colim}(\mathit{Costar}(t_2))$ it holds that $t_1 \in \mathcal{L}(\mathcal{M})$ if and only if $t_2 \in \mathcal{L}(\mathcal{M})$ (*semantic consistency*) and
- all initial and final states are in Q_0, i. e. $\mathrm{I}_k = \emptyset$ and $\mathrm{F}_k = \emptyset$ for all $k \geq 1$.

Analogous to automaton functors, the *graph language* of a consistent tree automaton $\mathcal{M}$ is defined as $\mathcal{G} = \{\mathit{Colim}(t) \mid t \in \mathcal{L}(\mathcal{M})\}$.

The structural consistency condition corresponds to the functor property of automaton functors in Definition 6.1: it says that the automaton behavior for a graph with interfaces does not depend on the way the graph is decomposed into a term decomposition. In the case of automaton functors the functor property is sufficient because all cospans have exactly two interfaces, therefore we only have to test whether $\emptyset \to G \leftarrow \emptyset$ is accepted. However, a graph may have term decompositions of different types, for example one of type $\langle 0, 0\rangle \to 0$ and one of type $\langle 0, 0, 0\rangle \to 0$. The semantic consistency condition is needed to make sure that in this case the acceptance of a graph does not depend on its specific term decomposition.

As with automaton functors, checking that a given structure satisfies the consistency conditions is non-trivial. Supplying building blocks that enable users to easily specify consistent tree automata is subject to ongoing research.

Example 6.7. *Let $k \in \mathbb{N}$ be given. Consider the automaton functor accepting k-colorable graphs of Example 6.4. We define the consistent tree automaton $\mathcal{M} = \langle Q, \mathcal{O}ps, \Delta, \mathrm{I}, \mathrm{F}\rangle$ as follows:*

- *$Q_n = \{f \colon \mathbb{N}_n \to \mathbb{N}_k \mid f$ is a valid k-coloring of $D_n\}$, i. e. Q_n is the set of all k-colorings of the discrete graph with n nodes.*

[4] Note that the types of t_1 and t_2 *are not* necessary equal, i. e. the lengths of the 0-sequences may be different.

- $\mathrm{I} = \mathrm{F} = Q_0$, *i. e.* all *states in* Q_0 *are both initial and final states.*
- *For* $f \in \mathcal{O}ps \setminus \{join^n \mid n \in \mathbb{N}\}$, $\Delta_f = \mathcal{A}_{C_{(k)}}(c_f)$, *where* $\mathcal{A}_{C_{(k)}}$ *is the automaton functor from Example 6.4 and* c_f *is the atomic cospan which corresponds to* f.
- $\Delta_{join^n}(q, q) = \{q\}$ *and* $\Delta_{join^n}(q, q') = \emptyset$ *for* $q \neq q'$.

Now, $\mathcal{M}$ *recognizes* k*-colorable graphs.*

In the remainder of this subsection we prove the following theorem, which relates the notion of consistent tree automata to the notion of automaton functors of Definition 6.1.

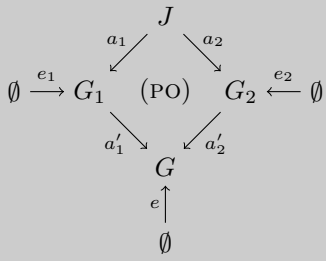
Theorem 6.8. Let $\mathcal{L}$ be a graph language. $\mathcal{L}$ is accepted by a consistent tree automaton if and only if $\mathcal{L}$ is accepted by an automaton functor.

The left-to-right case of the theorem is easily proved, because (modulo some technicalities) an automaton functor can be obtained from a consistent tree automaton by "forgetting" the transition functions for *join*. For the proof of the other direction we require some auxiliary machinery. First, we need an operation which composes two cospans in parallel by gluing over their common interface, that is, for cospans $c_1, c_2 \colon \emptyset \nrightarrow J$, we need to glue them together over J but keep J as the interface.

Definition 6.9. Let $c_1 \colon \emptyset - e_1 \to G_1 \leftarrow a_1 - J$ and $c_2 \colon \emptyset - e_2 \to G_2 \leftarrow a_2 - J$ be two cospans, where J is a discrete graph. We define the cospan

$$c_1 \mathbin{//} c_2 \colon \emptyset \to G \leftarrow J = \langle e, a_1 \mathbin{;} a_1' \rangle$$

by constructing a pushout as follows:



We define, for a given automaton functor $\mathcal{A}$, the equivalence relation $\equiv_{\mathcal{A}}$ as follows: $c_1 \equiv_{\mathcal{A}} c_2$ if $\mathcal{A}(c_1) = \mathcal{A}(c_2)$.

Lemma 6.10. The equivalence relation $\equiv_{\mathcal{A}}$ is a congruence, that is, if $c_1 \equiv_{\mathcal{A}} c_1'$ and $c_2 \equiv_{\mathcal{A}} c_2'$ then:

1. $c_1 \mathbin{;} c_2 \equiv_{\mathcal{A}} c_1' \mathbin{;} c_2'$ and
2. $c_1 \mathbin{//} c_2 \equiv_{\mathcal{A}} c_1' \mathbin{//} c_2'$.

Proof. Let $\mathcal{A} = \langle \mathcal{A}_0, I, F \rangle$.

1. Follows directly from the fact that $\mathcal{A}_0$ is a functor.

2. Define, for a cospan $d\colon \emptyset \nrightarrow J$ with $d = \langle d_{\mathrm{L}}, d_{\mathrm{R}} \rangle$, the cospan $\widehat{d}\colon J \nrightarrow J$ by $\widehat{d} = \langle d_{\mathrm{R}}, d_{\mathrm{R}} \rangle$.

   Consider the diagram of definition 6.9. It is the case that $c_1 \,;\, \widehat{c_2} = \langle e_1 \,;\, a_1', a_2 \,;\, a_2' \rangle$. Furthermore, $c_1 \mathbin{//} c_2 = \langle e, a_1 \,;\, a_1' \rangle = \langle e, a_2 \,;\, a_2' \rangle$. Since the graph morphism from $\emptyset$ to G is unique, it holds that $c_1 \mathbin{//} c_2 = c_1 \,;\, \widehat{c_2}$. Now assume $c_1 \equiv_{\mathcal{A}} c_1'$ and $c_2 \equiv_{\mathcal{A}} c_2'$. Note also that the definition of $c_1 \mathbin{//} c_2$ is completely symmetrical, and thus $c_1 \mathbin{//} c_2 = c_2 \mathbin{//} c_1$. Now we have:

$$\begin{aligned} c_1 \mathbin{//} c_2 = c_1 \,;\, \widehat{c_2} &\equiv_{\mathcal{A}} c_1' \,;\, \widehat{c_2} = c_1' \mathbin{//} c_2 = c_2 \mathbin{//} c_1' = c_2 \,;\, \widehat{c_1'} \\ &\equiv_{\mathcal{A}} c_2' \,;\, \widehat{c_1'} = c_2' \mathbin{//} c_1' = c_1' \mathbin{//} c_2', \end{aligned}$$

   as required.

□

The translation from automaton functors to consistent tree automata is carried out by the following construction. Note that it follows the same pattern as the construction of a finite automaton from the Myhill-Nerode equivalence classes of a language. In our case the equivalence is provided by the automaton functor $\mathcal{A}$ and is used to define equivalence classes of cospans of the form $\emptyset \nrightarrow D_i$ which then serve as states. Note that for a fixed i the number of these equivalence classes is finite since $\mathcal{A}$ maps (interface) graphs to finite (state) sets and there are only finitely many relations between two given finite sets.

Definition 6.11. Let an automaton functor $\mathcal{A} = \langle \mathcal{A}_0, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}} \rangle$ be given, and let $[\![c]\!]$ denote the $\equiv_{\mathcal{A}}$-equivalence class of c. We construct the consistent tree automaton $\mathcal{M}_{\mathcal{A}} = \langle Q, \Delta, \mathrm{I}_{\mathcal{M}}, \mathrm{F}_{\mathcal{M}} \rangle$ with:

- $Q_i = \{[\![c]\!] \mid c\colon \emptyset \nrightarrow D_i\}$.
- $\mathrm{I}_{\mathcal{M}} = \{[\![id_\emptyset\colon \emptyset \nrightarrow \emptyset]\!]\}$.
- $\mathrm{F}_{\mathcal{M}} = \{[\![c]\!] \mid c \in \mathcal{L}(\mathcal{A})\}$.
- $\Delta_{\square_k}([\![c]\!]) = \{[\![c]\!]\}$, for $c\colon \emptyset \nrightarrow D_k$.
- For all $f \in \mathcal{O}ps \setminus \{join^n \mid n \in \mathbb{N}\}$, $\Delta_f([\![c]\!]) = \{[\![c \,;\, c_f]\!]\}$, where c_f is the cospan corresponding to the function symbol f.
- $\Delta_{join^k}([\![c_1]\!], [\![c_2]\!]) = \{[\![c_1 \mathbin{//} c_2]\!]\}$, for $c_1, c_2\colon \emptyset \nrightarrow D_k$.

The construction of Definition 6.11 is well-defined because $\equiv_{\mathcal{A}}$ is a congruence (see Lemma 6.10). Observe that, by construction, Δ_t is deterministic for all terms, that is, it evaluates to a singleton.

Definition 6.12. Let $C = (\!|g, f_1, \ldots, f_n|\!)$ be a costar, where $g\colon J \to G$ and $f_i\colon K_i \to G$ (for $i \in \{1, \ldots, n\}$). Furthermore, let $c_1, \ldots, c_n$ be cospans with $c_i\colon \emptyset \nrightarrow K_i$ (for $i \in \{1, \ldots, n\}$). We define the cospan

$$C(c_1, \ldots, c_n) = \emptyset \xrightarrow{!_{G'}} G' \xleftarrow{g'} J,$$

where $g'\colon J \to G'$ is the first tentacle of the composed costar

$$(\!|g', !_{G'}, \ldots, !_{G'}|\!) = (\cdots (C \;;_{2,1} c_1) \cdots) \;;_{2,1} c_n.$$

Lemma 6.13. Let $\mathcal{A}$ be an automaton functor and $\mathcal{M}_\mathcal{A} = \langle Q, \Delta, \mathrm{I}_\mathcal{M}, \mathrm{F}_\mathcal{M}\rangle$ a consistent tree automaton as constructed by Definition 6.11. Furthermore, let $t\colon \langle k_1, \ldots, k_n\rangle \to k_0$ be a term decomposition and take $C_t = \mathit{Costar}(t)$. For all cospans $c_1, \ldots, c_n$ with $c_i\colon \emptyset \nrightarrow D_{k_i}$ $(i \in \{1, \ldots, n\})$ it holds that

$$\widehat{\Delta}_t([\![c_1]\!], \ldots, [\![c_n]\!]) = \Big\{ [\![C_t(c_1, \ldots, c_n)]\!] \Big\}.$$

Proof. By structural induction on t. □

Proposition 6.14. Let $\mathcal{A}$ be an automaton functor. $\mathcal{M}_\mathcal{A}$ as defined in Definition 6.11 is a consistent tree automaton.

Proof. Let $\mathcal{M}_\mathcal{A} = \langle Q, \Delta, \mathrm{I}_\mathcal{M}, \mathrm{F}_\mathcal{M}\rangle$. The structural consistency condition follows from the fact that, by Lemma 6.13, the transition function Δ_t depends only on $\mathit{Costar}(t)$.

To show the semantical consistency condition, observe that for costars

$$C_1 = (\!|f, \overbrace{!_G, \ldots, !_G}^{n \text{ times}}|\!) \text{ and } C_2 = (\!|f, \overbrace{!_G, \ldots, !_G}^{m \text{ times}}|\!)$$

where $f\colon J \to G$, it holds that $C_1(id_\emptyset, \ldots, id_\emptyset) = C_2(id_\emptyset, \ldots, id_\emptyset)$. Since, for term decompositions $t\colon \langle 0, \ldots, 0\rangle \to 0$, the costar $\mathit{Costar}(t)$ is of this form, the semantical consistency condition follows from Lemma 6.13. □

Now we have the necessary machinery to prove Theorem 6.8.

Proof of Theorem 6.8. ($\Rightarrow$): Let $\mathcal{M} = \langle Q, \Delta, I, F\rangle$ be a consistent tree automaton. We observe that all function symbols of the signature except join^n are unary, and therefore terms that do not contain join^n are isomorphic to cospan decompositions. Thus, by Lemma 5.5, every graph has a term decomposition which does not contain any occurrence of join^n.

Therefore, we can build an automaton functor from $\mathcal{M}$ by mapping each cospan c to the relation $\widehat{\Delta}(t)$, where t is a term (without join^n) isomorphic to c. Well-definedness of this definition and functoriality of the resulting automaton functor follow from the consistency condition on $\mathcal{M}$.

($\Leftarrow$): Let an automaton functor $\mathcal{A} = \langle \mathcal{A}_0, I, F \rangle$ be given and let $\mathcal{M}_\mathcal{A} = \langle Q, \Delta, I_\mathcal{M}, F_\mathcal{M} \rangle$ be the consistent tree automaton constructed as in Definition 6.11. By Proposition 6.14 $\mathcal{M}_\mathcal{A}$ is a consistent tree automaton which, in particular, satisfies the structural and semantic consistency conditions. It remains to show that $\mathcal{M}_\mathcal{A}$ and $\mathcal{A}$ accept the same language.

Let G be a graph, $c\colon \emptyset \nrightarrow \emptyset$ a cospan decomposition of G and $t\colon \langle 0, \dots, 0 \rangle \to 0$ a term decomposition of G. Suppose $c = c_1 \,;\, \cdots \,;\, c_n$. We define $t' = c_n(\cdots(c_1(\square_0))\cdots)$. By construction it holds that $\mathit{Costar}(t') = c$ (where the cospan c is interpreted as a costar consisting of two tentacles). Because all the interfaces of $\mathit{Costar}(t)$ and of c are empty, and the center graph of both is G, it must hold by the semantic consistency condition that $t \in \mathcal{L}(\mathcal{M})$ if and only if $t' \in \mathcal{L}(\mathcal{M})$. By construction it holds that $\Delta_{t'}([\![id_\emptyset]\!]_\equiv) = \{[\![c]\!]_\equiv\}$. Thus:

$$t \in \mathcal{L}(\mathcal{M}_\mathcal{A}) \iff t' \in \mathcal{L}(\mathcal{M}_\mathcal{A}) \iff ([\![id_\emptyset]\!]_\equiv \in I \text{ and } [\![c]\!]_\equiv \in F) \iff c \in \mathcal{L}(\mathcal{A}),$$

as required. □

6.1.3. Properties of Recognizable Graph Languages

Now we want to restate some properties of recognizable graph languages which are not very surprising. The proofs of most of the results are straightforward and can be found in [33, 34]. Later on we will use these results for the logic presented in Section 6.2.2 and for the algorithms shown in Section 7.2.

Proposition 6.15 ([33])**.** Let $s, t \in \mathbb{N}$ and $\mathcal{L}$ be a graph language over **OLCG**, containing cospans from D_s to D_t. Then $\mathcal{L}$ is recognizable if and only if there exists a locally finite congruence $\equiv_R$ such that $\mathcal{L}$ is the union of (finitely many) equivalence classes of $\equiv_R$.

Proposition 6.16 (Determinisation [33])**.** For every automaton functor there exists an equivalent deterministic automaton functor.

Proof Sketch. The construction is similar to the case of finite automata. Every state set will be replaced by a powerset of states. □

Proposition 6.17 (Closure under Boolean Operators [33])**.** Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two recognizable graph languages. Then also

- $\overline{\mathcal{G}_1}$ (the complement of $\mathcal{G}_1$),
- $\mathcal{G}_1 \cap \mathcal{G}_2$ (the intersection of $\mathcal{G}_1$ and $\mathcal{G}_2$) and
- $\mathcal{G}_1 \cup \mathcal{G}_2$ (the union of $\mathcal{G}_1$ and $\mathcal{G}_2$)

are recognizable.

Proof Sketch. The constructions are similar to the case of finite automata. □

As stated above, the constructions for the boolean operations are similar to the case of finite automata. Therefore, it is possible to compute the intersection and the union very efficiently, if the corresponding recognizable graph languages are given in terms of automaton functors. In the case of the complement construction, this is only true if the corresponding automaton functor is deterministic. Otherwise one needs to determinize the given automaton functor first, which depends on a construction with exponential space consumption (in the number of states of the automaton functor).

Note that the functoriality property of automaton functors still holds, i. e. the acceptance behavior of the constructed automaton functors is still independent of the concrete decomposition of a graph. Also note that the constructions for intersection and union are implemented in Raven (see Chapters 7 and 8 for more details).

Now, we want to focus on the concatenation of two recognizable (graph) languages over some common interface [34, 95]. Therefore, we do not only consider recognizable graph languages, but languages of cospans (such that the outer interface of the first language is equal to the inner interface of the second language).

Proposition 6.18 (Closure under Concatenation, [34])**.** Let $\mathcal{L}_{X,Y}$ and $\mathcal{L}_{Y,Z}$ be two recognizable languages (of linear[5] cospans of the form $c\colon X \nrightarrow Y$ and $d\colon Y \nrightarrow Z$ respectively). Then the language

$$\mathcal{L}_{X,Y} \,;\, \mathcal{L}_{Y,Z} = \{c \,;\, d \mid c \in \mathcal{L}_{X,Y} \text{ and } d \in \mathcal{L}_{Y,Z}\}$$

(the concatenation of $\mathcal{L}_{X,Y}$ and $\mathcal{L}_{Y,Z}$) is recognizable, too.

Proof. The proof can be found in [34] for the general case of recognizable languages in adhesive categories. □

Note that functors play the role of monoid morphisms in classical monoid theory. The following result is analogous to the well-known result that regular word languages are closed under inverse morphism application.

Proposition 6.19 (Closure under Inverse Morphisms)**.** Let $\mathcal{F}\colon \mathbf{OLCG} \to \mathbf{OLCG}$ be a functor. If $\mathcal{L}$ is a recognizable graph language, then $\mathcal{F}^{-1}(\mathcal{L})$ is a recognizable graph language.

Proof. By composing $\mathcal{F}$ with the automaton functor for $\mathcal{L}$, we obtain an automaton functor for $\mathcal{F}^{-1}(\mathcal{L})$. □

Next, we will lift the generalized theorem of Myhill-Nerode (Proposition 2.11) from the framework of word languages to our framework of graph languages [15]. But first we need the notion of Myhill-Nerode quasi-order. Note that while the variant of this theorem for word languages uses orders that are both left-monotone and right-monotone, here we work only with right-monotone orders. Intuitively this is sufficient

[5] Remember that a cospan $c\colon I \xrightarrow{c_\mathrm{L}} G \xleftarrow{c_\mathrm{R}} J$ is linear if both the left leg c_L and the right leg c_R are injective.

since we can assume that we start with the empty interface. This is possible because we can "simulate" the left-composition of any cospan by a right-composition of another equivalent cospan.

Definition 6.20 (Myhill-Nerode quasi-order)**.** Let $\mathcal{L}$ be a graph language over **OLCG**. A quasi-order $\leq_{\mathcal{L}}$ on **OLCG** is called *Myhill-Nerode quasi order (relative to $\mathcal{L}$)*, if for arbitrary cospans $c, c' \colon \emptyset \nrightarrow D_n$ the following condition is satisfied:

$$c \leq_{\mathcal{L}} c' \quad \text{iff} \quad \forall (d \colon D_n \nrightarrow \emptyset) \colon ((c \,;\, d) \in \mathcal{L} \implies (c' \,;\, d) \in \mathcal{L}) \,.$$

Based on $\leq_{\mathcal{L}}$ we can define the *Myhill-Nerode equivalence* $\equiv_{\mathcal{L}}$ on cospans $c, c' \colon \emptyset \nrightarrow D_n$ as follows:

$$c \equiv_{\mathcal{L}} c' \quad \text{iff} \quad c \leq_{\mathcal{L}} c' \text{ and } c' \leq_{\mathcal{L}} c.$$

One can prove that the Myhill-Nerode quasi-order is in fact a quasi-order on **OLCG**. It also possesses two other properties which will be important in the following.

Proposition 6.21. Let $\mathcal{L}$ be a graph language over **OLCG**. The Myhill-Nerode quasi-order (relative to $\mathcal{L}$) is right-monotone and the language $\mathcal{L}$ is upward-closed with respect to $\leq_{\mathcal{L}}$.

Proof.

- Right-monotone: Let arbitrary cospans $a, b \colon \emptyset \nrightarrow M$ with $a \leq_{\mathcal{L}} b$ be given. By definition, it holds that

  $$\forall (c \colon M \nrightarrow \emptyset) \colon (a \,;\, c) \in \mathcal{L} \implies (b \,;\, c) \in \mathcal{L}.$$

  Now take cospans $c' \colon M \nrightarrow N$ and $c'' \colon N \nrightarrow \emptyset$ such that $c = c' \,;\, c''$. Now we have $((a \,;\, c') \,;\, c'') \in \mathcal{L} \implies ((b \,;\, c') \,;\, c'') \in \mathcal{L}$. By definition, we have $(a \,;\, c') \leq_{\mathcal{L}} (b \,;\, c')$.

- Upward-closure: Let $a, b \colon \emptyset \nrightarrow \emptyset$ be arbitrary cospans such that $a \in \mathcal{L}$ and $a \leq_{\mathcal{L}} b$. Then $a = a \,;\, e \in \mathcal{L}$ implies $b = b \,;\, e \in \mathcal{L}$, where e is the empty cospan.

□

As an important result we now can re-state the generalized theorem of Myhill-Nerode for our setting:

Theorem 6.22 (Generalized Myhill-Nerode Theorem, [15])**.** Let a graph language $\mathcal{L}$ over **OLCG** be given. The following statements are equivalent:

(i) $\mathcal{L}$ is a recognizable graph language,

(ii) $\equiv_{\mathcal{L}}$ is locally finite and $\mathcal{L}$ is the union of (finitely many) equivalence classes of $\equiv_{\mathcal{L}}$,

(iii) $\mathcal{L}$ is upward closed with respect to some right-monotone well-quasi-order $\sqsubseteq_{\mathcal{L}}$,

(iv) The Myhill-Nerode quasi-order $\leq_{\mathcal{L}}$ is a well quasi-order.

6.1.4. An axiomatization of Output-linear Cospans

In the following we give an axiomatization of the output-linear cospans in terms of atomic cospans. The main advantage of the axiomization is to make it easier to verify that the functoriality property of automaton functors holds for concrete examples. In practice it is often very hard to check whether the functoriality property is satisfied or not. By means of the axiomatization we can split up this problem such that one needs to check only a small number of equations. We introduce a new normal form for atomic cospans and prove the soundness and completeness of the axiomatization.

Definition 6.23. Let two sequences $\vec{c}, \vec{d}$ of composable cospans in the category **OLCG** be given. The sequences $\vec{c}$ and $\vec{d}$ are *equivalent*, $\vec{c} \equiv \vec{d}$, if and only if $\mathit{Colim}(\vec{c}) \simeq \mathit{Colim}(\vec{d})$.

As an abbreviation, we extend the *fuse*-cospan to arbitrary equivalences in the following way. Let δ be an arbitrary equivalence on $\mathbb{N}_n$, then fuse^n_δ denotes the sequence

$$\mathit{fuse}^n_\delta = \begin{cases} \mathit{id}_{n,n}, & \text{if } n \leq 1 \\ \mathit{fuse}^{n-1}_{\delta|_{\mathbb{N}_{n-1}}}, & \text{if } n > 1 \wedge |[\![n]\!]_\delta| = 1 \\ \mathit{fuse}^n_{\min [\![n]\!]_\delta, n} \,;\, \mathit{fuse}^{n-1}_{\delta|_{\mathbb{N}_{n-1}}}, & \text{if } n > 1 \wedge |[\![n]\!]_\delta| \neq 1 \end{cases},$$

where $\mathit{id}_{n,n}$ denotes the identity cospan on D_n, i. e. $\mathit{id}_{n,n} \colon D_n - {\mathit{id}_n} \rightarrow D_n \leftarrow {\mathit{id}_n} - D_n$, and $\delta|_{\mathbb{N}_{n-1}}$ is the restriction of δ to $\mathbb{N}_{n-1}$, i. e. $\delta|_{\mathbb{N}_{n-1}} = \{\langle m, n\rangle \in \delta \mid m, n \in \mathbb{N}_{n-1}\}$.

The atomic cospan normal form can then be defined as follows:

Definition 6.24 (Atomic Cospan Normal Form)**.** A sequence of composable cospans $\vec{c}$, with $\mathit{Colim}(\vec{c}) = \langle V, E, \mathit{att}, \mathit{lab}\rangle$, is in *atomic cospan normal form* if it is of the following form:

$$\begin{aligned} &\mathit{vertex}^k_{k+1} \,;\, \mathit{vertex}^{k+1}_{k+2} \,;\, \cdots \,;\, \mathit{vertex}^{k+n-1}_{k+n} \,; \\ &\mathit{connect}^{k+n}_{A_1,\theta_1} \,;\, \cdots \,;\, \mathit{connect}^{k+n}_{A_{|E|},\theta_{|E|}} \,; \\ &\mathit{fuse}^{k+n}_\delta \,; \\ &\mathit{perm}^{k+n-|V/\delta|}_\pi \,; \\ &\mathit{res}^{k+n-|V/\delta|}_{k+n-|V/\delta|} \,;\, \cdots \,;\, \mathit{res}^{\ell+1}_{\ell+1} \end{aligned}$$

where

- $n \in \mathbb{N}$ and k and ℓ are the sizes of the left-most and right-most interface of $\vec{c}$, respectively,

- $A_i = lab(e_i)$ for every edge $e_i \in E$, θ_i is a function

$$\theta_i \colon \{0, \dots, |att(e_i)| - 1\} \to \{n, \dots |V| + n - 1\},$$

- δ is an equivalence on $\{1, \dots, |V| + n - 1\}$,
- π is a bijection on $\{0, \dots, k + n - |V/\delta| - 1\}$.

Proposition 6.25. For every sequence of composable cospans in the category **OLCG** exists an equivalent sequence of composable cospans which is in atomic cospan normal form.

Proof. See Appendix A.2 □

In order to prove the completeness of the axiomatization we introduce a number of algebraic properties which are satisfied by the atomic cospans defined in Chapter 3. The list consists of the following 17 equation schemes:

$$vertex_k^n \equiv vertex_n^n \,;\, perm_\pi^{n+1}, \tag{6.1}$$

where π is defined as

$$\pi \colon \mathbb{N}_{n+1} \to \mathbb{N}_{n+1}, \quad \pi(x) = \begin{cases} x, & \text{if } x < k \\ n+1, & \text{if } x = k \\ x-1, & \text{if } x > k \end{cases}.$$

$$vertex_k^n \,;\, vertex_\ell^{n+1} \equiv vertex_{f(\ell)}^n \,;\, vertex_{g(k)}^{n+1}, \tag{6.2}$$

where f is defined as

$$f \colon \mathbb{N}_{n+1} \to \mathbb{N}_n, \quad f(x) = \begin{cases} x, & \text{if } x \leq k \\ x-1, & \text{if } x > k \end{cases}$$

and g is defined as

$$g \colon \mathbb{N}_n \to \mathbb{N}_{n+1}, \quad g(x) = \begin{cases} x, & \text{if } x < \ell \\ x+1, & \text{if } x \geq \ell \end{cases}.$$

$$connect_{A,\theta}^n \,;\, vertex_k^n \equiv vertex_k^n \,;\, connect_{A,\theta'}^{n+1}, \tag{6.3}$$

where θ' is defined as

$$\theta' \colon \mathbb{N}_{ar(A)} \to \mathbb{N}_{n+1}, \quad \theta'(x) = \begin{cases} \theta(x), & \text{if } \theta(x) < k \\ \theta(x)+1, & \text{if } \theta(x) \geq k \end{cases}.$$

$$vertex_k^n \,;\, fuse_{i,k}^{n+1} \equiv id_{D_n}, \tag{6.4}$$

$$fuse_{i,j}^n \,;\, vertex_k^{n-1} \equiv vertex_{f(k)}^n \,;\, fuse_{g(i),g(j)}^{n+1}, \tag{6.5}$$

where f and g are defined as

$$f\colon \mathbb{N}_{n-1} \to \mathbb{N}_n, \quad f(x) = \begin{cases} x, & \text{if } x < j \\ x+1, & \text{if } x \geq j \end{cases}$$

and

$$g\colon \mathbb{N}_n \to \mathbb{N}_{n+1}, \quad g(x) = \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } x \geq k \end{cases}.$$

$$perm_\pi^n \,;\, vertex_k^n \equiv vertex_k^n \,;\, perm_{\pi'}^{n+1}, \tag{6.6}$$

where π' is defined as

$$\pi'\colon \mathbb{N}_{n+1} \to \mathbb{N}_{n+1}, \quad \pi'(x) = \begin{cases} \pi(x), & \text{if } x < k, \pi(x) < k \\ \pi(x)+1, & \text{if } x < k, \pi(x) \geq k \\ k, & \text{if } x = k \\ \pi(x-1), & \text{if } x > k, \pi(x-1) < k \\ \pi(x-1)+1, & \text{if } x > k, \pi(x-1) \geq k \end{cases}.$$

$$res_k^n \,;\, vertex_\ell^{n-1} \equiv vertex_{f(\ell)}^n \,;\, res_{g(k)}^{n+1}, \tag{6.7}$$

where f and g are defined as

$$f\colon \mathbb{N}_{n-1} \to \mathbb{N}_n, \quad f(x) = \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } x \geq k \end{cases}$$

and

$$g\colon \mathbb{N}_n \to \mathbb{N}_{n+1}, \quad g(x) = \begin{cases} x, & \text{if } x < \ell \\ x+1, & \text{if } x \geq \ell \end{cases}.$$

$$connect_{A,\theta}^n \,;\, connect_{A',\theta'}^n \equiv connect_{A',\theta'}^n \,;\, connect_{A,\theta}^n, \tag{6.8}$$

$$fuse_{i,j}^n \,;\, connect_{A,\theta}^{n-1} \equiv connect_{A,\theta'}^n \,;\, fuse_{i,j}^n, \tag{6.9}$$

where θ' is defined as

$$\theta'\colon \mathbb{N}_{ar(A)} \to \mathbb{N}_n, \quad \theta'(x) = \begin{cases} \theta(x), & \text{if } \theta(x) < j \\ \theta(x)+1, & \text{if } \theta(x) \geq j \end{cases}.$$

$$perm^n_\pi \,;\, connect^n_{A,\theta} \equiv connect^n_{A,\theta'} \,;\, perm^n_\pi, \tag{6.10}$$

where θ' is defined as

$$\theta' \colon \mathbb{N}_{ar(A)} \to \mathbb{N}_n, \quad \theta'(x) = \pi(\theta(x)).$$

$$res^n_k \,;\, connect^{n-1}_{A,\theta} \equiv connect^n_{A,\theta'} \,;\, res^n_k, \tag{6.11}$$

where θ' is defined as

$$\theta' \colon \mathbb{N}_{ar(A)} \to \mathbb{N}_n, \quad \theta'(x) = \begin{cases} \theta(x), & \text{if } \theta(x) < k \\ \theta(x)+1, & \text{if } \theta(x) \geq k \end{cases}.$$

$$fuse^n_{i,j} \,;\, fuse^{n-1}_{k,\ell} \equiv fuse^n_{f(k),f(\ell)} \,;\, fuse^{n-1}_{g(i),g(j)} \tag{6.12}$$

where f and g are defined as

$$f \colon \mathbb{N}_{n-1} \to \mathbb{N}_n, \quad f(x) = \begin{cases} x, & \text{if } x < j \\ x+1, & \text{if } x \geq j \end{cases}$$

and

$$g \colon \mathbb{N}_n \to \mathbb{N}_{n-1}, \quad g(x) = \begin{cases} x, & \text{if } x < \ell \\ x-1, & \text{if } x \geq \ell \end{cases}.$$

$$perm^n_\pi \,;\, fuse^n_{i,j} \equiv fuse^n_{\pi^{-1}(i),\pi^{-1}(j)} \,;\, perm^{n-1}_{\pi'}, \tag{6.13}$$

where π' is defined as

$$\pi' \colon \mathbb{N}_{n-1} \to \mathbb{N}_{n-1}, \quad \pi'(x) = \begin{cases} \pi(x), & \text{if } x < j, \pi(x) < j \\ \pi(x)-1, & \text{if } x < j, \pi(x) \geq j \\ \pi(x+1), & \text{if } x \geq j, \pi(x+1) < j \\ \pi(x+1)-1, & \text{if } x \geq j, \pi(x+1) \geq j \end{cases}.$$

$$perm^n_\pi \,;\, perm^n_{\pi'} \equiv perm^n_{\pi';\pi} \tag{6.14}$$

$$res^n_k \,;\, perm^{n-1}_\pi \equiv perm^n_{\pi'} \,;\, res^n_k, \tag{6.15}$$

where π' is defined as

$$\pi' \colon \mathbb{N}_{n+1} \to \mathbb{N}_{n+1}, \quad \pi'(x) = \begin{cases} \pi(x), & \text{if } x < k, \pi(x) < k \\ \pi(x)+1, & \text{if } x < k, \pi(x) \geq k \\ k, & \text{if } x = k \\ \pi(x-1), & \text{if } x > k, \pi(x-1) < k \\ \pi(x-1)+1, & \text{if } x > k, \pi(x-1) \geq k \end{cases}.$$

$$res^n_k \equiv perm^n_\pi \,;\, res^n_n, \tag{6.16}$$

where π is defined as

$$\pi\colon \mathbb{N}_n \to \mathbb{N}_n, \quad \pi(x) = \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } k \leq x < n \\ k, & \text{if } x = n \end{cases}.$$

$$res^n_k \,;\, res^{n-1}_\ell \equiv res^n_{f(\ell)} \,;\, res^{n-1}_{g(k)}, \tag{6.17}$$

where f is defined as

$$f\colon \mathbb{N}_{n-1} \to \mathbb{N}_n, \quad f(x) = \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } x \geq k \end{cases}$$

and g is defined as

$$g\colon \mathbb{N}_n \to \mathbb{N}_{n-1}, \quad g(x) = \begin{cases} x, & \text{if } x \leq \ell \\ x-1, & \text{if } x > \ell \end{cases}.$$

Proposition 6.26 (Algebraic Properties of Atomic Cospans)**.** The atomic cospans defined in Definition 3.10 satisfy the equation schemes 6.1–6.17.

Proof. See Appendix A.2 □

Now we are able to establish the two main theorems of this subsection.

Theorem 6.27 (Soundness)**.** If two sequences of composable cospans in the category **OLCG** (with the same inner and outer interface) can be transformed into each other via the equations schemes 6.1–6.17, they are equivalent.

Proof. See Appendix A.2 □

Theorem 6.28 (Completeness)**.** If two sequences of composable cospans in the category **OLCG** (with the same inner and outer interface) are equivalent, they can be transformed into each other via the equation schemes 6.1–6.17.

Proof. See Appendix A.2 □

We can immediately derive the following corollary.

Corollary 6.29. Two sequences of composable cospans in the category **Graph** (with the same inner and outer interface) are equivalent if and only if they have the same atomic cospan normal form.

Proof. The statement follows immediately from Proposition 6.25 and the Theorems 6.28 and 6.27. □

6.1.5. Graph Automata – A More Automaton-theoretic Model

In this subsection we want to introduce another automaton model which has a much more automaton-theoretic view on recognizable languages [12]. Furthermore, we will restrict ourselves to finite structures by limiting the maximum width of the considered cospans for the rest of this thesis, i. e. we only consider the category $\mathbf{OLCG}_n$. The reason behind this decision is that we want to establish a theory of recognizable graph languages which can be directly used as a basis for the tool implementation presented in Chapter 8.

But before we give the definition of the new automaton model, we introduce a special alphabet here. This alphabet is used as the input alphabet for the automaton model. Each letter of the alphabet represents an atomic cospan such that the concatenation of these letters (or cospans respectively) yields a graph (seen as a cospan with empty interfaces).

Let $n \in \mathbb{N}$ and a *doubly-ranked alphabet* $\Sigma = (\Sigma_{i,j})_{i,j \leq n}$ be given. The set of *(doubly-ranked) sequences* $S_\Sigma = (S_{i,j})_{i,j \leq n}$ over a doubly-ranked alphabet Σ is defined inductively:

- for every $i \leq n$ the *empty sequence* ε_i is in $S_{i,i}$
- for every $i, j \leq n$ every letter $\sigma \in \Sigma_{i,j}$ is in $S_{i,j}$
- for every $i, j, k \leq n$ and for every $\vec{\sigma} \in S_{i,j}$, $\vec{\sigma}' \in S_{j,k}$ the *concatenation* $\vec{\sigma}\,;\vec{\sigma}'$ of $\vec{\sigma}$ and $\vec{\sigma}'$ is in $S_{i,k}$

The *width* of a sequence is the maximum rank of its letters. We will also write S instead of S_Σ if the underlying alphabet is clear from the context.

Let Λ be a set of labels. By $\mathcal{S}ig = (\mathcal{S}ig_i)_{i \leq n}$ we denote the doubly-ranked alphabet containing the letters[6] as shown in Table 6.1:

Letter:	$connect^i_A$	$fuse^i$	$shift^i$	res^i	$trans^i$	$vertex^i$
Type:	(i, i)	$(i, i-1)$	(i, i)	$(i, i-1)$	(i, i)	$(i, i+1)$
Constraint:	$A \in \Lambda$, $ar(A) \leq i$	$i \geq 2$	$i \geq 3$	$i \geq 1$	$i \geq 2$	$i < n$

Table 6.1.: Constraints for the letters of the alphabet $\mathcal{S}ig$

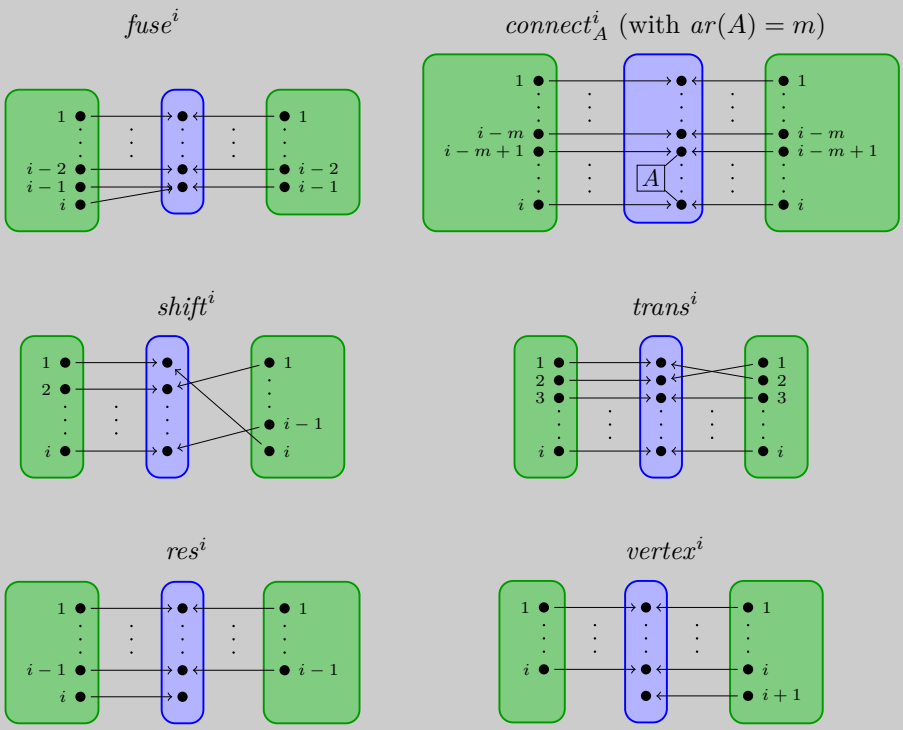
The meaning of these letters is given by the evaluation function defined below which maps each letter of the alphabet $\mathcal{S}ig$ to one of the atomic cospans[7] defined in Chapter 3.

[6] In this thesis we have renamed the *perm*-operation introduced in [12] to *shift* to avoid confusion with the *perm*-operation introduced in Chapter 3 and in [13, 14]

[7] Note that we have slightly changed the set of atomic cospans. Instead of the *perm*-cospan we have

Definition 6.30 (Evaluation function)**.** Let Λ be a set of labels.

1. The *evaluation function* $\eta\colon \mathcal{S}ig \to \mathbf{OLCG}_n$ maps each letter to an output linear cospan as shown below:



2. The *extended evaluation function* $\hat{\eta}\colon S_{\mathcal{S}ig} \to \mathbf{OLCG}_n$ is defined as

$$\hat{\eta}(\vec{\sigma}) = \begin{cases} D_j \xrightarrow{id} D_j \xleftarrow{id} D_j, & \text{if } \vec{\sigma} = \varepsilon_j \in S_{j,j} \\ \eta(\sigma), & \text{if } \vec{\sigma} = \sigma \in \mathcal{S}ig \\ \hat{\eta}(\vec{\sigma}_1)\ ;\ \hat{\eta}(\vec{\sigma}_2), & \text{if } \vec{\sigma} = \vec{\sigma}_1\ ;\ \vec{\sigma}_2 \end{cases}$$

Note that the definition of the extended evaluation function is well-defined due to the associativity of the cospan composition. Due to the fact that for two elements shifting and transposition are identical operations the constraint of the letter $shift^i$ is $i \geq 3$ as seen in Table 6.1. Since every letter can be identified with an atomic cospan and vice versa, we will not distinguish between the atomic cospan and the letter mapped to it. Let c be an output-linear cospan. The *width* of c is the minimal width of all $\vec{\sigma}$ such that $\hat{\eta}(\vec{\sigma}) = c$.

Now we are able to introduce the notion of bounded graph automata. Note that

two other atomic cospans, namely *shift* and *trans*, which play the role of *perm*. The reason behind this change lies in the implementation of Raven.

the bound is introduced to obtain a finite model, i. e. graph automata only accept graphs or copsans respectively up to a given maximum pathwidth. The main difference between bounded graph automata and bounded automaton functors is that bounded automaton functors are defined on *all* cospans of bounded size (of which there are infinitely many), while graph automata are only defined for the letters of the input alphabet defined above, which only correspond to the atomic cospans (of which there are finitely many). But this is not a limitation, since we can adapt Proposition 6.25 to the set of atomic cospans given in Definition 6.30. Hence, we are able to represent all cospans of bounded size by a sequence of letters which are given in Table 6.1.

Definition 6.31 (Bounded graph automaton, [12])**.** Let $n \in \mathbb{N}$ and $k, \ell \leq n$ be given. An *n-bounded $\langle k, \ell \rangle$-graph automaton* $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ consists of

- $Q = (Q_i)_{i \leq n}$ the *family of finite state sets*,
- $\Sigma = (\mathcal{S}ig_i)_{i \leq n}$ the *doubly-ranked input alphabet* (see Table 6.1),
- $\delta = (\delta_{i,j})_{i,j \leq n}$ the *family of transition functions*, where

$$\delta_{i,j} \colon Q_i \times \Sigma_{i,j} \to \wp(Q_j),$$

- $\mathrm{I} \subseteq Q_k$ the set of *initial states* and
- $\mathrm{F} \subseteq Q_\ell$ the set of *final states*

such that the following *consistency condition* holds for all states $q \in Q$ and sequences $\vec{\sigma}_1, \vec{\sigma}_2 \in S_{i,j}$:

$$\text{if } \hat{\eta}(\vec{\sigma}_1) \simeq \hat{\eta}(\vec{\sigma}_2) \text{ then } \hat{\delta}_{i,j}(\{q\}, \vec{\sigma}_1) = \hat{\delta}_{i,j}(\{q\}, \vec{\sigma}_2), \tag{$\star$}$$

where $\hat{\delta}_{i,j} \colon \wp(Q_i) \times S_{i,j} \to \wp(Q_j)$ is defined as follows:

$$\hat{\delta}_{i,j}(R, \vec{\sigma}) := \begin{cases} R & \text{if } \vec{\sigma} = \varepsilon_i \in \Sigma_{i,i} \text{ and } i = j \\ \delta(R, \sigma) & \text{if } \vec{\sigma} = \sigma \in \Sigma_{i,j} \\ \hat{\delta}_{k,j}(\delta_{i,k}(R, \vec{\sigma}_1), \vec{\sigma}_2) & \text{if } \vec{\sigma} = (\vec{\sigma}_1 \,;\, \vec{\sigma}_2), \vec{\sigma}_1 \in S_{i,k}, \vec{\sigma}_2 \in S_{k,j} \end{cases}.$$

A sequence $\vec{\sigma} \in S_{k,\ell}$ over Σ is *accepted* by $\mathcal{A}$ if and only if $\hat{\delta}_{k,\ell}(\mathrm{I}, \vec{\sigma}) \cap \mathrm{F} \neq \emptyset$.

Remember that the function $\hat{\eta}$ denotes the evaluation function defined in Definition 6.30.

In contrast to automaton functors as defined in Section 6.1.1 a graph automaton does not process the complete input graph at once but "piece by piece" in the following sense: First, the input graph has to be given in terms of an atomic cospan decomposition (see Chapter 5). Then, the graph automaton processes each atomic cospan one after another. After finishing this process the graph automaton either accepts or rejects the given atomic cospan decomposition and hence the given input graph. This is a difference to automaton functors which can process the whole graph (seen as a cospan) at once. The consistency condition ($\star$) guarantees that the graph automaton accepts an input graph independently of the decomposition of the graph and therefore corresponds to

the functor property of automaton functors. Showing that this condition holds for some prospective graph automaton is not trivial in general. One solution is to use the axiomatization presented in Subsection 6.1.4, but still a number of equations have to be checked. Another solution would be to automatically translate formulas of monadic second-order logic to correct graph automata. This approach will be further discussed in Section 6.2.2.

Definition 6.32 (Accepted language)**.** Let an n-bounded $\langle k, \ell\rangle$-graph automaton $\mathcal{A}$ be given. The *language* accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is

$$\mathcal{L}(\mathcal{A}) = \{c\colon D_k \nrightarrow D_\ell \mid \hat{\eta}(\vec{\sigma}) = c \text{ for some } \vec{\sigma} \text{ accepted by } \mathcal{A}\}.$$

The language of a bounded graph automaton contains cospans. Similar to the situation for automaton functors, if we want to accept graphs, we can interpret the cospan $[G]$ as the graph G.

Since a graph automaton is bounded, it is a kind of non-deterministic finite automaton (NFA). Therefore, we can apply standard algorithms from formal language theory, such as the subset construction and constructing the cross product of two automata. It can be shown that these constructions preserve condition $(\star)$ of graph automata. Thus, the languages accepted by n-bounded graph automata are closed under boolean operations, and many important decision problems (such as the membership, emptiness and language inclusion problems) are decidable. Note that the language inclusion algorithm for NFA is PSPACE-complete, and thus no algorithms for the problem are known, which always run in polynomial time.

Example 6.33. *Let $k \in \mathbb{N}$. We take another look at the k-colorability example (see Example 6.4) and give a graph automaton accepting the language $C_{(k)}$ of all k-colorable graphs of bounded pathwidth.*

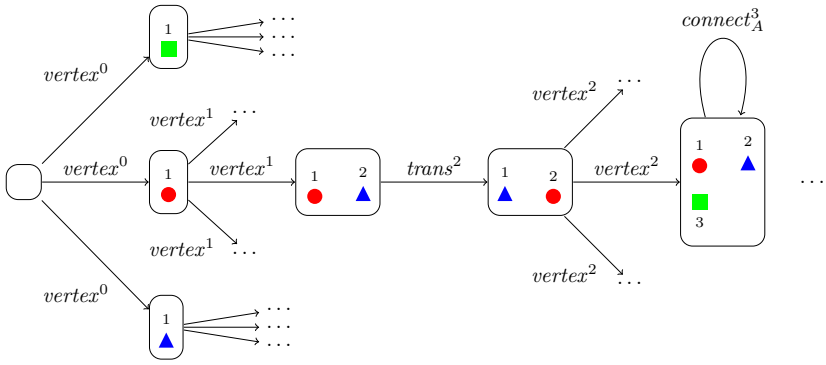


Figure 6.1.: Excerpt of the n-bounded $\langle 0, 0\rangle$-graph automaton accepting the language $C_{(3)}$

The idea of the graph automaton $\mathcal{A}_{(k)}$ accepting all k-colorable graphs is as follows:

Every state is a valid k-coloring of D_i, i. e.

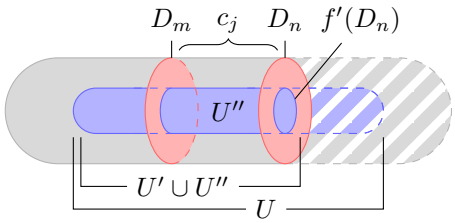
$$Q_i = \{f \colon V_{D_i} \to \mathbb{N}_k \mid f \text{ is a valid } k\text{-coloring of } D_i\}.$$

The transition function $\delta_{i,j}$ maps a coloring $f \in Q_i$ and a letter $\sigma \in \Sigma$ to a coloring $f' \in Q_j$ if and only if the coloring of the inner nodes of $\eta(\sigma)$ according to f and the coloring of the outer nodes of $\eta(\sigma)$ according to f' lead to a valid coloring of $\eta(\sigma)$. In Figure 6.1, a small excerpt of the 3-colorability $\langle 0, 0\rangle$-graph automaton is shown. In the left-most state, no node is colored since the interface is empty. Assume that the next symbol is vertex^0. The graph automaton non-deterministically goes into the next state, which has one node in the interface. The new node can be colored by any of the colors (■, ● and ▲). So there are three possible successor states. Let us assume that the new node is colored ●. Now, let the next cospan of the input be vertex^1. The first node is colored with the same color as before, whereas the new node can be colored by any of the colors. This time, let us assume that that the new node is colored ▲ and let the next cospan of the input be trans. This cospan only changes the order of the nodes in the interface, therefore we color the first node with ▲ and the second node with ●. We assume that the next symbol is vertex^2. As before, the nodes 1 and 2 are colored with the same colors as in the previous state. The color of the new third node can be chosen non-deterministically. Here we assume that the color of the third node is ■. Let the last cospan be connect^3_A. This cospan is mapped to a relation which relates the (current) state with itself, since the coloring of the current state is also a valid coloring for the successor state, because the added edge is incident to the second and the third interface node which have different colors.

More details on graph automata for coloring can be found in Section 7.1 and in Appendix B.1.

Example 6.34. *Second we consider the language $\mathcal{L}_U$ of all graphs of bounded pathwidth which contain a fixed subgraph U. The bounded graph automaton $\mathcal{A}_U$ accepting this language works as follows:*

Every state in each of the state sets Q_i contains two pieces of information. The first piece of information states which parts of the subgraph have already been recognized. The second piece of information is a function which maps every outer node to a node which has already been recognized or to some "bottom element" to indicate that the interface node is not mapped to a node of the wanted subgraph U.



The picture above shows an example of an input graph (the gray-filled area) seen as cospan which is divided in three parts:

- *the first part, starting with the empty interface on the left and ending with the interface D_m, represents the piece of the input graph which has already been processed,*

- *the second part, starting with the interface D_m (depicted by the pink ellipse on the left) on the left and ending with the interface D_n (depicted by the pink ellipse on the right), represents the piece of the input graph which is currently processed by the subgraph automaton,*
- *the last part, starting with the interface D_n on the left and ending with the empty interface on the right, represents the piece of the input graph which has still to be processed.*

Before the cospan $c_j \colon D_m \nrightarrow D_n$ is processed as depicted in the picture above, the automaton has already recognized the graph U' which is a subgraph of the wanted subgraph U. The transition function "updates" this information according to the cospan c_j which is currently processed. Since the cospan c_j contains the subgraph U'' which is also a subgraph of U the found subgraph can be updated to $U' \cup U''$. Note that the function $f'(D_n)$ represents the part of subgraph $U' \cup U''$ which is currently accessible by the outer interface of the cospan c_j.

Since the input graph might contain several parts which are isomorphic to the wanted subgraph U, the bounded graph automaton is highly non-deterministic. More details about the construction of this graph automaton can be found in [10].

Next, we compare graph automata with automaton functors. Therefore, we give a slightly different notion of automaton functors, which are instantiated to the category $\mathbf{OLCG}_n$. Hence, this type of automaton functors is also bounded.

Definition 6.35 (Bounded Automaton Functor)**.** Let $n \in \mathbb{N}$ and $k, \ell \leq n$. An *n-bounded $\langle k, \ell \rangle$-automaton functor* is a structure $\mathcal{A} = \langle \mathcal{A}_0, I, F \rangle$, where

- $\mathcal{A}_0 \colon \mathbf{OLCG}_n \to \mathbf{Rel}$ is a functor which maps every discrete graph D_i of $\mathbf{OLCG}$ to a finite set $\mathcal{A}_0(D_i)$ (the *state set of* D_i) and every output linear cospan $c \colon D_i \nrightarrow D_j$ to a relation $\mathcal{A}_0(c) \subseteq \mathcal{A}_0(D_i) \times \mathcal{A}_0(D_j)$ (the *transition relation* of c),
- $\mathrm{I} \subseteq \mathcal{A}_0(D_k)$ is the *set of initial* states and
- $\mathrm{F} \subseteq \mathcal{A}_0(D_\ell)$ is the *set of final states.*

For a discrete graph D_i or a output linear cospan c we will, in the following, usually write $\mathcal{A}(D_i)$ and $\mathcal{A}(c)$ instead of $\mathcal{A}_0(D_i)$ and $\mathcal{A}_0(c)$, respectively. A cospan $c \colon D_k \nrightarrow D_\ell$ is accepted by $\mathcal{A}$, if $\langle q, q' \rangle \in \mathcal{A}(c)$ for some $q \in \mathrm{I}$ and $q' \in \mathrm{F}$.

The following theorem relates the notions of automaton functors and graph automata.

Theorem 6.36. Let $\mathcal{L}$ be a language of cospans from D_k to D_ℓ. Then $\mathcal{L}$ is the language of an n-bounded $\langle k, \ell \rangle$-graph automaton if and only if it is the language of an n-bounded $\langle k, \ell \rangle$-automaton functor.

Proof. ($\Leftarrow$): Let Λ be a set of labels and let $\mathcal{B} = \langle (Q_i)_{i \leq n}, \mathcal{S}ig, \delta, \mathrm{I}', \mathrm{F}' \rangle$ be an n-bounded graph automaton. We construct an n-bounded automaton functor $\mathcal{A} = \langle \mathcal{A}_0, \mathrm{I}, \mathrm{F} \rangle$ as follows:

- $\mathcal{A}_0(D_i) = Q_i$
- Let $c\colon J_0 \to G \leftarrow J_n$ (where J_0 and J_n are discrete) be a cospan and let $c = c_1 \,;\, \ldots \,;\, c_n$, where $c_1, \ldots, c_n$ are atomic cospans. Assume that $c_i\colon J_{i-1} \to G_i \leftarrow J_i$ (where J_{i-1} and J_i are discrete). Then $\mathcal{A}_0(c) = \delta_{j_0,j_1}(\eta^{-1}(c_1)) \,;\, \ldots \,;\, \delta_{j_{n-1},j_n}(\eta^{-1}(c_n))$ where j_i is the size of J_i $(0 \leq i \leq n)$. This is well-defined independent of the decomposition of c, since the consistency condition $(\star)$ holds for $\mathcal{B}$. Note that the identity cospan id_{D_i} can be seen as the empty decomposition (in atomic cospans), hence $\mathcal{A}_0(id_{D_i}) = \delta_{i,i}(\varepsilon_i)$.

Now we show that $\mathcal{A}_0$ is a functor. First, we observe that $\mathcal{A}$ preserves identities:

$$\mathcal{A}_0(id_{D_i}) = \delta_{i,i}(\varepsilon_i) \overset{\text{Def. 6.31}}{=} id_{Q_i}.$$

Next, we show that $\mathcal{A}$ preserves composition. Let $c^1 = c^1_1 \,;\, \ldots \,;\, c^1_n$ and $c^2 = c^2_1 \,;\, \ldots \,;\, c^2_m$ be cospans, where c^1_i and c^2_j $(1 \leq i \leq n,\ 1 \leq j \leq m)$ are atomic cospans, such that c^1 and c^2 are composable. We choose the atomic building blocks $\sigma^1_1, \ldots, \sigma^1_n, \sigma^2_1, \ldots, \sigma^2_m$ such that $\eta(\sigma^1_i) = c^1_i$ and $\eta(\sigma^2_j) = c^2_j$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$. Then we have

$$\mathcal{A}_0(c^1) \,;\, \mathcal{A}(c^2) = \delta_{j_0,j_1}(\sigma^1_1) \,;\, \ldots \,;\, \delta_{j_{n-1},j_n}(\sigma^1_n) \,;\, \delta_{j_n,j_{n+1}}(\sigma^2_1) \,;\, \ldots \,;\, \delta_{j_{n+m-1},j_{n+m}}(\sigma^2_m) = \mathcal{A}_0(c^1 \,;\, c^2)$$

$(\Rightarrow)$: Let Λ be a set of labels and let $\mathcal{A} = \langle \mathcal{A}_0, \mathrm{I}, \mathrm{F} \rangle$ be an n-bounded automaton functor. We construct an n-bounded graph automaton $\mathcal{B} = \langle Q, \Sigma, \delta, \mathrm{I}', \mathrm{F}' \rangle$ as follows:

- $Q = (\mathcal{A}_0(D_i))_{i \leq n}$
- $\Sigma = \mathcal{S}ig$
- $\mathrm{I}' = \mathrm{I}$
- $\mathrm{F}' = \mathrm{F}$
- for all $\sigma \in \Sigma$, $\delta(\sigma) = \mathcal{A}_o(c)$, where c is the atomic cospan corresponding to σ, i.e. $\eta(\sigma) = c$

Now, we show that the "consistency condition" holds. Let $\vec{\sigma}^1 = \sigma^1_1 \,;\, \ldots \,;\, \sigma^1_n$, $\vec{\sigma}^2 = \sigma^2_1 \,;\, \ldots \,;\, \sigma^2_m \in S_\Sigma$ with $\hat{\eta}(\vec{\sigma}^1) \simeq \hat{\eta}(\vec{\sigma}^2)$ and let $c^1_1, \ldots, c^1_n, c^2_1, \ldots, c^2_m$ be the atomic cospans corresponding to the atomic building blocks $\sigma^1_1, \ldots, \sigma^1_n$, $\sigma^2_1, \ldots, \sigma^2_m$.

$$\hat{\delta}(\vec{\sigma}^1) \overset{\text{Def.}}{=} \delta(\sigma^1_1) \,;\, \ldots \,;\, \delta(\sigma^1_n) \overset{\text{Def.}}{=} \mathcal{A}_0(c^1_1) \,;\, \ldots \,;\, \mathcal{A}_0(c^1_n) \overset{(*)}{=} \mathcal{A}_0(c^1_1 \,;\, \ldots \,;\, c^1_n)$$
$$\overset{(**)}{=} \mathcal{A}_0(c^2_1 \,;\, \ldots \,;\, c^2_m) \overset{(*)}{=} \mathcal{A}_0(c^2_1) \,;\, \ldots \,;\, \mathcal{A}_0(c^2_m) \overset{\text{Def.}}{=} \delta(\sigma^2_1) \,;\, \ldots \,;\, \delta(\sigma^2_m) \overset{\text{Def.}}{=} \hat{\delta}(\vec{\sigma}^2)$$

The equation $(*)$ holds due to the functor property of $\mathcal{A}$. The equation $(**)$ holds, since the cospans $c^1_1 \,;\, \ldots \,;\, c^1_n = \hat{\eta}(\vec{\sigma}^1)$ and $c^2_1 \,;\, \ldots \,;\, c^2_m = \hat{\eta}(\vec{\sigma}^2)$ are isomorphic by definition.

□

In the remainder of this section we give an introduction to a straightforward approach to verification which is based on recognizable graph languages. The main idea is to provide an invariant and to check that it is preserved by all graph transformation rules (see Definition 3.6).

Definition 6.37 (Invariant)**.** A language $\mathcal{L}$ is an *invariant* with respect to a graph transformation rule ρ if it holds for all graphs G and H with $G \overset{\rho}{\Longrightarrow} H$ that $[G] \in \mathcal{L}$ implies $[H] \in \mathcal{L}$.

If we assume that the language $\mathcal{L}$ is a recognizable graph language, we can make use of Theorem 6.22. Hence, the recognizable graph language $\mathcal{L}$ is an invariant for some graph transformation rule $\rho = \langle \ell, r \rangle$ if and only if ℓ and r are related w. r. t. a monotone well-quasi-order such that $\mathcal{L}$ is upward-closed w. r. t. this well-quasi-order. The coarsest such order is the Myhill-Nerode quasi-order of a language $\mathcal{L}$ and it can be computed by a fixed-point iteration similar to the computation of the minimal finite automaton [15]. The main disadvantage of this approach is that the algorithm for computing the Myhill-Nerode quasi-order applies only to deterministic (graph) automata, whereas in general our graph automata are highly non-deterministic. Determinization is not an option because it would lead to an exponential blow-up of already huge automata. Therefore, we had to settle for an approximation which even led to new difficulties [15].

But there is also another connection between invariants on the one hand and the language inclusion problem on the other hand. Before we give the theorem, we need a further definition. For an output-linear cospan $c\colon D_i \nrightarrow D_j$ and an n-bounded $\langle i, k \rangle$-graph automaton $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ we obtain a new n-bounded $\langle j, k \rangle$-graph automaton $\mathcal{A}[c] = \langle Q, \Sigma, \delta, \mathrm{I}', \mathrm{F} \rangle$ with $\mathrm{I}' = \hat{\delta}_{i,j}(\mathrm{I}, \vec{\sigma})$, where $\vec{\sigma}$ is some word over the alphabet $\mathcal{S}ig$ such that $\hat{\eta}(\vec{\sigma}) = c$. (If the width of c is larger than n, such a $\vec{\sigma}$ does not exist, and we take $\mathrm{I}' = \emptyset$, such that $\mathcal{L}(\mathcal{A}[c]) = \emptyset$.) The initial states of the new automaton are all states reachable from the initial states of the original automaton by processing c. Note that I' is independent of the specific decomposition of c into a sequence $\vec{\sigma}$.

Theorem 6.38 (Invariant Checking)**.** Let $\mathcal{A}$ be an n-bounded $\langle 0, k \rangle$-graph automaton accepting the cospan language $\mathcal{L}$, and let $\rho = \langle \ell, r \rangle$ be a transformation rule. The cospan language $\mathcal{L}$ is an invariant of ρ if and only if $\mathcal{L}(\mathcal{A}[\ell]) \subseteq \mathcal{L}(\mathcal{A}[r])$.

Proof. Trivial. □

The quintessence of the theorem is that we solve the language inclusion problem to decide whether a language is an invariant for a given transformation rule. As already mentioned before, we can see graph automata as (very large) non-deterministic finite automata. Hence, we can use recent approaches to solve the language inclusion problem for NFAs which still need exponential time in general, but in practice often achieve better runtimes. In Chapter 7 we will explain how approaches to solve the language inclusion problem for NFAs can be adapted to the setting of graph automata.

6.2. Logic and Recognizability

The topic of this section is the strong connection between monadic second-order logic and recognizable languages. Monadic second-order logic is an extension of first-order logic, which allows quantification not only over individual elements, but also over sets of elements. The quantification over arbitrary relations is not permitted, in contrast to general second-order logic.

In their seminal work Büchi and Elgot [38, 65] and indepently Trakhtenbrot [126] showed that the class of regular (word) languages is exactly the class of languages definable in monadic second-order logic (on words), which is called the theorem of Büchi-Elgot-Trakhtenbrot. Since the notion of recognizability can be straightforwardly generalized to languages of trees, it is not surprising that the Büchi-Elgot-Trakhtenbrot theorem can be carried over to the setting of trees, too [123, 124]. For the case of recognizable graph languages we can only obtain the result that every set of graphs definable in monadic second-order logic is recognizable, which is due to Courcelle [44]. The opposite direction does not hold due to the fact that there are only countably many different monadic second-order definable sets of graphs but uncountably many recognizable graph languages.

6.2.1. Monadic Second-Order Logic on Graphs

In this section, we introduce the *monadic second-order logic on graphs*, or MSOGL for short, [43, 44, 48], which is a fragment of the sorted second-order logic, used as one of the most important specification logics for graphs. Especially Courcelle's theorem [44] says that every graph property definable in MSOGL is decidable in linear time on (finite) graphs of bounded treewidth (see also below). Note that we do not introduce MSOGL to use it as a specification language, but to compare it to the logic LCL, which is presented in Subsection 6.2.2.

The syntax of MSOGL which is based on the two sorts v (for vertices) and e (for edges) is defined as follows:

Definition 6.39 (Syntax of MSOGL)**.** Let $\mathcal{V} = \{x_1, x_2, x_3, \ldots, X_1, X_2, X_3, \ldots\}$ be a set of variables where $x_1, x_2, x_3, \ldots$ denote first-order variables and $X_1, X_2, X_3, \ldots$ denote second-order variables. We define the *syntax* inductively:

- $x \in X$, $x = y$ and $edge_A(x, y_1, \ldots, y_{ar(A)})$ are formulas,
- if φ and ψ are formulas, then also $\neg\varphi$, $\varphi \wedge \psi$ are formulas,
- if φ is a formula, then also $(\exists x\colon v)\,\varphi$, $(\exists x\colon e)\,\varphi$, $(\exists X\colon V)\,\varphi$ and $(\exists X\colon E)\,\varphi$ are formulas,

where typing must be respected, that is, in formulas of the form $x = y$ both variables have the same type, in formulas of the form $x \in X$ it holds that x is a first-order vertex (edge) variable and X a second-order vertex (edge) variable and in formulas of the form $edge_A(x, y_1, \ldots, y_{ar(A)})$ it holds that x is a first-order edge variable and $y_1, \ldots, y_n$ are first-order vertex variables.

As usual we define the following abbrevations:

$$x \notin X := \neg(x \in X), \qquad x \neq y := \neg(x = y)$$
$$\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi) \qquad \varphi \to \psi := \neg\varphi \vee \psi$$
$$\varphi \leftrightarrow \psi := (\varphi \to \psi) \wedge (\psi \to \varphi)$$
$$(\forall x\colon v)\,\varphi := \neg(\exists x\colon v)\,\neg\varphi \qquad (\forall x\colon e)\,\varphi := \neg(\exists x\colon e)\,\neg\varphi$$
$$(\forall x\colon V)\,\varphi := \neg(\exists x\colon V)\,\neg\varphi \qquad (\forall x\colon E)\,\varphi := \neg(\exists x\colon E)\,\neg\varphi$$

Similar to Definition 4.2 we have a valuation for each variable (for which the valuation is defined).

Definition 6.40 (Valuation)**.** Let $\mathcal{V} = \{x_1, x_2, x_3, \ldots, X_1, X_2, X_3, \ldots\}$ be a set of variables and let $G = \langle V_G, E_G, att_G, lab_G\rangle$ be a graph. A *$\mathcal{V}$-valuation θ (in the graph G)* is a function assigning to each first-order variable $x_i \in \mathcal{V}$ of type v (e) a node $\theta(x_i) \in V_G$ (an edge $\theta(x_i) \in E_G$) and to each second-order variable $X_i \in \mathcal{V}$ of type V (E) a set of nodes $\theta(X_i) \subseteq V_G$ (a set of edges $\theta(X_i) \subseteq E_G$).

Let θ be a $\mathcal{V}$-valuation in the graph G, $x \in \mathcal{V}$ be a first-order variable of type v and $t \in V_G$ be a node. By $\theta[x \mapsto t]$ we denote the $\mathcal{V}$-valuation defined as

$$\theta[x \mapsto t]\colon \mathcal{V} \to G, \quad y \mapsto \begin{cases} t, & \text{if } y = x \\ \theta(y), & \text{else} \end{cases}$$

The substitution for first-order edge variables and second-order variables is defined analogously. This can be generalized to an arbitrary number of substitutions in the obvious manner.

Now we can define the semantics of the monadic second-order logic of graphs:

Definition 6.41 (Semantics of MSOGL)**.** A graph $G = \langle V_G, E_G, att_G, lab_G\rangle$ satisfies a formula φ, written $G \models \varphi$, if there exists a $\mathcal{V}$-valuation θ such that $G, \theta \models \varphi$, where:

- $G, \theta \models x = y$ if $\theta(x) = \theta(y)$ and $G, \theta \models x \in X$ if $\theta(x) \in \theta(X)$.
- $G, \theta \models edge_A(x, y_1, \ldots, y_n)$ if $lab_G(\theta(x)) = A$ and $att_G(\theta(x)) = \theta(y_1)\ldots\theta(y_n)$.
- $G, \theta \models \varphi_1 \wedge \varphi_2$ if $G, \theta \models \varphi_1$ and $G, \theta \models \varphi_2$.
- $G, \theta \models \neg\varphi$ if $G, \theta \not\models \varphi$.
- $G, \theta \models (\exists x\colon v)\,\varphi$ if there is a $v' \in V_G$ such that $G, \theta[x \mapsto v'] \models \varphi$.
- $G, \theta \models (\exists x\colon e)\,\varphi$ if there is a $e' \in E_G$ such that $G, \theta[x \mapsto e'] \models \varphi$.
- $G, \theta \models (\exists X\colon V)\,\varphi$ if there is a $V \subseteq V_G$ such that $G, \theta[X \mapsto V] \models \varphi$.

- $G, \theta \models (\exists X \colon E)\, \varphi$ if there is a $E \subseteq E_G$ such that $G, \theta[X \mapsto E] \models \varphi$.

Let φ be an arbitrary MSOGL-formula, by $\mathcal{L}(\varphi)$ we denote the language of φ, i. e. the set of graphs satisfying the property φ, or more formally $\mathcal{L}(\varphi) = \{G \mid G \models \varphi\}$.

Many NP-complete problems can be defined using MSOGL. One example, the set of all 3-colorable graphs, can be expressed by the following MSOGL-formula:

Example 6.42.

$$
\begin{aligned}
&(\exists X_1 \colon V)\,(\exists X_2 \colon V)\,(\exists X_3 \colon V) \\
&(\forall x \colon v)\ (x \in X_1 \vee x \in X_2 \vee x \in X_3) \wedge \\
&(\forall x \colon v)\ ((x \notin X_1 \vee x \notin X_2) \wedge (x \notin X_1 \vee x \notin X_3) \wedge (x \notin X_2 \vee x \notin X_3)) \wedge \\
&(\forall x_1 \colon v)\,(\forall x_2 \colon v)\,(\forall y \colon e)\,\Big(edge_\star(y, x_1, x_2) \rightarrow \\
&\qquad \big((x_1 \notin X_1 \vee x_2 \notin X_1) \wedge (x_1 \notin X_2 \vee x_2 \notin X_2) \wedge (x_1 \notin X_3 \vee x_2 \notin X_3)\big)\Big)
\end{aligned}
$$

The first line defines the existence of three colors, the second and third line state that every node has exactly one color and the last line guarantees that adjacent nodes have different colors.

Other problems definable in MSOGL are sets of graphs which are planar, which have a Hamiltonian circuit[8], a dominating set (of size less than some constant k)[9], or a vertex cover (of size less than k).

As already mentioned above, a well-known result about graph languages and monadic second-order logic is the following famous theorem by Courcelle:

Theorem 6.43 (Courcelle's Theorem [44])**.** Any property of graphs definable in monadic second-order logic can be decided in linear time for graphs of bounded treewidth.

The idea is to construct a tree automaton for a MSOGL-formula φ and a bound on the maximum treewidth w that accepts a tree decomposition of size at most w if and only if the tree decomposition, corresponding to the input graph, satisfies the formula φ.

It directly follows that many problems, which are NP-complete, become linear-time solvable if we restrict the input to graphs of bounded treewidth. However, the result is only of theoretical interest, since the multiplicative constants in the running time are usually exponential (or worse) in the treewidth. This is due to the fact that for every negation occuring in the formula φ one has to first determinize the tree automaton, which leads to an exponential blow-up, and afterwards build the "complement automaton".

Therefore, we will study another logic in the following section, for which we try to avoid the issues above.

[8] A graph contains a *Hamiltonian circuit* if there exists a cycle which contains every node of the graph exactly once.

[9] A *dominating set (of size at most k)* of a graph is a set D (of size at most k) such that every node of the graph is either contained in D or adjacent to some node in D.

6.2.2. Linear Cospan Logic

In this section, we introduce another logic which describes properties of cospans of graphs and which has therefore a strong connection to the graph automata defined above in Section 6.1.5.

The idea of a logic which "operates" in a more categorical setting and which has the expressive power of monadic second-order logic has been inspired by the so-called *subobject logic* of Bruggink and König [32]. The greatest problem of this logic is that the construction of graph automata from a given subobject logic formula requires exponential space (and time), even if the formula is rather small. This is due to the powerset construction which is needed in case of negations.

To overcome this issue we introduce another logic called *Linear Cospan Logic* (LCL). The basic idea is to enrich this logic with a number of predefined properties, e. g. k-colorability or planarity, which are handled as atomic formulas and to avoid the cost-intensive negation (see below). This idea is similar to an approach by Courcelle and Durand as well as Courcelle and Engelfriet [46, 48].

The syntax of LCL is defined as follows:

Definition 6.44 (Syntax of LCL)**.** Let $\Pi = (\Pi_{i,j})_{i,j\in\mathbb{N}}$ be a (doubly-ranked) set of *predicates*. We define the *syntax* inductively:

Atomic Formulas: Every predicate $\pi\colon i \to j$ with $\pi \in \Pi_{i,j}$ is a formula.

Negation and Conjunction: If $\varphi\colon i \to j$ and $\psi\colon i \to j$ are both formulas, then $\neg\varphi\colon i \to j$ and $\varphi \wedge \psi\colon i \to j$ are formulas.

Existential Quantification: If $\varphi\colon i \to j$ and $\psi\colon j \to k$ are formulas, then $(\exists\varphi\colon \psi)\colon i \to k$ is a formula.

Existential Shift: If $\varphi\colon i \to k$ and $\psi\colon i \to j$ are formulas, then $(\varphi{\downarrow}^{\exists}_{\psi})\colon j \to k$ is a formula.

The set of all formulas of type $i \to j$ over the alphabet Σ is denoted by $\Phi^{\Sigma}_{i,j}$. We set $\Phi^{\Sigma}_{n} = \bigcup_{i,j\leq n} \Phi^{\Sigma}_{i,j}$ and $\Phi^{\Sigma} = \bigcup \Phi^{\Sigma}_{n}$. If the alphabet is obvious from the context, we will also write Φ.

The semantics of the logic can then be defined as follows, where we fix the category $\mathbf{OLCG}_n$ as universe and we assume that we have an graph automaton $\mathcal{A}_\pi$ for each predicate $\pi \in \Sigma$.

Definition 6.45 (Semantics of LCL)**.** The *semantics* are defined inductively:

$$\begin{aligned} [\![\pi]\!] &= \mathcal{L}(\mathcal{A}_\pi), \text{ for } \pi\colon i \to j, \\ [\![\neg\varphi]\!] &= \mathcal{M}(D_i, D_j) \setminus [\![\varphi]\!], \text{ for } \varphi\colon i \to j, \\ [\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!], \end{aligned}$$

$$[\![\exists\varphi\colon \psi]\!] = [\![\varphi]\!] \,;\, [\![\psi]\!] = \{c\colon D_i \nrightarrow D_k \mid \exists c'\colon D_i \to D_j, c''\colon D_j \nrightarrow D_k\colon c = c' \,;\, c'' \wedge c' \in [\![\varphi]\!] \wedge c'' \in [\![\psi]\!]\},\ \text{for } \varphi\colon i \to j,\ \psi\colon j \to k,$$

$$\left[\!\!\left[\varphi{\downarrow}^{\exists}_{\psi}\right]\!\!\right] = \{c'\colon D_j \nrightarrow D_k \mid \exists c\colon D_i \nrightarrow D_j\colon c \in [\![\psi]\!] \wedge c \,;\, c' \in [\![\varphi]\!]\} \quad \text{for } \varphi\colon i \to k,\ \psi\colon i \to j.$$

The idea of the shift operation is to partially evaluate a formula φ on a set of cospans which satisfy ψ. Hence, the "shifted" formula describes exactly those cospans which satisfy φ whenever these cospans are precomposed by a cospan satisfying ψ. This originates from an operation for nested application conditions (for transformation rules), introduced by Pennemann [108].

As stated above, we assume that automata for the predicates are given before. The conjunction of two formulas can be obtained by computing the automaton for the intersection of the underlying automata (see Proposition 6.17).

Furthermore, the automaton for the existential quantification can be efficiently obtained from the underlying automata by computing the concatenation automaton (see Proposition 6.18). The automaton $\mathcal{A}_{\varphi{\downarrow}^{\exists}_{\psi}}$ for the existential shift is computed by a construction which takes the automata $\mathcal{A}_\varphi = \langle Q_\varphi, \Sigma, \delta_\varphi, \mathrm{I}_\varphi, \mathrm{F}_\varphi\rangle$ for φ and $\mathcal{A}_\psi = \langle Q_\psi, \Sigma, \delta_\psi, \mathrm{I}_\psi, \mathrm{F}_\psi\rangle$ for ψ. The set of states, the transition function and the set of final states of $\mathcal{A}_{\varphi{\downarrow}^{\exists}_{\psi}}$ are exactly those of $\mathcal{A}_\varphi$. The set of initial states of $\mathcal{A}_{\varphi{\downarrow}^{\exists}_{\psi}}$ are determined as follows: A state $s \in Q_\varphi$ is an initial state of $\mathcal{A}_{\varphi{\downarrow}^{\exists}_{\psi}}$ if and only if there exists a sequence $\vec{\sigma} \in S_\Sigma$ such that $s \in \hat{\delta}_\varphi(\mathrm{I}_\varphi, \vec{\sigma})$ and $\hat{\delta}_\psi(\mathrm{I}_\psi, \vec{\sigma}) \cap \mathrm{F}_\psi \neq \emptyset$. For more details of the construction of the automaton $\mathcal{A}_{\varphi{\downarrow}^{\exists}_{\psi}}$ see also the proof of Theorem 6.52 below.

We assume that the automaton for the existential quantification can also be obtained in an efficient manner. But it is planned for the future to find a more efficient construction. The computation of the "negation automaton" (for a negated formula) involves the determinization of the automaton for the underlying formula which causes an exponential blow up. Hence, this operation cannot be implemented efficiently. Therefore, we try to avoid negations by transforming formulas containing negations. Further details of this idea will be explained below.

If $c\colon D_i \nrightarrow D_j$ is a cospan and $\varphi\colon i \to j$ is a formula, we will write $c \models \varphi$ ($c \not\models \varphi$) if and only if $c \in [\![\varphi]\!]$ ($c \notin [\![\varphi]\!]$). Furthermore, we define the following abbreviations:

$$\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi) \qquad \varphi \to \psi := \neg\varphi \vee \psi \qquad \varphi \leftrightarrow \psi := (\varphi \to \psi) \wedge (\psi \to \varphi)$$

$$\forall\psi\colon \varphi := \neg\,(\exists\psi\colon \neg\varphi) \qquad \varphi{\downarrow}^{\forall}_{\psi} := \neg\left(\neg\varphi{\downarrow}^{\exists}_{\psi}\right)$$

Similar to the existential shift above, we call the formula $\varphi{\downarrow}^{\forall}_{\psi}$ *universal shift*.

Additionally, if $c\colon D_i \nrightarrow D_j$ is a cospan, $C \subseteq \mathcal{M}(D_i, D_j)$ is a (not neccessarily finite) set of cospans and $\varphi\colon i \to k$ is a formula, then we define the formulas $\exists c\colon \varphi$, $\exists C\colon \varphi$ and $\varphi{\downarrow}_c$ as follows

$$[\![\exists c\colon \varphi]\!] = \{d\colon D_i \nrightarrow D_k \mid \exists c'\colon D_j \nrightarrow D_k\colon d = c \,;\, c' \wedge c' \models \varphi\}$$
$$[\![\exists C\colon \varphi]\!] = \{d\colon D_i \nrightarrow D_k \mid \exists c'' \in C, c'\colon D_j \nrightarrow D_k\colon d = c'' \,;\, c' \wedge c' \models \varphi\}$$
$$[\![\varphi{\downarrow}_c]\!] = \{c'\colon D_j \nrightarrow D_k \mid c \,;\, c' \models \varphi\}$$

Note that the first two operations are just special cases of $\exists\psi\colon \varphi$, where $[\![\psi]\!]$ is a single cospan or a fixed set of cospans respectively. The third operation is just a special case of the existential or universal shift respectively, i. e. if $[\![\psi]\!] = \{c\}$, then $[\![\varphi{\downarrow_c}]\!] = \left[\!\!\left[\varphi{\downarrow_\psi^\exists}\right]\!\!\right] = \left[\!\!\left[\varphi{\downarrow_\psi^\forall}\right]\!\!\right]$. Furthermore, the first two operations can be defined for universal quantification in the obvious way.

For the modified shift operations we give the following proposition which can be immediately obtained from the definitions:

Proposition 6.46. Let $c\colon D_i \nrightarrow D_j$ and $d\colon D_j \nrightarrow D_k$ be cospans and let $\varphi\colon i \to k$ be a formula. Then

$$c\,;d \models \varphi \iff d \models \varphi{\downarrow_c}.$$

Proof. Trivial, by definition. □

That the universal quantification and the universal shift can be seen as special cases of the existential quantification and the existential shift respectively, is motivated by the following two propositions.

The following lemma states that our definition of the universal quantification is adequate.

Proposition 6.47. Let $\varphi\colon i \to j$ and $\psi\colon j \to k$ be formulas. Then

$$[\![\forall\varphi\colon \psi]\!] = \{c\colon D_i \nrightarrow D_k \mid \forall c'\colon D_i \nrightarrow D_j, c''\colon D_j \nrightarrow D_k\colon (c = c'\,;c'' \wedge c' \models \varphi) \implies c'' \models \psi\}.$$

Proof. See Appendix A.2. □

The lemma below states the same result for the universal shift.

Proposition 6.48. Let $\varphi\colon i \to k$ and $\psi\colon i \to j$ be formulas. Then

$$\left[\!\!\left[\varphi{\downarrow_\psi^\forall}\right]\!\!\right] = \{c'\colon D_j \nrightarrow D_k \mid \forall c\colon D_i \nrightarrow D_j\colon c \models \psi \implies c\,;c' \models \varphi\}.$$

Proof. See Appendix A.2. □

The existential and the universal shift operations give rise to a very general invariant checking technique. For this setting we use a very universal graph transformation approach which differs from the approach presented in Chapter 3. Despite the replacement of a fixed left-hand side by a fixed right-hand side the approach considered here replaces a subgraph satisfying a formula ψ_1 by a subgraph satisfying a formula ψ_2. Note that the subgraphs for the left-hand and the right-hand side can be somewhat arbitrary (as long as the formulas are satisfied). Then the property specified by φ is preserved by such a transformation rule if and only if $\left[\!\!\left[\varphi{\downarrow_{\psi_1}^\exists}\right]\!\!\right] \subseteq \left[\!\!\left[\varphi{\downarrow_{\psi_2}^\forall}\right]\!\!\right]$.

Another approach which is based on the shift operations and which can be used for the automatic generation of invariants by computing weakest pre-conditions and

strongest post-conditions will be presented below. Before we can introduce this approach we first need some more machinary which is given below.

Next, we introduce the entailment relation on LCL-formulas.

Definition 6.49 (Entailment Relation)**.** We define the *entailment relation* $\models \subseteq \Phi_{i,j} \times \Phi_{i,j}$ as follows:

$$\varphi \models \psi \iff [\![\varphi]\!] \subseteq [\![\psi]\!]$$

As usual we also write $\varphi \equiv \psi$ if and only if $\varphi \models \psi$ and $\psi \models \varphi$ holds. It is easy to see that $\Phi_{i,j}$ together with the entailment relation is a quasi-ordered set.

Additionally, we immediately obtain the following laws:

Proposition 6.50. Let $\varphi_1, \varphi_2 \colon i \to k$, $\psi \colon i \to j$ and $\xi_1, \xi_2 \colon j \to k$ be formulas.

$$(\varphi_1 \wedge \varphi_2)\downarrow_\psi^\exists \equiv \varphi_1\downarrow_\psi^\exists \wedge \varphi_2\downarrow_\psi^\exists \qquad (\varphi_1 \wedge \varphi_2)\downarrow_\psi^\forall \equiv \varphi_1\downarrow_\psi^\forall \wedge \varphi_2\downarrow_\psi^\forall$$
$$(\varphi_1 \vee \varphi_2)\downarrow_\psi^\exists \equiv \varphi_1\downarrow_\psi^\exists \vee \varphi_2\downarrow_\psi^\exists \qquad (\varphi_1 \vee \varphi_2)\downarrow_\psi^\forall \equiv \varphi_1\downarrow_\psi^\forall \vee \varphi_2\downarrow_\psi^\forall$$
$$\exists\psi \colon (\xi_1 \vee \xi_2) \equiv \exists\psi \colon \xi_1 \vee \exists\psi \colon \xi_2 \qquad \forall\psi \colon (\xi_1 \wedge \xi_2) \equiv \forall\psi \colon \xi_1 \wedge \forall\psi \colon \xi_2$$

Proof. See Appendix A.2. □

Proposition 6.51. Let $c \colon D_i \nrightarrow D_j$ be a cospan and let $\varphi \colon i \to k$, $\psi \colon j \to k$ be formulas. Then

$$\varphi \models \exists c \colon \psi \iff \varphi\downarrow_c \models \psi \iff \varphi \models \forall c \colon \psi.$$

Proof. Trivial, by definition. □

Theorem 6.52. Let $n \in \mathbb{N}$ and let $\Pi = (\Pi_{i,j})_{i,j\in\mathbb{N}}$ be a (doubly-ranked) set of predicates. Then for every formula $\varphi \in \Phi_n^\Sigma$ the language $[\![\varphi]\!]$ is recognizable.

Proof. We prove this proposition by induction over the structure of φ.

Base case: Let $\varphi = \pi \in \Sigma_{i,j}$. By definition we have $[\![\pi]\!] = \mathcal{L}(\mathcal{A}^\pi)$ for some graph automaton $\mathcal{A}^\pi$, therefore $[\![\pi]\!]$ is recognizable.

Inductive step:

- Let $\varphi = \neg\psi$. By induction, $[\![\psi]\!]$ is recognizable. By Proposition 6.17 the recognizable languages are closed under complementation, hence we can build a graph automaton for φ using the automaton for ψ.
- Let $\varphi = \psi_1 \wedge \psi_2$. By induction, $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$ are recognizable. By Proposition 6.17 the recognizable languages are closed under intersection, hence we can build a graph automaton for φ using the automata for ψ_1 and ψ_2.

- Let $\varphi = \exists\psi_1 : \psi_2$. By induction, $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$ are recognizable. By Proposition 2.13 the recognizable languages are closed under concatenation.
- Let $\varphi = \psi_1{\downarrow}^{\exists}_{\psi_2}$. By induction, $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$ are recognizable by an $\langle i,k\rangle$-graph automaton $\mathcal{A}^{\psi_1} = \langle Q^{\psi_1}, \Sigma, \delta^{\psi_1}, \mathrm{I}^{\psi_1}, \mathrm{F}^{\psi_1}\rangle$ and a $\langle i,j\rangle$-graph automaton $\mathcal{A}^{\psi_2} = \langle Q^{\psi_2}, \Sigma, \delta^{\psi_2}, \mathrm{I}^{\psi_2}, \mathrm{F}^{\psi_2}\rangle$.

  Now, we define a $\langle j,k\rangle$-graph automaton $\mathcal{A}^{\varphi}$ which accepts the language $[\![\varphi]\!]$ as follows:

$$\mathcal{A}^{\varphi} = \langle Q^{\psi_1}, \Sigma, \delta^{\psi_1}, \mathrm{I}, \mathrm{F}^{\psi_1}\rangle$$

  with

$$\mathrm{I} = \{q_1' \mid \exists\vec{\sigma} \in S_{i,j} : \exists(q_1,q_2) \in \mathrm{I}^{\psi_1} \times \mathrm{I}^{\psi_2} : (q_1',q_2') \in \delta^{\psi_1\times\psi_2}(q_1,q_2,\vec{\sigma}) \wedge (q_1',q_2') \in Q^{\psi_1} \times \mathrm{F}^{\psi_2}\} \quad (6.18)$$

  and

$$\delta^{\psi_1\times\psi_2} : Q^{\psi_1} \times Q^{\psi_2} \times \Sigma \to \wp(Q^{\psi_1}) \times \wp(Q^{\psi_2}) \\ (q_1,q_2,\sigma) \mapsto (\delta^{\psi_1}(q_1,\sigma), \delta^{\psi_2}(q_2,\sigma)).$$

  That $\mathcal{A}^{\varphi}$ is a $\langle j,k\rangle$-graph automaton is obvious, since $\mathcal{A}^{\psi_1}$ is $\langle i,k\rangle$-graph automaton and $\mathcal{A}^{\psi_2}$ is $\langle i,j\rangle$-graph automaton. We now show that $\mathcal{L}(\mathcal{A}^{\varphi}) = [\![\varphi]\!]$, as follows:

$$\begin{aligned}
&\mathcal{L}(\mathcal{A}^{\varphi}) \\
&= \{c : D_j \nrightarrow D_k \mid \exists\vec{\sigma} \in S_{j,k}, \exists q \in I : \hat{\eta}(\vec{\sigma}) = c \wedge \delta^{\psi_1}(q,\vec{\sigma}) \cap F^{\psi_1} \neq \emptyset\} \\
&\overset{6.18}{=} \{c : D_j \nrightarrow D_k \mid \exists\vec{\sigma} \in S_{j,k}, \exists\vec{\sigma}' \in S_{i,j}, \exists q_1 \in \mathrm{I}^{\psi_1}, \exists q_2 \in \mathrm{I}^{\psi_2} : \\
&\quad \hat{\eta}(\vec{\sigma}) = c \wedge \delta^{\psi_1}(q_1, \vec{\sigma}' \,;\, \vec{\sigma}) \cap F^{\psi_1} \neq \emptyset \wedge \delta^{\psi_2}(q_2,\vec{\sigma}') \cap F^{\psi_2} \neq \emptyset\} \\
&= \{c : D_j \nrightarrow D_k \mid \exists\vec{\sigma} \in S_{j,k}, \exists\vec{\sigma}' \in S_{i,j} : \hat{\eta}(\vec{\sigma}) = c \wedge \\
&\quad \hat{\eta}(\vec{\sigma}' \,;\, \vec{\sigma}) \in \mathcal{L}\left(\mathcal{A}^{\psi_1}\right) \wedge \hat{\eta}(\vec{\sigma}') \in \mathcal{L}\left(\mathcal{A}^{\psi_2}\right)\} \\
&= \{c : D_j \nrightarrow D_k \mid \exists\vec{\sigma} \in S_{j,k}, \exists\vec{\sigma}' \in S_{i,j} : \hat{\eta}(\vec{\sigma}' \,;\, \vec{\sigma}) = c \wedge \\
&\quad \hat{\eta}(\vec{\sigma}' \,;\, \vec{\sigma}) \models [\![\psi_1]\!] \wedge \hat{\eta}(\vec{\sigma}') \models [\![\psi_2]\!]\} \\
&= \{c : D_j \nrightarrow D_k \mid \exists c' : D_i \nrightarrow D_j : c' \,;\, c \models [\![\psi_1]\!] \wedge c' \models [\![\psi_2]\!]\} \\
&= [\![\varphi]\!]
\end{aligned}$$

This completes the induction.

□

In the following, we investigate the properties of LCL. The focus will be on properties which can later on be used for the applications we have in mind, i. e. verification of dynamically evolving systems (see Chapter 7 for more details). Since the construction of an automaton accepting the language described by a property is usually not feasible we try to avoid the construction of the complete automaton. Instead of that our approach is to transform the property to an equivalent statement which can be checked more easily.

Above we have already seen examples of such transformations. By Proposition 6.51 we can avoid to construct the automata for the existential or the universal quantification by transforming the statement to one depending only on a shift of a single cospan. As seen before, the corresponding automaton can be constructed very easily.

Proposition 6.53. Let $c\colon D_i \nrightarrow D_j$ and $c'\colon D_j \nrightarrow D_k$ be cospans, $\psi\colon i \to j$ and $\varphi\colon j \to k$ be formulas. Then

$$(c \models \psi \wedge c\,;\, c' \models \forall\psi\colon \varphi) \implies c' \models \varphi$$

Proof. Let $c\colon D_i \nrightarrow D_j$ and $c'\colon D_j \nrightarrow D_k$ be cospans, such that $c \models \psi$ and $c\,;\, c' \models \forall\varphi\colon \psi$. We assume that $c' \not\models \varphi$. By $c\,;\, c' \models \forall\psi\colon \varphi$ we have that there exists no pair of cospans $d\colon D_i \nrightarrow D_j$ and $d'\colon D_j \nrightarrow D_k$ such that $c\,;\, c' = d\,;\, d'$, $d \models \psi$ and $d' \models \neg\varphi$. Therefore, it must hold (if we choose $c = d$ and $c' = d'$) that $c' \models \varphi$. But this is a contradiction to our assumption. □

Proposition 6.54. Let $c\colon D_i \nrightarrow D_j$ and $c'\colon D_j \nrightarrow D_k$ be cospans, $\psi\colon i \to j$ and $\varphi\colon j \to k$ be formulas. Then

$$(c \models \psi \wedge c' \models \varphi{\downarrow_\psi^\forall}) \implies c\,;\, c' \models \varphi$$

Proof. Let $c\colon D_i \nrightarrow D_j$ and $c'\colon D_j \nrightarrow D_k$ be cospans, such that $c \models \psi$ and $c' \models \varphi{\downarrow_\psi^\forall}$. We assume that $c\,;\, c' \not\models \varphi$. Since $c' \models \varphi{\downarrow_\psi^\forall}$ holds if and only if for all cospans $d\colon D_i \nrightarrow D_j$ satisfying $d \models \psi$ it holds that $d\,;\, c' \models \varphi$, we can conclude from $c \models \psi$ that $c\,;\, c' \models \varphi$. But this is a contradiction to our assumption. □

In the following we introduce the notion of Galois connections. Our aim is to use the theory of Galois connections in order to avoid impractical computations on graph automata. How this can be achieved will be explained below.

Definition 6.55 (Galois Connection)**.** Let $\langle L, \leq_L\rangle$ and $\langle M, \leq_M\rangle$ be two partially ordered sets. A *Galois connection* $\langle\alpha, \gamma\rangle$ (between L and M) consists of two functions $\alpha\colon L \to M$ and $\gamma\colon M \to L$, called *abstraction* and *concretion*, which satisfy the following conditions:

- α and γ are monotone, i. e.

  $$\forall\ell_1, \ell_2 \in L\colon \ell_1 \leq_L \ell_2 \implies \alpha(\ell_1) \leq_M \alpha(\ell_2)$$

  and

  $$\forall m_1, m_2 \in M\colon m \leq_M m \implies \gamma(m_1) \leq_L \gamma(m_2)$$

- $\alpha\,;\,\gamma$ is extensive and $\gamma\,;\,\alpha$ is reductive, i. e.

  $$\forall\ell \in L\colon \ell \leq_L \gamma(\alpha(\ell)) \quad \text{and} \quad \forall m \in M\colon \alpha(\gamma(m)) \leq_M m$$

We can immediately obtain the following properties:

Proposition 6.56 (Properties of Galois Connections)**.** Let $\langle \alpha, \gamma \rangle$ be a Galois connection between two partially ordered sets L and M. Then the following conditions hold:

(i) $\alpha(\ell) \leq_M m \iff \ell \leq_L \gamma(m)$

(ii) $\gamma(m) = \bigsqcup\{\ell \mid \alpha(\ell) \leq_M m\}$

(iii) $\alpha(\ell) = \bigsqcap\{\alpha(\ell) \mid \ell \leq_L \gamma(m)\}$

(iv) $\alpha\left(\bigsqcup L'\right) = \bigsqcup\{\alpha(\ell) \mid \ell \in L'\}$, for $L' \subseteq L$

(v) $\gamma\left(\bigsqcap M'\right) = \bigsqcap\{\gamma(m) \mid m \in M'\}$, for $M' \subseteq M$

Proof. See Appendix A.2. □

In order to make use of the laws given above, we introduce two concrete Galois connections. For this we overload the entailment relation in the following sense: To obtain a partially ordered set, we interpret the entailment relation $\models$ on equivalence classes of LCL-formulas. Note that this yields a lattice where $\vee$ is the supremum and $\wedge$ is the infimum of the lattice.

The laws presented above are very useful, especially if we take a further look at the quantification and the shift defined above. The quantification, $\mathcal{Q}\psi\colon \varphi$, and the shift $\varphi{\downarrow_\psi^{\mathcal{Q}}}$, for $\mathcal{Q} \in \{\forall, \exists\}$, can be seen as functions with two parameters, ψ and φ. Therefore, if we fix the formula ψ we obtain two functions (in one parameter) which can be used as abstractions and concretions forming a Galois connection as seen below. Hence, we can use the above laws for Galois connections to transform (universal) quantifications and (universal) shifts. This helps to avoid the costly determinization of the huge underlying graph automata.

Proposition 6.57. Let $n \in \mathbb{N}$, $\psi\colon i \to k$ be a formula and $i, j, k \leq n$. The functions

$$\alpha_\psi^\exists\colon \Phi_{k,j} \to \Phi_{i,j} \qquad \gamma_\psi^\forall\colon \Phi_{i,j} \to \Phi_{k,j}$$
$$\alpha_\psi^\exists(\varphi) = \exists\psi\colon \varphi \qquad \gamma_\psi^\forall(\varphi') = \varphi'{\downarrow_\psi^\forall}$$

are a Galois connection between $\langle \Phi_{k,j}, \models \rangle$ and $\langle \Phi_{i,j}, \models \rangle$.

Proof. See Appendix A.2. □

Proposition 6.58. Let $n \in \mathbb{N}$, $\psi\colon i \to j$ be a formula and $i, j, k \leq n$. The functions

$$\alpha_\psi^\forall\colon \Phi_{j,k} \to \Phi_{i,k} \qquad \gamma_\psi^\exists\colon \Phi_{i,k} \to \Phi_{j,k}$$
$$\alpha_\psi^\forall(\varphi) = \forall\psi\colon \varphi \qquad \gamma_\psi^\exists(\varphi') = \varphi'{\downarrow_\psi^\exists}$$

are a Galois connection between $\langle \Phi_{j,k}, \models \rangle$ and $\langle \Phi_{i,k}, \models \rangle$.

Proof. See Appendix A.2. □

Another application of the Lcl-logic is the automatic generation of invariants. For this purpose we can use the machinery presented above to compute the weakest pre-condition and the strongest post-condition for a given set of transformation rules.

Definition 6.59 (Hoare Triple, (Weakest) Pre-Condition, (Strongest) Post-Condition)**.** Let $\rho = \langle \ell, r \rangle$ be a transformation rule and let $\varphi, \psi \colon 0 \to j$ be formulas. The triple $\langle \varphi, \rho, \psi \rangle$ is a *Hoare Triple*, also written as $\{\varphi\}\ \rho\ \{\psi\}$, if for all graphs G and H such that $[G] \models \varphi$ and $G \overset{\rho}{\Longrightarrow} H$ it holds that $[H] \models \psi$.

For a Hoare triple $\{\varphi\}\ \rho\ \{\psi\}$, φ is called a *pre-condition* for ρ and ψ and ψ is called a *post-condition* for φ and ρ.

A formula φ is the *weakest pre-condition* for ρ and ψ, written as $wp\,(\rho, \psi)$, if it is a pre-condition for ρ and ψ and for every other pre-condition φ' for ρ and ψ it holds that $\varphi' \models \varphi$. A formula ψ is the *strongest post-condition* for φ and ρ, written as $sp\,(\varphi, \rho)$, if it is a post-condition for φ and ρ and for every other post-condition ψ' for φ and ρ it holds that $\psi \models \psi'$.

To compute the weakest pre-condition (or strongest post-condition respectively), we use a method similar to that presented by Bruggink, Cauderlier, Hülsbusch and König [31]. The basic idea is as follows: The weakest pre-condition of a transformation rule $\rho = \langle \ell, r \rangle$ and a formula ψ has to guarantee that for every occurrence of the left-hand side the part of the post-condition ψ which is not affected by the right-hand side is satisfied, i. e. every left-hand side must satisfy the formula $\psi{\downarrow_r}$. The strongest post-condition of a formula φ and a transformation rule $\rho = \langle \ell, r \rangle$ has to guarantee that there exists an occurrence of the right-hand side such that this right-hand side satisfies the part of the pre-condition which is not affected by the left-hand side, i. e. some right-hand side must satisfy the formula $\varphi{\downarrow_\ell}$.

This can be spelled out by the following proposition:

Proposition 6.60. Let $\rho = \langle \ell, r \rangle$ be a transformation rule and let $\varphi, \psi \colon 0 \to j$ be formulas. It holds that

- $wp\,(\rho, \psi) = \forall \ell \colon (\psi{\downarrow_r})$
- $sp\,(\varphi, \rho) = \exists r \colon (\varphi{\downarrow_\ell})$

Proof.

- We show that $wp\,(\rho, \psi) = \forall \ell \colon (\psi{\downarrow_r})$ holds in two steps:
  1. We prove that $\forall \ell \colon (\psi{\downarrow_r})$ is a pre-condition of ρ and ψ. Let G and H be graphs such that $[G] \models \forall \ell \colon (\psi{\downarrow_r})$ and $G \overset{\rho}{\Longrightarrow} H$. Hence, we have that $[G] = \ell\ ;\ c$ and $[H] = r\ ;\ c$ for some cospan c. By this we can follow that $c \models \psi{\downarrow_r}$. By proposition 6.46 we then have $[H] = r\ ;\ c \models \psi$. Therefore, $\forall \ell \colon (\psi{\downarrow_r})$ is a pre-condition of ρ and ψ.

2. We show that $\forall \ell \colon (\psi{\downarrow_r})$ is the weakest pre-condition. Let φ' be any pre-condition for ρ and ψ and let G be a graph such that $[G] \models \varphi'$, which implies that for every graph H with $G \overset{\rho}{\Longrightarrow} H$ we have that $[H] \models \psi$. This means that for every cospan c such that $[G] = \ell \,;\, c$ we have that $[H] = r \,;\, c \models \psi$ holds which is equivalent to $c \models \psi{\downarrow_r}$ by proposition 6.46. This implies that $[G] \models \forall \ell \colon (\psi{\downarrow_r})$ and hence $\forall \ell \colon (\psi{\downarrow_r})$ is the weakest pre-condition.

- We show that $sp\,(\varphi, \rho) = \exists r \colon (\varphi{\downarrow_\ell})$ holds in two steps:

  1. We prove that $\exists r \colon (\varphi{\downarrow_\ell})$ is a post-condition of φ and ρ. Let G and H be graphs such that $[G] \models \varphi$ and $G \overset{\rho}{\Longrightarrow} H$. Hence, we have that $[G] = \ell \,;\, c$ and $[H] = r \,;\, c$ for some cospan c. By proposition 6.46 we immediately obtain that $c \models \varphi{\downarrow_\ell}$ and since $[H] = r \,;\, c$ we can conclude that $[H] \models \exists r \colon (\varphi{\downarrow_\ell})$. Therefore, we have that $\exists r \colon (\varphi{\downarrow_\ell})$ is a post-condition of φ and ρ.
  2. We show that $\exists r \colon (\varphi{\downarrow_\ell})$ is the strongest post-condition. Let ψ' be any post-condition for φ and ρ. Let H be a graph such that $[H] \models \exists r \colon (\varphi{\downarrow_\ell})$. This implies that there exists a cospan c such that $[H] = r \,;\, c$ and $c \models \varphi{\downarrow_\ell}$. By proposition 6.46 we have that $\ell \,;\, c \models \varphi$ and if we set $[G] = \ell \,;\, c$, we have that $G \overset{\rho}{\Longrightarrow} H$. Since ψ' is any post-condition, this implies $[H] \models \psi'$. Hence, we obtain $\exists r \colon (\varphi{\downarrow_\ell}) \models \psi'$. Therefore, $\exists r \colon (\varphi{\downarrow_\ell})$ is the strongest post-condition.

□

Now we can check whether a given set of graphs satisfies the weakest pre-condition or strongest post-condition respectively. Such a method could serve as the basis for automatic invariant generation.

Note that we have some potential for optimizations here. Let $\rho = \langle \ell, r \rangle$ be a transformation rule and let $\varphi, \psi \colon 0 \to j$ be formulas. By Proposition 6.51 we can optimize the check whether φ satisfies the weakest pre-condition of ρ and ψ and the check whether ψ satisfies the strongest post-condition of φ and ρ in the following way: We have that

$$\varphi \models wp\,(\rho, \psi) \iff \varphi{\downarrow_\ell} \models \psi{\downarrow_r} \iff sp\,(\varphi, \rho) \models \psi.$$

Hence, we can simplify both checks by computing the "shift automata" (of $\varphi{\downarrow_\ell}$ and $\psi{\downarrow_r}$) instead of computing the more costly "quantification automata".

It remains future work to investigate the expressiveness of LCL and the connection to other well-known logics. But we assume that LCL is equivalent to first-order logic if the set of predicates is instantiated only to singletons. However, since we can instantiate predicates with languages which are not MSOGL-definable (but recognizable), it is possible to express properties which cannot be defined in MSOGL.

6.3. Conclusion

In this chapter we have introduced the notion of recognizable graph languages in terms of three different automaton models. The first one has been the categorical notion of automaton functors which accept the class of recognizable graph languages introduced by Courcelle (when instantiated to the category of (cospans of) graphs). Since we are

interested in a more automaton-theoretic view due to our applications (see Chapters 7 and 8) we have also introduced the notions of consistent tree automata and graph automata which are closer to classical tree automata and finite automata respectively. But we have shown that in the case of consistent tree automata we obtain the same notion of recognizability. In the case of graph automata we have only considered languages of graphs with a bounded pathwidth. Hence, we have shown that this class of languages coincides with the class of languages accepted by bounded automaton functors (which also accept only graphs with a bounded pathwidth, instead of graphs with arbitrary path-/treewidth accepted by unbounded automaton functors). We have introduced this bound in order to get finitely representable automata, which we will use in the next chapters as basis for our tool-suite Raven. If we dropped this bound (and allowed graphs with arbitrary pathwidth) the corresponding graph automata would consist of an infinite number of finite state sets. Hence, the graph automata become non-implementable.

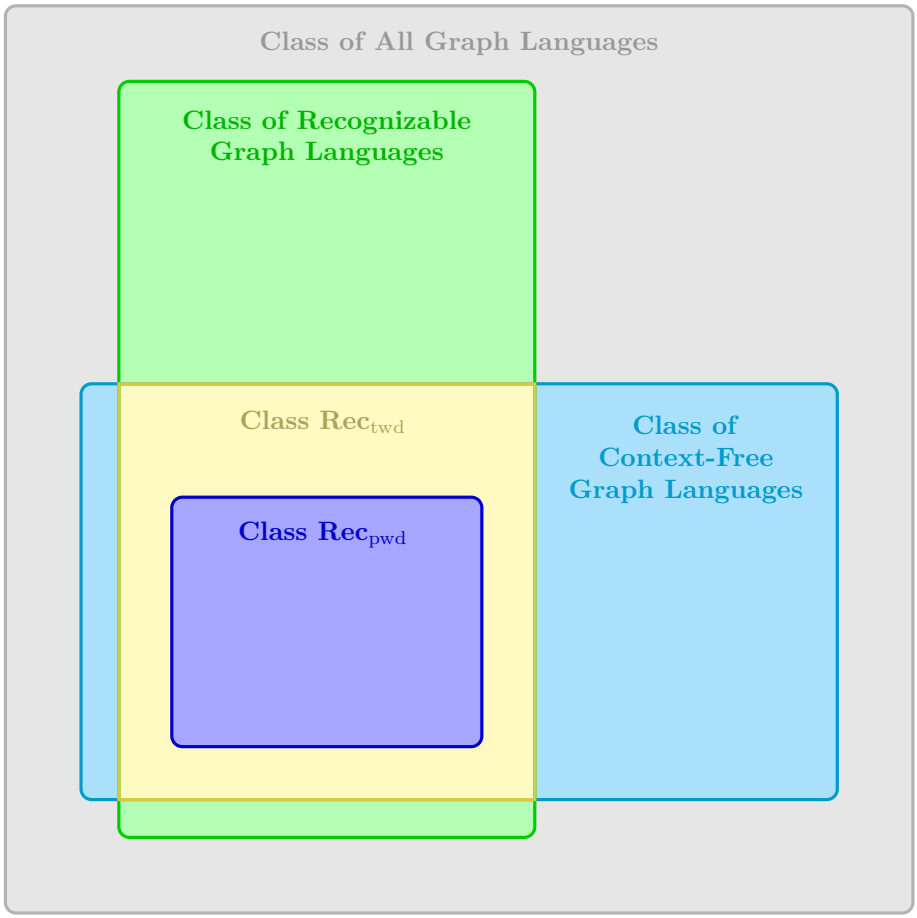


Figure 6.2.: Language Hierarchy

Note that despite the situation for word languages the class of recognizable graph languages and the class of context-free graph languages are incomparable (see [44] for a proof). But due to a result by Courcelle [48, Cor. 4.38] we have that every recognizable graph language with bounded treewidth is also context-free. However, currently we have only considered bounded graph automata accepting graphs with bounded pathwidth. But we could also introduce bounded consistent tree automata, which accept languages of graphs with a bounded treewidth. It is obvious that the class of all recognizable graph languages with bounded treewidth ($\mathbf{Rec}_{\text{twd}}$) is a super class of the class of all recognizable graph languages with bounded pathwidth ($\mathbf{Rec}_{\text{pwd}}$), since $twd(G) \leq pwd(G)$ for all graphs G. Altogether, we have the language hierarchy depicted in Figure 6.2. However, the class $\mathbf{Rec}_{\text{pwd}}$ already contains many interesting graph languages.

Beside these different automata models we have taken a look at two different logics for graphs: on the one hand the monadic second-order logic on graphs and on the other hand the linear cospan logic, which we have introduced in this thesis. The former one is of great interest for us for two reasons. First, it is well-known that every MSOGL-formula induces a recognizable graph language. Second, the famous Courcelle's theorem states that every MSOGL-definable graph language can be decided in linear time for graphs of bounded treewidth. The drawback of this latter result is that the construction of an automaton for the corresponding MSOGL-formula is very costly in practice. Hence, we have considered the linear cospan logic which depends on the usual logical operations (i. e. conjunction, negation, existential quantification) plus a shift-operation and a number of pre-defined predicates which describe some basic graph properties, e. g. k-colorability, dominating sets, etc. The shift-operation is very useful for different applications, such as invariant checking and the computation of weakest pre-conditions (strongest post-conditions). In case of the invariant checking problem we have seen that this problem can be reduced to a language inclusion problem of two automata obtained by shifting the left-hand side and right-hand side respectively over the original automaton. For the other application we have presented above, Hoare logic, we have seen that the shift-operation is quite useful for the computation of the weakest pre-condition (strongest post-condition).

In the next chapter we will explain how graph automata can be efficiently represented by binary decision diagrams. Furthermore, we will present different algorithms to solve the language inclusion problem. These algorithms will then be used to solve the invariant checking problem as stated above.

"In theory, there is no difference between theory and practice. But, in practice, there is."

Johannes L. A. van de Snepscheut (1953 – 1994)

7

Symbolically Represented Graph Automata

In general an automaton functor consists of infinitely many finite state sets, since graphs with arbitrary large interface size have to be considered. But if only recognizable graph languages of bounded interface size are permitted, it is possible to use graph automata of bounded size, since the size of the interfaces of considered graphs is bounded. However, the graph automaton might still be very large. In general the sizes of the state sets will be exponential in the maximum interface size.

In order to tackle this important problem and to obtain an implementable representation of graph automata we will investigate how binary decision diagrams (as introduced in Chapter 4) can be used to reduce the size of the representation. Furthermore, we will take a look at three different algorithms solving the language inclusion problem for non-deterministic finite automata, which have been published in the last years, and we will show how these algorithms can be adapted to symbolically represented automata.

Parts of this chapter have already been published in [12].

7.1. Representation of Graph Automata using Binary Decision Diagrams

Even for rather small interface sizes graph automata become very large very quickly. This is due to the fact that the sizes of the state sets grow exponentially (or worse) in the maximum permitted interface size. For instance, the number of states and the number of Bdd nodes used to encode the state set for the graph automaton accepting all graphs of bounded interface size which are 4-colorable (cf. Example 6.33) are depicted in Figure 7.1 for various maximum interface sizes. Note that the scale on the x-axis is linear and on the y-axis the scale is logarithmic, i. e. the number of states

grows exponentially and the number of Bdd nodes used to encode the state set grows sub-exponentially.

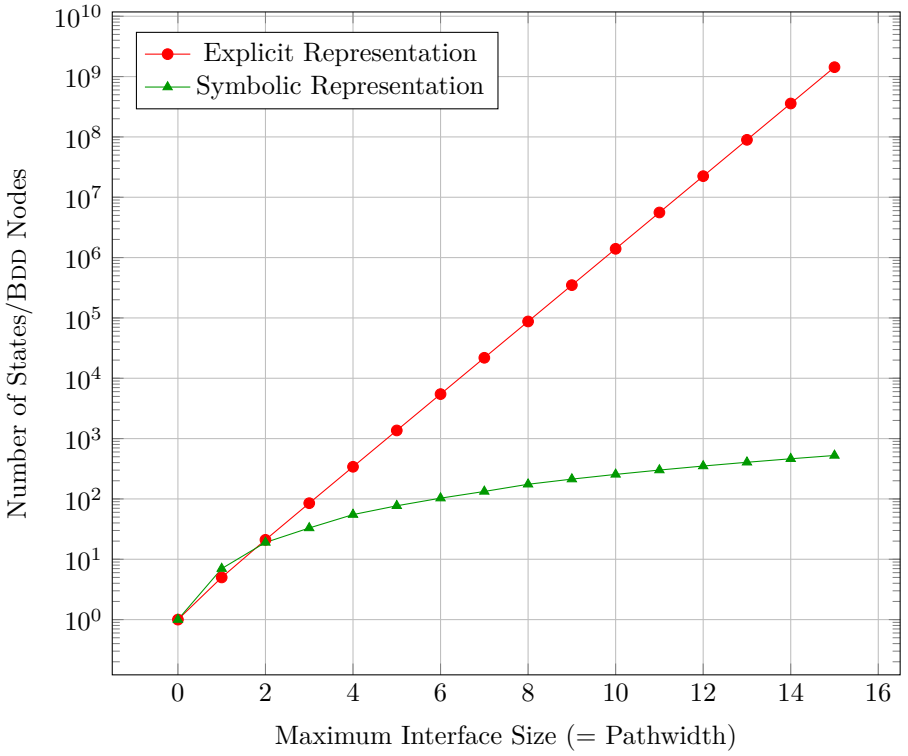


Figure 7.1.: Number of states and number of Bdd nodes used for the 4-colorability graph automaton (Note: y-axis in logarithmic scale)

As one can see, if the maximum interface size is five the automaton contains over 1 300 states, for the maximum interface size of ten the number of states is about 1 400 000 states and if the maximum interface size is fifteen the number of states grows up to more than 1 400 000 000 states. In addition to the several state sets of the graph automaton there is more information which must be stored. On the one hand this includes the sets of initial and final states and on the other hand the transition relations for the different letters of the input alphabet.

In order to represent graph automata efficiently we encode the different state sets and the various transition relations of the graph automata as bit strings which can then be represented by Bdds (cf. Chapter 4 for more information about Bdds). In contrast to the size of the state sets the number of Bdd nodes to encode the state sets of the different graph automata is rather small. For the example mentioned above, the corresponding Bdd contains 77 nodes for the maximum interface size of five, for maximum interface size of ten 255 nodes are needed and for the maximum interface size of fifteen the Bdd consists of 525 nodes. Hence, the rate of growth of the Bdds is several dimensions smaller than the rate of growth of the explicit state space representation.

For the rest of this section we present and explain how graph automata can be encoded by the means of BDDs. The state encoding for every graph automaton has to take care of the following information:

- the index of the state set to which the state belongs (the index corresponds directly to the size of the outer interface of the cospan decomposition seen so far) and
- the specific information of the different graph automata

In the following, we use the k-colorability graph automaton as an example to explain how the encoding works. In this case the specific information encoded in the second part of the state encoding includes the color of each node which is accessible by the outer interface[1] of the cospan decomposition seen so far.

As already mentioned in Chaper 4 the size of a BDD depends highly on the ordering of the BDD nodes. Therefore, it is crucial to find good orderings of the bits holding the several pieces of information to take advantage of the encoding of state sets as BDDs. But before we investigate the differences between a good and a bad encoding, we first start with an explanation how to obtain an encoding. For the k-colorability example we have experimented with different orderings and found the following to be the best:

Let n be the maximum interface size and k the number of colors. Furthermore, let $m = \lceil \log_2 n \rceil$ be the number of bits required to store the maximum interface size, and $\ell = \lceil \log_2(k+1) \rceil$ the number of bits to store one color. The additional "color" is used to indicate uncolored or unused nodes. This is due to the fact that all bit strings encoding the several states have the same length, independent of the number of interface nodes which are currently accessible. Therefore, we use this "uncolor" to indicate which nodes are currently not in the interface. In the following we distinguish between three different types of nodes: the first type are the BDD nodes, the second type are the graph nodes, i. e. the nodes occurring in the middle graph of the input cospan, the third and last type are the interface nodes, i. e. the nodes occurring in the (outer) interface of the input cospan. Furthermore, we assume that the color, which indicates that the node is "uncolored", is $\vec{0}$. A state is then encoded by the bit sequence

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} = b_1\ldots b_m(c_{1,1}\ldots c_{1,\ell})\ldots(c_{n,1}\ldots c_{n,\ell}),$$

where $\vec{b} = b_1\ldots b_m$ encodes the current interface size as a binary number and $\vec{c_j} = (c_{j,1}\ldots c_{j,\ell})$ (for $1 \leq j \leq n$) represents the color of the j-th interface node. In addition to represent relations of states we interleave the BDD nodes of the corresponding states. Note if the current interface size is i, that the bit strings $\vec{c}_{i+1}$ to $\vec{c}_n$ are set to $\vec{0}$ to indicate that the respective nodes are not in the current interface.

The set of initial states of the n-bounded $\langle s,t\rangle$-graph automaton accepting all graphs which are k-colorable is defined by the following propositional formula:

$$(\vec{b} = s) \wedge \bigwedge_{x=1}^{s} \bigvee_{y=1}^{k} (\vec{c}_x = y) \wedge \bigwedge_{x=s+1}^{n} (\vec{c}_x = \vec{0}).$$

[1] In the following, we will also simply write "... node in the inner/outer interface ..." rather than "... node accessible by the inner/outer interface ...)

The formula expresses that the current interface size corresponds to the inner interface size of all cospans accepted by the graph automaton and that the first s nodes which are accessible by the inner interface are colored valid, i. e. each node has a color from the set $\{1, \dots k\}$ and that the other $(n-s) \cdot \ell$ bits of the encoding are set to 0, since these bits encode nodes which are currently not accessible by the interface. The formulas for the set of final states and the set of all states can be defined analogously.

The next step is to encode the transition functions of the graph automaton. As described in Section 6.1.5 each graph automaton consists of a family of transition functions – one transition function which maps a predecessor state and an input letter to a set of successor states for each pair of inner and outer interface size (of the corresponding atomic cospan). Instead of representing each transition function as a Bdd we first fix an input letter and then define a propositional formula describing all transitions from a fixed state set of the graph automaton. Then the formula is transformed into a Bdd. We present the formulas f_{vertex^i} and $f_{connect^i_\diamond}$ which encode all *vertex*-transitions and all $connect_\diamond$-transitions, respectively, from the i-th state set as an example. Since the transition functions are relations we interleave the bits of the domain and the codomain as described in Chapter 4. To distinguish between the bits for the current state and the bits for the successor state we indicate the successor state encoding by $\vec{b'}\vec{c'_1} \dots \vec{c'_n}$.

The formula f_{vertex^i} consists of four parts:

$$f_1 := (i < n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i+1) \qquad f_3 := \bigwedge_{\substack{j=1 \\ j \neq i+1}}^{n} (\vec{c}_j = \vec{c'}_j)$$

$$f_2 := \bigwedge_{j=i+2}^{n} (\vec{c}_j = 0) \qquad f_4 := (\vec{c}_{i+1} = 0) \wedge (1 \leq \vec{c'}_{i+1} \leq k)$$

The subformula f_1 expresses that the size of the current interface is less than the maximum interface size, otherwise no new node can be added, and that the interface size of the current state is i, whereas the interface size of the successor state is $i+1$. The subformula f_2 expresses that the nodes of the encoding which do not belong to the current interface, that is the last $n-i+1$ nodes in the encoding, have not been colored. Next, f_3 expresses that all nodes have the same color in the source and the target state – except for the $(i+1)$-th node which is currently added. Finally, f_4 expresses that the node which is to be added does not exist in the interface of the current state, which is encoded by 0-bits, and the new node has a valid color in the successor state, i. e. the color of the node is between 1 and k. Now, we take $f_{vertex} := f_1 \wedge f_2 \wedge f_3 \wedge f_4$, that is, a transition $q \xrightarrow{vertex^i} q'$ exists if and only if the above four conditions hold.

The formula $f_{connect_\diamond}$ also consists of four parts. We set $p = i - ar(\diamond) + 1$ indicating the index of the first node attached to the new edge:

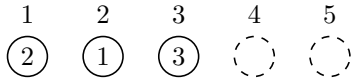
$$f_5 := (ar(\diamond) \leq i) \wedge (\vec{b} = i) \wedge (\vec{b} = \vec{b'}) \qquad f_7 := \bigwedge_{j=1}^{n} (\vec{c}_j = \vec{c'}_j)$$

$$f_6 := \bigwedge_{j=i+1}^{n} (\vec{c}_j = 0) \qquad f_8 := \bigwedge_{j=p}^{i} \bigwedge_{j'=p}^{i} (j \neq j') \rightarrow (\vec{c}_j \neq \vec{c}_{j'})$$

The subformula f_5 expresses that the arity of the added edge is less than or equal to the current interface and that the interface size of both the current state and the successor state is i. The subformulas f_6 and f_7 are similar to the subformulas f_2 and f_3. Next, f_8 expresses that the nodes which are connected by the new edge have different colors. Let $f_{connect_\diamond} := f_5 \wedge f_6 \wedge f_7 \wedge f_8$, that is, a transition $q \xrightarrow{-connect^i_\diamond} q'$ exists if and only if the four corresponding conditions hold.

Note that each BDD generated by the formulas presented above represents a whole set of states or a whole relation respectively, rather than a single state or pair of states. Hence we do not deal with single states but perform the different computations on the whole state set. On the one hand this is very efficient because we do not have to process one state after the other. On the other hand we have to adapt the techniques which are designed for explicitly represented states to our setting of symbolically represented automata (see the next section of this chapter for more details).

Example 7.1. *We consider the* 3*-colorability graph automaton with a maximum interface size of* 5*. The size of the state encoding is* $3 + (5 \cdot 2) = 13$ *bits. Consider the state* q *depicted in Figure 7.2 (on the left): We have three nodes in the current interface, colored with color* 2*,* 1 *and* 3*, respectively. The bit string which encodes this state is given in Figure 7.2 on the right. Note that the least significant bit is left.*



b_1	b_2	b_3	$\vec{c_1}$	$\vec{c_2}$	$\vec{c_3}$	$\vec{c_4}$	$\vec{c_5}$
1	1	0	01	10	11	00	00

Figure 7.2.: State q and its representation as bit string

The BDD *which encodes the vertex*^{3}*-transition of the* 3*-colorability graph automaton is shown in Figure 7.3. Since the nodes which represent the predecessor and successor states are interleaved, the nodes with even numbers encode the predessor states and the odd numbered nodes encode the successor states.*

The BDD *nodes labeled with* 0 *to* 5 *guarantee that the predecessor state has an interface size of* 3 *(binary encoded as* 110*, with least significant bit left) and that the successor state's interface size is* 4 *(or* 001 *as a binary string). The* BDD *nodes labeled with* 6 *to* 9 *encode that the color of the first interface node is not changed by the vertex*^{3}*-transition. This condition is sufficient to guarantee that the first interface has a valid color, since every initial state guarantees that the interface nodes are colored with a valid color. The* BDD *nodes* 10 *to* 13 *and* 14 *to* 17 *encode the analogous condition for the second and the third interface node. Since the inner interface size of the vertex*^{3}*-cospan is* 3 *the* BDD *nodes* 18 *and* 20 *have to be set to* 0*, which indicates that there is no fourth interface node in the predeseccor state. But the fourth interface node is added by the vertex*^{3}*-operation. Therefore, it needs a valid color in the successor state, i. e. the* BDD *nodes* 19 *and* 21 *must be either* 1*,* 2 *or* 3 *(binary encoded* 10*,* 01 *or* 11*). The fifth interface node occurs neither in the predecessor nor in the successor node. Hence the* BDD *nodes* 22 *to* 25 *are set to* 0*.*

Suppose that the graph automaton is currently in state q*, and that the next letter is vertex*^{3}*. Then there exist three different states,* q'*,* q''*,* q'''*, which are reachable from* q

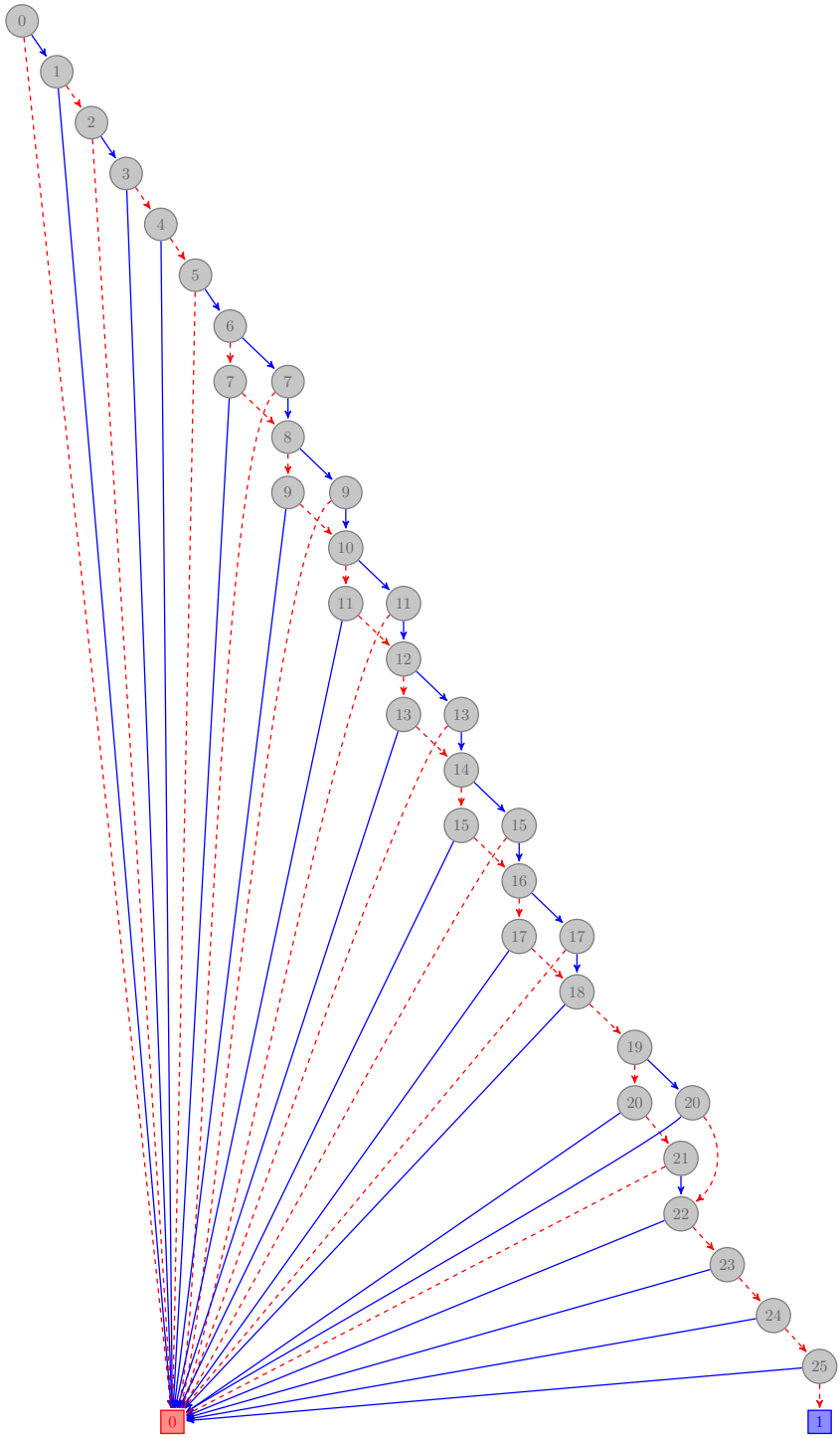


Figure 7.3.: The BDD encoding of the $vertex^3$-transition

by the vertex^3-transition. These states are encoded by the following bit strings:

$$\begin{array}{rcccccccc} & b'_1 & b'_2 & b'_3 & \vec{c'_1} & \vec{c'_2} & \vec{c'_3} & \vec{c'_4} & \vec{c'_5} \\ q' : & 0 & 0 & 1 & 01 & 10 & 11 & 10 & 00 \\ q'' : & 0 & 0 & 1 & 01 & 10 & 11 & 01 & 00 \\ q''' : & 0 & 0 & 1 & 01 & 10 & 11 & 11 & 00 \end{array}$$

Now, we interleave, for example, the bits encoding the states q and q' and obtain the following bit string:

$$10\ 10\ 01\ 00\ 11\ 11\ 00\ 11\ 11\ 01\ 00\ 00\ 00.$$

If we traverse the Bdd *in Figure 7.3 according to this bit string we end up in the terminal node labeled with* 1 *which indicates that $q \xrightarrow{-vertex^3\rightarrow} q'$ is a valid transition. For the other two states, q'' and q''', we get the same result, therefore $q \xrightarrow{-vertex^3\rightarrow} q''$ and $q \xrightarrow{-vertex^3\rightarrow} q'''$ are also valid transitions.*

Suppose on the other hand that we want to check whether the state $\overline{q'}$ encoded by

$$\begin{array}{rcccccccc} & b'_1 & b'_2 & b'_3 & \vec{c'_1} & \vec{c'_2} & \vec{c'_3} & \vec{c'_4} & \vec{c'_5} \\ \overline{q'} : & 0 & 0 & 1 & 11 & 10 & 11 & 10 & 00 \end{array}$$

is a valid vertex^3-successor state of q. The bit string encoding the transition is as follows:

$$10\ 10\ 01\ 01\ 11\ 11\ 00\ 11\ 11\ 01\ 00\ 00\ 00.$$

Again, we traverse the Bdd *for the vertex^3-transition, but this time we reach the terminal labeled with* 0 *since the color of the first node accessible by the interface is different in the states q and $\overline{q'}$.*

Next, we turn to the letter connect$^3_\diamond$, where $\diamond$ is a label of arity 2*. In Figure 7.4 we present the* Bdd *for the connect$^3_\diamond$-transitions of the* 3*-colorability graph automaton. Again, the nodes representing the predecessor and successor states are interleaved, such that the even numbered* Bdd *nodes encode the predessor states and the odd numbered* Bdd *nodes encode the successor states.*

The information encoded by the different Bdd *nodes is the same as described for the* Bdd *in Figure 7.3. The* Bdd *nodes labeled with* 0 *to* 5 *ensure that the interface size for the predecessor and the successor state is* 3 *(or* 110 *as a bit string). The* Bdd *nodes labeled* 6 *to* 9 *guarantee that the color of the first graph node is not changed by the connect$^3_\diamond$-transitions. The nodes in the* Bdd *labeled* 10 *to* 17 *are used to distinguish three cases, depending on the color of the second interface node. The three cases are: The graph node has the color* 1 *(bit string* 10*), the color* 2 *(bit string* 01*) or the color* 3 *(bit string* 11*). In each case three conditions must be fulfilled:*

- *the color of the second interface node must not be changed by the transition,*
- *the color of the third interface node must not be changed by the transition, and*
- *the color of the second interface node must be different from the color of the third interface node.*

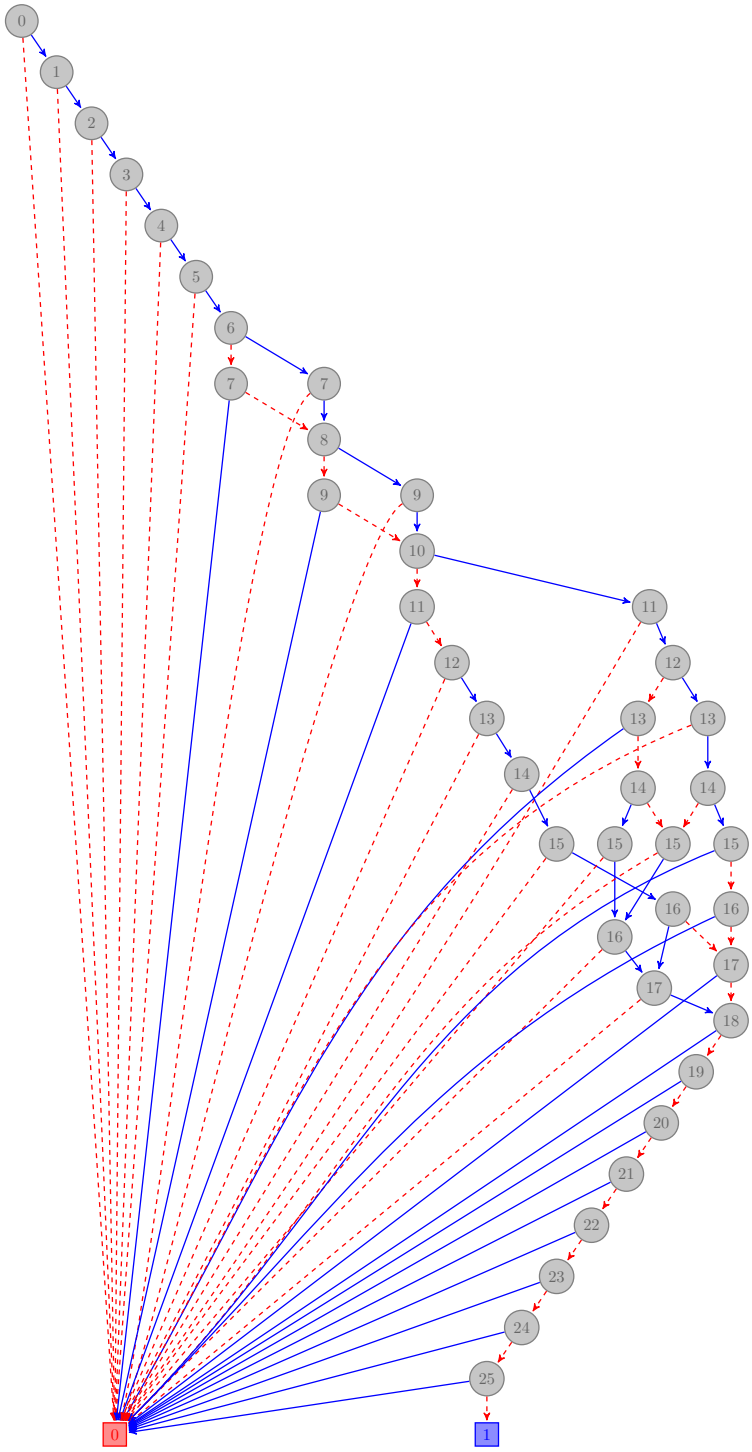


Figure 7.4.: The BDD encoding of the $connect^3_\diamond$-transition

Since there are three distinguished cases it is sufficient for the third condition to check that, if the color of the second interface node is 1*, for instance, that the color of the third interface node is either* 2 *or* 3*. The same must hold for the cases where the color of the second interface node is* 2 *or* 3*, respectively.*

Now, suppose that the 3*-colorability graph automaton is currently in state* q *and that the next letter is* $\mathit{connect}^3_\diamond$*. Since the* $\mathit{connect}^3_\diamond$*-transition does not change any bit, but only checks that the coloring is still valid, the successor state is also* q*. Therefore, the bit string*

$$11\ 11\ 00\ 00\ 11\ 11\ 00\ 11\ 11\ 00\ 00\ 00\ 00$$

leads to the terminal labeled with 1 *of the* Bdd *shown in Figure 7.4.*

Finally, we suppose that the graph automaton is currently in the state $\overline{q}$ *which is given by the following bit string:*

$$\begin{array}{rccccccc} & b_1 & b_2 & b_3 & \vec{c_1} & \vec{c_2} & \vec{c_3} & \vec{c_4}\ \vec{c_5} \\ \overline{q}: & 1 & 1 & 0 & 11 & 10 & 10 & 00\ 00 \end{array}$$

and that the next letter is again $\mathit{connect}^3_\diamond$*. This time the second and the third interface node have the same color. Hence, there exists no* $\mathit{connect}^3_\diamond$*-transition from* $\overline{q}$ *and thus, the bit string*

$$11\ 11\ 00\ 11\ 11\ 11\ 00\ 11\ 00\ 00\ 00\ 00\ 00$$

leads to the terminal labeled 0*.*

Apart from a graph automaton which accepts k-colorable graphs, we also implemented graph automata for various other graph problems for arbitrary inner and outer interface sizes. This list includes:

- *k-Dominating Set:* Graph automata which accept a cospan if and only if the middle graph G of the cospan contains a dominating set of size at most k. That is a set D of nodes of G with size at most k such that each node of G is either in D or adjacent to a node in D.
- *k-Vertex Cover:* Graph automata accepting a cospan if and only if the middle graph G of the cospan has a vertex cover of size at most k. That is there exists a set C of nodes of G with size at most k such that each edge is incident to at least one node of C. See also Example 6.5.
- *Edge/Node-Counting:* Graph automata which accept a cospan if and only if number of edges with a specific label or nodes of the middle graph of the input cospan are equal to a given number (modulo a fixed divisor).
- *Maximum/Minimum Edge/Node:* Graph automata accepting only cospans whose middle graph has a particular maximum (minimum) number of edges or nodes.
- *No Isolated Nodes:* Graph automata which accept only those cospans whose middle graphs do not contain any isolated node.
- *Link:* Graph automata which check that the middle graph of the input cospan consists of an edge which is incident to at least one node of the inner interface and to at least one node of the outer interface of the input cospan.

- *Path:* Graph automata which check that there exists an $\langle S, T \rangle$-path in the middle graph of the input cospan, where S is a subset of the inner interface and T is a subset of the outer interface. That is there exists a path starting at some node contained in S to some other node contained in T.
- *Subgraph:* Graph automata which accept all cospans whose middle graph contains a specific graph as a subgraph. See also Example 6.34.
- *Intersection/Union:* Graph automata whose accepted language is the intersection or union, respectively, of the languages of two given graph automata.

The states of the graph automata for the different graph problems mentioned above have to encode different pieces of information. But we assume that each encoding is of the form: $b_1 \ \dots \ b_m \ s_1 \ \dots \ s_n$, where the bits $b_1 \ \dots \ b_m$ encode the size of the outer interface of the cospan seen so far and the bits $s_1 \ \dots \ s_n$ encode the problem-specific state information. In the following we will describe which problem-specific information is needed and what the initial/final states for several graph problems are. We assume that each graph automaton accepts languages from i to j, i. e. all accepted cospans are of the form $D_i \nrightarrow D_j$:

- *k-Dominating Set:*
  - *Problem-specific information:* which nodes of the current interface are part of the dominating set; which nodes of the current interface are dominated by some node of the dominating set; the size of the dominating set (if nodes which belong to the dominating set are fused, the size has to be updated),
  - *Initial states:* the initial states are exactly those states which belong to Q_i and which have at most $\min(i, k)$ interface nodes contained in the dominating set; the size of the dominating set is equal to the number of interface nodes which belong to the dominating set,
  - *Final states:* the final states are exactly those states such that all interface nodes are dominated by some node (or belong to the dominating set); the size of the dominating set has to be less or equal to k.
- *k-Vertex Cover:*
  - *Problem-specific information:* which nodes of the current interface are part of the vertex cover; the size of the vertex cover (if nodes which belong to the vertex cover are fused, the size has to be updated),
  - *Initial states:* the initial states are exactly those states belonging to Q_i and which have at most $\min(i, k)$ interface nodes contained in the vertex cover; the size of the vertex cover is equal to the number of interface nodes which belong to the vertex cover,
  - *Final states:* the final states are exactly those states such that the size of vertex cover is less or equal to k.
- *Edge/Node-Counting:*
  - *Problem-specific information:* the remainder (modulo the fixed divisor) of the number of edges (nodes) which have been seen so far,

  - *Initial states:* in case of the edge-counting automaton, the only initial state which belongs to Q_i is the one with a remainder which is equal to zero; in case of the node-counting automaton, the only initial state which belongs to Q_i is the one with a remainder which is equal to i (modulo the divisor),
  - *Final states:* the only final state, which belongs to Q_j, is the one with a remainder which is equal to the wanted value.

- *Maximum/Minimum Edge:*
  - *Problem-specific information:* the current number of edges up to the maximum (minimum); if the number of edges exceeds the desired maximum (minimum) further edges are no longer counted,
  - *Initial states:* the only initial state which belongs to Q_i is the one having a counter equal to zero,
  - *Final states:* in case of the maximum automaton, the final states are those whose counter are less or equal to the maximum; in case of the minimum automaton, the final states are those whose counters are greater or equal to the minimum.

- *Maximum/Minimum Node:*
  - *Problem-specific information:* the current number of nodes up to the maximum interfaces size; if the number of nodes exceeds the desired maximum (minimum) further nodes *have* to be counted, since they can be fused again,
  - *Initial states:* the only initial state which belongs to Q_i is the one with a counter which is equal to i,
  - *Final states:* in case of the maximum automaton, the final states are those whose counters are less or equal to the maximum; in case of the minimum automaton, the final states are those whose counters are greater or equal to the minimum.

- *No Isolated Nodes:*
  - *Problem-specific information:* whether the several interface nodes are isolated; whether the parts of the input cospan which no longer belong to the interface contain an isolated node,
  - *Initial states:* the only initial state which belongs to Q_i is the one whose i interface nodes are all isolated, but which has no isolated node in the part of the input cospan which is not accessible anymore,
  - *Final states:* the only final state which belongs to Q_j is the one without any isolated interface node and which has no isolated node in the part of the input cospan which is not accessible anymore.

- *Link:*
  - *Problem-specific information:* which interface nodes belong to the inner interface of the input cospan; which nodes of the interface are connected to at least one interface node which belongs to the inner interface of the input cospan,

  - *Initial states:* the only initial state which belongs to Q_i is the one whose i interface nodes are all marked as inner interface nodes and which has no interface node which is connected to an inner interface node (since there is no edge),
  - *Final states:* the final states are exactly those which contain at least one interface node which is connected to an inner interface node of the input cospan.

- *Path:*
  - *Problem-specific information:* the transitive closure of the adjacency matrix of the subgraph induced by the interface nodes; for which interface node exists a path to at least one of the given source nodes,
  - *Initial states:* the initial states are exactly those states, which belong to Q_i and whose entries in the adjacency matrix are all 0 except for the main diagonal (of the length i); the only interface nodes for which a path to the source nodes exist are the source nodes themselves,
  - *Final states:* the final states are exactly those states, which belong to Q_j and whose adjacency matrix (for the nodes 1 to j) can be arbitrary and for all other nodes are set to 0; there exists at least one interface node which is contained in the target nodes for which a path to the source nodes exist.

- *Subgraph:*
  - *Problem-specific information:* the parts of the subgraph which have been recognized so far; the overlap of the parts (of the subgraph) with the current interface,
  - *Initial states:* the initial states are exactly those states, which belong to Q_i and which recognize a part of the subgraph consisting of at most i nodes (and no edge); every (recognized) node of the subgraph has to be mapped by at least one interface node,
  - *Final states:* the final states are exactly those states, which belong to Q_j and which recognize the complete subgraph; the nodes of the subgraph may or may not overlap with the interface (depends on whether an interface node corresponds to a subgraph node).

- *Intersection/Union:*
  - *Problem-specific information:* the specific information of the graph automata which have been used for intersection/union are concatenated,
  - *Initial states:* in case of the intersection automaton, the initial states are the conjunction of the initial states of the underlying automata; in case of the union automaton, the initial states are the disjunction of the initial states of the underlying automata,
  - *Final states:* in case of the intersection automaton, the final states are the conjunction of the final states of the underlying automata; in case of the union automaton, the final states are the disjunction of the final states of the underlying automata.

For the rest of this section we want to discuss why it is important to find a "good" encoding of the state sets of the several automata or rather of the BDDs representing the state sets. Since the state spaces of the automata listed above become very large very quickly, as seen in the introduction to this section, we have experimented with different encodings. The results of these experiments have led to the following advice:

- First of all, the index of the state set to which the encoded state belongs, i. e. the current interface size should be encoded with least significant bit first,
- Similar pieces of information should be grouped, i. e. BDD nodes encoding similar pieces of information should be encoded by subsequent BDD nodes,
- The BDD nodes used to encode the current states and the successor states (when encoding the transition relations) should be arranged in an interleaved fashion.

As explained above, all states are encoded by bit strings of the same length. Therefore, we have to check that only those BDD nodes are used which encode information that is valid for the current state. The BDD nodes which are not necessary to encode information for the current state are set to some unused default value. The first point is motivated by the thought that it is advantageous to first encode the current interface size in order to be able to check the remaining information. That is, if some state of the graph automaton above belongs to the i-th state set, then we can use this information to check that only the BDD nodes are valid which encode the status of the graph nodes accessible by the i interface nodes and that all other BDD nodes do not encode any information. The idea behind the second point is the fact that the size of a BDD grows exponentially in the length of a path between two BDD nodes which depend on each other. The third point is motivated by a similar argument like the one for the second point. Since the BDD nodes (encoding the information) of the successor state are arranged in the same way as in the current state this groups the BDD nodes automatically.

But even if we take all the considerations above into account, there could be a huge difference between two encodings. As an example we compare two encodings for a graph automaton which accepts all graphs up to a maximum interface size of n which have a dominating set of size at most 5. For this example the input alphabet of the graph automaton contains the letters $\mathit{connect}^i_\diamond$, fuse^i, shift^i, res^i, trans^i, vertex^i, where $\diamond$ is a label of arity 2. The encodings represent a state of the graph automaton as the bit string

$$\vec{b}\,(m_1\,d_1)\ldots(m_n\,d_n)\,(s_1\,s_2\,s_3) \qquad \text{(First Encoding)}$$

or

$$\vec{b}\,m_1\ldots m_n\,d_1\ldots d_n\,(s_1\,s_2\,s_3) \qquad \text{(Second Encoding)}$$

respectively, where $\vec{b} = b_1\ldots b_m$ encodes the current interface size as a binary number, m_i encodes whether the i-th interface node is a member of the dominating set and d_i encodes whether the i-th interface node is dominated by some other node (or itself). The bits $s_1\,s_2\,s_3$ are used to encode the number of nodes which belong to the dominating

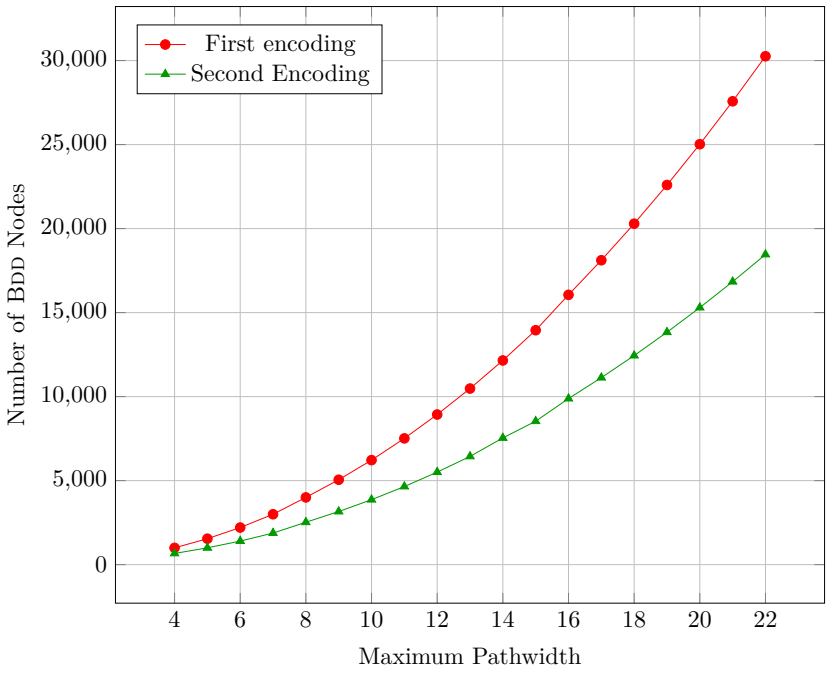
set, but which have already been removed from the interface, i. e. which are no longer accessible.

In Figure 7.5a the number of Bdd nodes which are needed to encode the Bdds representing the transitions of the graph automaton for various maximum interface sizes are shown. As we can see, it is more efficient to first encode all bits representing the (non-)membership of the i-th interface node, before we encode whether the several graph nodes which can be accessed through the interface are dominated. The difference in the number of Bdd nodes is in large part due to the $shift^i$-letter. For rather small maximum interface sizes the size of the Bdd representing all $shift^i$-transitions using the first encoding amounts to about 50% of the overall Bdd nodes, but this portion increases up to 70% for the maximum interface sizes of 20 and more. The reason for this is that the first encoding groups the bit for (non-)membership with the bit for the (non-)domination (for each node). Hence, in case of the $shift^i$-transition the corresponding Bdd has to relate bits which lay on a path of length $2n - 1$. This is rather costly (compared to the bit arrangement of the second encoding). For the second encoding the Bdd representing all $shift^i$-transitions only amounts to 25% for the smaller maximum interface sizes and the portion grows up to 35%. This fact is not suprising, since the Bdd for the $shift^i$-transitions has to relate the bit m_1 (d_1) with the bit m_n (d_n). As stated above, in case of the first encoding, the path from the first to the second bit is twice as long as in the case of the second encoding. Hence it is more efficient (in size of the Bdd nodes) to use the second encoding.

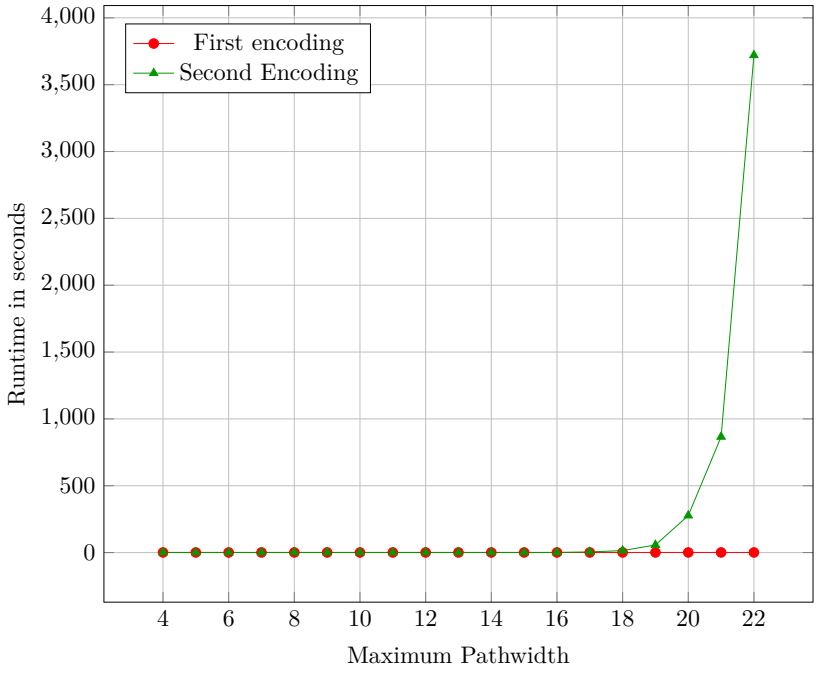
But beside the number of Bdd nodes needed to represent a graph automaton, it is also important to measure the time required to compute the different Bdds. Figure 7.5b shows the runtimes in seconds for the computation of the transition-Bdds. It is clear that the first encoding is much more efficient than the second encoding. Even for a maximum interface size of 100 the first encoding requires only a few seconds to compute all transition-Bdds. For maximum interface sizes larger than 20 the second encoding is not practically usable. The reason for this huge difference in the runtime is probably arising from the construction of the several Bdds. The resulting Bdd which encodes the state space according to the different encodings is not computed at once, but in several steps. Even if the resulting Bdd is rather small, as in the case of the second encoding, it might be the case that Bdds which are computed in between are exponentially larger (in the number of Bdd nodes). Hence, the operations which are applied to the several Bdds need much more time to compute. It seems that the Bdds which are created temporarily to compute the resulting Bdd for the first encoding are smaller and thus the computation of the first encoding Bdd is more efficient in matters of runtime.

7.2. Graph Automata and the Language Inclusion Problem

One of the main applications of Raven is to automatically check invariants of graph transformation systems as described in Chapter 6. In [15], Bruggink and König together with the author of this thesis presented a technique for checking invariants based on the Myhill-Nerode quasi-order which has been introduced in Section 6.1.3 (cf. Definition 6.20). But the computation of the Myhill-Nerode quasi-order depends on deterministic graph automata. To avoid determinization we extended the approach and



(a) Comparison of the number of Bdd nodes



(b) Comparison of runtimes to compute the Bdds

Figure 7.5.: Number of Bdd nodes and runtimes to compute the Bdds used to encode all $\{connect^i_\diamond, fuse^i, shift^i, res^i, trans^i, vertex^i\}$-transitions of the 5-Dominating Set automaton.

used an over-approximation, namely a simulation quasi-order, instead of the Myhill-Nerode quasi-order. However, the over-approximation led to the problem of one-sided errors.

To overcome this problem, we will use recent algorithms for checking language inclusion, which can be applied to non-deterministic (finite) automata. The connection between checking invariants and the language inclusion problem has been shown in Theorem 6.38.

These approaches which we will investigate further in the next subsections are:

- An antichain-based approach using either a forwards or a backwards search [132],
- an enhanced antichain-based approach which is equipped with an additional simulation relation [1] and
- an approach using bisimulations up to congruence for checking language equivalence, which can be used for the language inclusion problem as well [23].

In the next subsections, we forget typing information of the states and consider bounded automata as regular finite automata.

7.2.1. An Antichain-Based Approach for Language Inclusion

In this subsection we introduce an antichain-based algorithm in four variants – two forwards and two backwards searching variants – developed by De Wulf, Doyen, Henzinger and Raskin [132]. Recall that an antichain is a set of elements which are incomparable with respect to some ordering. How the elements look like and what ordering is used depends on the application; for the algorithms presented below which are used to decide language inclusion we use an order which is based on the usual subset ordering, but which is extended to pairs. The details of the ordering will be explained below.

Let $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, \mathrm{I}_\mathcal{A}, \mathrm{F}_\mathcal{A} \rangle$ and $\mathcal{B} = \langle Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, \mathrm{I}_\mathcal{B}, \mathrm{F}_\mathcal{B} \rangle$ be n-bounded graph automata. Remember that we denote the set of non-initial states of $\mathcal{B}$ by $\overline{\mathrm{I}_\mathcal{B}}$ and the set of non-accepting states of $\mathcal{B}$ by $\overline{\mathrm{F}_\mathcal{B}}$, i. e. $\overline{\mathrm{I}_\mathcal{B}} = Q_\mathcal{B} \setminus \mathrm{I}_\mathcal{B}$ and $\overline{\mathrm{F}_\mathcal{B}} = Q_\mathcal{B} \setminus \mathrm{F}_\mathcal{B}$. Furthermore, we introduce two new kinds of transitions, which play an important role for the algorithms given below: *reversed transitions*, denoted by $\delta_{\mathcal{A}^{-1}}$, and *complemented transitions*, denoted by $\delta_{\overline{\mathcal{A}}}$. The first kind is obtained by "turning a given transition around", i. e. $q \in \delta_{\mathcal{A}^{-1}}(q', \sigma)$ if and only if $q' \in \delta_\mathcal{A}(q, \sigma)$. Analogously, the reversed transitions are defined for the graph automaton $\mathcal{B}$. The second kind is a bit more complicated. It can be obtained as follows: $\delta_{\overline{\mathcal{A}}}(q, \sigma) := \overline{\delta_\mathcal{A}(\overline{\{q\}}, \sigma)}$. The intuition behind the complemented transitions is that there exists a complemented transition between a state q and a state q' if and only if there exists no (normal) transition between a state in $\overline{\{q\}}$ and q'. The complemented transitions for the graph automaton $\mathcal{B}$ are defined as $\delta_{\overline{\mathcal{B}}}(Q, \sigma) := \overline{\delta_\mathcal{B}(\overline{Q}, \sigma)}$.

To come back to our actual aim, which is to decide the language inclusion problem of $\mathcal{A}$ and $\mathcal{B}$, i. e. whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ or not, we now introduce the antichain algorithm. In particular, the algorithm tries to falsify the claim that the language inclusion holds by one of four different variants:

Normal Forwards Searching Variant: By starting in an initial state $i \in \mathrm{I}_\mathcal{A}$ of $\mathcal{A}$ and in the set of initial states $\mathrm{I}_\mathcal{B}$ of $\mathcal{B}$ this variant tries to reach a state $q' \in \mathrm{F}_\mathcal{A}$ and a

set of states $Q' \subseteq \overline{\mathrm{F}_{\mathcal{B}}}$ by performing (forward) transitions such that $q' \in \hat{\delta}_{\mathcal{A}}(i, \vec{\sigma})$ and $Q' = \hat{\delta}_{\mathcal{B}}(\mathrm{I}_{\mathcal{B}}, \vec{\sigma})$ for some sequence $\vec{\sigma}$.

Normal Backwards Searching Variant: By starting in a final state $q' \in \mathrm{F}_{\mathcal{A}}$ of $\mathcal{A}$ and in the set of final states $\mathrm{F}_{\mathcal{B}}$ of $\mathcal{B}$ this variant tries to reach a state $i \in \mathrm{I}_{\mathcal{A}}$ and a set of states $Q \subseteq \overline{\mathrm{I}_{\mathcal{B}}}$ by performing reversed transitions such that $i \in \hat{\delta}_{\mathcal{A}^{-1}}(q', \vec{\sigma})$ and $Q = \hat{\delta}_{\mathcal{B}^{-1}}(\mathrm{F}_{\mathcal{B}}, \vec{\sigma})$ for some sequence $\vec{\sigma}$.

Complement Forwards Searching Variant: By starting in an initial state $i \in \mathrm{I}_{\mathcal{A}}$ of $\mathcal{A}$ and in the set of non-initial states $\overline{\mathrm{I}_{\mathcal{B}}}$ of $\mathcal{B}$ this variant tries to reach a state $q' \in \mathrm{F}_{\mathcal{A}}$ and a set of states $Q' \supseteq \mathrm{F}_{\mathcal{B}}$ by performing complemented transitions such that $q' \in \hat{\delta}_{\overline{\mathcal{A}}}(i, \vec{\sigma})$ and $Q' = \hat{\delta}_{\overline{\mathcal{B}}}(\overline{\mathrm{I}_{\mathcal{B}}}, \vec{\sigma})$ for some sequence $\vec{\sigma}$.

Complement Backwards Searching Variant: By starting in a final state $q' \in \mathrm{F}_{\mathcal{A}}$ of $\mathcal{A}$ and the set of non-final states $\overline{\mathrm{F}_{\mathcal{B}}}$ of $\mathcal{B}$ this variant tries to reach an initial state $i \in \mathrm{I}_{\mathcal{A}}$ of $\mathcal{A}$ and a set of states $Q \supseteq \mathrm{I}_{\mathcal{B}}$ by performing reversed and complemented transitions such that $i \in \hat{\delta}_{\overline{\mathcal{A}^{-1}}}(q', \vec{\sigma})$ and $Q = \hat{\delta}_{\overline{\mathcal{B}^{-1}}}(\overline{\mathrm{F}_{\mathcal{B}}}, \vec{\sigma})$ for some sequence $\vec{\sigma}$.

For any of these four variants it holds that if the particular algorithm has found a sequence $\vec{\sigma}$, which satisfies the conditions stated above, the algorithm has found a witness of the falsification of the language inclusion, since the sequence is accepted by the graph automaton $\mathcal{A}$, but is rejected by the graph automaton $\mathcal{B}$.

There is a strong connection between the two forwards and the two backwards variants. The connection between the normal and the complement forwards searching variant is made as follows: Due to the definition of the complemented transitions, one can reach merely non-final states when starting from the initial states if and only if the final states are only reachable from the non-initial states by performing complemented transitions. Therefore, the normal and the complement forwards searching variants are equivalent.

The same argument holds for the two backwards searching variants and the reversed transitions. Hence, these two variants are also equivalent.

The relationship between the normal forwards and the complement backwards searching variant is a litte bit different: By Henzinger et al. [132] we have that for every instance of the language inclusion problem which is difficult to solve with the normal forward searching variant, there exists an equally difficult instance for the complement backwards searching variant.

Again, we can argue that the same holds for the relationship between the complement forwards and the normal backwards searching variant.

In the following we introduce the algorithms for normal forwards searching and the complement backwards searching variant. At the end of this section, we will then take a further look at all four variants. We will also explain why it is not necessary to implement all four variants if using BDDs. We use a slightly different version of the algorithms stated by Henzinger et al., which have been published by Blume, Bruggink, Engelke and König in [12]. Let $\mathcal{U} = Q_{\mathcal{A}} \times \wp(Q_{\mathcal{B}})$ and $\langle q, Q\rangle, \langle q', Q'\rangle \in \mathcal{U}$, we define

$$\langle q, Q\rangle \leq \langle q', Q'\rangle \quad \text{if } q = q' \text{ and } Q \subseteq Q'.$$

Remember that an *antichain* (for language inclusion) is a set of pairwise incomparable elements.

Normal Forwards Searching Antichain Algorithm. For the forwards searching algorithm, we define the following function:

$$\mathsf{Post}_{\mathcal{A},\mathcal{B}}(K) = \left\{ \langle q', Q' \rangle \;\middle|\; \exists \sigma \in \Sigma : \exists \langle q, Q \rangle \in K : q' \in \delta_{\mathcal{A}}(q, \sigma) \wedge \hat{\delta}_{\mathcal{B}}(Q, \sigma) = Q' \right\}.$$

The function does the following: For each $\langle q, Q \rangle \in K$, we take the pairs $\langle q', Q' \rangle$ such that, for some symbol σ, q' is an σ-successor of q and Q' is the set of states, which is reached from Q when reading σ.

The forwards searching algorithm, which returns *true* if and only if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, works as follows:

Algorithm 7.1: Forwards Searching Antichain Algorithm

Input: $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, \mathrm{I}_{\mathcal{B}}, \mathrm{F}_{\mathcal{B}} \rangle$
Output: *true* if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, *false* otherwise

```
1: K ← I_A × {I_B}
2: repeat
3:     K' ← K
4:     K ← ⌊K ∪ Post_{A,B}(K)⌋
5: until K = K'
6: return there exist ⟨q, Q⟩ ∈ K such that q ∈ F_A and Q ⊆ complement(F_B)
```

The line $K \leftarrow \lfloor K \cup \mathsf{Post}_{\mathcal{A},\mathcal{B}}(K) \rfloor$ adds new elements to the current antichain and removes all but the minimal ones. The idea behind the minimization step is as follows: Let $\langle q, Q \rangle, \langle q', Q' \rangle \in K$ be two state pairs such that $\langle q, Q \rangle \leq \langle q', Q' \rangle$. It is obvious that whenever a final state pair[2] is reachable from the state pair $\langle q, Q \rangle$ by some sequence there exists a final state pair which is reachable from the state pair $\langle q', Q' \rangle$ by the same sequence.

Complement Backwards Searching Antichain Algorithm. This variant is similar to the forward algorithm above, except that it starts with the antichain $\mathrm{F}_{\mathcal{A}} \times \{\overline{\mathrm{F}_{\mathcal{B}}}\}$ and iterates the function

$$\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K) = \left\{ \langle q, Q \rangle \;\middle|\; \exists \sigma \in \Sigma : \exists \langle q', Q' \rangle \in K : q' \in \delta_{\mathcal{A}}(q, \sigma) \wedge \hat{\delta}_{\mathcal{B}}(Q, \sigma) \subseteq Q' \right\}.$$

The function does the following: For each $\langle q', Q' \rangle \in K$, we take the pairs $\langle q, Q \rangle$ such that, for some symbol σ, q is an σ-predecessor of q' and Q is the set of states, from which only states in Q' are surely reachable when reading σ.

Formally, the basic version of the algorithm, which returns *true* if and only if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, works as follows:

[2] A pair $\langle q, Q \rangle$ is said to be *final* if and only if $q \in \mathrm{F}_{\mathcal{A}}$ and $Q \not\subseteq \mathrm{F}_{\mathcal{B}}$.

Algorithm 7.2: Backwards Searching Antichain Algorithm

Input: $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, \mathrm{I}_{\mathcal{B}}, \mathrm{F}_{\mathcal{B}} \rangle$
Output: *true* if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, *false* otherwise
1: $K \leftarrow \mathrm{F}_{\mathcal{A}} \times \{\overline{\mathrm{F}_{\mathcal{B}}}\}$
2: **repeat**
3: $\quad K' \leftarrow K$
4: $\quad K \leftarrow \lceil K \cup \mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K) \rceil$
5: **until** $K = K'$
6: **return** there exist $\langle q, Q \rangle \in K$ such that $q \in \mathrm{I}_{\mathcal{A}}$ and $\mathrm{I}_{\mathcal{B}} \subseteq Q$

The line $K \leftarrow \lceil K \cup \mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K) \rceil$ adds new elements to the current antichain and removes all but the maximal ones.

The basic algorithms can be optimized in various ways. First, only new elements need to be processed in each step instead of all the elements in K. Second, since the function is monotone, the algorithm can return *true* as soon as the final condition (in line 6) is satisfied (meaning that $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$). Both optimizations (for both algorithms) have been implemented in RAVEN. But note that in the implementation that we used in the tool, both the automata and the pairs in the antichains are represented symbolically as BDDs. For a correctness proof of the algorithms, we refer to [132].

Example 7.2. *Since graph automata can be seen as (very large) non-deterministic finite automata, we use (small)* NFAs *instead of graph automata for this example.*

Let two non-deterministic finite automata

$$\mathcal{A} = \langle \{s_0, s_1, s_2\}, \{a, b\}, \delta_{\mathcal{A}}, \{s_0, s_1\}, \{s_2\} \rangle$$

and

$$\mathcal{B} = \langle \{t_0, t_1, t_2, t_3\}, \{a, b\}, \delta_{\mathcal{B}}, \{t_0, t_1\}, \{t_0, t_2\} \rangle,$$

be given which are depicted in Figure 7.6. We want to check whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ or not, using the two algorithms explained above.

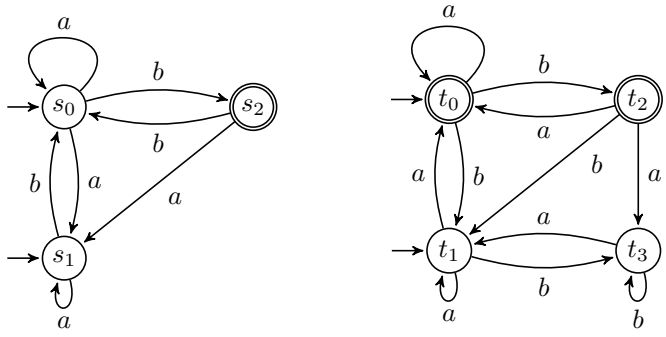


Figure 7.6.: The graph automata $\mathcal{A}$ (depicted on the left) and $\mathcal{B}$ (depicted on the right).

In case of the (optimized) normal forwards searching variant, the algorithm is initialized with the state set $K = \{\langle s_0, \{t_0, t_1\}\rangle, \langle s_1, \{t_0, t_1\}\rangle\}$ which is depicted in

Figure 7.7 by the solid-bordered state pairs. Starting at these two initial state pairs, the algorithm computes the successor states by means of the function $\mathsf{Post}_{\mathcal{A},\mathcal{B}}(K)$*. The result of this function is the set*

$$\mathsf{Post}_{\mathcal{A},\mathcal{B}}(\{\langle s_0,\{t_0,t_1\}\rangle,\langle s_1,\{t_0,t_1\}\rangle\}) = \{\langle s_0,\{t_0,t_1\}\rangle,\langle s_1,\{t_0,t_1\}\rangle,\\ \langle s_2,\{t_1,t_2,t_3\}\rangle,\langle s_0,\{t_1,t_2,t_3\}\rangle\}$$

which is also depicted in the second row of Figure 7.7. The letters on the arrows indicate how the several successor pairs have been computed. The dotted-bordered boxes indicate that the respective state pairs have already been computed in previous steps of the algorithm and need not be processed again. Hence the new value of K *is the set*

$$K = \{\langle s_2,\{t_1,t_2,t_3\}\rangle,\langle s_0,\{t_1,t_2,t_3\}\rangle\}$$

and the algorithm iterates further, since the antichain has been changed. Again, the successors of the set K *are computed:*

$$\mathsf{Post}_{\mathcal{A},\mathcal{B}}(\{\langle s_2,\{t_1,t_2,t_3\}\rangle,\langle s_0,\{t_1,t_2,t_3\}\rangle\}) = \{\langle s_1,\{t_0,t_1,t_3\}\rangle,\langle s_0,\{t_1,t_3\}\rangle,\\ \langle s_0,\{t_0,t_1,t_3\}\rangle,\langle s_2,\{t_1,t_3\}\rangle\}$$

Since $\langle s_1,\{t_0,t_1\}\rangle \leq \langle s_1,\{t_0,t_1,t_3\}\rangle$ *and* $\langle s_0,\{t_0,t_1\}\rangle \leq \langle s_0,\{t_0,t_1,t_3\}\rangle$*, which is indicated by the dashed-bordered state pairs, we have that*

$$\lfloor\{\langle s_1,\{t_0,t_1,t_3\}\rangle,\langle s_0,\{t_1,t_3\}\rangle,\langle s_0,\{t_0,t_1,t_3\}\rangle,\langle s_2,\{t_1,t_3\}\rangle\}\rfloor =\\ \{\langle s_0,\{t_1,t_3\}\rangle,\langle s_2,\{t_1,t_3\}\rangle\}.$$

The optimized algorithm terminates at this point and returns true*, i. e.* $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$*, since the state pair* $\langle s_2,\{t_1,t_3\}\rangle$ *contains the accepting state* s_2 *of* $\mathcal{A}$*, but the state set* $\{t_1,t_3\}$ *of* $\mathcal{B}$ *is non-accepting, which is indicated by the doubly-bordered box. The word, found by the algorithm, which is a counterexample for the language inclusion is bb.*

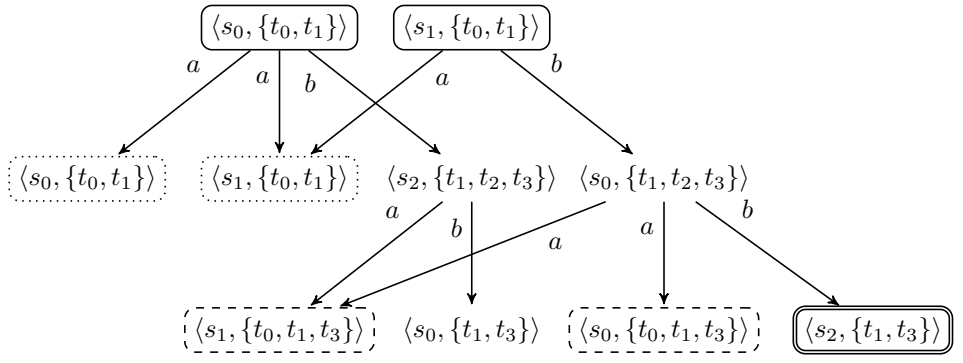


Figure 7.7.: Run of the (optimized) forwards searching antichain algorithm on the automata $\mathcal{A}$ and $\mathcal{B}$

Now, we want to perform the language inclusion check using the complemented backwards searching algorithm. Initially, the set K *contains only the state pair* $\langle s_2,\{t_1,t_3\}\rangle$*,*

which is shown in Figure 7.8. Again, the solid-bordered box indicates that the state has been added in the initial phase. Now, the first iteration of the loop starts, computing the set

$$\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(\{\langle s_2, \{t_1, t_3\}\rangle\}) = \{\langle s_0, \{t_1, t_2, t_3\}\rangle\}.$$

Since the pair $\langle s_0, \{t_1, t_2, t_3\}\rangle$ *is incomparable to* $\langle s_2, \{t_1, t_3\}\rangle$ *and the set* $\{t_1, t_2, t_3\}$ *does not contain all initial states of* $\mathcal{B}$ *the algorithm continues with another iteration. The predecessors of* $\langle s_0, \{t_1, t_2, t_3\}\rangle$ *are*

$$\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(\{\langle s_0, \{t_1, t_2, t_3\}\rangle\}) = \{\langle s_0, \{t_1, t_3\}\rangle, \langle s_1, \{t_0, t_1, t_2, t_3\}\rangle, \langle s_2, \{t_0, t_1, t_2, t_3\}\rangle\}.$$

Since $\{t_1, t_3\} \subset \{t_1, t_2, t_3\}$*, we have that* $\langle s_0, \{t_1, t_3\}\rangle \leq \langle s_0, \{t_1, t_2, t_3\}\rangle$*. Therefore, the pair* $\langle s_0, \{t_1, t_3\}\rangle$ *can be omitted, indicated by the dashed-bordered box in Figure 7.8. Anyway, the algorithm terminates and returns* true*, i. e.* $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$*, due to the pair* $\langle s_0, \{t_0, t_1, t_2, t_3\}\rangle$*. This is because the state* s_0 *is an initial state of* $\mathcal{A}$ *and the state set* $\{t_0, t_1, t_2, t_3\}$ *contains all initial states of* $\mathcal{B}$*. The witness which is found to disprove the language inclusion is again the word bb.*

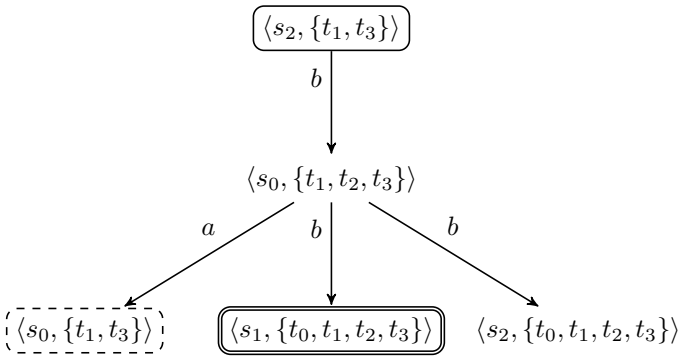


Figure 7.8.: Run of the (optimized) backwards searching antichain algorithm on the automata $\mathcal{A}$ and $\mathcal{B}$

Now, we want to take another look at the different variants of the antichain-based approach. As presented above, there are two forwards searching variants and two backwards searching variants. In the following we will discuss how the two variants, which were not presented in detail, could be implemented. But we will see that it is not necessary to implement them on BDDs, since they are subsumed by the other variants.

At first, we further inspect the two forwards searching variants. The algorithm for the normal forwards searching variant starts with an arbitrary initial state of the first and the set of all initial states of the second automaton, searches for the corresponding successor states and chooses only the minimal ones (with respect to set inclusion). But instead of starting in the initial states of both automata and minimizing the resulting successor state pairs one could do the opposite. That is one could perform the search on the "complement" of the original antichain, whereas the complement of a set of state pairs K is the set $\overline{K} = \{\langle q, \overline{Q}\rangle \mid \langle q, Q\rangle \in K\}$. This leads to the following changes to the original forwards searching variant:

- the algorithm is initiated with all pairs containing an initial state of the first automaton and the set of non-initial states of the second automaton,
- the function $\mathsf{Post}_{\mathcal{A},\mathcal{B}}(K)$ is replaced by $\overline{\mathsf{Post}_{\mathcal{A},\mathcal{B}}(\overline{K})}$ and the result is maximized and not minimized,
- the algorithm returns *true* to indicate that the language inclusion does not hold, if and only if there exists a state pair containing a final state of the first and a set containing all final states of the second automaton.

Next, we discuss the two backwards searching variants. As explained above, the algorithm for the complement backwards searching variant starts with an arbitrary final state of the first and the set of all non-final states of the second automaton. Subsequently, the corresponding predecessor state pairs are computed and the maximal ones (with respect to set inclusion) are chosen. Again, we could initiate the algorithm with the "complement" set and perform the opposite operations, which would lead to the following changes:

- the algorithm is initiated with all pairs containing a final state of the first automaton and the set of final states of the second automaton,
- the function[3] $\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K)$ is replaced by $\overline{\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(\overline{K})}$ and the result is minimized and not maximized,
- the algorithm returns *true* to indicate that the language inclusion does not hold if and only if there exists a state pair containing an initial state of the first and a set without any initial state of the second automaton.

An overview of the four different forwards and backwards searching algorithms is given in Table 7.1.

As mentioned above, it is not necessary to implement all four variants by means of BDDs. This is due to the fact that a set (of states) and its complement set can be represented by two almost identical BDDs. The only difference is that whenever a path within a BDD leads to the terminal labeled with 0, the same path leads to the terminal labeled with 1 of the "complement BDD" and vice versa. Therefore, every BDD representing a state set computed during the run of normal forwards searching algorithms can be converted into a state set of a run of the complement forwards searching algorithms (by switching the two terminal nodes) and vice versa. The same holds for the normal and the complement backwards searching algorithm.

7.2.2. Simulation-based Antichain Algorithms

In this subsection we want to present an improved version of the antichain approach introduced in the previous subsection. The main idea is to combine two different types of approaches used to prove (or disprove) language inclusion.

These two types can be divided into *simulation-based* approaches and approaches using the *subset construction*. The former ones compute and use a simulation relation

[3] Note that this is the "usual" predecessor function, i. e. the successor function of the "reverted" automaton.

Input: $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, \mathrm{I}_{\mathcal{B}}, \mathrm{F}_{\mathcal{B}} \rangle$
Output: *true* if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, *false* otherwise
  $K \leftarrow$ ①
  repeat
    $K' \leftarrow K$
    $K \leftarrow$ ②
  until $K = K'$
  return there exist $\langle q, Q \rangle \in K$ such that ③

	Forwards	Backwards
Normal	① $\mathrm{I}_{\mathcal{A}} \times \{\mathrm{I}_{\mathcal{B}}\}$	① $\mathrm{F}_{\mathcal{A}} \times \{\mathrm{F}_{\mathcal{B}}\}$
	② $\lfloor K \cup \mathsf{Post}_{\mathcal{A},\mathcal{B}}(K) \rfloor$	② $\lfloor K \cup \overline{\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(\overline{K})} \rfloor$
	③ $q \in \mathrm{F}_{\mathcal{A}} \wedge Q \subseteq \overline{\mathrm{F}_{\mathcal{B}}}$	③ $q \in \mathrm{I}_{\mathcal{A}} \wedge Q \subseteq \overline{\mathrm{I}_{\mathcal{B}}}$
Complement	① $\mathrm{I}_{\mathcal{A}} \times \{\overline{\mathrm{I}_{\mathcal{B}}}\}$	① $\mathrm{F}_{\mathcal{A}} \times \{\overline{\mathrm{F}_{\mathcal{B}}}\}$
	② $\lceil K \cup \overline{\mathsf{Post}_{\mathcal{A},\mathcal{B}}(\overline{K})} \rceil$	② $\lceil K \cup \mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K) \rceil$
	③ $q \in \mathrm{F}_{\mathcal{A}} \wedge \mathrm{F}_{\mathcal{B}} \subseteq Q$	③ $q \in \mathrm{I}_{\mathcal{A}} \wedge \mathrm{I}_{\mathcal{B}} \subseteq Q$

Table 7.1.: Overview of the different variants of the antichain-based algorithm

on the states of the two involved automata $\mathcal{A}$ and $\mathcal{B}$ and then check if all of the initial states of the automaton $\mathcal{A}$ can be simulated by some initial state of the automaton $\mathcal{B}$ [55]. In this case the language inclusion has been proven. The latter ones, like the algorithm from Subsection 7.2.1, typically build the product automaton $\mathcal{A} \times \overline{\mathcal{B}}$, which consists of the automaton $\mathcal{A}$ and the complement of the automaton $\mathcal{B}$, and then try to find a word which is accepted by the product automaton. In this case the language inclusion has been disproven. The advantage of the simulation-based approaches is that the simulation can be computed in polynomial time in the size of the involved automata. The disadvantage is that simulation implies language inclusion, but not vice versa. This problem does not occur with techniques using the subset construction, since these approaches are complete. However, the subset construction often causes an exponential blow-up of the state space which leads to an exponential runtime.

The algorithm by Abdulla, Chen, Holík, Mayr and Vojnar [1], which is presented below, combines these two approaches by using a simulation to rule out unneccessary state pairs which have been computed by the antichain-based algorithm given in Subsection 7.2.1. The idea behind this new algorithm is as follows: The antichain-based algorithm uses equality (resp. set inclusion) to determine whether two states (resp. two state sets) are related w. r. t. language inclusion. The algorithm presented below uses a simulation relation, which is defined in Definition 7.3, instead of the equality and set inclusion, since simulations are more general (cf. Lemma 7.4).

Definition 7.3 (Simulation)**.** Let $n \in \mathbb{N}$ and $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, \mathrm{I}_\mathcal{A}, \mathrm{F}_\mathcal{A} \rangle$ be an n-bounded graph automaton with $Q_\mathcal{A} = (Q_i)_{0 \leq i \leq n}$. A *simulation* (on $\mathcal{A}$) is a family of relations

$$\preceq = \{ R_i \mid 0 \leq i \leq n \},$$

where the components $R_i \subseteq Q_i \times Q_i$ are relations such that $p \, R_i \, q$ implies that the following two conditions are satisfied:

- $p \in \mathrm{F}_\mathcal{A}$ implies $q \in \mathrm{F}_\mathcal{A}$ and
- for every state p' with $p' \in \delta_\mathcal{A}(p, \sigma)$ there exists a state q' with $q' \in \delta_\mathcal{A}(q, \sigma)$ such that $p' \, R_i \, q'$.

Let $q, q' \in Q_i$ with $Q_i \subseteq Q_\mathcal{A}$. We usually write $q \preceq q'$, if $q \, R_i \, q'$. For two sets $Q, Q' \subseteq Q_\mathcal{A}$ of states we define

$$Q \preceq^{\forall\exists} Q' \quad \text{if } \forall\, q \in Q \colon \exists\, q' \in Q' \colon q \preceq q'.$$

The following lemma is well-known for non-deterministic automata and can easily be generalized to graph automata.

Lemma 7.4. Let $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, \mathrm{I}_\mathcal{A}, \mathrm{F}_\mathcal{A} \rangle$ be an n-bounded graph automaton, $\preceq$ be a simulation and $q_1, q_2 \in Q_\mathcal{A}$. Then $q_1 \preceq q_2$ implies $\mathcal{L}(\mathcal{A})(q_1) \subseteq \mathcal{L}(\mathcal{A})(q_2)$.

In the following we give the enhanced version of the antichain-based language inclusion algorithm. In contrast to the basic antichain algorithm we only have a forwards searching variant. For this new variant we have to slighty alter some definitions from above. First, we use a different ordering. Let $\mathcal{U} = Q_\mathcal{A} \times \wp(Q_\mathcal{B})$ and $\langle q, Q \rangle, \langle q', Q' \rangle \in \mathcal{U}$, we define

$$\langle q, Q \rangle \leq \langle q', Q' \rangle \quad \text{if } q' \preceq q \text{ and } Q \preceq^{\forall\exists} Q'. \tag{7.1}$$

Additionally, we need a reduction[4] function Reduce which does the following:

Input: $Q \subseteq Q_i$ with $Q_i \in Q_\mathcal{B}$ a set of $\mathcal{B}$-states
Output: $Q' \subseteq Q$

```
Q' := Q
for q ∈ Q do
    if ∃q' ∈ Q' such that q ⪯ q' then
        Q' := Q' \ {q}
return Q'
```

The function Reduce(Q) iterates over all states $q \in Q$ and removes q from the resulting set Q' if there exists another state $q' \in Q'$ such that $q \preceq q'$. Hence, the resulting set contains only the minimal representatives w. r. t. simulation of the states contained in Q. Furthermore, for each "equivalence class" w. r. t. simulation the function chooses arbitrarily one representative. How this function can be implemented will be explained below (cf. Equation 7.2).

[4] In [1] this function is called Minimize. We rename it, to avoid confusion with the minimal elements.

Forwards searching algorithm. First, we redefine the function $\mathsf{Post}_{\mathcal{A},\mathcal{B}}$ from subsection 7.2.1 for the forwards searching algorithm:

$$\mathsf{RPost}_{\mathcal{A},\mathcal{B}}(K) = \big\{(q', \mathsf{Reduce}(Q')) \;\big|\; \exists \sigma \in \Sigma\colon \exists \langle q, Q\rangle \in K\colon q' \in \delta_{\mathcal{A}}(q,\sigma) \wedge \hat{\delta}_{\mathcal{B}}(Q,\sigma) = Q'\big\}.$$

The function does the same as the function $\mathsf{Post}_{\mathcal{A},\mathcal{B}}$ with the difference that the successors of $\mathcal{B}$ are minimized additionally.

The enhanced forwards searching algorithm, which returns *true* if and only if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, is then defined as follows:

Algorithm 7.3: Forwards Searching Simulation-based Antichain Algorithm

Input: $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}}\rangle$ and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, \mathrm{I}_{\mathcal{B}}, \mathrm{F}_{\mathcal{B}}\rangle$
Output: *true* if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, *false* otherwise
1: $K \leftarrow \mathrm{I}_{\mathcal{A}} \times \{\mathsf{Reduce}(\mathrm{I}_{\mathcal{B}})\}$
2: **repeat**
3: $\quad K' \leftarrow K$
4: $\quad K \leftarrow \lfloor K \cup \mathsf{RPost}_{\mathcal{A},\mathcal{B}}(K)\rfloor$
5: **until** $K = K'$
6: **return** there exist $q \in \mathrm{I}_{\mathcal{A}}$ and $Q \preceq^{\forall\exists} \overline{\mathrm{F}_{\mathcal{B}}}$ such that $\langle q, Q\rangle \in K$

Note that in the first line, the set of initial states of $\mathcal{B}$ is already reduced. Additionally, the line $K \leftarrow \lfloor K \cup \mathsf{RPost}_{\mathcal{A},\mathcal{B}}(K)\rfloor$ adds only those state pairs to the current antichain whose sets of $\mathcal{B}$-states have been reduced. At all times it holds that for all $\langle q, Q\rangle \in K$ there is a word $\vec{\sigma}$ such that $\hat{\delta}_{\mathcal{A}}(\{q\}, \vec{\sigma}) \cap \mathrm{F}_{\mathcal{A}} \neq \emptyset$ and $\hat{\delta}_{\mathcal{B}}(Q, \vec{\sigma}) \subseteq \overline{\mathrm{F}_{\mathcal{B}}}$.

Once more, the algorithm can be optimized. The first two optimizations can be adapted from the antichain-based algorithms from Subsection 7.2.1. The third optimization can be obtained from the following observation:

Lemma 7.5. Let $\langle q, Q\rangle \in \mathcal{U}$ be some state pair. For every $q' \in Q$ such that $q \preceq q'$ it holds that $\mathcal{L}(\mathcal{A})(q) \subseteq \mathcal{L}(\mathcal{B})(q')$.

Furthermore, if for all $i_{\mathcal{A}} \in \mathrm{I}_{\mathcal{A}}$ it holds that there exists an $i_{\mathcal{B}} \in \mathrm{I}_{\mathcal{B}}$ such that $i_{\mathcal{A}} \preceq i_{\mathcal{B}}$ then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

Proof. Trivial. □

Therefore, the algorithm can stop if $\mathrm{I}_{\mathcal{A}} \preceq^{\forall\exists} \mathrm{I}_{\mathcal{B}}$ holds and return *false*, meaning $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

All optimizations have been implemented in Raven. But note that in the implementation that we used in the tool, both the automata and the pairs in the antichains are represented symbolically as Bdds. For a correctness proof of the algorithms, we refer to [1].

Example 7.6. *Let the two non-deterministic finite automata $\mathcal{A}$ and $\mathcal{B}$ from Example 7.2, which were depicted in Figure 7.6, be given. Again, we want to check whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ or not. But this time we use the simulation-based antichain algorithm*

explained above. Before we can start, we have to compute a simulation. For this example we use the simulation given by the two tables below:

x	s_0	✔	✘	✘	✘	✘	✘	✘
	s_1	✘	✔	✔	✘	✘	✘	✘
	s_2	✘	✘	✔	✘	✘	✘	✘
$x \preceq y$		s_0	s_1	s_2	t_0	t_1	t_2	t_3
		y						

x	t_0	✔	✘	✘	✘
	t_1	✔	✔	✔	✘
	t_2	✔	✘	✔	✘
	t_3	✔	✔	✔	✔
$x \preceq y$		t_0	t_1	t_2	t_3
		y			

In case of the (optimized) forwards searching variant, the algorithm is initialized with the state set $K = \{\langle s_0, \{t_0\}\rangle, \langle s_1, \{t_0\}\rangle\}$ *which is depicted in Figure 7.9 by the solid-bordered state pairs. The fact that the state pairs only contain the set* $\{t_0\}$ *but not the set* $\{t_0, t_1\}$ *is due to the simulation. Since* $t_1 \preceq t_0$ *and the forwards searching variant uses only the minimal elements, we can omit the state* t_1*. Now, the algorithm starts with these two initial state pairs and computes the successor states using the function* $\mathsf{RPost}_{\mathcal{A},\mathcal{B}}(K)$*. The result of this function is*

$$\mathsf{RPost}_{\mathcal{A},\mathcal{B}}(\langle s_0, \{t_0\}\rangle, \langle s_1, \{t_0\}\rangle) = \{\langle s_0, \{t_0\}\rangle, \langle s_1, \{t_0\}\rangle, \langle s_2, \{t_2\}\rangle, \langle s_0, \{t_2\}\rangle\}$$

which is depicted in the second row of Figure 7.9. The letters at the arrows indicate by which transition of the respective state pair the successor pair can be reached. The dotted-bordered boxes indicate that the respective state pairs have already been computed in the previous step of the algorithm and must not be processed again. Hence, the new value of K *is the set*

$$K = \{\langle s_2, \{t_2\}\rangle, \langle s_0, \{t_2\}\rangle\}.$$

Due to the change of the value of K*, the algorithm computes the successors of* K *again:*

$$\mathsf{RPost}_{\mathcal{A},\mathcal{B}}(\langle s_2, \{t_2\}\rangle, \langle s_0, \{t_2\}\rangle) = \{\langle s_1, \{t_1\}\rangle, \langle s_1, \{t_0\}\rangle, \langle s_0, \{t_0\}\rangle, \langle s_2, \{t_1\}\rangle\}$$

Due to $\langle s_1, \{t_1\}\rangle \leq \langle s_1, \{t_0\}\rangle$ *the state pair* $\langle s_1, \{t_1\}\rangle$ *can be omitted, which is indicated by the dashed-bordered state pair, and since the state pairs* $\langle s_1, \{t_0\}\rangle, \langle s_0, \{t_0\}\rangle$ *have already been computed, we have that*

$$\lfloor\{\langle s_1, \{t_1\}\rangle, \langle s_1, \{t_0\}\rangle, \langle s_0, \{t_0\}\rangle, \langle s_2, \{t_1\}\rangle\}\rfloor = \{\langle s_2, \{t_1\}\rangle\}.$$

However, the optimized algorithm terminates at this point and returns true*, i. e.* $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$*, since the state pair* $\langle s_2, \{t_1\}\rangle$ *contains the accepting state* s_2 *of* $\mathcal{A}$*, but for the state set* $\{t_1\}$ *of* $\mathcal{B}$ *it holds that* $\overline{\mathrm{F}_{\mathcal{B}}} \preceq^{\forall\exists} \{t_1\}$*, which is indicated by the doubly-bordered box. The word, found by the algorithm, which is a counterexample for the language inclusion, is bb.*

To implement the simulation-based antichain algorithm in a symbolic fashion, we have to take a further look at the simulation relation. The problem is that the algorithm takes only the maximum pairs into account and removes all other pairs. But since the simulation relation is *not* antisymmetric, there could be an arbitrary number of maximum pairs which are all equivalent w. r. t. simulation. If we process the pairs one after another this fact does not lead to any difficulties, but since we want to work with

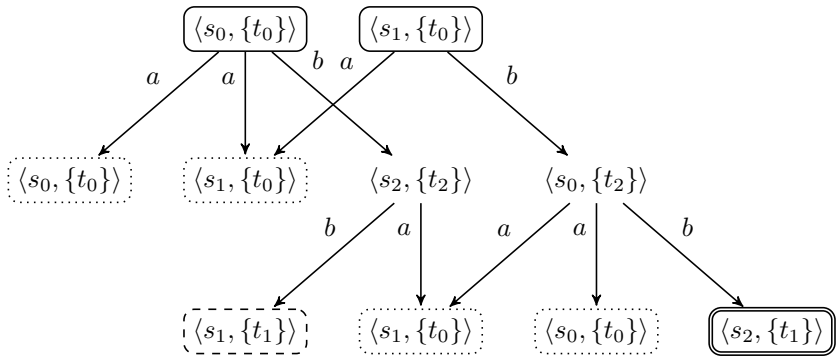


Figure 7.9.: Run of the (optimized) forwards searching simulation-based antichain algorithm on the automata $\mathcal{A}$ and $\mathcal{B}$ (shown in Figure 7.6)

BDDs we want to avoid processing every pair seperately. Therefore, we slightly alter the order given in equation 7.1:

$$\langle q, Q\rangle \leq \langle q', Q'\rangle \quad \text{if } q' \preceq q \text{ and } Q \sqsubseteq^{\forall\exists} Q'. \tag{7.2}$$

where $\sqsubseteq$ denotes an arbitrary antisymmetric order which implies language inclusion and which satisfies the following condition for all states q, q':

$$\text{if } q \preceq q' \wedge q' \preceq q \text{ then } q \sqsubseteq q' \vee q' \sqsubseteq q.$$

Since we require the order $\sqsubseteq$ to be antisymmetric, the condition ensures that for states $q \neq q'$ the order relates either q to q' or q' to q. This guarantees that there is exactly one $\leq$-maximum pair.

7.2.3. Language Equivalence and Bisimulation up to Congruence

In this subsection we present a recent algorithm by Bonchi and Pous [23] which checks the language equivalence of two automata using a technique called bisimulation up to congruence. Their approach is an enhanced version of Hopcroft and Karp's algorithm [85]. This algorithm uses a *coinduction proof principle* [117, 119] for checking the language equivalence of two states of two (not necessarily different) determinisitic finite automata. But in contrast Bonchi and Pous' algorithm does not require the two automata to be deterministic. This advantage relies on a new proof technique called *bisimulation up to congruence*, which extends the *bisimulation up to equivalence*-technique used in the algorithm of Hopcroft and Karp.

In order to explain the different proof techniques, we first introduce the notion of bisimulation in terms of graph automata.

Definition 7.7 (Bisimulation)**.** Let $n \in \mathbb{N}$ and $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}}\rangle$ be an n-bounded graph automaton with $Q_{\mathcal{A}} = (Q_i)_{0\leq i\leq n}$. A *bisimulation (on $\mathcal{A}$)* is a family of relations

$$\sim = \{R_i \mid 0 \leq i \leq n\},$$

where the components $R_i \subseteq Q_i \times Q_i$ are relations such that $p\, R_i\, q$ implies that the following three conditions are satisfied:

- $p \in \mathrm{F}_{\mathcal{A}}$ if and only if $q \in \mathrm{F}_{\mathcal{A}}$,
- for every $\sigma \in \Sigma_{i,j}$ and for every $p' \in \delta_{\mathcal{A}}(p, \sigma)$ there exists a $q' \in \delta_{\mathcal{A}}(q, \sigma)$ such that $p'\, R_j\, q'$, and
- for every $\sigma \in \Sigma_{i,j}$ and for every $q' \in \delta_{\mathcal{A}}(q, \sigma)$ there exists a $p' \in \delta_{\mathcal{A}}(p, \sigma)$ such that $p'\, R_j\, q'$.

Obviously, bisimulation implies language equivalence, i. e. for states $p, q \in Q_{\mathcal{A}}$ with $p \sim q$ we can immediately conclude that $\mathcal{L}(p) = \mathcal{L}(q)$. But note that it is not necessary for a bisimulation to be an equivalence. However, let p, q, r be states such that $p \sim q$ and $q \sim r$, but $p \not\sim r$. From the consideration above, we can conclude that $\mathcal{L}(p) = \mathcal{L}(q) = \mathcal{L}(r)$. An example of a bisimulation which is also an equivalence is the largest bisimulation.

The definition above can be lifted to sets of states, rather than single states, in the following way: Let $P, Q \subseteq Q_i$ be two sets of $\mathcal{A}$-states such that $P \sim Q$ implies

- P is accepting if and only if Q is accepting,
- for every $\sigma \in \Sigma_{i,j}$ it holds that $\hat{\delta}_{\mathcal{A}}(P, \sigma) \sim \hat{\delta}_{\mathcal{A}}(Q, \sigma)$.

Furthermore, we have that for all state sets $P, Q \subseteq Q_{\mathcal{A}}$ that $P \sim Q$ implies $\mathcal{L}(P) = \mathcal{L}(Q)$. Additionally, we have that for all state sets $P, Q \subseteq Q_{\mathcal{A}}$ it holds that $\mathcal{L}(P \cup Q) = \mathcal{L}(P) \cup \mathcal{L}(Q)$. This leads to the following observations:

1. Let $Q, Q', Q'' \subseteq Q_{\mathcal{A}}$ be state sets such that $Q \sim Q'$ and $Q' \sim Q''$, but not $Q \sim Q''$, then we can conclude that $\mathcal{L}(Q) = \mathcal{L}(Q'')$, since we have that $\mathcal{L}(Q) = \mathcal{L}(Q')$ and $\mathcal{L}(Q') = \mathcal{L}(Q'')$.
2. Let $P, P', Q, Q' \subseteq Q_{\mathcal{A}}$ be state sets such that $P \sim P'$ and $Q \sim Q'$, then we have that $\mathcal{L}(P \cup Q) = \mathcal{L}(P' \cup Q')$.

From the first obversation we conclude that a bisimulation does not need to relate Q and Q'' if there exists a state set Q' related to both. This leads us to bisimulations up to equivalence. As a result of the second observation we can conclude that a bisimulation does not need to relate $P \cup Q$ and $P' \cup Q'$, if P (resp. Q) and P' (resp. Q') are related. This leads us to bisimulations up to context. If we take the two up-to techniques together, we obtain bisimulations up to congruence. We now give the explicit definition for the sake of clarity:

Definition 7.8 (Bisimulation up to congruence)**.** Let $n \in \mathbb{N}$ and an n-bounded graph automaton $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}} \rangle$ with $Q_{\mathcal{A}} = (Q_i)_{0 \leq i \leq n}$ be given. A *bisimulation up to congruence (on $\mathcal{A}$)* is a family of relations

$$\sim = \{R_i \mid 0 \leq i \leq n\},$$

where the components $R_i \subseteq \wp(Q_i) \times \wp(Q_i)$ are relations such that $P\, R_i\, Q$ implies that the following three conditions are satisfied:

- $P \cap \mathrm{F}_{\mathcal{A}} \neq \emptyset$ if and only if $Q \cap \mathrm{F}_{\mathcal{A}} \neq \emptyset$,
- for every $\sigma \in \Sigma_{i,j}$, it holds that $P' = \hat{\delta}_{\mathcal{A}}(P, \sigma)$ and $Q' = \hat{\delta}_{\mathcal{A}}(Q, \sigma)$ are related by $c(\sim)$, i. e. $P'\ c(\sim)\ Q'$,

where $c(\sim)$ denotes the smallest equivalence which is closed with respect to $\cup$ and which includes $\sim$.

We now present the algorithm which checks whether two given graph automata are language equivalent. But first, we need a further function

$$\mathrm{isFinal}_{\mathcal{A}} \colon \wp(Q_{\mathcal{A}}) \to \{\mathit{false}, \mathit{true}\}$$
$$\mathrm{isFinal}_{\mathcal{A}}(Q) = \begin{cases} \mathit{false}, & \text{if } Q \cap \mathrm{F}_{\mathcal{A}} = \emptyset \\ \mathit{true}, & \text{else} \end{cases}$$

The function does the following: it returns *true* if and only if the given state set is accepting, otherwise *false* is returned.

Forwards searching algorithm. The forwards searching algorithm, which returns *true* if and only if $\mathcal{L}(\mathcal{A}) \neq \mathcal{L}(\mathcal{B})$, is then defined as follows:

Algorithm 7.4: Forwards searching Bisimulation up to Congruence Algorithm

Input: $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, \mathrm{I}_{\mathcal{A}}, \mathrm{F}_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, \mathrm{I}_{\mathcal{B}}, \mathrm{F}_{\mathcal{B}} \rangle$
Output: *true* if $\mathcal{L}(\mathcal{A}) \neq \mathcal{L}(\mathcal{B})$, *false* otherwise

```
1:  ∼ ← ∅
2:  todo ← {⟨I_A, I_B⟩}
3:  repeat
4:      extract ⟨P, Q⟩ from todo
5:      if ⟨P, Q⟩ ∈ c(∼ ∪ todo) then skip
6:      if isFinal_A(P) ≠ isFinal_B(Q) then return true
7:      for all σ ∈ Σ do
8:          todo ← todo ∪ {⟨δ̂_A(P, σ), δ̂_B(Q, σ)⟩}
9:      ∼ ← ∼ ∪ {⟨P, Q⟩}
10: until todo is empty
11: return false
```

The line $\langle P, Q \rangle \in c(\sim \cup \mathit{todo})$ checks whether the pair $\langle P, Q \rangle$ is already related by the bisimulation up to congruence based on the current bisimulation $\sim$. If this is the case it is not necessary to add the pair to the bisimulation. At all times it holds that for all $\langle P, Q \rangle \in \sim$ there is no word $\vec{\sigma}$ such that $\hat{\delta}_{\mathcal{A}}(P, \vec{\sigma})$ is accepting and $\hat{\delta}_{\mathcal{B}}(Q, \vec{\sigma})$ is not accepting or vice versa.

Note that the algorithm given above slighty differs from the original algorithm (presented in [23]) in that it returns *true* if and only if the two automata are *not* language equivalent. This modification is made to have the same result as the algorithms presented in the previous subsections. For the correctness we refer to [23].

The practicability of the algorithm above highly depends on an efficient way to check whether the state sets of some pair (in *todo*) are related or not w. r. t. bisimulation up

to congruence. The solution is to compute the largest representative, i. e. the largest set, of the involved equivalence classes. This can be done by considering every pair $\langle P, Q\rangle$ in the relation $\sim$ as rewriting rules of the form $P \to P \cup Q$ and $Q \to P \cup Q$. We formalize this as follows: Let $n \in \mathbb{N}$ and $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, \mathrm{I}_\mathcal{A}, \mathrm{F}_\mathcal{A}\rangle$ be an n-bounded graph automaton with $Q_\mathcal{A} = (Q_i)_{0\leq i\leq n}$ and let $R = \{R_i \mid 0 \leq i \leq n\}$ be a family of relations where each R_i is a binary relation on Q_i. We define

$$\rightsquigarrow_R = \{S_i \mid 0 \leq i \leq n\}$$

to be a family of relations such that each $S_i \subseteq Q_i \times Q_i$ is the smallest irreflexive relation that satisfies the following conditions:

- For every pair $\langle P, Q\rangle \in S_i$ it holds $\langle P, P \cup Q\rangle, \langle Q, P \cup Q\rangle \in \rightsquigarrow_R$ and
- for every pair $\langle P, Q\rangle \in S_i$ it holds $\langle P \cup Q', Q \cup Q'\rangle \in \rightsquigarrow_R$ for all $Q' \subset Q_i$.

Furthermore, we denote the *normal form* of a state set $Q \subseteq Q_i$ w. r. t. $\rightsquigarrow_R$, i. e. the largest set of its equivalence class, by $Q{\downarrow_R}$. The key to efficiency is the following proposition:

Proposition 7.9 (Bonchi, Pous [23])**.** Let $n \in \mathbb{N}$ and $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F}\rangle$ be an n-bounded graph automaton with $Q = (Q_i)_{0\leq i\leq n}$ and let

$$R = \{R_i \mid 0 \leq i \leq n\}$$

be a family of relations where each R_i is a binary relation on Q_i. For all $P, Q \subseteq Q_i$ it holds $P{\downarrow_R} = Q{\downarrow_R}$ if and only if $\langle P, Q\rangle \in c(R)$.

The bisimulation up to congruence algorithm can be implemented on Bdds straightforwardly. Each pair of state sets $\langle P, Q\rangle$ can be mapped to a single Bdd such that the odd-numbered Bdd nodes represent the first state set P and the even-numbered Bdd nodes represent the second state set Q (the interleaving is due to encoding as explained in Section 7.1). The bisimulation up to congruence is then (explicitly) represented by a set of such Bdds. This way, the computation of the normal form w. r. t. $\rightsquigarrow_R$ can be done by the following method which is proposed by Bonchi and Pous [23]: Let $\langle P, Q\rangle$ be a pair of state sets for which the normal w. r. t. $\rightsquigarrow_R$ should be generated. For each pair $\langle U, V\rangle \in R$ we try to perform the following rewriting steps one after another:

1. Rewriting of P:
   - If $U \subseteq P$, apply the rule $U \to U \cup V$ to the set P which yields the pair $\langle P \cup V, Q\rangle$,
   - else if $V \subseteq P$, apply the rule $V \to U \cup V$ to the set P which yields the pair $\langle P \cup U, Q\rangle$.
2. Rewriting of Q:
   - If $U \subseteq Q$, apply the rule $U \to U \cup V$ to the set P which yields the pair $\langle P, Q \cup V\rangle$,
   - else if $V \subseteq Q$, apply the rule $V \to U \cup V$ to the set P which yields the pair $\langle P, Q \cup U\rangle$.

The rewriting can easily be implemented on BDDs due to the following observations: We explain the idea at the example of the first case given above. First, the "subset check", i. e. whether $U \subseteq P$ holds or not, can be performed on BDDs by checking that the implication between the BDD representing the set U and the BDD representing the set P is a tautology. Second, the rewriting can be performed by computing the disjunction of the two BDDs representing the set P and the set V. The other cases can be handled analogously. Hence, tests for both state sets can be efficiently implemented with BDDs. Finally, the test $\text{isFinal}_{\mathcal{A}}(P) \neq \text{isFinal}_{\mathcal{B}}(Q)$ can be easily obtained by the BDD operations $\wedge$, $\leftrightarrow$ as well as the existential quantification of BDDs and the final state sets of the underlying automata which are also represented as BDDs.

As a special case the algorithm can be used to solve the language inclusion problem for two automata $\mathcal{A}$ and $\mathcal{B}$. This is based on the following fact:

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \iff \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}).$$

Example 7.10. *Let the two non-deterministic finite automata $\mathcal{A}$ and $\mathcal{B}$ from Example 7.2, which where depicted in Figure 7.6, be given. Once again we want to prove that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ does not hold. In this example we use the bisimulation up to congruence algorithm explained above. But before we can start the algorithm, we need to construct the union automaton of $\mathcal{A}$ and $\mathcal{B}$, i. e. we construct the automaton $\mathcal{C}$ which accepts the languages $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$. As explained in Section 6.1.3 this can be done efficiently. In the following, we denote by $\sim_i$ and $todo_i$ the relations which have been computed during the i-th round of the algorithm.*

In case of the (optimized) forwards searching variant, the algorithm is initialized with the relations $\sim_0 = \emptyset$ and $todo_0 = \{\langle\{s_0, s_1, t_0, t_1\}, \{t_0, t_1\}\rangle\}$ which is depicted in Figure 7.10 by the solid-bordered pairs. Due to the fact that there is currently only one pair contained in the relation $todo_0$ the algorithm extracts the pair $\langle\{s_0, s_1, t_0, t_1\}, \{t_0, t_1\}\rangle$. Since the relations $\sim_0$ and $todo_0$ are now both empty and $\{s_0, s_1, t_0, t_1\} \neq \{t_0, t_1\}$, the algorithm can skip the test in line 5. Furthermore, neither $\{s_0, s_1, t_0, t_1\}$ nor $\{t_0, t_1\}$ are accepting state sets, the algorithm continues with line 8 and computes the successor states of these two state sets as follows,

$$\hat{\delta}_{\mathcal{C}}(\{s_0, s_1, t_0, t_1\}, a) = \{s_0, s_1, t_0, t_1\}, \qquad \hat{\delta}_{\mathcal{B}}(\{t_0, t_1\}, a) = \{t_0, t_1\},$$
$$\hat{\delta}_{\mathcal{C}}(\{s_0, s_1, t_0, t_1\}, b) = \{s_0, s_2, t_1, t_2, t_3\}, \qquad \hat{\delta}_{\mathcal{B}}(\{t_0, t_1\}, b) = \{t_1, t_2, t_3\},$$

which yields the pairs $\langle\{s_0, s_1, t_0, t_1\}, \{t_0, t_1\}\rangle$ and $\langle\{s_0, s_2, t_1, t_2, t_3\}, \{t_1, t_2, t_3\}\rangle$. Since the first pair has already been processed, we can skip it. The other pair is added to the relation $todo_1$ and the pair which has been extracted in this round is added to the relation $\sim_1$. Hence we have the following situation:

$$\sim_1 = \{\langle\{s_0, s_1, t_0, t_1\}, \{t_0, t_1\}\rangle\}, \qquad todo_1 = \{\langle\{s_0, s_2, t_1, t_2, t_3\}, \{t_1, t_2, t_3\}\rangle\}.$$

The algorithm continues with the second round and extracts the only pair in $todo_1$: $\langle\{s_0, s_2, t_1, t_2, t_3\}, \{t_1, t_2, t_3\}\rangle$. In order to test whether

$$\{s_0, s_2, t_1, t_2, t_3\}\ c(\sim_1 \cup todo_1)\ \{t_1, t_2, t_3\},$$

the algorithm computes and checks $\{s_0, s_2, t_1, t_2, t_3\}{\downarrow_{\sim_1}} \stackrel{?}{=} \{t_1, t_2, t_3\}{\downarrow_{\sim_1}}$. But since no rule is applicable to either of both sides and $\{s_0, s_2, t_1, t_2, t_3\} \neq \{t_1, t_2, t_3\}$, the test is

negative and the algorithm continues with line 6. The test on this line is also negative, because both state sets are accepting. Therefore the algorithm computes the successor states of these two state sets,

$$\hat{\delta}_{\mathcal{C}}(\{s_0, s_2, t_1, t_2, t_3\}, a) = \{s_0, s_1, t_0, t_1, t_3\}, \qquad \hat{\delta}_{\mathcal{B}}(\{t_1, t_2, t_3\}, a) = \{t_0, t_1, t_3\},$$
$$\hat{\delta}_{\mathcal{C}}(\{s_0, s_2, t_1, t_2, t_3\}, b) = \{s_0, s_2, t_1, t_3\}, \qquad \hat{\delta}_{\mathcal{B}}(\{t_1, t_2, t_3\}, b) = \{t_1, t_3\},$$

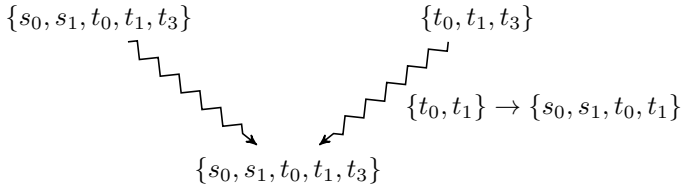
and adds the pairs $\langle\{s_0, s_1, t_0, t_1, t_3\}, \{t_0, t_1, t_3\}\rangle$ *and* $\langle\{s_0, s_2, t_1, t_3\}, \{t_1, t_3\}\rangle$ *to the relation* $todo_2$*. The relation* $\sim_2$ *is updated by the pair* $\langle\{s_0, s_2, t_1, t_2, t_3\}, \{t_1, t_2, t_3\}\rangle$*. This leads to the following situation:*

$$\sim_2 = \{\langle\{s_0, s_1, t_0, t_1\}, \{t_0, t_1\}\rangle, \langle\{s_0, s_2, t_1, t_2, t_3\}, \{t_1, t_2, t_3\}\rangle\},$$
$$todo_2 = \{\langle\{s_0, s_1, t_0, t_1, t_3\}, \{t_0, t_1, t_3\}\rangle, \langle\{s_0, s_2, t_1, t_3\}, \{t_1, t_3\}\rangle\}.$$

In the third round, the algorithm can choose between two pairs. We assume that it chooses the pair $\langle\{s_0, s_1, t_0, t_1, t_3\}, \{t_0, t_1, t_3\}\rangle$ *first and extracts it from* $todo_2$*. Once again it must be tested whether*

$$\{s_0, s_1, t_0, t_1, t_3\}\ c(\sim_2 \cup todo_2)\ \{t_0, t_1, t_3\}.$$

Both state sets have the same normal form (w. r. t. $\rightsquigarrow_{\sim_2 \cup todo_2}$*) as we can see below. The set on the right can be rewritten to the set on the left by the rewriting rule* $\{t_0, t_1\} \to \{s_0, s_1, t_0, t_1\}$*. Therefore, the pair can be skipped.*



This yields the following situation:

$$\sim_3 = \{\langle\{s_0, s_1, t_0, t_1\}, \{t_0, t_1\}\rangle, \langle\{s_0, s_2, t_1, t_2, t_3\}, \{t_1, t_2, t_3\}\rangle\},$$
$$todo_3 = \{\langle\{s_0, s_2, t_1, t_3\}, \{t_1, t_3\}\rangle\}.$$

In the next round, the algorithm extracts $\langle\{s_0, s_2, t_1, t_3\}, \{t_1, t_3\}\rangle$ *which is the only pair in* $todo_3$*. The test in line 5 whether*

$$\{s_0, s_2, t_1, t_3\}\ c(\sim_3 \cup todo_3)\ \{t_1, t_3\}$$

can be skipped, since there is no applicable rule (obtainable from $\sim_3$*) and* $\{s_0, s_2, t_1, t_3\} \neq \{t_1, t_3\}$*. But this time the test in line 6 is positive, because the state set* $\{s_0, s_2, t_1, t_3\}$ *is accepting, but the state set* $\{t_1, t_3\}$ *is not. Hence, the algorithm terminates and returns* true*. The word found by the algorithm, which is a counterexample for the language inclusion, is again bb.*

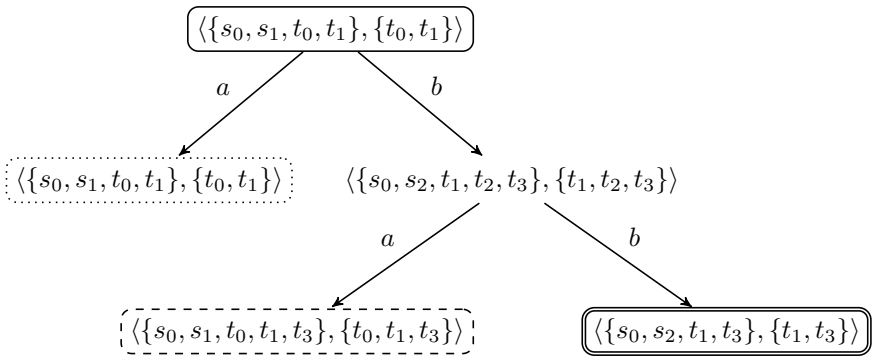


Figure 7.10.: Run of the (optimized) forwards searching bisimulation up to congruence algorithm on the automata $\mathcal{A}$ and $\mathcal{B}$

7.3. Conclusion

In this chapter we explained how graph automata could be implemented by means of binary decision diagrams (BDDs). The main challenge here is to find good state encodings in the sense that the achieved BDDs are as small as possible (in the number of used BDD nodes) and that the runtime which is needed to compute these BDDs is as short as possible. We have seen, using the example of the 5-dominating set graph automaton, that it is in general not possible to optimize both parameters. Hence, one has to find a compromise between small BDDs and short runtimes.

Furthermore, we have introduced three different algorithms which can be used to solve the language inclusion problem. We will also use these algorithms to check whether a given recognizable graph language is an invariant for some graph transformation rule. For this purpose we showed that the presented algorithms could be implemented symbolically for the use with graph automata which are also implemented symbolically.

In the next chapter, we will present our tool suite RAVEN, which uses the techniques explained in this chapter.

Part III.

Applications/Tools

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Ervin Knuth (1938 – present)

8

Raven – A Verification Tool Suite Based on Recognizability

In this chapter we describe the tool suite Raven, which is an important achievement and one of the goals of this thesis. Raven has been developed to handle and manipulate graph automata of bounded interface size as described in Chapter 6. Furthermore, we implemented the techniques presented in Chapter 7 in order to perform invariant checks, which can be used for the verification of graph transformation systems.

8.1. Brief Description of Raven

The implementation of Raven[1] started in 2008 with a tool which implements the graph automaton accepting all graphs containing a given subgraph (see Example 6.34) [10]. The application of that tool was to check whether the language (accepted by the implemented graph automaton) is an invariant for a given graph transformation rule by computing the Myhill-Nerode quasi-order (see Definition 6.20) and checking whether the left-hand side of the transformation rule is related to the right-hand side (w. r. t. the Myhill-Nerode quasi-order).

This tool had two main drawbacks which led to the development of Raven. On the one hand the graph automaton had been implemented in an explicit fashion. As already described in Chapter 7, the number of states grows exponentially in the size of the maximum permitted interface of the graph automaton, which has a direct impact on the size of the explicit representation. As shown by practical examples, it is not possible to compute graph automata which exceed a rather small maximum interface size. On the other hand one needs to build the deterministic graph automaton to be able to compute the Myhill-Nerode quasi-order. But since the computation of the

[1] Available at http://www.ti.inf.uni-due.de/research/tools/raven/

deterministic graph automaton (by the powerset construction) is not feasible due to the exponential blow-up, we used the simulation quasi-order (see Definition 7.3) as an over-approximation. This quasi-order had also been represented explicitly which led to the problem that the simulation quasi-order had been computable only for graph automata up to a maximum interface size of 4, even for rather simple subgraphs [15].

To overcome these issues the development of Raven has been started. The key to an efficient new tool has been the usage of Bdds as data structures to represent graph automata as explained in Chapter 7. In addition to the algorithms for checking language inclusion introduced in the previous chapter, we have also implemented some other algorithms which comprise universality checks (based on the algorithms presented in Chapter 7), emptiness checks, memberships checks and algorithms to compute atomic cospan decompositions (cf. Chapter 5) for a given cospan (which includes graphs seen as cospans with empty inner and outer interfaces).

8.2. System Architecture of Raven

In this section we will describe the system architecture of Raven which is depicted in Figure 8.1.

The architecture of Raven can roughly be divided into six parts:

- the input components (depicted on the left of Figure 8.1) which are again split up into two groups: the *user interface* and the *file readers*,
- the *repository* (depicted in the center) which is one of the core components of Raven, that is a database for all current objects,
- the *decomposer* unit (depicted on the bottom) which is used to transform graphs and cospans to equivalent atomic cospan decompositions,
- the *goal* components (depicted in the middle around the repository) which are also core components providing different techniques to perform universality, language inclusion, invariant, emptiness and membership checks,
- the *algorithms* unit (depicted on the top) which is used by the different goal components,
- the output components (depicted on the right) which are also split up into two groups: the *user interface* and the *file writers*.

In the following we will describe the components in detail.

The system starts by reading in the user's input. Depending on which goals the user wants to achieve different data structures must be provided. The data structures which can be handled by Raven are:

- *Graphs* can either be directly created by the user via the user interface or by loading a file in Gxl format[2] (see Appendix C.1),

[2] Gxl is a Xml-based standard exchange language for different kinds of graphs widely spread in the graph transformation community [84, 94, 129], see also http://www.gupro.de/GXL/

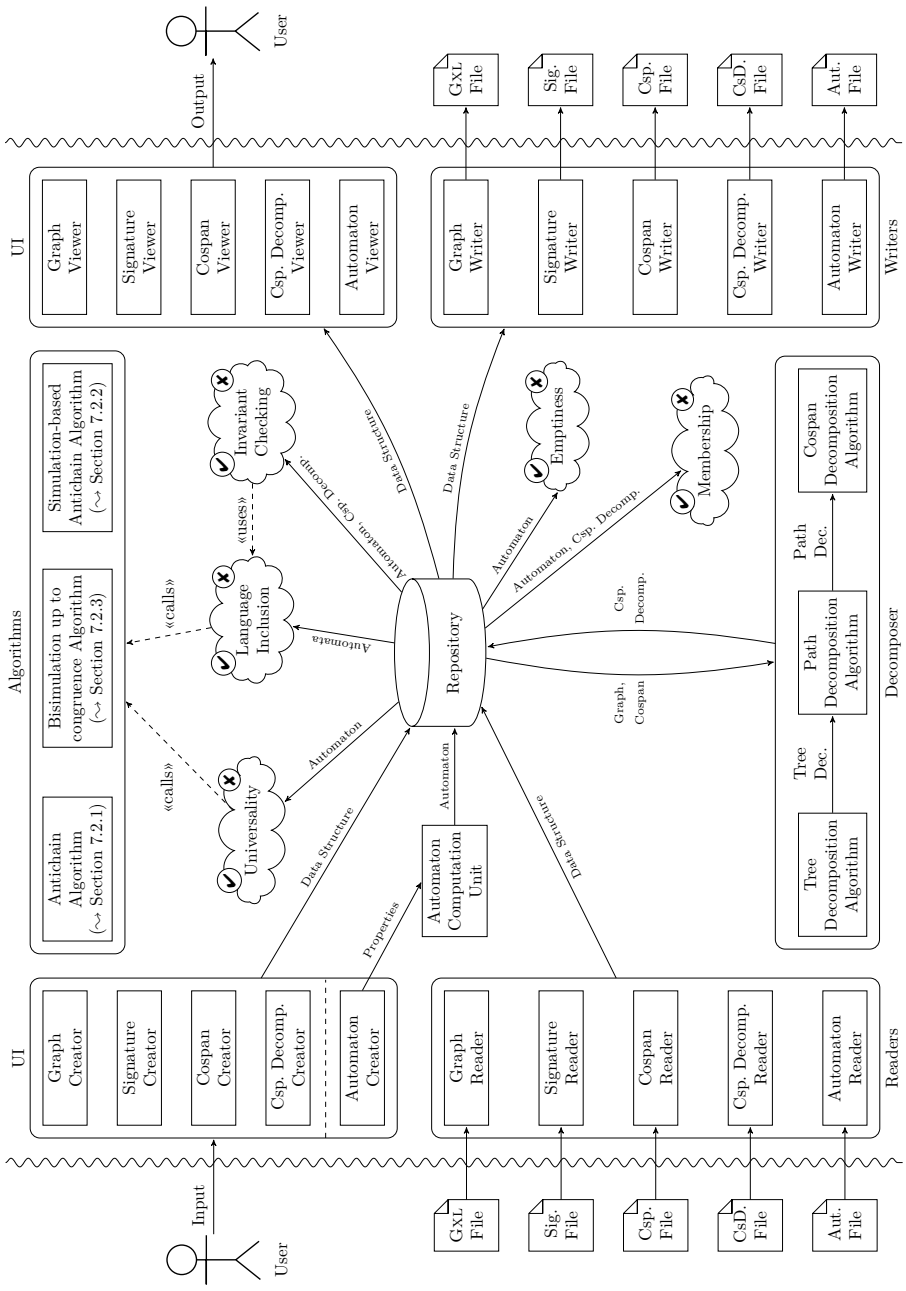


Figure 8.1.: System Architecture of Raven

- *Signatures* contain strings representing the letters of the alphabet $\Sigma(\Lambda)$ (see Table 6.1) which are used to define the input alphabet of the different graph automata. Signatures can also be created in the user interface or by loading a file in a special line-oriented format,
- *Cospans* can be produced through the user interface or by loading a file in Gxl format (also see Appendix C.2),
- *(Atomic) Cospan Decomposition* can either be generated by the user calling the decomposer unit on a graph or a cospan, which will be explained in detail below, or by loading a file in a special line-oriented format,
- *Graph Automata* can be created by choosing an automaton type (out of a list of predefined automata, also see Section 7.1) and defining the automaton's properties, such as the permitted inner, outer and maximum interface and some further specific properties, or by loading a graph automaton from a file (see Appendix C.3 for further information).

After the data structures have been read, every object is stored under a certain name in the repository. Later on, the user can use the objects in the repository by handing over the names of the desired objects to the several goals.

Some of the goals, namely the membership and the invariant checking goal, expect the user to input an atomic cospan decomposition. Hence, Raven provides the opportunity to automatically decompose a graph or a cospan into such an atomic cospan decomposition. This is done in several steps depending on the object one wants to decompose:

- In case of graphs, the first step is to compute a tree decomposition of the graph (see Definition 5.1). This is done by two algorithms. The first one is a heuristic which computes a list of the nodes of the graph depending on different criteria [20]. The second algorithm generates the actual tree decomposition depending on the node list. The heuristic to find the node list can either be chosen by the user or the default heuristic, called GreedyDegree in the literature, is chosen.

  In the second step, the tree decomposition (which is essentially a tree) is traversed to obtain a linearization of the decomposition. In order to get a path decomposition, one needs to perform further operations, since it could be the case that the bags (of the linearization) containing a given node (of the graph) do not form a path, which would be a violation of the definition of a path decomposition. Therefore, it is checked for every graph node and for every path from one bag to another bag which both contain the given node, whether the node is also contained in all bags on the path. If this is not the case, the node is added to all bags on the path which do not contain the node. This yields a valid path decomposition of the graph.

  In the last step, the path decomposition is transformed into an atomic cospan decomposition. This is done in a "bag-wise" manner, i. e. the algorithm processes the path decomposition bag by bag based on the procedure described in the proof of Lemma 5.9.

- In case of cospans, five different steps are required to obtain an atomic cospan decomposition. The idea is based on (the proof of) Proposition 6.25. The first step is to add a *vertex*-operation for each node in the middle graph of the given cospan. Note that there are additional nodes which are accessible from the inner interface (of the given cospan). In the second step all edges of the middle graph are added by permuting the incident nodes to the last position of the outer interface with the help of *shift*- and *trans*-operations and a subsequent $connect_{\diamond}$-operation, where $\diamond$ is just a placeholder for the corresponding label. Next, in step three, the nodes added in step one are fused with the already existing nodes according to the inner interface (of the given cospan). At this point the middle graph obtained by the atomic cospan decomposition matches the middle graph of the given cospan. Hence, in step four, we can permute the outer interface (of the atomic cospan decomposition) by adding *shift*- and *trans*-operations repeatedly such that the nodes which are not accessible by the outer interface of the given cospan allocate the last positions of the outer interface of the atomic cospan decomposition. Then, the fifth and last step is to remove these last nodes from the outer interface by adding enough *res*-operations. This yields the desired atomic cospan decomposition.

An object in the repository can be used as input for the different goal components:

- The *universality* method checks whether the given graph automaton is universal, i. e. whether the accepted language of the $\langle i, j\rangle$-graph automaton contains all cospans of the form $c\colon D_i \nrightarrow D_j$. In case that the given graph automaton is *not* universal, a counterexample is returned.

- The *language inclusion* goal expects two $\langle i, j\rangle$-graph automata and checks whether the language of the first graph automaton is contained in the language of the second graph automaton. In case the language inclusion does not hold, each algorithm computes a counterexample and returns it to the user.

- The *invariant checking* goal checks whether the language of a given $\langle 0, j\rangle$-graph automaton is an invariant for a graph transformation rule $\rho = \langle \ell, r\rangle$ given as two cospans of the form $\ell, r\colon \emptyset \nrightarrow D_i$ (see Section 3.3). As a pre-processing step the $\langle i, j\rangle$-graph automata $\mathcal{A}[\ell]$ and $\mathcal{A}[r]$ are computed as described in Section 7.2. The check is then based on the language inclusion goal for the graph automata $\mathcal{A}[\ell]$ and $\mathcal{A}[r]$. Again, if the language of the given graph automaton is not an invariant for the given transformation rule, a counterexample is computed and presented to the user.

- The *emptiness checking* goal expects a graph automaton and checks whether the language of the automaton is empty.

- The *membership checking* goal runs a given $\langle i, j\rangle$-graph automaton on a given cospan of the form $c\colon D_i \nrightarrow D_j$ and checks whether the cospan is accepted by the graph automaton.

If the language inclusion goal has been chosen, one of the algorithms presented in the Sections 7.2.1, 7.2.2 and 7.2.3 is called to solve the language inclusion problem.

Note that in case of the simulation-based antichain algorithm, the simulation pre-order for the given automata is computed as a pre-processing step.

If the user has chosen the universality goal, there exist algorithms which are essentially the same as for the language inclusion goal (see [1, 23, 132] for further information about these universality algorithms).

If the invariant checking goal has been selected by the user, the input is transformed as described above and afterwards the chosen language inclusion algorithm is called.

The other two goals are directly implemented on top of the underlying data structures, which are given as input, i. e. both the emptiness and the membership check goal call methods which are implemented in the graph automaton datastructure. Therefore, no further algorithms are needed here.

At last, the system can either visualize the data structures contained in the repository using the different viewers or write the data structures in the different file formats which have been described above.

8.3. Tutorial: Functionality and Usage

The tool suite Raven has been implemented in the Java programming language[3] and offers both a command-line and a graphical user interface. Furthermore, Raven depends on a number of libraries/programs which are listed in the following:

- JavaBDD, a Java library for manipulating Bdds which offers an interface to the well-known BuDDy library [128]
- BuDDy, a highly efficient Bdd library written in C [101]
- LibTW, a Java library for computing tree decompositions of graphs [54]
- ANTLR, a parser generator library written in Java,
- JDOM, a library for reading, manipulating and writing Xml documents written in Java
- JGraphX, a Java library for visualizing graphs
- JAnsi, a library written in Java to use ANSI escape sequences to format console outputs
- Graphviz, a program for automatically layouting graphs

All the libraries and programs given above are free software (also see Appendix D for further information). Since Raven as well as many of the libraries above are written in Java and since BuDDy as well as Graphviz are available for many platforms, Raven can be used on Linux, MacOS and Windows. The full description can be found in the Raven program documentation[4].

In the rest of this section we give a short tutorial to the main features of Raven. But note that we will limit our attention to the graphical user interface. The program can be called from the shell by the following command

[3] Raven depends on a Java Runtime Environment 1.7 or higher.
[4] Available at http://www.ti.inf.uni-due.de/research/tools/raven/documentation.pdf

```
java -jar raven.jar
```

which brings up the main window of RAVEN (see Figure 8.2). This window consists of five main componentes:

① The *goal panel* on which the user can choose the different goals (membership check, language inclusion check, ...) to use with RAVEN,

② The *output panel* which shows the user text-based status information about RAVEN,

③ The *command-line input field* which provides the user with the opportunity to directly use console commands in the graphical user interface,

④ The *repository panel* which presents a list of all data objects (signatures, cospans, ...) which are currently contained in the repository,

⑤ The *data information panel* which gives detailed information about the currently selected data object.

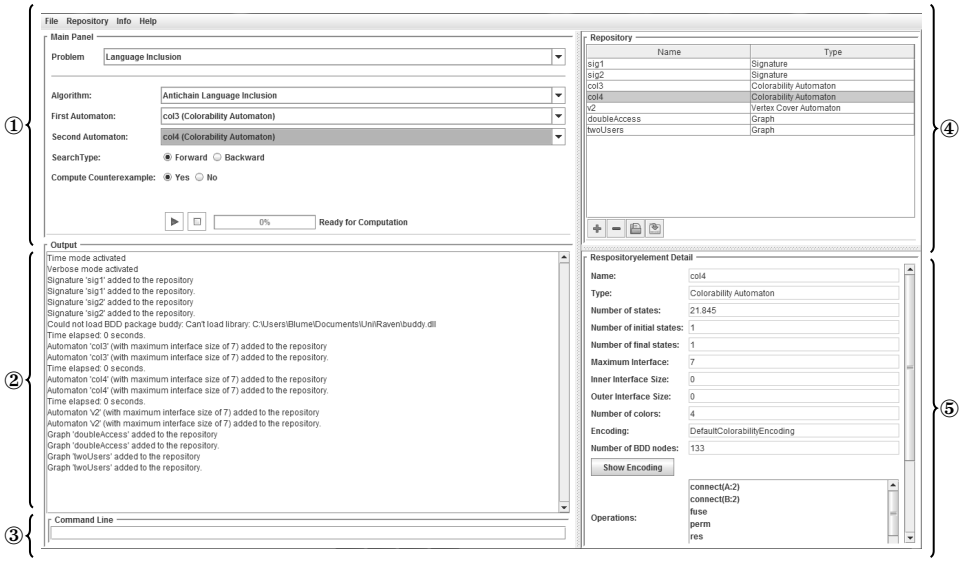


Figure 8.2.: RAVEN GUI

We start by creating some data structures which will be used later on for the analysis. First, we create a new graph and add it to the repository. This can be done by choosing either `Repository -> Create graph` from the main menu or the item `Create graph` which appears when pressing the button (✚) labeled with a green plus sign located on the repository panel (④). The *visual graph creator* component appears, which is depicted in Figure 8.3. The use is rather intuitive. In the text field on the top side of the window, the user has to define a (unique) name for the new graph which is used to address the graph after it has been added to the repository. By pressing the `Add`

`node` button the user can add new nodes to the graph. A click on the `Add edge` button brings up a new dialog window on which the user can specify the (unique) name, the label and incident nodes of a new edge. The graph is then added to the repository by pressing the `Accept` button.

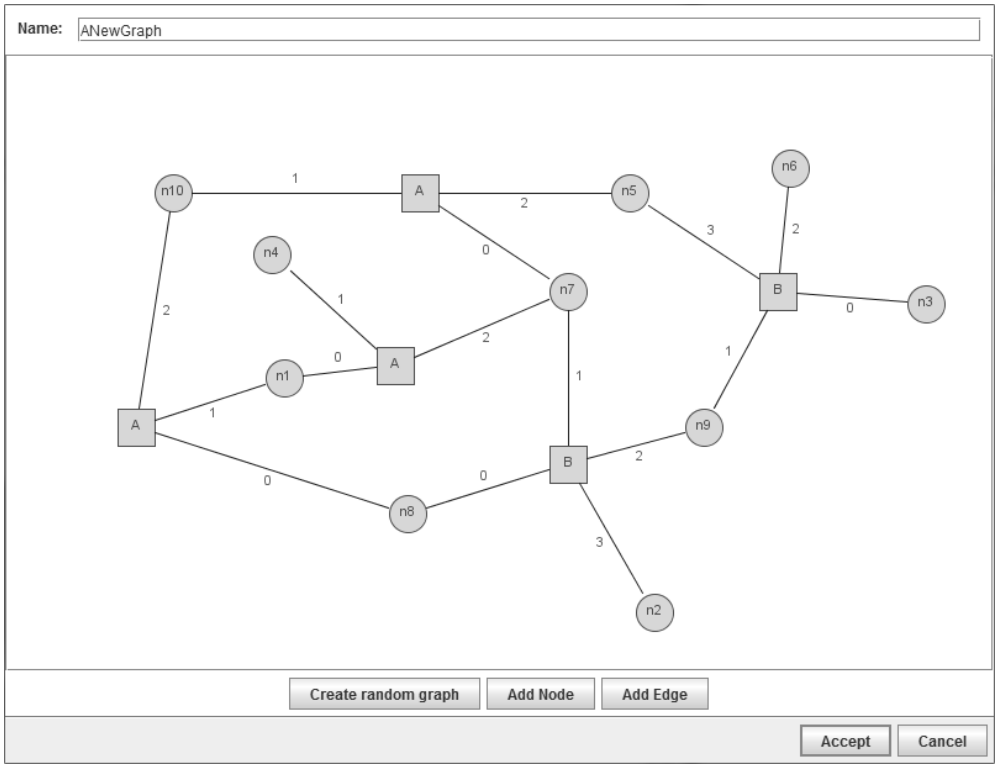


Figure 8.3.: Raven Visual Graph Creator

Next, we want to add a graph automaton. But before we can do this, we need to add a signature to define the automaton's input alphabet. Again, we either choose `Repository -> Create signature` from the main menu or the item `Create signature` which appears when pressing the button (✚) labeled with a green plus sign located on the repository panel. The *visual signature creator* appears, which is shown in Figure 8.4. First of all, the user has to define a (unique) name for the new signature. Subsequently the user can add the letters which should be contained in the signature by pressing the button (✚) labeled with a green plus sign. The available letters consist of the letters introduced in Chapter 6: $connect_\diamond$ (where $\diamond$ is a placeholder for an arbitrary label which can be determined by the user), *fuse*, *perm*, *res*, *trans* and *vertex*. If the user wants to remove a letter, this can be done by selecting the respective letter and pressing the button (▬) labeled with a red minus sign. Once all operations have been added to the signature, the user can press the `Accept` button and add it to the repository.

Now, we can create a new graph automaton. As before, the *visual automaton creator* dialog can be invoked either by the main menu, choosing `Repository -> Create`

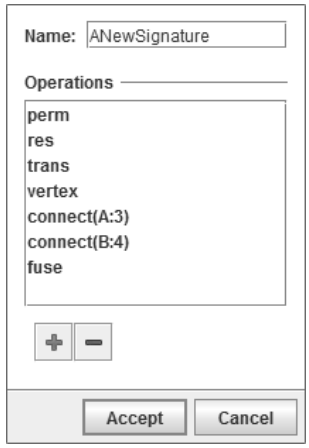


Figure 8.4.: RAVEN Visual Signature Creator

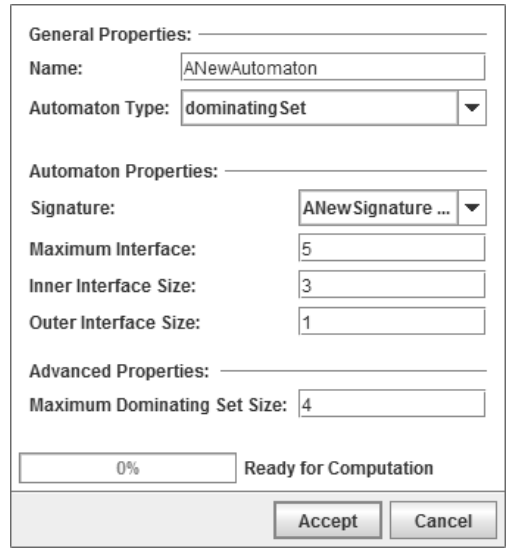


Figure 8.5.: RAVEN Visual Automaton Creator

`graph automaton`, or by pressing the button (✚) labeled with a green plus sign located on the repository panel and selecting the `Create automaton` item. Either way the visual automaton creator, which is depicted in Figure 8.5, is shown to the user. Analogously to the dialogs explained above, first of all the user has to define a (unique) name for the new graph automaton. Next, the user can choose the type of the graph automaton, which specifies the language accepted by the automaton. The list (at the time of writing this thesis) consists of types which represents the languages presented in Section 7.1. Therefore, the type must be one of the following: *colorability*, *dominating set*, *edge counting*, *link*, *maximum edge*, *maximum vertex*, *minimum edge*, *minimum vertex*, *no isolated nodes*, *path*, *product*, *subgraph*, *union*, *vertex cover*, *vertex counting*. Furthermore, the user has to specify the signature used by the new graph automaton as well as the inner, outer and maximum interface size of the cospans accepted by the automaton. However, it is recommended to choose the maximum interface size as low as possible, since the size (and therefore the time of computation) of a graph automaton depends exponentially on the maximum interface size, as explained in Section 7.1. At last, the user has to define some type-specific properties such as the number of colors in case of the colorability type or the maximum size of the dominating set in case of the dominating set type, et cetera. If all required properties have been given, the user can start the computation of the new graph automaton by clicking the `Accept` button. When the computation is finished, the graph automaton is added to the repository.

The last data structure we want to create is a new cospan. For this purpose the user has two possibilities in RAVEN. The first is to directly create a new cospan by creating the middle graph of the cospan and defining which nodes are in the inner and which ones are in the outer interface. The second is to give the cospan in terms of an atomic cospan decomposition (see Chapter 5). The advantage of the latter option is

that the user has more influence on the cospan decomposition, since no decomposition must be computed. The disadvantage is that the user has to know the decomposition of the desired cospan. In order to create a new cospan the user has to choose either `Repository -> Create cospan` from the main menu or the button (✚) labeled with a green plus sign located on the repository panel and selecting the `Create cospan` item. The *visual cospan creator* dialog appears which is shown in Figure 8.6. The use of this dialog is similar to that of the visual graph creator. Therefore, we omit the explanation of the visual cospan creator as far as possible. The only difference is that the user can add nodes of the middle graph to the inner and outer interface respectively. This can be done by selecting the certain node and clicking on the buttons labeled with the different arrows. To create a new cospan decomposition the user has to choose the `Create cospan decomposition` item instead of the `Create cospan` item. This brings up the *visual cospan decomposition creator* dialog, which can be seen in Figure 8.7. First of all, the user has to define the (unique) name of the cospan decomposition. Afterwards the user can compose the new atomic cospan decomposition out of the list of available atomic cospans which are depicted on the upper left of the dialog window by selecting the desired atomic cospan and pressing the arrow buttons. Every time a new atomic cospan is added to the decomposition, the cospan list, depicted in the upper middle of the dialog, and the cospan view, depicted in the bottom part of the dialog, are updated. Furthermore, the user is provided with the current (outer) interface of the cospan by the list on the right. Once the atomic cospan decomposition is complete, the user can hit the `Accept` button such that the decomposition is added to the repository.

After the user has created some data structures, the user can get further information about these objects by selecting them on the repository panel (④). The details about the selected object are then depicted on the data information panel (⑤). For example, if the user selects a graph automaton, the depicted details consist of properties which have been used during the creation and of further information such as the number of (all/the initial/the final) states of the automaton, the name of the Bdd encoding (see Section 7.1) or the (visual representation of the) Bdds used to encode the several transitions functions of the graph automaton. For the other data structures the user can also get additional information by selecting an appropriate object on the repository panel.

Now, we turn to the analysis of the graph automaton created beforehand. We assume that the user has created a graph with the help of the visual graph creator. As an example we want to check whether the given graph is accepted by the graph automaton, i. e. we want to solve the membership problem. To do so, the user has to select the `Membership` goal from the goal panel (①) first. Then the desired graph automaton and the desired graph have to be selected from the respective lists which appear on the goal panel. To start the computation of the underlying membership algorithm the user has to hit the button (▶) labeled with a green triangle. In a similar way, the other goals can be chosen and started by the user. Therefore, we omit the description here and refer to the documentation[5] for further information.

During the whole runtime of Raven the output panel (②) provides the user with additional information about the status of the program. Once the computation has finished, the result is displayed to the user. In case of the membership check only a

[5] Available at http://www.ti.inf.uni-due.de/research/tools/raven/documentation.pdf

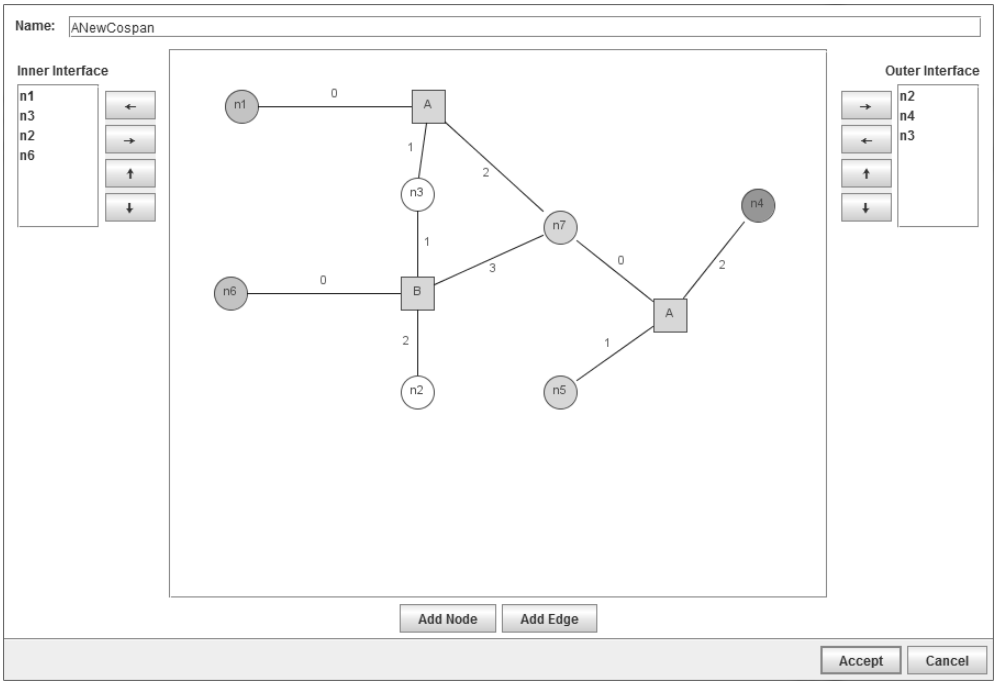


Figure 8.6.: RAVEN Visual Cospan Creator

small text dialog is displayed which is used to indicate whether the check has been successful or not. In the case where the user has chosen the universality, language inclusion or invariant check, RAVEN also displays a counterexample if the result is negative.

8.4. Comparison with other tools

In this section we want to give a short comparison between RAVEN and some other software tools, which use techniques similar to the ones presented here.

In his seminal work Courcelle [44] has shown that graph properties definable in monadic second-order logic can be encoded as finite (tree) automata. That is a result which is very important not only for RAVEN, but also for other tools. The challenge in the application of this result is the practicability for graphs with a reasonable treewidth (pathwidth), since the treewidth (pathwidth) has an exponential impact of the size of the finite (tree) automaton. Initial work to tackle this problem has been done by the authors of the tool MONA [77, 89]. This tool is used to encode formulas of (a fragment of) monadic second-order logic on strings and trees into finite (tree) automata. Similar to RAVEN the tool MONA also benefits from the symbolical encoding of automata by BDDs. But in case of MONA the BDDs are used differently, since instead of the state space only the alphabet is encoded by means of BDDs. Therefore, for MONA it is beneficial that one has a huge alphabet which must be encoded, whereas RAVEN is

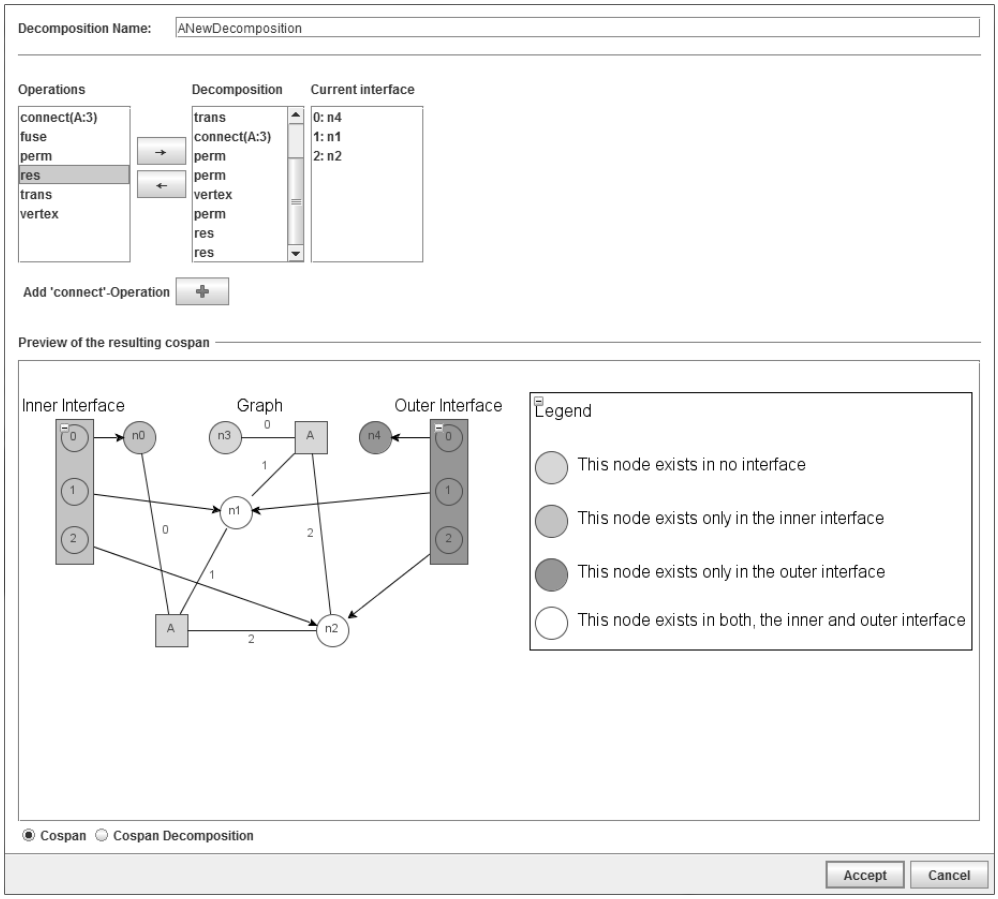


Figure 8.7.: Raven Visual Cospan Decomposition Creator

designed for huge state spaces. However, this is not the only difference between both tools. In contrast to Raven which is used for the verification of graph transformation systems, Mona is designed for the analysis of so-called *pointer programs* [111, 112] and for *shape analysis* [118] respectively. These techniques are very suitable if the data structures under consideration (i. e. single or doubly linked lists, different kind of trees, maps, ...) are known in advance. In this case only specific graph-like structures, which are used to model the underlying data structures, must be considered in the verification, which potentially makes these techniques more feasible. But if the inspected system is much more heterogeneous in a way that the system must be modelled by arbitrary graphs or is evolving in a rule-based but otherwise unpredictable manner or has different components with different behavior, one needs a more general approach. This is one of the advantages of Raven in contrast to Mona, since Raven is only restricted by the class of (bounded) recognizable graph languages. Hence, in case of Raven, the systems under consideration can be very heterogeneous.

The software AutoWrite [58, 59] is another tool which is used to verify monadic

second-order logic definable properties on graphs of bounded tree- or clique-width[6] respectively. The verification is done by transforming the input formula into a so-called fly-automaton, in which the state set and the transition function are represented as computable functions rather than in an explicit way [45–47]. Furthermore, it is possible to compute more complex fly-automata using basic automata and the usual automata-theoretic operations such as union, intersection, complementation, et cetera, which is similar to the features of Raven. The differences between both tools lie in the distinct fields of application. AutoWrite has much potential in solving the membership problem, since the tool has to compute and explicitly store only states which are reached by a given input. This is due to the representation of the transition functions, which yields small fly-automata on the one hand and less overhead due to the on-the-fly computation on the other hand. The disadvantage is, that due to the use of computable functions to represent the (huge) state sets and the (huge) transition function, many decision problems which are of special interest for verification purposes (e. g. emptiness, universality, language inclusion, ...) become undecidable, if arbitrary computable functions are considered. In contrast the advantage of Raven is that these problems are decidable if one uses the techniques and symbolical representation implemented in Raven. Therefore, AutoWrite and Raven are useful for different purposes, even if the theoretical basis of both tools is similar.

A further tool which depends on techniques also used for Raven is the software tool Alaska [57]. Just as Raven, the Alaska-tool is based on the antichain-based approach to solve the language inclusion problem presented in Chapter 7. Another similarity between both tools is the representation of the state sets of the involved automata as Bdds. But Alaska converts these symbolically represented state sets into an explicit representation when computing the successor or predecessor state sets respectively. This is different to Raven as described in Section 7.2, which is implemented fully symbolically. In case of Raven this conversion would be infeasible due to the huge size of the state spaces of the certain automata. But in contrast to Raven, the Alaska tool deals with much smaller automata which originates from the different areas of application: Alaska is applied to solve the Ltl-model checking problem whereas Raven can be used for monadic second-order model checking. Hence, Alaska deals with alternating finite automata or alternating Büchi automata respectively to represent (ω-regular) word languages which are rather small compared to the graph automata to represent recognizable graph languages used by Raven.

8.5. Conclusion

In this chapter we presented a prototype implementation for a tool suite which is able to handle and manipulate graph automata up to a bounded interface size as introduced in Chapter 6. In order to represent these automata in an efficient manner the implementation depends on a symbolic representation realised as binary decision diagrams as described in Section 7.1. The functionality of Raven covers among others the computation of pre-defined classes of automata (k-colorability automaton, subgraph isomorphism automaton, vertex cover automaton, ...), which can be combined by

[6] The notion of *clique-width* is similar to the notion of pathwidth and treewidth and depends on a clique decomposition of a graph. For further information about clique-width see [49].

computing union or product automata, thus realising closure properties, and to check the membership of a given graph in the language defined by an automaton. For our purposes, which are in the area of verification of dynamically evolving systems, decision procedures on automata are a central ingredient and hence we implemented both emptiness and universality checks. Another feature of Raven is the possibility to solve the language inclusion problem for two given graph automata which is implemented via the algorithms presented in Section 7.2. In the next chapter, we will give some experimental results which have been obtained by using Raven.

"The only truly secure system is one that is powered off."
Eugene Howard Spafford (1956 – present)

9 Experimental Results

In this chapter, we present different case studies which we have conducted using the tool suite RAVEN presented in the previous chapter. The case studies are of different types:

Multi-User File System We considered the multi-user file system from [15]. In this example a system state is represented by a single graph: Nodes represent users and files, edges represent permissions, either "read" or "write". Graph transformation rules are used to model the access rules of the system such as "add new user", "change permission", "delete file", ... The task of this case study is to check whether the file system can reach a forbidden state which violates some consistency property using the system's access rules.

Invariant checking We considered additional case studies in connection with invariant checking to evaluate our approach. In particular we analyzed a transformation rule which switches a single edge for which the language of all graphs containing a triangle subgraph is an invariant. Furthermore, we examined two other examples. The first one consists of a transformation rule which extends a path (segment) from length one to length three for which the language of all 2-colorable graphs is an invariant. The second one consists of a transformation rule which replaces a triangulated rectangular subgraph by a more complex subgraph for which the language of all 3-colorable graphs is an invariant.

Results from graph theory and counterexample generation Language inclusion allowed us to prove some results from graph theory (up to bounded pathwidth) and to obtain counterexamples if the inclusion fails.

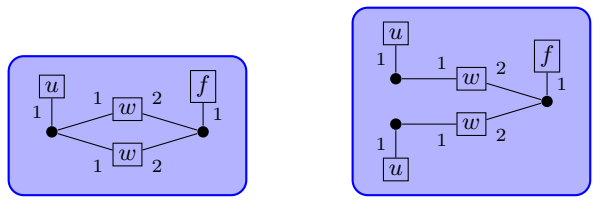
Membership tests We obtained runtime results for various membership tests. We computed random graphs (for different number of nodes), obtained their tree decompositions and then decomposed them into atomic cospans and processed them with our tool.

All tests in this chapter, except for the membership tests, were performed on a 64-bit Linux machine with a Xeon Dualcore 5150 processor and 8 GB of available main memory. For the membership tests we used a different machine running a 64-bit version of Windows 7 on a Core i5-2500 processor and 8 GB of available main memory. This hardware exchange has been necessary, since the other machine was no longer available. For all case studies we used only signatures which do *not* contain the letter *fuse*. This decision is motivated by two reasons: on the one hand we only consider loop-free graphs which can be created without the *fuse* cospan, on the other hand the computation of the *fuse*-transition is usually very expensive (measured in the time of computation).

9.1. Case Study "Multi-user File System"

In this section we want to validate the multi-user file system[1] from [15], where the access to the system is controlled by several rules in order to guarantee some consistency properties. In this example, a system state is modelled as a graph: users and files are nodes with a unary edge, either labeled by u (for user) or f (for file), attached to each node to distinguish users from files. Furthermore, access permissions (either "read" or "write") are binary edges, either labeled by r or w. The system behavior (add new user, change access permissions, ...) is modelled as transformation rules (see below). As signature we take the following (families of) letters: $connect_u^i$, $connect_f^i$, $connect_r^i$, $connect_w^i$, $perm^i$, res^i, $trans^i$ and $vertex^i$, where u as well as f are both labels of arity 1 and r as well as w are both labels of arity 2 (see also Table 6.1 on page 72).

We consider two properties which describe each a violation of the consistency of the multi-user file system. The system is in a consistent state as long as these properties are *not* satisfied. The first property is the double write access of a user to a file (*double access*), i.e. a user has twice a write access to the same file at the same time. The second property is the write access of two different users to the same file at the same time (*two users*). These two forbidden properties can be modeled by the two graphs depicted in Figure 9.1.



(a) **(b)**

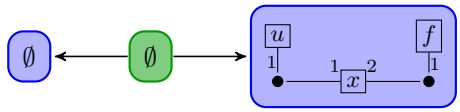
Figure 9.1.: Forbidden subgraphs "Double Access" and "Two Users"

Note that it is not forbidden that a user has more than one read access to a file at the same time and that two or more users can have read access to the same file at the same time even if one user has write access to that file.

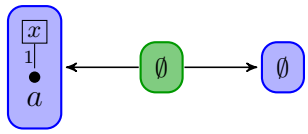
[1] This case study has been inspired by [92].

The multi-user file system offers the usual operations which will be explained in more detail below:

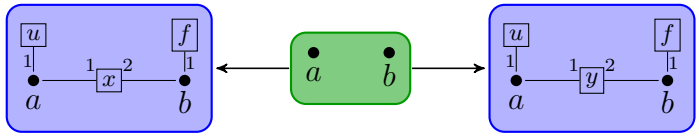
Create new user (with read/write access): The rule “Create new user” creates a new user u and a new file f and gives the user a read (write) access to this file. It can be modeled by the following span where x is either r or w:



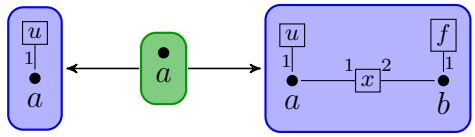
Delete user/file: The rule “Delete user/file” applied for some user u (file f) removes the user (file) from the system. The following span models this rule where x is either u or f:



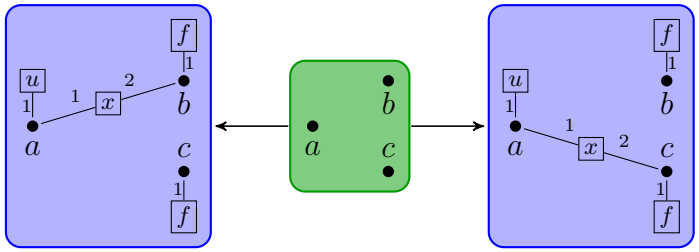
User converts (read/write) access: The rule “User converts access” applied to some user u with read (write) access to some file f converts the access of the user from read to write access to this file (or vice versa). It can be modeled by the following span where either x is r and y is w or vice versa:



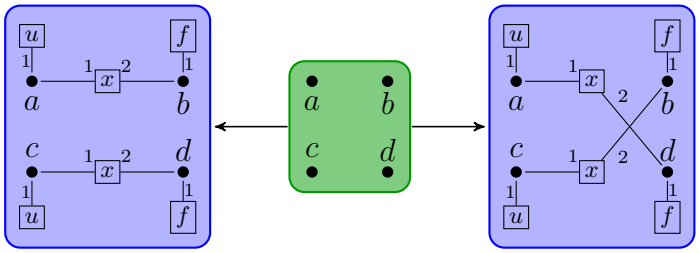
User creates new file (with read/write access): The rule “User creates new file” applied to some user u creates a new file f and gives the user a read (write) access to this file. It can be modeled by the following span where x is either r or w:



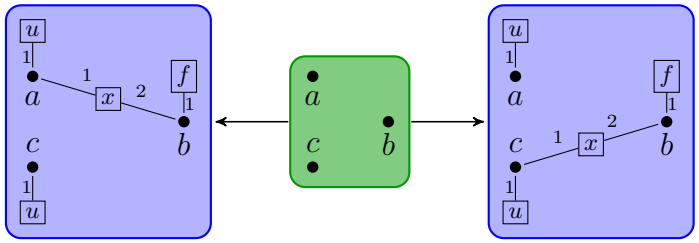
User requests file (with read/write access): The rule “User requests file” applied to some user u sets the read (write) access of this user from the current file to some other existing file. The following span models this rule where x is either r or w:



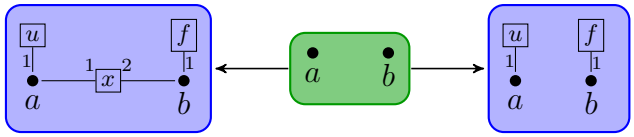
Users swap (read/write) access: The rule "Users swap access" applied to two user u_1 and u_2 with read (write) access to files f_1 and f_2 swaps the access to these files. The following span models this rule where x is either r or w:



User transfers (read/write) access: The rule "User transfers access" applied to two users u_1, u_2 and a file f read (write) accessible by u_1 transfers the access of the file to the user u_2. The following span models this rule where x is either r or w:



Withdraw (read/write) access: The rule "Withdraw access" applied to some user u with read (write) access to some file f widthdraws the access of the user to this file. The following span models this rule where x is either r or w:



As already mentioned in Section 3.3 every rewriting rule $\rho\colon L \leftarrow \ell - I - r \rightarrow R$ can be considered as two cospans $c_\ell\colon \emptyset \rightarrow L \leftarrow \ell - I$ and $c_r\colon \emptyset \rightarrow R \leftarrow r - I$ which are the left-hand side and right-hand side of the corresponding rewriting rule.

In Section 6.1.3 we have stated that recognizable languages are closed under boolean operations and in Example 6.34 we have described how the graph automaton accepting

		Max. Interface Size	
Access Rule	Result	4	10
Create new user with read access	✔	2	3 246
Create new user with write access	✔	2	3 260
Delete user	✔	2	3 153
Delete file	✔	2	3 137
User converts read access	✘	< 1	< 1
User converts write access	✔	2	3 084
User creates new file with read access	✔	2	3 145
User creates new file with write access	✔	2	3 329
User requests file with read access	✔	2	3 054
User requests file with write access	✘	< 1	< 1
Users swap read access	✔	2	3 232
Users swap write access	✔	2	3 312
User transfers read access	✔	2	3 098
User transfers write access	✔	2	3 120
Withdraw read access	✔	2	2 663
Withdraw write access	✔	2	2 628

Table 9.1.: Results and runtimes (in seconds) for the case study “Multi-user file system”

the language of all graphs containing a fixed subgraph works. With these considerations we can now construct a graph automaton that recognizes all graphs violating one of the two properties, i. e. all graphs that contain either of the two forbidden subgraphs. Note that this graph automaton accepts the complement of the language of all consistent states, i. e. all graphs that *do* contain one of the forbidden subgraphs. Hence, we perform a backwards analysis on each rewriting rule and check whether $\mathcal{L}(\mathcal{A}[r]) \subseteq \mathcal{L}(\mathcal{A}[\ell])$. If the language inclusion holds, then the original rewriting rule does not violate the consistency of the multi-user file system. This can be seen as follows: After the application of the rule the consistency of the system is violated only if it was already violated before the rule application, hence the language is verified to be an invariant.

In Table 9.1 the result of the backwards analysis and the runtime results obtained by the complement backwards searching antichain algorithm (see Subsection 7.2.1) are presented. In the first column the name of the rewriting rule is given. The result of the language inclusion check is either denoted by a checkmark (✔), if the transformation rule is consistent, or by a crossmark (✘), if the transformation rule is not consistent. In the columns three and four we present the times (in seconds) needed to run the complement backwards searching antichain algorithm for both a setting with a maximum permitted interface size of 4 and a setting with a maximum permitted interface size of 10.

As we can see, the only rules which are not consistent are the two rules “User converts

read access" and "User requests file with write access". To prove the non-consistency two counterexamples – one for each rule – can be obtained from the complement backwards searching antichain algorithm. For the rule "User converts read access" the counterexample depicted in Figure 9.2 is computed. Obviously, the graph on the left-hand side is consistent, i. e. it does violate neither the "double access" nor the "two users" condition. But if we apply the rule "User converts read access" we obtain the graph depicted on the right, which violates the consistency property "double access".

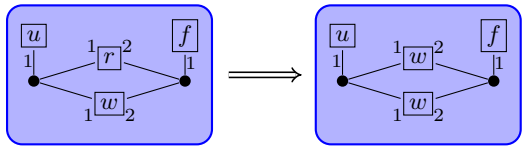


Figure 9.2.: Counterexample for the rule "User converts read access"

For the rule "User requests file with write access" the counterexample depicted in Figure 9.3 is computed. Similarly, the graph depicted on the left is consistent, but the graph on the right, obtained by applying the rule "User requests file with write access", obviously violates the consistency property "double access", too.

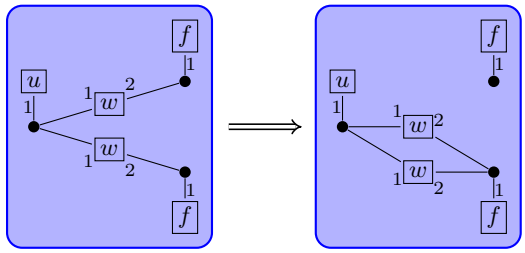


Figure 9.3.: Counterexample for the rule "User requests file with write access"

Note that in [10] and [15] the author together with Bruggink and König used a different approach based on the generalized Myhill-Nerode theorem (cf. Theorem 6.22) to verify the access rules depicted above. The idea is that a language $\mathcal{L}$ is an invariant according to a rule $\rho\colon L \leftarrow\ell- I -r\rightarrow R$ if the corresponding cospans[2] c_ℓ and c_r are related by the Myhill-Nerode quasi-order, i. e. $c_\ell \leq_{\mathcal{L}} c_r$ holds. To compute the Myhill-Nerode quasi order there exists an algorithm which is similar to the one used for the computation of the minimal DFA. But for practical purposes this approach is not useful due to the fact that the Myhill-Nerode quasi order is only computable for deterministic graph automata which are in general exponentially larger than the equivalent non-deterministic graph automata. Hence, a simulation relation was used to approximate

[2]For the connection between transformation rules and cospans we refer to page 27.

the Myhill-Nerode quasi-order, which can also be computed for non-deterministic graph automata. Due to this approximation the validation of the rewriting rules "User transfers write access" and "Users swap write access" was unsuccessful, although the language is an invariant w. r. t. these rules. Now we succesfully verified them.

9.2. Case Studies: Invariants for Subgraph Containment & Colorability

In this section we want to validate three other invariants, namely the "triangle subgraph invariant", the "2-colorability with path extension invariant" and the "3-Colorability with Node Replacement" invariant. We explain these two case studies in more detail below:

Triangle Subgraph: For this example we take the graph T (which is depicted in Figure 9.4) as fixed subgraph.

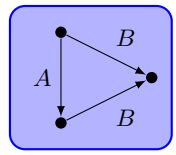


Figure 9.4.: Wanted subgraph T

The signature of this case study contains the following (families of) letters: $connect^i_A$, $connect^i_B$, $perm^i$, res^i, $trans^i$ and $vertex^i$, where A and B are both labels of arity 2 (see also Table 6.1 on page 72). How the graph automaton can be obtained which accepts the language $\mathcal{L}_T$ of all graphs containing T as a subgraph has been explained in Example 6.34. The language $\mathcal{L}_T$ is an invariant for the rule ρ_A (shown in Figure 9.5) which "switches" an A-labeled edge.

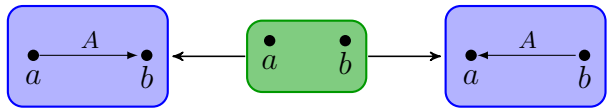


Figure 9.5.: Transformation rule ρ_A

This can be seen as follows: Every graph which contains T as subgraph before the application of ρ_A does contain T also after the rule application. But due to the "switch" of the A-labeled edge, also the B-labeled edges switch their role.

2-Colorability with Path Extension: For this example we consider the language $C_{(2)}$ of all 2-colorable graphs, which has been introduced in Example 6.4. The corresponding graph automaton has been explained in Example 6.33. The corresponding signature consists of the following (families of) letters: $connect^i_\diamond$, $connect^i_w$, $perm^i$,

res^i, $trans^i$ and $vertex^i$, where $\diamond$ is an arbitrary label of arity 2 (see also Table 6.1 on page 72). This language is an invariant for the transformation rule α_n depicted in Figure 9.6 which adds two new nodes between two adjacent nodes on a path.

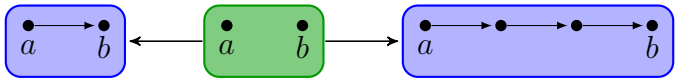


Figure 9.6.: Transformation rule α_n

It is obvious that a graph with an even number of nodes also has an even number of nodes after the rule application. Hence, the language $C_{(2)}$ is an invariant for the rule α_n, since every path with an even number of nodes is 2-colorable. Note that this holds also for other k-colorable graphs (with $k \geq 2$).

3-Colorability with Node Replacement: For this example we consider the language $C_{(3)}$ of all 3-colorable graphs (see Example 6.4 and Example 6.33, the signature is the same as for the example above). This language is an invariant for the transformation rule α_r depicted in Figure 9.7 which replaces the center node of the left-hand side by a directed cycle consisting of four nodes which are connected to a new center node.

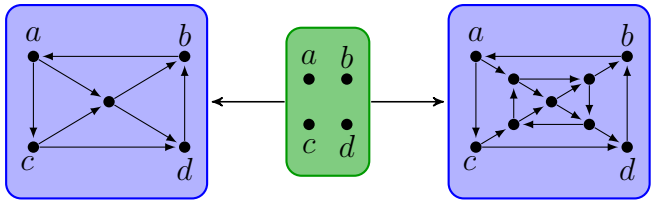


Figure 9.7.: Transformation rule α_r

That the language $C_{(3)}$ is an invariant for the rule α_r can be seen as follows: The outer nodes of the graph of the left-hand side can be colored alternating. Hence, the third color can be used to color the center node. This coloring can be extended for the graph of the right-hand side. The nodes of the inner cycle can be colored alternatingly, too.

We now present the runtime results for these three case studies in Table 9.2 (for the "triangle subgraph" case study) in Table 9.3 (for the "2-Colorability with path extension" case study) and in Table 9.4 (for the "3-Colorability with Node Replacement" case study). Note that we have used the four language inclusion algorithms presented in Section 7.2 to check the invariants. That is, we used the following algorithms:

- normal forwards searching antichain algorithm (forwards antichain for short), see also Algorithm 7.1,
- complement backwards searching antichain algorithm (backwards antichain for short), see also Algorithm 7.2,

- simulation-based antichain algorithm, see also Algorithm 7.3 and
- bisimulation up-to congruence algorithm, see also Algorithm 7.4.

Furthermore, all tests have been run with a time limit of 86 400 seconds which equals 24 hours. Whenever this time limit was exceeded, the respective test was aborted.

	Maximum Interface Size							
Algorithm	3	4	5	6	7	8	9	10
Forwards Antichain	11	1 202	69 829	TO	–	–	–	–
Backwards Antichain	1	3	17	190	1 525	10 200	47 001	TO
Sim.-Based Antichain	150	9 756	TO	–	–	–	–	–
Bisim. up-to Cong.	TO	–	–	–	–	–	–	–

Table 9.2.: Runtimes (in seconds) for the case study "Triangle Subgraph"; TO: timed out

	Maximum Interface Size							
Algorithm	3	4	5	6	7	8	9	10
Forwards Antichain	< 1	< 1	1	2	25	1 311	60 853	TO
Backwards Antichain	< 1	< 1	1	3	37	1 501	49 638	TO
Sim.-Based Antichain	< 1	1	7	527	40 433	TO	–	–
Bisim. up-to Cong.	< 1	< 1	1	12	821	45 271	TO	–

Table 9.3.: Runtimes (in seconds) for the case study "2-Colorability with path extension"; TO: timed out

	Max. Interface Size		
Algorithm	5	6	7
Forwards Antichain	63	30 747	TO
Backwards Antichain	112	TO	–
Sim.-Based Antichain	< 1	TO	–
Bisim. up-to Cong.	TO	–	–

Table 9.4.: Runtimes (in seconds) for the case study "3-Colorability with Node Replacement"; TO: timed out

Now, we compare and interpret the runtimes of the four algorithms. In all three case studies the best runtime results are always obtained by one of the antichain algorithms. In the first case study the backwards antichain algorithm is much faster

and applicable to much higher maximum interface sizes than the other three algorithms. In the second case study the forwards and the backwards antichain algorithm are nearly equal. In the third case study only the forward antichain algorithm is applicable to a maximum interface size greater than 5. In contrast to the antichain algorithms the simulation-based antichain algorithms is rather slow and the bisimulation up-to congruence algorithm is not usable in the first and the third case study, the results in the second case study are also rather poor.

The reason for the good results of the backwards antichain algorithms in the first case study may be found in the implementation of the subgraph graph automaton. In order to minimize the size of the BDDs needed to encode the states of the graph automaton, we also permitted bit strings which do not encode valid states of the graph automaton, but which are unreachable from the initial states. Thus, the number of states of the graph automaton becomes larger and conversely the BDD which encodes the state set becomes smaller. This fact may yield a more compact representation of (state) information.

The reason why the runtime results of the forwards antichain algorithm for the third case study are much better than that of the backwards antichain algorithm can be found by taking the "starting states" of both algorithms into account. The forwards algorithm starts with the initial states of the "shifted" automata, i. e. those states which are reachable from the initial state of the 3-colorability automaton by processing the left- and right-hand side of the transformation rule respectively. Hence, the initial states of the "shifted" automata represented valid colorings of the left- and right-hand side. Then the forwards algorithms tries to reach an accepting state pair (as decripted in subsection 7.2.1). The backwards algorithm starts with the opposite states, i. e. the algorithm is initialized with the set of accepting state pairs and tries to (backward-)reach the initial states of the "shifted" automata (as described in subsection 7.2.1). Due to this backwards steps, the backwards antichain algorithm may also (backward-)reach states which are not reachable from the initial states. Hence, the search space of this variant is larger than that of the forwards antichain algorithm, which results in worse runtimes.

Altogether, we can handle some non-trivial examples up to relatively large interface sizes (note that in practical applications the width, and thus the interface size of graphs, is in general relatively small). For example, the "triangle subgraph automaton" has 37 440 states in case of maximum interface 3, 19 173 952 states in case of interface size 6 and 2 147 483 647 in case of the interface size 9.

9.3. Case Studies from Graph Theory

In this section we want to perform some language inclusion checks which allow us to prove some results from graph theory (up to bounded pathwidth). In case the language inclusion does not hold we want to obtain counterexamples. To be precise we checked the following six results (for $2 \leq k \leq 4$):

- $C_{(3)} \subseteq C_{(4)}$ and $C_{(4)} \not\subseteq C_{(3)}$. We computed results to show that every 3-colorable graph is also 4-colorable, but that not every 4-colorable graph is 3-colorable. For the corresponding graph automata we refer to Example 6.33.
- $V_{(k)} \not\subseteq D_{(k)}$ and $D_{(k)} \not\subseteq V_{(k)}$. We computed results to show that graphs which

have a vertex cover of size at most k do not necessarily have a dominating set of size at most k. We also showed that the reverse inclusion does not hold as well. For the vertex cover graph automaton we refer to Example 6.5. For an idea how to obtain the graph automaton for the language $V_{(k)}$ we refer to the list on page 103.

- $\mathit{NonIso} \cap V_{(k)} \subseteq D_{(k)}$ and $D_{(k)} \not\subseteq \mathit{NonIso} \cap V_{(k)}$. We computed results to show that all graphs without isolated nodes which have a vertex cover of size at most k also have a dominating set of size at most k. We also showed that the reverse inclusion does not hold. The information which is needed to encode the graph automaton for the language *NonIso* is explained in the list on page 103.

Similar to the case studies in the previous section, we use the four algorithms presented in Section 7.2 to check the language inclusion. In Table 9.5 we present the results for the check $C_{(3)} \subseteq C_{(4)}$ and in Table 9.6 the results for the check $C_{(4)} \not\subseteq C_{(3)}$. As signature we take the following (families of) letters: $\mathit{connect}^i_\diamond$, perm^i, res^i, trans^i and vertex^i, where $\diamond$ is an arbitrary label of arity 2 (see also Table 6.1 on page 72).

	Maximum Interface Size				
Algorithm	3	4	5	6	7
Forwards Antichain	< 1	1	246	TO	–
Backwards Antichain	< 1	1	269	TO	–
Sim.-Based Antichain	< 1	5	1 127	57 556	TO
Bisim. up-to Cong.	< 1	2	TO	–	–

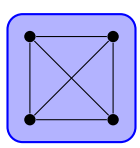
Table 9.5.: Runtimes (in seconds) for the case study $C_{(3)} \subseteq C_{(4)}$; TO: timed out

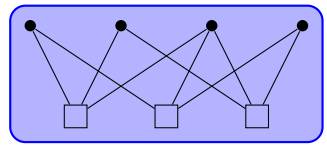
	Maximum Interface Size						
Algorithm	4	5	6	7	8	9	10
Forwards Antichain	5	7 899	TO	–	–	–	–
Backwards Antichain	7	7 846	TO	–	–	–	–
Sim.-Based Antichain	8	1 319	69 999	TO	–	–	–
Bisim. up-to Cong.	< 1	8	156	763	2 351	6 703	7 200

Table 9.6.: Runtimes (in seconds) for the case study $C_{(4)} \not\subseteq C_{(3)}$; TO: timed out

Note that for each test given in Table 9.6 which has completed successfully, a counterexample has been computed, too. The computed counterexample is depicted below on the left. Obviously, the graph is 4-colorable, since it consists only of 4 nodes. But it is not 3-colorable, because every node is pairwise incident to every other node. Furthermore, we want to point out that we obtain another counterexample if we change the signature by replacing the $\mathit{connect}^i_\diamond$-operation with $\mathit{connect}^i_\square$, where $\square$ is a label of

arity 3. In this case we would obtain the counterexample depicted below on the right. This graph is also a 4-clique, but the nodes are connected pairwise by three $\Box$-edges.





In Table 9.7 we present the results for the check $V_{(k)} \not\subseteq D_{(k)}$ and in Table 9.8 the results for the check $D_{(k)} \not\subseteq V_{(k)}$. Note that the results depicted in the two tables have been obtained for different values of k ranging from 2 to 4. For both case studies we used the same signature which contains the following (families of) letters: $connect^i_\diamond$, $perm^i$, res^i, $trans^i$ and $vertex^i$ (see also Table 6.1 on page 72).

		Maximum Interface Size							
Algorithm	k	3	4	5	6	7	8	9	10
Forwards	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Antichain	3	< 1	< 1	< 1	< 1	< 1	1	1	1
	4	< 1	3	15	48	91	108	118	138
Backwards	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Antichain	3	< 1	< 1	< 1	< 1	< 1	1	1	1
	4	< 1	< 1	< 1	1	2	4	11	28
Sim.-Based	2	< 1	8	108	1 542	21 031	TO	–	–
Antichain	3	1	14	240	2 883	35 081	TO	–	–
	4	2	46	577	8 155	TO	–	–	–
Bisim. up-to	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Cong.	3	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	4	< 1	< 1	2	4	4	9	3	9

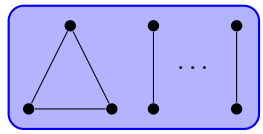
Table 9.7.: Runtimes (in seconds) for the case study $V_{(k)} \not\subseteq D_{(k)}$; TO: timed out

In both cases we obtained a counterexample for each test which has completed successfully. For the case study $V_{(k)} \not\subseteq D_{(k)}$ the counterexample simply consists of a graph with $k+1$ isolated nodes, i. e. the graph D_{k+1}. Due to the fact that this graph does not contain any edge the vertex cover can be left empty. But since there are $k+1$ nodes which are pairwise non-adjacent each node must be dominated by itself. Hence the dominating set has to contain all $k+1$ nodes, which is not allowed due to the restriction of the size of the dominating set.

For the other case study $D_{(k)} \not\subseteq V_{(k)}$ a counterexample of the following form is computed:

		Maximum Interface Size							
Algorithm	k	3	4	5	6	7	8	9	10
Forwards	2	< 1	2	7	22	53	63	134	306
Antichain	3	< 1	22	854	7 121	21 272	58 294	TO	–
	4	3	685	TO	–	–	–	–	–
Backwards	2	1	551	86 266	TO	–	–	–	–
Antichain	3	6	6 922	TO	–	–	–	–	–
	4	24	38 672	TO	–	–	–	–	–
Sim.-Based	2	< 1	8	121	1 582	1 480	TO	–	–
Antichain	3	< 1	13	182	2224	35 070	TO	–	–
	4	2	43	747	14 305	TO	–	–	–
Bisim. up-to	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	1
Cong.	3	1	17	101	147	151	179	215	199
	4	64	1 990	18 342	TO	–	–	–	–

Table 9.8.: Runtimes (in seconds) for the case study $D_{(k)} \not\subseteq V_{(k)}$; TO: timed out



where the number of 2-cliques is equal to $k - 1$. This is obviously a valid counterexample witnessing $D_{(k)} \not\subseteq V_{(k)}$, since the 3-clique can be dominated by a single node, the remaining 2-cliques can be dominated by another $k - 1$ nodes (one node dominating each 2-clique). But to cover the three edges of the 3-clique at least two of the three nodes of the 3-clique have to be contained in the vertex cover. Additionally, for every 2-clique one of the two nodes must be contained in the vertex cover. Hence, the minimum vertex cover has a size of $k + 1$.

In Table 9.9 we present the results for the case study $\mathit{NonIso} \cap V_{(2)} \subseteq D_{(2)}$ and in Table 9.10 the results for the case study $D_{(2)} \not\subseteq \mathit{NonIso} \cap V_{(2)}$. For both case studies we used the same signature which contains the following (families of) letters: $\mathit{connect}^i_\diamond$, perm^i, res^i, trans^i and vertex^i (see also Table 6.1 on page 72). Note that the results for bisimulation up-to congruence algorithm in Table 9.9 in case of $k = 3, 4$ for the maximum interface size are somewhat suprising.

Again, for each test given in Table 9.10 which has completed successfully a counterexample has been computed. In case of the $D_{(2)} \not\subseteq \mathit{NonIso} \cap V_{(2)}$ case study the counterexample consists only of a single node, i. e. the graph D_1. Obviously, this graph has a dominating set of size at most 2 (the dominating set contains the only node), but since the single node is isolated, the graph cannot be contained in the language $\mathit{NonIso} \cap V_{(2)}$.

		Maximum Interface Size					
Algorithm	k	3	4	5	6	7	8
Forwards	2	< 1	2	31	868	19 114	TO
Antichain	3	1	204	18 329	TO	–	–
	4	11	6 272	TO	–	–	–
Backwards	2	1	412	TO	–	–	–
Antichain	3	6	3 873	TO	–	–	–
	4	31	21 053	TO	–	–	–
Sim.-Based	2	1	11	157	2 036	26 359	TO
Antichain	3	3	104	3 018	66 616	TO	–
	4	9	216	5 550	TO	–	–
Bisim. up-to	2	12	TO	–	–	–	–
Cong.	3	19 064	TO	–	–	–	–
	4	18 735	TO	–	–	–	–

Table 9.9.: Runtimes (in seconds) for the case study $NonIso \cap V_{(k)} \subseteq D_{(k)}$; TO: timed out

		Maximum Interface Size							
Algorithm	k	3	4	5	6	7	8	9	10
Forwards	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Antichain	3	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	4	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Backwards	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Antichain	3	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	4	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Sim.-Based	2	< 1	10	135	1 790	28 592	TO	–	–
Antichain	3	3	100	2 670	64 231	TO	–	–	–
	4	8	200	4 890	TO	–	–	–	–
Bisim. up-to	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Cong.	3	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	4	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1

Table 9.10.: Runtimes (in seconds) for the case study $D_{(k)} \not\subseteq NonIso \cap V_{(k)}$; TO: timed out

Next, we interpret the runtimes of the six case studies. In case $C_{(3)} \subseteq C_{(4)}$ (cf. Table 9.5) the simulation-based antichain algorithm is applicable to higher maximum interface sizes than the other three algorithms. The reason for this is that the algorithm benefits from the simulation relation which is costly to compute but helps to optimize the antichain algorithm.

This result can also be seen in the case study $C_{(4)} \not\subseteq C_{(3)}$ (cf. Table 9.6). The simulation-based antichain algorithm is still applicable to the maximum interface size of 6, for which both the forwards and the backwards antichain algorithm timed out. But the best algorithm for this case study is the bisimulation up-to congruence algorithm.

In the case study $V_{(k)} \subseteq D_{(k)}$ (cf. Table 9.7) the simulation-based antichain algorithms is only applicable to a maximum interface size of 7 (if k equals 2 or 3) or 6 respectively (if k equals 4). Furthermore, we can see that the computation of the simulation relation is rather costly. Here, the other three algorithms are much better, since the runtimes are significantly better and the algorithms are applicable to much higher maximum interface sizes.

For the case study $D_{(k)} \not\subseteq V_{(k)}$ (cf. Table 9.8) only the forwards antichain algorithm and the bisimulation up-to congruence algorithm are usable for higher maximum interface sizes (if k equals 2 or 3). Here, the backwards antichain algorithms delivers the worst results, but also the simulation-based antichain algorithm is not as efficient as the other two algorithms. Only if k equals 4 the simulation-based antichain algorithm is still applicable for a maximum interface size of 6 while the other algorithms are not applicable anymore.

In the case study $\mathit{NonIso} \cap V_{(k)} \subseteq D_{(k)}$ (cf. Table 9.9) the three antichain algorithms – forwards, backwards and simulation-based – are all applicable up to a maximum interface size of 4, whereas the bisimulation up-to congruence algorithm is only applicable up to a maximum interface size of 3. But again, we can see that the computation of the simulation is very costly and not always beneficial.

In case of the case study $D_{(k)} \not\subseteq \mathit{NonIso} \cap V_{(k)}$ (cf. Table 9.10) the simulation can only be computed for a maximum interface size of 4. Hence, the simulation-based antichain algorithm is only applicable up to that interface size. The other three algorithms can be used up to maximum interface sizes of 10 and beyond. The good results of these algorithms (for this case study) are not suprising, since the counterexample consists only of a single node.

From the case studies above, it is apparent that the runtimes are better when the first automaton is small (the automaton for $C_{(3)}$ and $D_{(2)}$, respectively). This is unsurprising, because the states of the first automaton are explicitly represented (more formally, as a Bdd representing a singleton set), whereas the (sets of) states of the second automaton are collectively represented by a Bdd. A more detailed evaluation of the four algorithms is given in Section 9.5. There, we discuss not only the reults of this section, but we also compare the results of other sections for all algorithms.

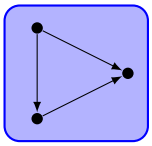
9.4. Membership Case Studies

In this section we give some results for solving the membership problem for different languages. For each membership case study we computed 100 random graphs. The number of nodes of these graphs varies between 10 and 50, depending on the parameters of the different case studies. But in each case we computed graphs with an “edge density”

of 10%, i. e. for each pair of nodes of the graph there is a 10% chance that an edge (with label $\diamond$ of arity 2) is added connecting these two nodes. For all membership case studies we used the same signature which contains the following (families of) letters: $connect^i_\diamond$, $perm^i$, res^i, $trans^i$ and $vertex^i$ (see also Table 6.1 on page 72).

For our membership experiments we used the following languages as case studies:

- the language $\mathcal{L}_{3C}$ of all graphs which contain the 3-clique (see graph below) as subgraph,



- the language $C_{(3)}$ of all graphs which are 3-colorable,
- the language $C_{(4)}$ of all graphs which are 4-colorable,
- the language $D_{(k)}$ of all graphs which have a domating set of size at most k,
- the language $V_{(k)}$ of all graphs which have a vertex cover of size at most k,
- the language $NonIso \cap V_{(k)}$ of all graphs which have no isolated nodes and have a vertex cover of size at most k.

For each case study we performed tests with graphs of different size (in the number of nodes) which were used as input. But note that for each test all graphs have the same size. Depending on the number of nodes of the input graphs we also altered the maximum interface size of the graph automata, i. e. depending on the number of nodes of the random graphs we have chosen a fixed maximum interface size:

Number of Nodes	10	20	30	40	50
Maximum Interface Size	5	10	15	20	25

This is due to the idea that we want to compute the graph automaton only once and to use it over and over again to process all graphs (with the same number of nodes).

If we use a different graph automaton for each graph it might be more advisable to choose the maximum interface size depending on the width of the atomic cospan decomposition of the given graph. Since in this case the maximum interface size needed to process the graph might be smaller than the maximum interface size defined in the table above, this results in a smaller graph automaton. However, even if we set the maximum interface size to a fixed value which may be greater than needed, this does not effect the runtime of the membership test. This is due to the fact that states which encode information for interfaces with a size greater than the width of the atomic cospan decomposition (of the processed graph) are not reachable while processing the atomic cospan decompostion.

But also note that we do not guarantee to be able to process all graphs with the given number of nodes. But due to the “sparse edge density”, it is very likely that the

bigger part of the random graphs can be processed, i. e. that the pathwidth is equal or less to the chosen maximum path width. Furthermore, each instance of RAVEN (used for the tests) had 2 GB of available memory.

In order to obtain the atomic cospan decompositions, RAVEN first uses the GREEDY-FILLIN-algorithm (cf. Bodlaender and Koster [20]) to get a tree decomposition of the input graph. Second, the tool transforms this tree decomposition into a path decomposition and then computes the atomic cospan decomposition.

In Table 9.11 we present the results of the membership tests for $[G] \in \mathcal{L}_{3C}$. In the first column we give the number of nodes of the several graphs which serve as input. The second column gives the ratio of acceptance, i. e. the portion of graphs which have been accepted by the graph automaton. The columns three to five give the minimum, the average and the maximum width of the atomic cospan decompositions which were processed. In the columns six to eight the minimum, average and maximum length of the processed atomic cospan decompositions is given. The last three columns give the minimum, average and maximum time (in seconds) which were needed to process the input graphs.

		Width			Length			Time (in sec)		
$\lvert V_G \rvert$	Ratio	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10	11%	1	2.32	4	20	31.80	78	< 1	< 1	< 1
20	64%	3	5.24	9	74	201.28	564	< 1	1	45
30	97%	6	8.74	11	246	737.06	1 345	< 1	283	2894
40	–	–	–	–	–	–	–	–	–	–
50	–	–	–	–	–	–	–	–	–	–

Table 9.11.: Results for the membership tests $[G] \in \mathcal{L}_{3C}$

The tests for graphs with 40 (50) nodes could not be finished successfully due to a high memory consumption. But note that the graph automaton accepting the language $\mathcal{L}_{3C}$ with a maximum interface size of 15 already contains 2 573 485 501 354 560 states.

In Table 9.12 and in Table 9.13 we present the results of the membership tests for $[G] \in C_{(3)}$ and for $[G] \in C_{(4)}$. The information given in the several columns are the same as in Table 9.11.

		Width			Length			Time (in sec)		
$\lvert V_G \rvert$	Ratio	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10	100%	2	2.40	5	21	33.52	111	< 1	< 1	< 1
20	100%	2	5.22	8	61	196.13	512	< 1	< 1	< 1
30	95%	5	9.24	13	176	829.06	1 591	< 1	< 1	< 1
40	65%	10	14.54	20	1 393	2 682.40	5 163	< 1	2	69
50	14%	15	20.09	25	3 623	6 069.74	8 828	< 1	18	293

Table 9.12.: Results for the membership tests $[G] \in C_{(3)}$

		Width			Length			Time (in sec)		
$\lvert V_G \rvert$	Ratio	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10	100%	1	2.43	5	20	34.70	98	< 1	< 1	< 1
20	100%	2	5.07	10	68	181.02	528	< 1	< 1	< 1
30	100%	6	9.12	13	268	823.61	1 630	< 1	1	15
40	100%	9	14.07	18	1 217	2 507.70	3 870	< 1	350	3 172
50	–	–	–	–	–	–	–	–	–	–

Table 9.13.: Results for the membership tests $[G] \in C_{(4)}$

The tests for the $[G] \in C_{(4)}$ case study were only successful for graphs with 40 or less nodes. In case of graphs with 50 nodes the test could not be finished successfully due to a high memory consumption. Note that even if the graph automaton with a maximum interface size of 25 for the language $C_{(4)}$ contains "only" 1 501 199 875 790 165 states, the graph automaton is highly non-deterministic which leads to BDDs with a rather great number of BDD nodes. This is due to the fact that in many cases there exists a great number of valid colorings (which are all equivalent).

In Table 9.14 we present the results of the membership tests for $[G] \in D_{(k)}$. In the second column the different values for k are given, i. e. the bound of the size of the dominating set. All other information given in the several columns are the same as in Table 9.11. Again, the tests for the $[G] \in D_{(14)}$ case study for graphs with 50 nodes have been cancelled prematurely due to a high memory consumption. Also in this case the problem is rather the size of the graph automaton with maximum interface size of 25 accepting the language $D_{(14)}$, which contains 19 063 993 712 460 states, but the high non-determinism which is instrinsic to this kind of graph automata.

			Width			Length			Time (in sec)		
$\lvert V_G \rvert$	k	Ratio	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10	6	52%	1	2.36	5	20	32.74	92	< 1	< 1	< 1
20	8	44%	2	5.08	8	59	192.14	478	< 1	< 1	< 1
30	10	84%	5	9.49	13	297	903.01	2 146	< 1	1	18
40	12	95%	10	14.23	19	1 318	2 590.67	4 367	< 1	142	1 627
50	14	–	–	–	–	–	–	–	–	–	–

Table 9.14.: Results for the membership tests $[G] \in D_{(k)}$

In Table 9.15 and in Table 9.16 we present the results of the membership tests for $[G] \in V_{(k)}$ and for $[G] \in \mathit{NonIso} \cap V_{(k)}$. In the second column of both tables the different values for k are given, i. e. the bound of the size of the vertex cover. All other information given in the several columns are the same as in Table 9.11.

As already mentioned in Chapter 5 we supervised a student, Weixiang Guan, to work on the development of more efficient heuristics to compute atomic cospan decompositions [79]. Since the results of this master's thesis seem to be very promising, perhaps we could tackle graphs with a greater pathwidth (which corresponds to the maximum

			Width			Length			Time (in sec)		
$\lvert V_G \rvert$	k	Ratio	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10	6	100%	2	2.46	5	21	33.82	90	< 1	< 1	< 1
20	7	39%	2	4.87	8	68	173.71	551	< 1	< 1	< 1
30	14	46%	6	9.27	13	246	860.97	1 687	< 1	< 1	< 1
40	21	54%	10	14.22	20	1 127	2 682.81	4 199	< 1	< 1	3
50	28	50%	15	19.87	24	3 450	5 917.50	8 606	2	19	108

Table 9.15.: Results for the membership tests $[G] \in V_{(k)}$

			Width			Length			Time (in sec)		
$\lvert V_G \rvert$	k	Ratio	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10	6	3%	2	2.40	4	23	34.02	85	< 1	< 1	< 1
20	7	1%	2	5.00	9	62	181.34	527	< 1	< 1	< 1
30	14	6%	6	9.47	14	357	874.03	1 508	< 1	< 1	< 1
40	21	26%	11	14.64	18	1 430	2 707.77	4 610	< 1	< 1	3
50	28	30%	15	20.05	25	3 501	6 071,29	9 896	< 1	22	139

Table 9.16.: Results for the membership tests $[G] \in \mathit{NonIso} \cap V_{(k)}$

interface size of the graph automata). But even without these optimizations Raven can compete with other tools which are used to solve the membership problem for different graph problems. In [91] Kneis, Langer and Rossmanith solved the membership problem for minimum vertex cover and 3-colorability only for grids of graphs (with approximately 200 nodes). We have not given results for these case studies obtained by Raven, but since grids are somewhat path-like it is obvious that we can simply decompose and process this kind of graph. In [47, 59] Courcelle and Durand gave a short overview of membership problems which can be solved by the tool AutoWrite. In case of the k-colorability Courcelle and Durand could obtain automata for graphs with a cliquewidth[3] which is less or equal to 2 (for $k = 3$) or for graphs with a cliquewidth which is less or equal to 3 (for $k = 2$). Since they gave no concrete runtimes it is quite difficult to compare Raven and AutoWrite in detail. But even if the results of the membership tests are rather good, Raven can not compete with heuristics which can be used to solve a fixed problem such as k-colorability [53, 100, 107]. However, the focus of Raven lies on the area of verification, anyway. Hence, we are more interested in closure properties and decision procedures for language inclusion/equivalence rather than membership.

[3] *Cliquewidth* is a notion similar to treewidth and pathwidth with the following connection: A graph with bounded treewidth has also a bounded cliquewidth and a graph with a bounded cliquewidth has also a bounded pathwidth.

9.5. Conclusion

In this chapter we presented different experimental results which we have obtained by the use of the tool suite Raven (cf. Chapter 8). First of all we successfully verified the multi-user file system case study from [15]. Furthermore, we have seen that the forwards antichain algorithm is in most cases very efficient. However, the simulation-based antichain algorithm as well as the bisimulation up-to congruence algorithm have been useful for some case studies.

The reason that the simulation-based antichain algorithm is not very efficient in most cases is that the computation of the simulation relation is often very costly (measured in the runtime) – compared to the time needed to run the several antichain algorithms. Hence, the computation of the simulation relation is only advisable if

- either both automata are rather huge (or the concrete state space which has to be searched is rather huge), but have a similar behavior
- or at least the first automaton is rather huge compared to the second automaton.

An example for the first case can be found in the case study $C_{(3)} \subseteq C_{(4)}$. Due to the high non-determinism of both automata the state space (containing pairs of states of both automata, cf. Section 7.2.1) which has to be searched by the antichain algorithms is rather huge. But many of the states are behaviorially equivalent, since the colorings are often interchangeable. The simulation relation can be used to speed up the search of the state space, since it takes the behavior of states into account. Therefore, the simulation-based antichain algorithm is better applicable in this case than the other two antichain algorithms.

	Maximum Interface Size							
Automaton	3	4	5	6	7	8	9	10
$\mathcal{A}_{D_{(2)}}$	120	363	1 092	3 279	9 840	29 523	88 572	265 719
$\mathcal{A}_{D_{(3)}}$	160	484	1 456	4 372	13 120	39 364	118 096	354 292
$\mathcal{A}_{D_{(4)}}$	200	605	1 820	5 465	16 400	49 205	147 620	442 865

Table 9.17.: Number of states for different dominating set graph automata depending on the maximum interface size

	Maximum Interface Size							
Automaton	3	4	5	6	7	8	9	10
$\mathcal{A}_{V_{(2)}}$	45	93	189	381	765	1 533	3 069	6 141
$\mathcal{A}_{V_{(3)}}$	60	124	252	508	1 020	2 044	4 092	8 188
$\mathcal{A}_{V_{(4)}}$	75	155	315	635	1 275	2 555	5 115	10 235

Table 9.18.: Number of states for different vertex cover graph automata depending on the maximum interface size

	Maximum Interface Size			
Automaton	3	4	5	6
$\mathcal{A}_{\text{NonIso}\cap V_{(2)}}$	510	2 046	8 190	32 766
$\mathcal{A}_{\text{NonIso}\cap V_{(3)}}$	680	2 728	10 920	43 688
$\mathcal{A}_{\text{NonIso}\cap V_{(4)}}$	850	3 410	13 650	54 610

	Maximum Interface Size			
Automaton	7	8	9	10
$\mathcal{A}_{\text{NonIso}\cap V_{(2)}}$	131 070	524 286	2 097 150	8 388 606
$\mathcal{A}_{\text{NonIso}\cap V_{(3)}}$	174 760	699 048	2 796 200	11 184 808
$\mathcal{A}_{\text{NonIso}\cap V_{(4)}}$	218 450	873 810	3 495 250	13 981 010

Table 9.19.: Number of states for different automata depending on the maximum interface size

An example for the second case can be found in the case studies $V_{(4)} \not\subseteq D_{(4)}$, $\mathit{NonIso} \cap V_{(3)} \subseteq D_{(3)}$ and $\mathit{NonIso} \cap V_{(4)} \subseteq D_{(4)}$. Since in all these cases the first automaton is rather huge and has much more states than the second automaton (see Table 9.17, Table 9.18 and Table 9.19), the state space cannot be represented very efficiently. This is due to the fact that the pairs (which contain a single state of the first automaton and a set of states of the second automaton) are encoded by BDDs. But BDDs are only advantageous if used to represent sets (of states), otherwise this is equivalent to an explicit encoding. To compensate this disadvantage it is important to reduce the number of pairs which have to be processed. Again, the simulation-based antichain algorithm is better applicable in these cases, since the simulation-relation can help to rule out unnecessary pairs.

The bisimulation up-to congruence algorithm has also obtained good results for some case studies. In the $C_{(4)} \not\subseteq C_{(3)}$ case study we were able to proof that the language inclusion does not hold up to a maximum size of 10 – we have not computed runtime results for a higher maximum interface size. With the other algorithms this has not been possible. On the one hand this is caused by the fact that the colorability automaton for the language $C_{(4)}$ is larger (in number of states) than the automaton for the language $C_{(3)}$ (cf. Table 9.20). As already mentioned above it is inappropriate for the forwards and backwards antichain algorithm if the first automaton has a larger size than the second automaton due to the "explicit representation" of the states of the first automaton. On the other hand it has not been possible to compute the simulation relation for colorability automata of a maximum interface size greater than 6 within the given time limit. Hence, the simulation-based antichain algorithm has not been applicable for tests with a maximum interface size of 7 or more.

In the case studies $V_{(3)} \not\subseteq D_{(3)}$ and $V_{(4)} \not\subseteq D_{(4)}$ the bisimulation up-to congruence algorithm has given results which are equivalent to that of the forwards and backwards antichain algorithms or are slighty better respectively. The higher the maximum interface size, the more noticeable is the margin between the runtimes of the bisimulation up-to congruence algorithm and the other algorithms. The advantage of the bisimulation

	Maximum Interface Size							
Automaton	3	4	5	6	7	8	9	10
$\mathcal{A}_{C_{(2)}}$	15	31	63	127	255	511	1 023	2 047
$\mathcal{A}_{C_{(3)}}$	40	121	364	1 093	3 280	9 841	29 524	88 573
$\mathcal{A}_{C_{(4)}}$	85	341	1 365	5 461	21 845	87 381	349 525	1 398 101

Table 9.20.: Number of states for different colorability graph automata depending on the maximum interface size

up-to congruence algorithm, in contrast to the forwards and backwards antichain algorithm, is that it considers the behavior of states to rule out unnecessary states. But in contrast to the simulation-based antichain algorithm the (bi)simulation relation is not computed before but on-the-fly. For this kind of example it is not wise to pre-compute such a relation, since the counter-examples can be computed quite fast.

A somewhat different situation can be observed for the case study $D_{(k)} \not\subseteq V_{(k)}$. For the cases $D_{(2)} \not\subseteq V_{(2)}$ and $D_{(3)} \not\subseteq V_{(3)}$ it is clear that the bisimulation up-to congruence algorithm is better than the other three algorithms. But in the case of $D_{(4)} \not\subseteq V_{(4)}$ the pre-computation of the simulation relation is more beneficial. The reason is that the bisimulation up-to congruence algorithm has the disadvantage that the language inclusion test is reduced to the language equivalence problem by computing the product automaton of both automata and checking whether the language of the product automaton is equivalent to the language of the second automaton (cf. Subsection 7.2.3). In contrast, for the simulation-based antichain algorithm it is not necessary to compute the product automaton.

However, we have seen that both the simulation-based antichain algorithm and the bisimulation up-to congruence algorithm have their benefits. To combine the advantages of the backwards antichain algorithm either with the simulation-based antichain algorithm or the bisimulation up-to congruence algorithm, we would like to consider (and implement) backwards variants of both the simulation-based and the bisimulation up-to congruence algorithm. But at least in case of the simulation-based antichain algorithm it is not obvious how to obtain a backwards variant.

"The best way to predict the future is to invent it."
Alan Kay (1940 – present)

10 Conclusion

In this chapter we draw the thesis to a close. First, we give an overview of related work. Subsequently, we summerize the theory and applications presented in this thesis. Finally, we finish with possible future work.

10.1. Related Work

In this section we discuss similarities and differences to related work.

Recognizability: The categorial notion of recognizability used in this thesis has originally been introduced by Bruggink and König [27]. When instantiated to the category of graphs, which yields the recognizable graph languages presented in Chapter 6, this notion is equivalent to Courcelle's notion of recognizability [44]. For a detailed comparison see [27]. Another notion of recognizable languages which is also equivalent to the notion of Bruggink and König has been introduced by Griffing [78]. In contrast to the other categorial approach Griffing does not introduce the notion of graph automata or automaton functors respectively, but characterizes recognizable languages (which are called composition-representative subsets by Griffing) in terms of two equivalent notions. The first notion is characterized via locally finite congruences. The second notion is based on a functor from some category into a category with finite hom-sets, whereas the recognizable languages are the preimages of subsets of a finite hom-set. Both characterizations yield the same notion of recognizability defined by Courcelle.

Furthermore, Bozapalidis and Kalampakas presented in [28] another notion of recognizable graph languages which is based on magmoids of graphs. A magmoid is a doubly-ranked alphabet equipped with two binary associative, distributive operations, called product and sum, and a family of units for each operation. In case of the magmoid of graphs the two ranks can be identified as an inner and an

outer interface respectively. In addition, the product of two graphs is obtained by taking the disjoint union of the two involved graphs and subsequently fusing the corresponding interfaces. The sum of two graphs is obtained by merely taking the disjoint union of the involved graphs and the concatenation of the corresponding interfaces. In [28] Bozapalidis and Kalampakas have shown that this notion is also equivalent to Courcelle's notion of recognizability (and therefore is also equivalent to the notion presented here), if only graphs with rank zero, i. e. with empty interfaces, are considered.

In [81], Habel, Kreowski and Vogler introduced the notion of compatible graph properties which arise in the context of hyperedge-replacement grammars. The idea of compatibility is as follows: A graph property is *compatible*, if the property can be checked for each graph generated by a hyperedge-replacement grammar by checking the property for the components and composing the sub-results to a result for the entire graph. Furthermore, Lengauer and Wanke introduced the notion of finite graph properties [99], which also originates from the context of hyperedge-replacement grammars. A graph property is said to be *finite* if there exist only a finite number of classes of graphs that behave differently w. r. t. the graph property. The connection between compatible, finite and recognizable graph properties (represented by inductive graph properties introduced by Bauderon and Courcelle [7, 44]) has been investigated by Habel, Kreowski and Lautemann in [80]. Habel et al. have shown that all three notions are essentially equivalent.

Graph Automata and Logics: In the last 40 years graph automata have already been considered by other authors. To some extent, their approaches differ considerably from the graph automata presented in Chapter 6. Originally, graph automata have been introduced by Milgram [105], as well as Wu and Rosenfield [131] as generalizations of finite automata which consist of graph-structured instead of string-oriented tapes. The main idea is to place an automaton on a node (or an edge respectively) of some labeled input graph and to move the automaton to another adjacent node (or another adjacent edge) while changing the state of the automaton and the label of the node (edge). Due to this method of operation the input graph has to be connected and, possibly, the degree of each node has to be bounded by some constant. However, the computational power of these automata is equivalent to Turing machines (in case of the automata introduced by Milgram) or linear bounded automata (in case of the automata introduced by Wu and Rosenfield) respectively.

Another kind of graph automata which have a similar concept to the graph automata of Milgram, Wu and Rosenfield, but which have less descriptive power have been introduced by Kreowski and Rozenberg [93]. These automata are a special kind of finite automata which accept so-called Eulerian circuit languages, i. e. languages which contain only connected graphs consisting of nested circuits. Hence, there is a great difference between the automaton models explained above and the automaton model introduced in this thesis.

A further notion of graph automata has been considered by Brandenburg and Skodinis [29, 30] with the aim of defining automata which accept a class of languages generated by a specific class of graph grammars. Their approach deals with automata for so-called linear graph languages which are generated by linear

NCE graph grammars [66]. In some sense these linear NCE graph grammars are an extension of linear context-free word grammars to the setting of graphs. Therefore, this kind of graph automaton is different to the graph automata presented here, since the graph automata of this thesis deal with recognizable graph languages rather than linear graph languages.

In [27] Bozapalidis and Kalampakas introduced a weaker notion of graph automata which are based on a special kind of magmoids, called graphoids. By this kind of graph automata a new class of recognizable graph languages is introduced which is a proper subclass of the class of recognizable graph languages presented in this thesis. For a detailed comparison between these two notions of recognizability see [27].

One of the issues when working with graph automata is their huge size (in the number of states) which grows exponentially in the maximum width of the accepted graphs. To overcome this difficulty, Courcelle and Durand [45, 46, 59] introduced so-called fly-automata, which are used to verify monadic second-order logic definable properties on graphs of bounded width. The main difference between fly-automata and the graph automata presented in Chapter 6 is that in fly-automata the state set and the transition function are represented as computable functions rather than in an explicit or a symbolic way. This is motivated by the fact that fly-automata of Courcelle and Durand are used primarily to decide whether a given graph satifies a given MSOGL-formula. This differs from the approach presented in this thesis where graph automata are used particularly to represent sets of graphs.

Another approach to overcome the state space explosion problem is presented by Kneis, Langer and Rossmanith [90, 91]. Their work abstains from the representation of a graph language as an automaton, but establishes a game-theoretic approach between two players, the *verifier* and the *falsifier*. The game is identified with an algorithm that bottom-up traverses a tree decomposition of the input graph. In simple terms, the algorithm tries to evaluate the given graph property at each bag of the decomposition. If the graph property can already be decided, the algorithm terminates, otherwise the tree decomposition is traversed further. This approach is useful for deciding the satisfiability of monadic second-order formulas, similar to the work of Courcelle and Durand. Hence, this approach has a different area of application than the approach presented in this thesis.

In Chapter 8 we have already mentioned the tools MONA, AUTOWRITE and ALASKA. The first one is used to encode monadic second-order tree formulas into tree automata [89]. The second one is a tool to verify monadic-second order definable properties on graphs of bounded tree- or clique-width. The last tool offers verification methods based on the antichain-based approach similar to the techniques presented in Chapter 7. For a comparison between these tools and RAVEN see Section 8.4.

Verification: Invariant checking for graph transformation systems has already been considered by other authors. In the work of Fradet and Le Métayer [69] and in the work of Bakewell, Plump and Runciman [6] so-called shapes are introduced which are used to describe sets of graphs satisfying specific properties. The former

approach is based on context-free graph grammars used to describe shapes, the latter uses more expressive graph reduction systems instead. However, the idea of both methods is to verify a graph transformation system by checking how it may change the shape of a graph.

Another related work by Habel, Pennemann and Rensink [82] presents an approach for computing weakest preconditions of application conditions, which are equivalent to first-order graph logic. It is possible to use this approach for invariant checking in the following way: for every transformation rule it has to be shown that the weakest precondition of the transformation rule is implied by the invariant. Note that recognizable graph languages are more expressive than application conditions, since every monadic second-order graph logic formula is known to specify a recognizable graph language [44].

In [8] Becker et al. consider an approach for verifying safety properties in the setting of mechatronic systems. Safety properties are modeled by the use of so-called graph patterns consisting of two distinguished parts: a positive and a negative one. Intuitively, a graph pattern represents all graphs that contain the positive part as a subgraph, but do not contain the (complete) negative part. In order to verify (or falsify) a safety property Becker et al. perform a backwards reachability analysis for hyperedge replacement grammars. If the initial graph is backwards reachable from some forbidden system state by the application of some transformation rule, the system has been falsified. Otherwise the system is verified. To make this method efficient a symbolic implementation based on binary decision diagrams is used, but in a somewhat different setting than the one presented in Chapter 7.

10.2. Summary

In this thesis we presented a concrete and automaton-theoretic view on automaton functors called graph automata. This new automaton model is designed to accept recognizable languages of cospans with a bounded width. Therefore, we have introduced several notions of cospan width which depend on different types of cospan decompositions. In contrast to the situation in classical graph theory, there is more than one choice how to obtain decompositions of cospans. On the one hand we can decompose a cospan in a path-like manner, similar to path decompostions, or in a tree-like manner, similar to tree decompositions. But on the other hand we have also the choice whether to identify bags with interfaces or with the center graph of a cospan. However, as we have seen these notions all coincide either with the classical notion of pathwidth or with the classical notion of treewidth. In addition we have introduced a further type of decomposition into atomic cospans for both path-like and tree-like cospan decompositions.

Especially the path-like atomic cospan decompositions play an important role, since graph automata do not consider all cospan decompostions but only decompositions in atomic cospans. This is a major difference between graph automata and automaton functors, because an automaton functor (in the category of cospans of graphs) has to be defined on all cospans of graphs. However, we have seen that it suffices to take only atomic cospans into account, since every (bounded) cospan can be obtained from a

(finite) sequence of atomic cospans.

Graph automata have also another advantage over automaton functors. Since graph automata can be seen as a special kind of finite automata (due to their boundedness), we can apply classical and recent algorithms for finite automata, if we perform some minor changes to these algorithms. The modifications are necessary not only because of the differences in the range of use between graph automata and finite automata, but also for the symbolic representation in terms of binary decision diagrams, which we have used. Note that this symbolic representation is one of the keystones to efficiently implement graph automata, because an explicit representation is impossible due to huge size of graph automata – even for a rather small maximum interface size and rather simple examples. Another keystone for an efficient implementation of graph automata are good, i. e. fast and scaling, algorithms. One of the main applications of the theory presented in this thesis is to automatically check whether a language, given as an automaton, is an invariant of a graph transformation system, which can be reformulated as a language inclusion problem. Hence, we can use recent algorithms by Henzinger et al. [132], Abdulla et al. [1] as well as Bonchi and Pous [23] for the language inclusion problem, which yield very promising results.

Furthermore we have started to tackle another problem: So far we had to construct graph automata very directly, but this is hard because of the consistency property which has to be enforced for graph automata. For this reason we have presented the LCL-logic which can be used for generating graph automata from formulas. But in contrast to the MSOGL-logic the atomic formulas of the LCL-logic are predefined graph properties which may be rather complex. The idea is to avoid the cost-intensive construction of graph automata for such graph properties from (complex) MSOGL-formulas to make our logic more efficient in practice. But the idea is not only to simply transform an LCL-formula to a graph automaton, but to use the logic to potentially simplify constructions or to automatize the computation of invariants.

Beside the theoretical results we have also presented a tool suite for computing and manipulating (bounded) graph automata, called RAVEN. The tool suite offers the possibility to compute a number of pre-defined graph automata for different graph languages. We have also implemented methods for computing the union and the intersection of graph automata, as well as a method to "shift" a cospan over a given automaton. In addition we have implemented different algorithms for both universality and language inclusion checking for recognizable graph languages which are based on different recent approaches. Finally, we have given a number of examples and corresponding runtime results.

10.3. Future Work

In this section we discuss some ideas which could be used as a starting point for future work.

As stated in Section 6.1 apart from graph automata which are used to process path-like cospan decompositions there is also the notion of consistent tree automata which are used to process tree-like cospan decompositions. So far we have not implemented this kind of automata, but it could be profitable, since in general the treewidth of a graph is much smaller than its pathwidth. Hence, the maximum interface size needed to process a graph can be reduced if one uses consistent tree automata instead of

graph automata. In turn consistent tree automata depend on the *join*-operation whose computation is often very costly. Therefore, it has to be investigated whether consistent tree automata can help to further increase the efficiency of Raven.

Another improvement of Raven could be achieved by considering node-labeled graphs. For example, the complexity of the multi-user file system case study of Section 9.1 could be decreased, because we could omit the edges of arity 1 which are used to "label" each node. But before this type of graphs can be used, the question of how the underlying theory has to be adapted must be answered. In any case, this would yield more atomic cospans, since a different *vertex*-operation is needed for every node label, similar to the situation with the *connect*-operation. Hence, the reduction in complexity, i. e. the size of the system model, causes a more complex alphabet of the underlying automaton. It has to be checked whether this modification can be used to increase the efficiency of Raven.

In Section 6.2.2 we introduced the Lcl-logic whose purpose is to be used as generator of graph automata from Lcl-formulas. But it is still unclear how expressive this logic is. Since the set of atomic formulas can be identified with (the language of) arbitrary graph automata, it is possible to provide a graph automaton which accepts a non-Msogl-definable language. Hence, it is possible to express properties which are not Msogl-expressible. But in return, we want to prohibit formulas which contain (arbitrary) negations. Therefore, it could be possible to express properties in Msogl, which are not necessarily Lcl-expressible.

To better classify the Lcl-logic we could consider a set of predefined atomic formulas such that each atomic formula is identified with a singleton set. It would be interesting to compare the expressive power of this variant with the expressive power of conditional reactive systems [31] (which are equivalent to first-order graph logic). But note, that in case of the Lcl-logic both interfaces of the considered cospans are fixed, whereas in case of the conditional reactive systems only the inner interface is fixed. Currently, it is not clear how to fix this gap. This is a point for future work.

Furthermore, it would be interesting to use the Lcl-logic in combination with Raven in order to compute the weakest pre-condition and the strongest post-condition for a given set of transformation rules and to check whether a given set of graphs satisfies the weakest pre-condition (strongest post-condition). Such a method could serve as the basis for automatic invariant generation as explained in Chapter 6.

Another approach to the verification of distributed and infinite-state systems is regular model checking [25, 26]. The idea of this approach is to represent sets of states as regular languages, which are given as finite automata, and transitions of the system as regular relations, given as finite-state transducers. Since regular languages are closed under rule applications, the set of successors of a set of states is also a regular language. However, a main problem of this approach is the size of the finite automaton resulting after a number of rule applications. Hence, widening techniques are needed in order to over-approximate the automata and to bound their size. Since this approach is very successful and has also been extended to the setting of regular tree languages, it is natural to ask for an extension to the setting of recognizable graph languages. For this purpose a kind of graph transducer has to be introduced. There already exists the notion of Msogl-definable transductions by Courcelle [43], but this notion has two major drawbacks. The first is that these transductions are very complex and the second is that they do not guarantee to transform a recognizable graph language into another

recognizable graph language in general. Therefore, it has to be further investigated how finite-state transducers can be generalized in a way that make them useful for "regular graph model checking".

In [35] Bruggink, König and Zantema introduce two approaches for proving termination of graph transformation systems, i. e. to show the absence of transformation sequences of infinite length. One of these two approaches is based on so-called type graphs which are used to define languages of graphs. The language of a type graph, which is just some fixed graph T, consists of all graphs G for which a morphism exists which maps G to T. The approach of Bruggink, König and Zantema then assigns weights to graphs by means of a weighted type graph. It can be shown that a graph transformation system which uses as initial graphs only the language of type graph is terminating if the weights of the transformation decrease in each derivation step. This idea is interesting for recognizable graph languages, because the language of a type graph is recognizable. In [95] Küpper has shown that a straight-forward generalization of an approach for proving termination of string rewriting systems which are closed under regularity to the setting of graph transformation systems and recognizable graph languages is not possible. This is due to the fact that recognizable graph languages do not have the same closure properties as regular word languages. Hence, it has to be further investigated how recognizable graph languages can be successfully used in the termination analysis of graph transformation systems.

Part IV.

Appendix

A
Proofs

A.1. Proofs of Chapter "Tree, Path and Cospan Decompositions"

Lemma 5.5. Let $c = J \xrightarrow{c_L} G \xleftarrow{c_R} K$ be a linear cospan, i. e. c_L, c_R are monos. Then there exist atomic cospans $a_1, \ldots, a_n$ such that $c = a_1 ; \cdots ; a_n$.

In fact, there exist such atomic cospans $a_1, \ldots, a_n$ such that the following condition holds: Let $a_i = D_{n_{i-1}} \to H_i \leftarrow D_{n_i}$, for $1 \leq i \leq n$. It holds that $|D_{n_i}| \leq |G|$ for all $0 \leq i \leq n$ and $|H_i| \leq |G|$ for all $1 \leq i \leq n$.

Proof. Let $J = D_n$, $K = D_m$ and $G = \langle V, E, att, lab \rangle$. We can assume without loss of generality that $V_G = \mathbb{N}_{|G|}$ and $c_L(v) = v$ for every node v of D_n. Assume furthermore, that $E = \{e_1, \ldots, e_k\}$ and define $A_i = lab(e_i)$.

We construct c using the following atomic cospans:

$vertex_n^n ; vertex_{n+1}^{n+1} ; \cdots ; vertex_{|V|-1}^{|V|-1} ;$	add enough nodes to the graph
$connect_{A_1,\theta_1}^{|V|} ; \cdots ; connect_{A_k,\theta_k}^{|V|} ;$	connect nodes with edges
$perm_\pi^{|V|} ;$	move nodes of outer interface to the front
$res_{|V|}^{|V|} ; res_{|V|-1}^{|V|-1} ; \cdots ; res_{|V|-m+1}^{|V|-m+1}$	remove appropriate nodes

where the θ_i are functions $\theta_i \colon \{1, \ldots, |att(e_i)|\} \to \{1, \ldots, |V|\}$ where $\theta_i(j)$ returns the j-th node attached to edge e_i. Furthermore, π is a bijection on $\{1, \ldots, |V|\}$ with $\pi(j) = f(j)$ for $1 \leq j \leq m$ and is arbitrary otherwise.

The second condition of the lemma also holds for the above atomic cospans. □

Lemma 5.9.

1. Let $\mathcal{P}$ be a path decomposition of a graph G. There exists a graph-bag decomposition $\vec{c}$ of G such that $wd_{\mathrm{gb}}(\vec{c}) = wd(\mathcal{P})$.
2. Let $\vec{c}$ be a graph-bag decomposition of G. There exists an interface-bag decomposition $\vec{d}$ of G such that $wd_{\mathrm{ib}}(\vec{d}) = wd_{\mathrm{gb}}(\vec{c})$.
3. Let $\vec{c}$ be a graph-bag decomposition of G. There exists an atomic cospan decomposition $\vec{d}$ of G such that $wd_{\mathrm{at}}(\vec{d}) \leq wd_{\mathrm{gb}}(\vec{c})$.
4. Let $\vec{c}$ be an interface-bag decomposition of G. There exists a path decomposition $\mathcal{P}$ of G such that $wd(\mathcal{P}) = wd_{\mathrm{ib}}(\vec{c})$.

Proof.

1. Let $\mathcal{P} = \langle P, X\rangle$, with $P = 1 - \cdots - n$ and $G = \langle V, E, att, lab\rangle$. We construct the cospan path decomposition $\vec{c} = c_1, \ldots, c_n$ (where $c_i = J_{i-1} \to G_i \leftarrow J_i$, for $1 \leq i \leq n$) as follows:

   Let $G_i = \langle V_i, E_i, att_i, lab_i\rangle$ be the graph which contains the nodes in X_i and all edges of G which are connected only to nodes of X_i and are not in some G_j, with $j < i$. Furthermore, let $J_0 := \emptyset$ and $J_n = \emptyset$ and, for $1 \leq i < n$, let J_i be the discrete graph with node set $V_i \cap V_{i+1}$.

   We claim that $\vec{c}$ is a graph-bag cospan path decomposition of G with $wd_{\mathrm{gb}}(\vec{c}) = wd(\mathcal{P})$. By construction, there is an injection from every J_i and G_i into G. Moreover, since $\mathcal{P}$ is a path decomposition of G, every node of G appears in at least one G_i, while every edge appears in exactly one G_i. Also, since the bags containing a node form a subpath, nodes that appear in more than one G_i will be fused. Thus, $Colim(\vec{c}) = G$.

   Also by construction, each G_i corresponds to some bag X_i. Thus, $wd(\mathcal{P}) = wd_{\mathrm{gb}}(\vec{c})$ directly follows.

2. Define, for a cospan $c = J \to[c_{\mathrm{L}}] G \leftarrow[c_{\mathrm{R}}] K$, the pair of cospans $\hat{c}, \check{c}$, where
   - $\hat{c} = J \to[c'_{\mathrm{L}}] G^- \leftarrow[id] G^-$ and
   - $\check{c} = G^- \to[id'] G \leftarrow[c_{\mathrm{R}}] K$.

   where G^- is the discrete graph with the same node set as G and c'_{L} and id' describe the same morphism as c_{L} and id, respectively, but have different codomains. We observe that $\hat{c}\,;\check{c} = c$.

   Let $\vec{c} = c_1, \ldots, c_n$. We define $\vec{d} := \hat{c_1}, \check{c_1}, \ldots, \hat{c_n}, \check{c_n}$. By the observation in the previous paragraph, it holds that $Colim(\vec{c}) = Colim(\vec{d})$.

   By construction, one interface of both $\hat{c}$ and $\check{c}$ consists of the nodes of G. Therefore, both $\hat{c}$ and $\check{c}$ are jointly node-surjective, all nodes of each edge occur together in some interface, and $|\hat{c}|_{\mathrm{ib}} = |\check{c}|_{\mathrm{ib}} = |c|_{\mathrm{gb}}$. From this it follows, that $\vec{d}$ is an interface-bag decomposition and $wd_{\mathrm{ib}}(\vec{d}) = wd_{\mathrm{gb}}(\vec{c})$.

3. Let $\vec{c} = c_1, \ldots, c_n$. By Lemma 5.5, there exist, for $1 \leq i \leq n$, atomic cospan decompositions $\vec{a_i} = a_{i,1}, \ldots, a_{i,m_i}$ such that $a_{i,1} ; \cdots ; a_{i,m_i} = c_i$. Define:

$$\vec{d} = a_{1,1}, \ldots, a_{1,m_1}, \ldots, a_{n,1}, \ldots, a_{n,m_n}.$$

It follows directly from the previous observations that $\mathit{Colim}(\vec{d}) = \mathit{Colim}(\vec{c})$. From the second part of Lemma 5.5 it follows that $wd_{\mathrm{at}}(\vec{d}) \leq wd_{\mathrm{gb}}(\vec{c})$.

4. Let $\vec{c} = c_1, \ldots, c_n$, where $c_i = J_{i-1} \to G_i \leftarrow J_i$. By assumption, $\mathit{Colim}(\vec{c}) = G$. Let $f_i \colon J_i \to G$ (for $0 \leq i \leq n$) and $g_i \colon G_i \to G$ (for $1 \leq i \leq n$) be the morphisms given by the colimit construction. We construct the path decomposition $\mathcal{P} = \langle P, X \rangle$ as follows: $P = 0 - \cdots - n$, with $X_i = f_i(V_{J_i})$, for $0 \leq i \leq n$.

This is a path decomposition by the following arguments: First of all, since $\mathit{Colim}(\vec{c}) = G$ and all cospans are jointly node-surjective, all nodes of G must have a pre-image in some interface J_i along the morphism f_i, and therefore appear in the bag X_i. Since $\vec{c}$ is an interface-bag decomposition, (the pre-images of) all nodes connected to a single edge must appear together in some interface and therefore the nodes must appear together in some bag. finally, suppose there is a node v of G and bags X_p, X_q and X_r with $p < q < r$, such that v has a pre-image in X_p and X_r (over f_p and f_r, respectively). Since the colimit construction on graphs takes the disjoint union and then factors through the smallest equivalence relation which equates nodes that have a common pre-image, it must be the case that X_q contains a pre-image of v (along f_q).

From the facts that, by construction, $X_i = f_i(V_{J_i})$, and all f_i are injective, it follows that $wd(\mathcal{P}) = wd_{\mathrm{ib}}(\vec{c})$.

□

Lemma 5.16.

1. Let $\mathcal{T}$ be a tree decomposition of G. There exists a star decomposition $\mathcal{S}$ of G such that $wd_\star(\mathcal{S}) = wd(\mathcal{T})$.
2. Let $\mathcal{S}$ be a star decomposition of G. There exists a costar decomposition $\mathcal{C}$ of G such that $wd_\star(\mathcal{C}) = wd_\star(\mathcal{S})$.
3. Let $\mathcal{C}$ be a costar decomposition of G. There exists a tree decomposition $\mathcal{T}$ of G such that $wd(\mathcal{T}) = wd_\star(\mathcal{C})$.
4. Let $\mathcal{S}$ be a star decomposition of G. There exists an atomic star decomposition $\mathcal{S}'$ of G such that $wd_\star(\mathcal{S}') = wd_\star(\mathcal{S})$.

Proof.

1. Let $\mathcal{T} = \langle T, X \rangle$. We choose an arbitrary total ordering $<$ on the edges of T. We construct the star decomposition $\mathcal{S} = \langle T, \tau \rangle$, where T is the tree component of $\mathcal{T}$ and τ is defined as follows: For each node $t \in V_T$ of T, we define $\tau(t) = \langle X_t, \emptyset, \emptyset, \emptyset \rangle$, that is the discrete hypergraph with node set X_t. For each edge $b = \{t_1, t_2\} \in E_T$

we define $\tau(b) = \tau(t_1) -id'\rightarrow G_b \leftarrow id''- \tau(t_2)$, where G_b is the graph with node set $X_{t_1} \cup X_{t_2}$ which contains those edges of G which are not contained in (the center graph of) some cospan $\tau(b')$, where $b' < b$, and id' and id'' are the respective embeddings.

Now we need to show that $\mathcal{S}$ is a star decomposition of G. Let a graph G' and, for each node $t \in V_T$ and edge $b = \{t_1, t_2\} \in E_T$ of T, morphisms $f_t \colon \tau(t) \to G'$ and $f_b \colon G_b \to G'$ be given. We define a mediating morphism $h \colon G \to G'$.

Since the bags containing a node of G form a subtree of T, and every node of G occurs in some bag, for each node v of G the set $\{v' \in V_{G'} \mid f_x(v) = v' \text{ for some } x \in V_T \cup E_T\}$[1] must be a singleton (otherwise the diagram does not commute). Let w be the only element of the singleton. We must take $h(v) := w$ (otherwise the diagram does not commute). By construction, every edge $e \in E_G$ of G occurs in the domain of exactly one f_b. We must take $h(e) := g_b(e)$ (otherwise h is not a morphism). Now, h is the desired morphism, and it is unique, so $G = \mathit{Colim}(\mathcal{S})$.

Because the bags of $\mathcal{T}$ correspond one-to-one to (the node sets of) the graphs $\tau(t)$ of T, it is clear that $wd_\star(\mathcal{S}) = wd(\mathcal{T})$ and for all edges $e \in V_G$ it holds that the nodes adjacent to e occur together in $\tau(t)$ for some $t \in T$.

2. Let $\mathcal{S} = \langle T, \tau_\mathcal{S} \rangle$. We choose an arbitrary total ordering $<$ on the nodes of T; let $V_T = \{t_1, \ldots, t_n\}$, with $t_1 < \cdots < t_n$. Let $f_t \colon \tau_\mathcal{S}(t) \to G$ and $f_b \colon G_b \to G$, where $\tau_\mathcal{S}(b) = J \to G_b \leftarrow K$, be the morphisms given by the colimit construction.

   First we define a "skeleton" costar decomposition $\mathcal{B} = \langle T, \tau_\mathcal{B} \rangle$, where $\tau_\mathcal{B}$ is defined as follows:

   - for all tree nodes $t \in V_T$, $\tau_\mathcal{B}(t) := \tau_\mathcal{S}(t)$;
   - for all tree edges $b = \{t_1, t_2\} \in E_T$, where $\tau_\mathcal{S}(b) = J -\phi\rightarrow G \leftarrow\psi- K$, we define $\tau_\mathcal{B}(b) := J \leftarrow\phi'- G' -\psi'\rightarrow K$, where G' is obtained by taking the pullback of ϕ and ψ.

   Now, we construct the final costar decomposition $\mathcal{C} = \langle T, \tau_\mathcal{C} \rangle$ by adding the edges to appropriate graphs. For all $t \in V_T$, $\tau_\mathcal{C}(t)$ is built from $\tau_\mathcal{B}$ by adding all edges of G of which the adjacent nodes all have a pre-image (along f_t) in $\tau_\mathcal{C}(t)$, but which have no pre-image in $\tau_\mathcal{C}(t')$ for some $t' < t$.

   Since the "bags" of $\mathcal{C}$ correspond one-to-one to the "bags" of $\mathcal{S}$, $wd_\star(\mathcal{C}) = wd_\star(\mathcal{S})$. Furthermore, by construction, nodes of the graph $\tau_\mathcal{S}(t)$ which were mapped to the same node by the morphisms in the cospan $\tau_\mathcal{S}(b)$, are the image of the same node along the morphisms of the span $\tau_\mathcal{C}(b)$. Since all the edges of G are in the image of exactly one f_b, it must be the case that $\mathit{Colim}(\mathcal{C}) = \mathit{Colim}(\mathcal{S})$.

3. Let $\mathcal{C} = \langle T, \tau \rangle$. By assumption, $G = \mathit{Colim}(\mathcal{C})$. Let, for each $t \in V_T$ and $\{t_1, t_2\} \in E_T$, $f_t \colon \tau(t) \to G$ and $f_{\{t_1,t_2\}} \colon J_{\{t_1,t_2\}} \to G$ be the morphisms given by the colimit construction.

   We construct the tree decomposition $\mathcal{T} = \langle T, X \rangle$, where T is the tree from $\mathcal{C}$. If $\tau(t) = H$, then we let $X_t := f_t(V_H)$. We show that the structure thus constructed is a tree decomposition of G.

[1] Note that, by construction, the node and edge sets of the graphs occurring in the tree decomposition, are subsets of node and edge sets of G, respectively. Therefore, f_x can actually be applied to v.

First of all, since $G = \mathit{Colim}(\mathcal{C})$, every node of G must have a pre-image along some f_t, thus every node of G occurs in some bag X_t. Since interfaces are discrete, every edge of G occurs in the domain G_t of exactly one f_t, and thus the nodes adjacent to this edge occur together in X_t.

Furthermore, assume that for some node v the subgraph of bags of which v is an element do not form a subtree of T. That is, there are tree nodes t, t' and a tree node u on the path between t and t' such that $v \in X_t$, $v \in X_{t'}$ but $v \notin X_u$. Then we can show that $G \neq \mathit{Colim}(\mathcal{C})$ in a similar way as in the proof of Lemma 5.9 (iv).

4. By Lemma 5.5, we can transform any cospan with discrete interfaces and injective morphisms into an atomic cospan decomposition. That is, we can transform the edges of a star decomposition (labeled with cospans) into paths labeled with atomic cospans.

□

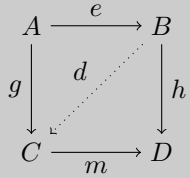
A.2. Proofs of Chapter "Recognizable Graph Languages"

Proposition 6.3 (Robustness)**.** Let a class $\mathcal{L}$ of output-linear cospans with discrete interfaces D_s and D_t be called *output-linear-recognizable* whenever $\mathcal{L}$ is recognizable in **OLCG**. Then $\mathcal{L}$ is recognizable in $\mathit{Cospan}(\mathbf{Graph})$ if and only if it is output-linear-recognizable.

For the proof of Proposition 6.3 we first need the notion of factorization structure.

Definition A.1 (Factorization Structure)**.** Let $\mathbf{C}$ be a category and let $\mathcal{E}, \mathcal{M}$ be classes of morphisms in $\mathbf{C}$. The pair $\langle \mathcal{E}, \mathcal{M} \rangle$ is called a *factorization structure* for $\mathbf{C}$ whenever

- $\mathcal{E}$ and $\mathcal{M}$ are closed under composition with isomorphisms,
- $\mathbf{C}$ has $\langle \mathcal{E}, \mathcal{M} \rangle$-factorizations of morphisms, i. e. each morphism $A \xrightarrow{f} C$ of $\mathbf{C}$ has a factorization $A \xrightarrow{f} C = A \xrightarrow{e} B \xrightarrow{m} C$ with $e \in \mathcal{E}$ and $m \in \mathcal{M}$, and
- $\mathbf{C}$ has the unique $\langle \mathcal{E}, \mathcal{M} \rangle$-*diagonalization property*: For each commutative square as shown below with $e \in \mathcal{E}$ and $m \in \mathcal{M}$ there exists a unique diagonal, i. e. a morphism d such that the diagram with the morphism $B \xrightarrow{d} C$ added commutes, i. e. $A \xrightarrow{e} B \xrightarrow{d} C = A \xrightarrow{g} C$, $B \xrightarrow{d} C \xrightarrow{m} D = B \xrightarrow{h} D$.



Note that in any category with an $\langle \mathcal{E}, \mathcal{M} \rangle$-factorization structure, the classes $\mathcal{E}$ and $\mathcal{M}$ are closed under composition and factorizations of morphisms are unique up to isomorphism [2].

For the proof of Proposition 6.3 below we consider the factorization structure $\langle \mathcal{E}, \mathcal{M} \rangle$ for the category **Graph** with $\mathcal{E} = \{e \mid e \text{ is an epimorphism in } \mathbf{Graph}\}$ and $\mathcal{M} = \{m \mid e \text{ is a monomorphism in } \mathbf{Graph}\}$. Note that the epimorphisms (monomorphisms) are the surjective (injective) morphisms in **Graph**.

Proof. We adapt a robustness proof by Bruggink and König [33]. For both directions of the proof, we use the characterization of recognizability via congruences (see Proposition 6.15).

(⇒): Trivial, because **OLCG** is a subcategory of *Cospan*(**Graph**), and thus we can use the congruence for *Cospan*(**Graph**) also in the case **OLCG**.

(⇐): Let $\equiv_R$ be a congruence for $\mathcal{L}$ in the sense of Definition 2.19 for the category **OLCG**. We now define an equivalence $\equiv'_R$ on *Cospan*(**Graph**) and show that it is a congruence of finite index (see Proposition 6.15).

For this we first need the notion of *merger pairs*. For a cospan $c\colon D_n \to_{c_\mathrm{L}} G \leftarrow_{c_\mathrm{R}} D_m$ the set $M(c)$ of merger pairs consists of all pairs of cospans $\langle a_n, a_m \rangle$ of the form

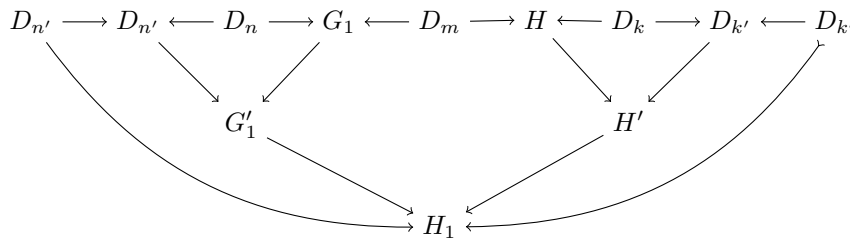
$$a_n\colon D_{n'} \xrightarrow{id} D_{n'} \xleftarrow{a_{n\mathrm{R}}} D_n, \qquad a_m\colon D_m \xrightarrow{a_{m\mathrm{L}}} D_{m'} \xleftarrow{id} D_{m'}$$

such that $a_{n\mathrm{R}}$ and $a_{m\mathrm{L}}$ are surjective and $a_n \,;\, c \,;\, a_m$ is an output-linear cospan. In a sense the merger pairs induce an equivalence on the interfaces which relates at least the interface items which have the same image under c_L or c_R. Furthermore, since we only deal with finite graphs, there are only finitely many "merger cospans" up to isomorphism.

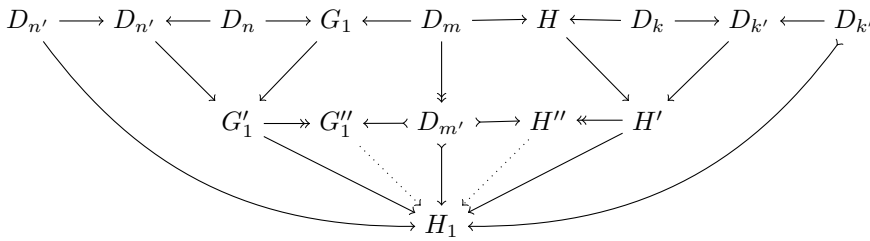
We now define $\equiv'_R$ as follows: for two cospans $c_1\colon D_n \to G_1 \leftarrow D_m$ and $c_2\colon D_n \to G_2 \leftarrow D_m$ it holds that $c_1 \equiv'_R c_2$ whenever $M(c_1) = M(c_2)$ and for all $\langle a_n, a_m \rangle \in M(c_1)$ we have $a_n \,;\, c_1 \,;\, a_m \equiv_R a_n \,;\, c_2 \,;\, a_m$. Note that for an input- and output-linear cospan c the pairs consisting of the identity cospans on D_n and D_m are among the merger pairs and hence $\equiv'_R$ is a refinement of $\equiv_R$. Furthermore it can be shown that $\equiv'_R$ is locally finite whenever $\equiv_R$ is.

We now have to show that $\equiv'_R$ is congruence. For this take two cospans $c_1 \equiv'_R c_2$ with $c_i\colon D_n \to G_i \leftarrow D_m$, where $i \in \{1, 2\}$, and another cospan $d\colon D_m \to H \leftarrow D_k$. We have to prove that $c_1 \,;\, d \equiv'_R c_2 \,;\, d$.

Now take a merger pair of $c_1 \,;\, d$, i. e., choose $\langle a_n, a_k \rangle \in M(c_1 \,;\, d)$. We will show that there are cospans $a_m, a_{m'}$ such that $\langle a_n, a_m \rangle \in M(c_1)$, $\langle a_{m'}, a_k \rangle \in M(d)$ and $a_n \,;\, c_1 \,;\, d \,;\, a_k = a_n \,;\, c_1 \,;\, a_m \,;\, a_{m'} \,;\, d \,;\, a_k$. For this consider the diagram below which represents the composition $(a_n \,;\, c_1) \,;\, (d \,;\, a_k)$ via pushouts where the order of composition is indicated by the brackets.

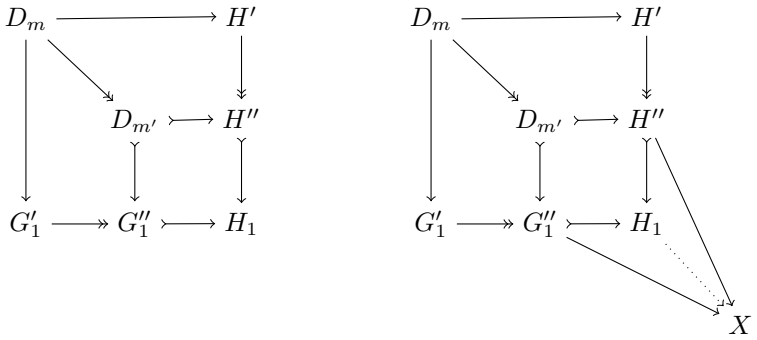


Note that the two "triangles" on left and right are pushouts and that the middle "pentagon" is also a pushout. In the following we will denote injective morphisms by $\rightarrowtail$ and surjective morphisms by $\twoheadrightarrow$. Now consider the morphism $D_m \to H_1$ and take its factorization into a surjective morphism $D_m \twoheadrightarrow D_{m'}$, followed by an injective morphism $D_{m'} \rightarrowtail H_1$. In the category of graphs this factorization always exists and is unique. In addition take the pushout of $D_m \to G_1'$, $D_m \twoheadrightarrow D_{m'}$ and of $D_m \to H'$, $D_m \twoheadrightarrow D_{m'}$, obtaining graphs G_1'' and H'' and corresponding mediating morphisms. We have that the morphism $D_{m'} \to G_1''$ is injectve, since it splits the injective morphism $D_{m'} \rightarrowtail H_1$. The same argument holds for the morphism $D_{m'} \to H''$. Hence the situation is as follows:



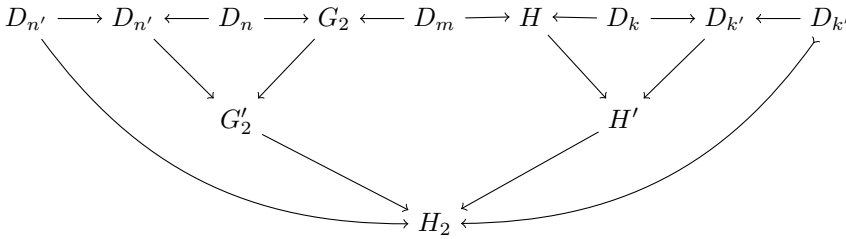
Note also that the morphism $D_{k'} \to D_{k'} \to H' \to H''$ must be injective, since it splits the injective morphism $D_{k'} \rightarrowtail H_1$.

We will now show that the square $D_{m'}, G_1'', H'', H_1$ commutes and that it is a pushout. We are in the situation depicted on the left below where the outer square is pushout. We have $D_m \twoheadrightarrow D_{m'} \rightarrowtail G_1'' \to H_1 = D_m \to G_1' \twoheadrightarrow G_1'' \to H_1 = D_m \to H' \twoheadrightarrow H'' \to H_1 = D_m \twoheadrightarrow D_{m'} \rightarrowtail H'' \to H_1$. Since $D_m \twoheadrightarrow D_{m'}$ is surjective we can conclude that $D_{m'} \rightarrowtail G_1'' \to H_1 = D_{m'} \rightarrowtail H'' \to H_1$. Now assume a commuting square $D_{m'}, G_1'', H'', X$ as shown below on the right. Since the outer square is a pushout we obtain a mediating morphism $H_1 \to X$. The triangles G_1'', H_1, X and H'', H_1, X commute since they commute if we precompose them with the morphisms $G_1' \twoheadrightarrow G_1''$ and $H' \twoheadrightarrow H''$ respectively. Furthermore the morphism $H_1 \to X$ must be unique since any other mediating morphism would also be a mediating morphism for the outer pushout.



Hence we get $a_m \colon D_m \twoheadrightarrow D_{m'} \leftarrow_{id} D_{m'}$ and $a_{m'} \colon D_{m'} \rightarrow_{id} D_{m'} \twoheadleftarrow D_m$ and from the diagram above it follows that $a_n \mathbin{;} c_1 \mathbin{;} d \mathbin{;} a_k = a_n \mathbin{;} c_1 \mathbin{;} a_m \mathbin{;} a_{m'} \mathbin{;} d \mathbin{;} a_k$. Due to the considerations above we know that the morphisms $D_{m'} \to G''_1$ and $D_{k'} \to H''$ are injective, which are the right-legs of the cospans $a_n \mathbin{;} c_1 \mathbin{;} a_m$ and $a_{m'} \mathbin{;} d \mathbin{;} a_k$ respectively. Hence, both cospans are also output-linear. By $c_1 \equiv'_R c_2$, we get $a_n \mathbin{;} c_1 \mathbin{;} a_m \mathbin{;} a_{m'} \mathbin{;} d \mathbin{;} a_k = a_n \mathbin{;} c_2 \mathbin{;} a_m \mathbin{;} a_{m'} \mathbin{;} d \mathbin{;} a_k$.

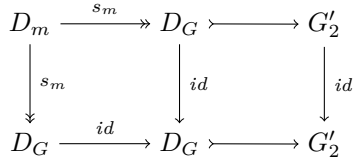
It is now left to show that $a_n \mathbin{;} c_2 \mathbin{;} a_m \mathbin{;} a_{m'} \mathbin{;} d \mathbin{;} a_k = a_n \mathbin{;} c_2 \mathbin{;} d \mathbin{;} a_k$ and that this is an output-linear cospan. We first consider the diagram for $(a_n \mathbin{;} c_2) \mathbin{;} (d \mathbin{;} a_k)$.



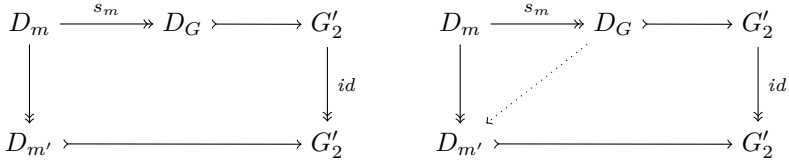
Next, we consider the factorization of $D_m \to G'_2$ into a surjective morphism followed by an injective morphism, i. e.,

$$D_m \to G'_2 = D_m \xrightarrow{s_m} D_G \rightarrowtail G'_2.$$

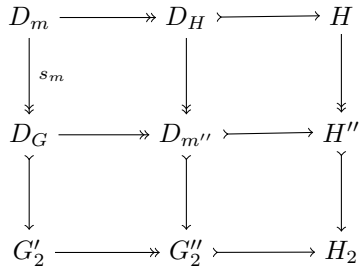
We construct the pushout of $s_m \colon D_m \twoheadrightarrow D_G$ and $D_m \twoheadrightarrow G'_2$ as shown below in two steps. First we take the pushout of s_m with itself, which gives us D_G and the identity morphisms since s_m is surjective. Then we take the pushout of $D_G \rightarrowtail G'_2$ and the identity on D_G, resulting in G'_2.



Now consider the cospan $\hat{a}\colon D_m \xrightarrow{s_m} D_G \xleftarrow{id} D_G$. Because of the considerations above we can conclude that $\langle a_n, \hat{a}\rangle \in M(c_2) = M(c_1)$. Furthermore, we can show that $\hat{a}$ is uniquely characterized to be the most "general" partner for a_n. Consider for instance another cospan $a_m\colon D_m \to D_{m'} \leftarrow D_{m'}$ with $\langle a_n, a_m\rangle \in M(c_2)$. Then we obtain the diagram below on the left.

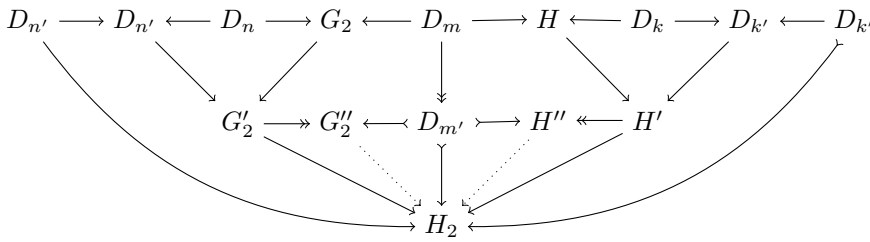


Because of the diagonalization property of factorizations we obtain a unique morphism $D_G \to D_{m'}$ which makes the diagram commute (see diagram above on the right). This property uniquely characterizes s_m (up to isomorphism). The same characterization holds for the factorization of $D_m \to G'_1$ (since the merger pairs are identical) and we can conclude that $D_m \to G'_1$ factorizes into $D_m \xrightarrow{s_m} D_{m'} \rightarrowtail G'_1$. Now consider also the factorization of $D_m \to H$ into $D_m \twoheadrightarrow D_H \rightarrowtail H$ and split the pushout of $D_m \to G'_2$ and $D_m \to H$ as shown below:



Hence taking the pushout of $D_m \twoheadrightarrow D_G$ and $D_m \twoheadrightarrow D_H$ gives us the unique factorization of $D_m \to H_2$. The same is true in the case of H_1 where the factorization is $D_m \to H_1 = D_m \twoheadrightarrow D_{m'} \rightarrowtail H_1$. Hence $D_{m'}$ and $D_{m''}$ are isomorphic.

Furthermore in the diagram for $(a_n\,;c_2)\,;(d\,;a_k)$ above we can split the middle pushout in a way identical to the splitting of $(a_n\,;c_1)\,;(d\,;a_k)$ and obtain a pushout consisting of $D_{m'}, G''_2, H'', H_2$ (see below).



This implies that $a_n \,;\, c_2 \,;\, d \,;\, a_k = a_n \,;\, c_2 \,;\, a_m \,;\, a_{m'} \,;\, d \,;\, a_k$. Furthermore $D_{k'} \rightarrowtail H_2$ must be injective: $D_{k'} \rightarrowtail H''$ is injective (since it is the right leg of the output-linear cospan $a_{m'} \,;\, d \,;\, a_k$) and if we compose with the injective morphism $H'' \rightarrowtail H_2$ we obtain $D_{k'} \rightarrowtail H_2$. Hence we have $M(c_1 \,;\, d) = M(c_2 \,;\, d)$ and this concludes the proof.

□

Proposition 6.25. For every sequence of composable cospans in the category **OLCG** exists an equivalent sequence of composable cospans which is in atomic cospan normal form.

Proof. Let $\vec{c}$ be a sequence of composable cospans such that the cospan obtainded from $\vec{c}$ has the form

$$\vec{c}\colon D_k \xrightarrow{c_{\mathrm{L}}} G \xleftarrow{c_{\mathrm{R}}} D_\ell,$$

where $G = \langle V, E, att, lab\rangle$. Without loss of generality we assume that $V = \{1, \ldots, n\}$ and $E = \{e_1, \ldots, e_m\}$. Furthermore, we define $A_i = lab(e_i)$, $E_i = \{e_j \in E \mid j < i\}$ and $s(i) = \sum_{e\in E_i} |att(e)|$ for every $i = 1, \ldots, m$.

We construct the equivalent sequence in atomic cospan normal form as follows:

$$\begin{aligned}
&\mathit{vertex}^{k}_{k+1} \,;\, \mathit{vertex}^{k+1}_{k+2} \,;\, \cdots \,;\, \mathit{vertex}^{k+s-1}_{k+s} \,;\\
&\mathit{connect}^{k+s}_{A_1,\theta_1} \,;\, \cdots \,;\, \mathit{connect}^{k+s}_{A_m,\theta_m} \,;\\
&\mathit{fuse}^{k+s}_{\delta} \,;\\
&\mathit{perm}^{n}_{\pi} \,;\\
&\mathit{res}^{n}_{n} \,;\, \cdots \,;\, \mathit{res}^{\ell+1}_{\ell+1}
\end{aligned}$$

where

- $s = s(m) + z = \sum_{e\in E} |att(e)| + z$ such that z is the number of isolated nodes of G,
- the θ_i are (injective) functions

  $$\theta_i\colon \{1, \ldots, |att(e_i)|\} \to \{k + s(i-1) + 1, \ldots, k + s(i-1) + |att(e_i)|\}$$

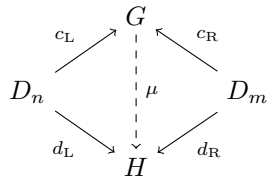
  where $\theta_i(j)$ returns the j-th node attached to edge $e_i \in E$ and
- δ is an equivalence which indicates which nodes must be fused, i. e. δ is the smallest equivalence containing the pair $\langle v, w\rangle$ if and only one of the following conditions hold:
  - the nodes v and w are the ends of a loop, i. e. let $v = \theta_i(j)$ be the j-th and $w = \theta_i(j')$ be the j'-th node of some edge $e_i \in E$ such that $att(e_i)[j] = att(e_i)[j']$,
  - the nodes v and w connect two edges, i. e. let $v = \theta_i(j)$ be the j-th node of some edge $e_i \in E$ and $w = \theta_{i'}(j')$ be the j'-th node of some edge $e_{i'} \in E$ such that $att(e_i)[j] = att(e_{i'})[j']$,

  - the node i-th node of the inner interface is mapped to the j-th node of the middle graph, i. e. $c_{\mathrm{L}}(i) = j$, $v = i$ and $w = k + j$.
- π is a bijection on $\{1, \ldots, n\}$ with $\pi(j) = c_{\mathrm{R}}(j)$ for $1 \leq j \leq \ell$ and is arbitrary otherwise.

□

Proposition 6.26 (Algebraic Properties of Atomic Cospans)**.** The atomic cospans defined in Definition 3.10 satisfy the equation schemes 6.1–6.17.

Proof. The proofs in the following are always of the form that we have two cospans $c\colon D_n - c_{\mathrm{L}} \to G \leftarrow c_{\mathrm{R}} - D_m$, $d\colon D_n - d_{\mathrm{L}} \to H \leftarrow d_{\mathrm{R}} - D_m$ and we have to show that the following diagram commutes



- Proof of equation 6.1: We have that

$$vertex_i^n = D_n \xrightarrow{\varphi} D_{n+1} \xleftarrow{id_{D_{n+1}}} D_{n+1}$$

and

$$vertex_n^n \mathbin{;} perm_\pi^{n+1} = D_n \xrightarrow{i} D_{n+1} \xleftarrow{\pi} D_{n+1}$$

where

$$\varphi\colon D_n \to D_{n+1}, \qquad x \mapsto \begin{cases} x, & \text{if } x < k \\ x+1, & \text{otherwise} \end{cases}$$

$$i\colon D_n \to D_{n+1}, \qquad x \mapsto x.$$

We show that

$$D_n \xrightarrow{\varphi} D_{n+1} \xleftarrow{id_{D_{n+1}}} D_{n+1} \simeq D_n \xrightarrow{i} D_{n+1} \xleftarrow{\pi} D_{n+1}$$

as follows: Let $\mu := \pi^{-1}$, i. e.

$$\mu\colon D_{n+1} \to D_{n+1}, \qquad x \mapsto \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } k \leq x \leq n \\ k, & \text{otherwise} \end{cases}$$

Obviously, we have that $\pi \mathbin{;} \mu = id_{D_{n+1}}$. Additionally, let $x \in D_n$, we then have

$$(i \mathbin{;} \mu)(x) = \pi^{-1}(i(x)) = \begin{cases} i(x), & \text{if } i(x) < k \\ i(x)+1, & \text{if } k \leq i(x) \leq n \\ k, & \text{otherwise} \end{cases}$$

$$= \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } k \leq x \leq n \\ k, & \text{otherwise} \end{cases}$$

Since $dom(i \,;\, \mu) = \mathbb{N}_n$, the last case never occurs. Therefore, we have that:

$$= \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } k \leq x \leq n \end{cases}$$
$$= \varphi(x)$$

- Proof of equation 6.2: We have that

$$vertex^n_k \,;\, vertex^n_\ell = D_n \xrightarrow{\varphi_1;\varphi'_1} D_{n+2} \xleftarrow{id_{D_{n+2}}} D_{n+2}$$

and

$$vertex^n_{f(\ell)} \,;\, vertex^{n+1}_{g(k)} = D_n \xrightarrow{\varphi_2;\varphi'_2} D_{n+2} \xleftarrow{id_{D_{n+2}}} D_{n+2}$$

where

$$\varphi_1 \colon D_n \to D_{n+1}, \qquad x \mapsto \begin{cases} x, & \text{if } x < k \\ x+1, & \text{otherwise} \end{cases}$$
$$\varphi'_1 \colon D_{n+1} \to D_{n+2}, \qquad x \mapsto \begin{cases} x, & \text{if } x < \ell \\ x+1, & \text{otherwise} \end{cases}$$
$$\varphi_2 \colon D_n \to D_{n+1}, \qquad x \mapsto \begin{cases} x, & \text{if } x < f(\ell) \\ x+1, & \text{otherwise} \end{cases}$$
$$\varphi'_2 \colon D_{n+1} \to D_{n+2}, \qquad x \mapsto \begin{cases} x, & \text{if } x < g(k) \\ x+1, & \text{otherwise} \end{cases}$$

We show that

$$D_n \xrightarrow{\varphi_1;\varphi'_1} D_{n+2} \xleftarrow{id_{D_{n+2}}} D_{n+2} \simeq D_n \xrightarrow{\varphi_2;\varphi'_2} D_{n+2} \xleftarrow{id_{D_{n+2}}} D_{n+2}$$

as follows: Let $\mu := id_{D_{n+2}}$. Obviously, we have that $id_{D_{n+2}} \,;\, \mu = id_{D_{n+2}}$. Additionally, let $x \in D_n$. We the have

$$(\varphi_1 \,;\, \varphi'_1 \,;\, \mu)(x) = \varphi'_1(\varphi_1(x)) = \begin{cases} \varphi_1(x), & \text{if } \varphi_1(x) < \ell \\ \varphi_1(x)+1, & \text{otherwise} \end{cases}$$
$$= \begin{cases} x, & \text{if } x < k, x < \ell \\ x+1, & \text{if } x \geq k, x+1 < \ell \\ x+1, & \text{if } x < k, x \geq \ell \\ x+2, & \text{otherwise} \end{cases}$$

and

$$(\varphi_2 \,;\, \varphi_2' \,;\, \mu)(x) = \varphi_2'(\varphi_2(x)) = \begin{cases} \varphi_2(x), & \text{if } \varphi_2(x) < g(k) \\ \varphi_2(x)+1, & \text{otherwise} \end{cases}$$
$$= \begin{cases} x, & \text{if } x < f(\ell), x < g(k) \\ x+1, & \text{if } x \geq f(\ell), x+1 < g(k) \\ x+1, & \text{if } x < f(\ell), x \geq g(k) \\ x+2, & \text{otherwise} \end{cases}$$

Now, we distinguish two different cases:

- Case 1: Let $k < \ell$. Hence, $f(\ell) = \ell - 1$ and $g(k) = k$. Furthermore, since $x < k \wedge x \geq \ell$ and $x < g(k) \wedge x \geq f(\ell)$ are contradictions, this yields

$$(\varphi_1 \,;\, \varphi_1' \,;\, \mu)(x) = \begin{cases} x, & \text{if } x < k, x < \ell \\ x+1, & \text{if } x \geq k, x+1 < \ell \\ x+2, & \text{otherwise} \end{cases}$$
$$= \begin{cases} x, & \text{if } x < k \\ x+1, & \text{if } x \geq k, x+1 < \ell \\ x+2, & \text{otherwise} \end{cases}$$
$$= \begin{cases} x, & \text{if } x < f(\ell), x < g(k) \\ x+1, & \text{if } x < f(\ell), x \geq g(k) \\ x+2, & \text{otherwise} \end{cases}$$
$$= \varphi_2'(\varphi_2(x)) = (\varphi_2 \,;\, \varphi_2' \,;\, \mu)(x)$$

- Case 2: Let $k \geq \ell$. Hence, $f(\ell) = \ell$ and $g(k) = k + 1$. Furthermore, since $x \geq k \wedge x + 1 < \ell$ and $x \geq g(k) \wedge x < f(\ell)$ are contradictions, this yields

$$(\varphi_1 \,;\, \varphi_1' \,;\, \mu)(x) = \begin{cases} x, & \text{if } x < k, x < \ell \\ x+1, & \text{if } x < k, x \geq \ell \\ x+2, & \text{otherwise} \end{cases}$$
$$= \begin{cases} x, & \text{if } x < \ell \\ x+1, & \text{if } x \geq \ell, x+1 < k+1 \\ x+2, & \text{otherwise} \end{cases}$$
$$= \begin{cases} x, & \text{if } x < f(\ell), x < g(k) \\ x+1, & \text{if } x \geq f(\ell), x+1 < g(k) \\ x+2, & \text{otherwise} \end{cases}$$
$$= \varphi_2'(\varphi_2(x)) = (\varphi_2 \,;\, \varphi_2' \,;\, \mu)(x)$$

- Proof of equation 6.6: We have that

$$\mathit{perm}_\pi^n \,;\, \mathit{vertex}_i^n = D_n \xrightarrow{\varphi} D_{n+1} \xleftarrow{id_{D_{n+1}}} D_{n+1}$$

and

$$vertex_i^n \ ; \ perm_{\pi'}^{n+1} = D_n \xrightarrow{\psi} D_{n+1} \xleftarrow{\pi'} D_{n+1}$$

where

$$\varphi\colon D_n \to D_{n+1}, \qquad x \mapsto \begin{cases} \pi^{-1}(x), & \text{if } x < k, \pi^{-1}(x) < k \\ \pi^{-1}(x)+1, & \text{if } x < k, \pi^{-1}(x) \geq k \\ \pi^{-1}(x-1), & \text{if } x \geq k, \pi^{-1}(x) < k \\ \pi^{-1}(x-1)+1, & \text{otherwise} \end{cases}$$

$$\psi\colon D_n \to D_{n+1}, \qquad x \mapsto \begin{cases} x, & \text{if } x < k \\ x+1, & \text{otherwise} \end{cases}$$

We show that

$$D_n \xrightarrow{\varphi} D_{n+1} \xleftarrow{id_{D_{n+1}}} D_{n+1} \simeq D_n \xrightarrow{\psi} D_{n+1} \xleftarrow{\pi'} D_{n+1}$$

as follows: Let $\mu := \pi'^{-1}$, i. e.

$$\mu\colon D_{n+1} \to D_{n+1}, \qquad x \mapsto \begin{cases} \pi^{-1}(x), & \text{if } x < k, \pi'^{-1}(x) < k \\ \pi^{-1}(x)+1, & \text{if } x < k, \pi'^{-1}(x) \geq k \\ k, & \text{if } x = k \\ \pi^{-1}(x-1), & \text{if } x > k, \pi'^{-1}(x) < k \\ \pi^{-1}(x-1)+1, & \text{otherwise} \end{cases}$$

Obviously, we have that $\pi' \ ; \mu = id_{D_{n+1}}$. Additionally, let $x \in \mathbb{N}_n$, we then have

$$\begin{aligned} \psi \ ; \mu(x) = \pi'^{-1}(\psi(x)) &= \begin{cases} \pi'(x), & \text{if } x < k \\ \pi'(x+1), & \text{otherwise} \end{cases} \\ &= \begin{cases} \pi^{-1}(x), & \text{if } x < k, \pi'^{-1}(x) < k \\ \pi^{-1}(x)+1, & \text{if } x < k, \pi'^{-1}(x) \geq k \\ \pi^{-1}(x-1), & \text{if } x \geq k, \pi'^{-1}(x) < k \\ \pi^{-1}(x-1)+1, & \text{otherwise} \end{cases} \\ &= \varphi(x) \end{aligned}$$

- Proof of equation 6.10: We have that

$$perm_\pi^n \ ; \ connect_{A,\theta}^n = D_n \xrightarrow{\pi'^{-1}} H \xleftarrow{i} D_n$$

and

$$connect_{A,\theta}^n \ ; \ perm_\pi^n = D_n \xrightarrow{i} H' \xleftarrow{\pi;i} D_n$$

where

$$H = \langle \mathbb{N}_n, \{e\}, att_H, lab_H \rangle \quad \text{and} \quad H' = \langle \mathbb{N}_n, \{e'\}, att_{H'}, lab_{H'} \rangle$$

such that $att_H(e) = \theta(0)\dots\theta(ar(A)-1)$, $att_{H'}(e') = \theta'(0)\dots\theta'(ar(A)-1)$ and $lab(e) = A = lab(e')$,

$$\begin{aligned} \pi'^{-1}\colon D_n &\to H, & x &\mapsto \pi^{-1}(x) \\ i\colon D_n &\to H & x &\mapsto x \\ i'\colon D_n &\to H' & x &\mapsto x. \end{aligned}$$

We show that

$$D_n \xrightarrow{\pi'^{-1}} H \xleftarrow{i} D_n \simeq D_n \xrightarrow{i} H' \xleftarrow{\pi;i} D_n$$

as follows: We define μ as

$$\mu\colon H' \to H, \qquad x \mapsto \begin{cases} \pi^{-1}(x), & \text{if } x \in V \\ e, & \text{otherwise} \end{cases}.$$

Let $x \in D_n$, we then have

$$(i\ ;\ \mu)(x) = \pi''^{-1}(i(x)) = \pi''^{-1}(x) = \pi^{-1}(x) = \pi'^{-1}(x)$$

and

$$(\pi\ ;\ i'\ ;\ \mu)(x) = \pi''^{-1}(i(\pi(x))) = \pi''^{-1}(pi(x)) = \pi^{-1}(\pi(x)) = i(x).$$

- Proof of equation 6.15: We have that

$$res_i^n\ ;\ perm_\pi^n = D_n \xrightarrow{id_{D_n}} D_n \xleftarrow{\pi;\varphi} D_{n-1}$$

and

$$perm_\pi^n\ ;\ res_i^n = D_n \xrightarrow{\pi'^{-1}} D_n \xleftarrow{\varphi} D_{n-1}$$

where

$$\varphi\colon D_{n-1} \to D_n, \qquad x \mapsto \begin{cases} x, & \text{if } x < k \\ x+1, & \text{otherwise} \end{cases}$$

We show that

$$D_n \xrightarrow{id_{D_n}} D_n \xleftarrow{\pi;\varphi} D_{n-1} \simeq D_n \xrightarrow{\pi'^{-1}} D_n \xleftarrow{\varphi} D_{n-1}$$

as follows: Let $\mu := \pi'$. Obviously we have that $\pi'^{-1}\ ;\ \mu = id_{D_n}$. Additionally, let $x \in \mathbb{N}_n$, we then have

$$\begin{aligned} (\varphi'\ ;\ \mu)(x) &= \pi'(\varphi(x)) \\ &= \begin{cases} \pi(\varphi(x)), & \text{if } \varphi(x) < k, \pi(\varphi(x)) < k \\ \pi(\varphi(x)) + 1, & \text{if } \varphi(x) < k, \pi(\varphi(x)) \geq k \\ \varphi(x), & \text{if } \varphi(x) = k \\ \pi(\varphi(x) - 1), & \text{if } \varphi(x) > k, \pi(\varphi(x)) < k \\ \pi(\varphi(x) - 1) + 1, & \text{otherwise} \end{cases} \end{aligned}$$

Since there exists no $\ell \in D_n$ such that $\varphi(\ell) = k$, the corresponding case never can occur. Therefore, we obtain:

$$
\begin{aligned}
&= \begin{cases} \pi(x), & \text{if } x < k, \pi(x) < k \\ \pi(x)+1, & \text{if } x < k, \pi(x) \geq k \\ \pi(x+1-1), & \text{if } x > k, \pi(x) < k \\ \pi(x+1-1)+1, & \text{otherwise} \end{cases} \\
&= \begin{cases} \pi(x), & \text{if } \pi(x) < k \\ \pi(x)+1, & \text{otherwise} \end{cases} \\
&= \varphi(\pi(x)) = (\pi \,;\, \varphi)(x)
\end{aligned}
$$

- Proof of equation 6.16: We have that

$$res_i^n = D_n \xrightarrow{id_{D_n}} D_n \xleftarrow{\varphi} D_{n-1}$$

and

$$perm_\pi^n \,;\, res_n^n = D_n \xrightarrow{\pi^{-1}} D_n \xleftarrow{i'} D_{n-1}$$

where

$$
\begin{aligned}
&\varphi \colon D_{n-1} \to D_n, && x \mapsto \begin{cases} x, & \text{if } x < k \\ x+1, & \text{otherwise} \end{cases} \\
&i' \colon D_{n-1} \to D_n, && x \mapsto x
\end{aligned}
$$

We show that

$$D_n \xrightarrow{id_{D_n}} D_n \xleftarrow{\varphi} D_{n-1} \simeq D_n \xrightarrow{\pi^{-1}} D_n \xleftarrow{i'} D_{n-1}$$

as follows: Let $\mu := \pi$. Obviously we have that $\pi^{-1} \,;\, \mu = id_{D_n}$. Additionally, let $x \in D_n$, we then have

$$
\begin{aligned}
(i' \,;\, \mu)(x) &= \pi(i'(x)) \\
&= \begin{cases} i'(x), & \text{if } i'(x) < k \\ i'(x)+1, & \text{if } k \leq i'(x) < n \\ k, & \text{otherwise} \end{cases}
\end{aligned}
$$

Since $n \notin dom(i')$, which implies that $n \notin dom(i' \,;\, \mu)$, we have that:

$$
\begin{aligned}
&= \begin{cases} x, & \text{if } x < k \\ x+1, & \text{otherwise} \end{cases} \\
&= \varphi(x)
\end{aligned}
$$

- Proofs of equations 6.3, 6.4, 6.7, 6.8, 6.9, 6.11, 6.12, 6.14: Trivial, by definition of the atomic cospans.

- Proofs of equation 6.5, 6.13, 6.17: These proofs are similar to the proof of equation 6.2.

□

To make the completeness proof easier, we introduce the following derived equation shemes which can easily be proved by Proposition 6.26.

Remark A.2. *Here are some derived schemes which will be useful in the forthcoming proof:*

$$vertex^n_{n+1} \,;\, vertex^{n+1}_{n+2} \equiv vertex^n_{n+1} \,;\, vertex^{n+1}_{n+2} \,;\, perm^{n+2}_{\pi}, \tag{A.1}$$

where π is defined as

$$\pi\colon \mathbb{N}_{n+2} \to \mathbb{N}_{n+2}, \quad \pi(x) = \begin{cases} x, & \text{if } x < n+1 \\ n+2, & \text{if } x = n+1 \\ n+1, & \text{if } x = n+2 \end{cases}$$

$$vertex^n_{n+1} \,;\, \ldots \,;\, vertex^{n+k}_{n+k+1} \equiv vertex^n_{n+1} \,;\, \ldots \,;\, vertex^{n+k}_{n+k+1} \,;\, perm^{n+k+1}_{\pi}, \tag{A.2}$$

where π is defined as

$$\pi\colon \mathbb{N}_{n+k+1} \to \mathbb{N}_{n+k+1}, \quad \pi(x) = \begin{cases} x, & \text{if } x < n+1 \\ x+1, & \text{if } k+1 \le x < n+k+1 \\ n+1, & \text{if } x = n+k+1 \end{cases}$$

$$perm^n_{\pi} \,;\, vertex^n_{n+1} \,;\, \ldots \,;\, vertex^{n+k}_{n+k+1} \equiv vertex^n_{n+1} \,;\, \ldots \,;\, vertex^{n+k}_{n+k+1} \,;\, perm^{n+k+1}_{\pi'}, \tag{A.3}$$

where π' is defined as

$$\pi'\colon \mathbb{N}_{n+k+1} \to \mathbb{N}_{n+k+1}, \quad \pi'(x) = \begin{cases} \pi(x), & \text{if } x < n+1 \\ x, & \text{if } x \ge k+1 \end{cases}$$

$$connect^n_{A,\theta} \equiv perm^n_{\pi} \,;\, connect^n_{A,\theta'} \,;\, perm^n_{\pi^{-1}}, \tag{A.4}$$

where θ' is defined as

$$\theta'\colon \mathbb{N}_n \to \mathbb{N}_n, \quad \theta'(x) = \pi^{-1}(\theta(x))$$

$$perm^n_{\pi} \,;\, fuse^n_{\delta} \equiv fuse^n_{\pi^{-1}(\delta)} \tag{A.5}$$

$$res^n_n \,;\, res^{n-1}_{n-1} \equiv perm^n_\pi \,;\, res^n_n \,;\, res^{n-1}_{n-1}, \tag{A.6}$$

where π *is defined as*

$$\pi\colon \mathbb{N}_n \to \mathbb{N}_n, \quad \pi(x) = \begin{cases} x, & \text{if } x < n-1 \\ n, & \text{if } x = n-1 \\ n-1, & \text{if } x = n \end{cases}$$

$$res^{n+k}_{n+k} \,;\, \ldots \,;\, res^n_n \equiv perm^n_\pi \,;\, res^{n+k}_{n+k} \,;\, \ldots \,;\, res^n_n, \tag{A.7}$$

where π *is defined as*

$$\pi\colon \mathbb{N}_{n+k} \to \mathbb{N}_{n+k}, \quad \pi(x) = \begin{cases} x, & \text{if } x < n \\ n+k, & \text{if } x = n \\ x-1, & \text{if } x > n \end{cases}$$

$$res^{n+k}_{n+k} \,;\, \ldots \,;\, res^n_n \,;\, perm^n_\pi \equiv perm^{n+k}_{\pi'} \,;\, res^{n+k}_{n+k} \,;\, \ldots \,;\, res^n_n, \tag{A.8}$$

where π' *is defined as*

$$\pi'\colon \mathbb{N}_{n+k} \to \mathbb{N}_{n+k}, \quad \pi'(x) = \begin{cases} \pi(x), & \text{if } x < n \\ x, & \text{if } x \geq n \end{cases}$$

Theorem 6.27 (Soundness)**.** If two sequences of composable cospans in the category **OLCG** (with the same inner and outer interface) can be transformed into each other via the equations schemes 6.1–6.17, they are equivalent.

Proof. Let $\vec{c}$ and $\vec{d}$ be two sequences of composable cospans. By proposition 6.25 we can transform both $\vec{c}$ and $\vec{d}$ in two sequences $\vec{c'}$ and $\vec{d'}$ which are both in atomic cospan normal form. By assumption we can transform the sequence $\vec{c'}$ into the sequence $\vec{d'}$ and vice versa. Hence, $\vec{c}$ and $\vec{d}$ are equivalent. □

Theorem 6.28 (Completeness)**.** If two sequences of composable cospans in the category **OLCG** (with the same inner and outer interface) are equivalent, they can be transformed into each other via the equation schemes 6.1–6.17.

Proof. Let $\vec{c}$ and $\vec{d}$ be two equivalent sequences of composable cospans from which we can obtain two cospans of the following form

$$\vec{c}\colon D_k \xrightarrow{c_{\mathrm{L}}} G \xleftarrow{c_{\mathrm{R}}} D_\ell \quad \text{and} \quad \vec{d}\colon D_k \xrightarrow{d_{\mathrm{L}}} G' \xleftarrow{d_{\mathrm{R}}} D_\ell.$$

Since $\vec{c}$ and $\vec{d}$ are isomorphic it also holds that $G \simeq G'$. Without loss of generality, we assume that

$$G = \langle V, E, att, lab \rangle \quad \text{and} \quad G' = \langle V, E', att', lab' \rangle,$$

where $V = \{1, \ldots, n\}$, $E = \{e_1, \ldots, e_m\}$ and $E' = \{e'_1, \ldots, e'_m\}$. Furthermore, we define $A_i = lab(e_i)$, $E_i = \{e_j \in E \mid j < i\}$ and $s(i) = \sum_{e \in E_i} |att(e)|$ for every $i = 1, \ldots, m$. Analogously we define A'_i, E'_i and $s'(i)$ for $i = 1, \ldots, m$.

By Proposition 6.25, we can transform the sequence $\vec{c}$ into an equivalent sequence $\vec{c'}$ which is in atomic normal form, i. e.

$$\begin{aligned} \vec{c} \equiv \vec{c'} = {} & \mathit{vertex}^{k}_{k+1} \,;\, \mathit{vertex}^{k+1}_{k+2} \,;\, \cdots \,;\, \mathit{vertex}^{k+s-1}_{k+s} \,; \\ & \mathit{connect}^{k+s}_{A_1,\theta_1} \,;\, \cdots \,;\, \mathit{connect}^{k+s}_{A_m,\theta_m} \,; \\ & \mathit{fuse}^{k+s}_{\delta} \,; \\ & \mathit{perm}^{n}_{\pi} \,; \\ & \mathit{res}^{n}_{n} \,;\, \cdots \,;\, \mathit{res}^{\ell+1}_{\ell+1} \end{aligned}$$

where

- $s = s(m) + z = \sum_{e \in E} |att(e)| + z$ where z is the number of isolated nodes of G,
- the θ_i are (injective) functions

  $$\theta_i \colon \{1, \ldots, |att(e_i)|\} \to \{k + s(i-1) + 1, \ldots, k + s(i-1) + |att(e_i)|\}$$

  where $\theta_i(j)$ returns the j-th node attached to edge $e_i \in E$ and
- δ is an equivalence which indicates which nodes must be fused, i. e. δ is the smallest equivalence containing the pair $\langle v, w \rangle$ if and only one of the following conditions hold:
  - the nodes v and w are the ends of a loop, i. e. let $v = \theta_i(j)$ be the j-th and $w = \theta_i(j')$ be the j'-th node of some edge $e_i \in E$ such that $att(e_i)[j] = att(e_i)[j']$,
  - the nodes v and w connect two edges, i. e. let $v = \theta_i(j)$ be the j-th node of some edge $e_i \in E$ and $w = \theta_{i'}(j')$ be the j'-th node of some edge $e_{i'} \in E$ such that $att(e_i)[j] = att(e_{i'})[j']$,
  - the node i-th node of the inner interface is mapped to the j-th node of the middle graph, i. e. $c_{\mathrm{L}}(i) = j$, $v = i$ and $w = k + j$.
- π is a bijection on $\{1, \ldots, n\}$ with $\pi(j) = c_{\mathrm{R}}(j)$ for $1 \leq j \leq \ell$ and is arbitrary otherwise,

Similarly, we can transform the sequence $\vec{d}$ by Proposition 6.25 into an equivalent sequence $\vec{d'}$ which is in atomic normal form, i. e.

$$\begin{aligned} \vec{d} \equiv \vec{d'} = {} & \mathit{vertex}^{k}_{k+1} \,;\, \mathit{vertex}^{k+1}_{k+2} \,;\, \cdots \,;\, \mathit{vertex}^{k+s'-1}_{k+s'} \,; \\ & \mathit{connect}^{k+s'}_{A'_1,\theta'_1} \,;\, \cdots \,;\, \mathit{connect}^{k+s'}_{A'_m,\theta'_m} \,; \\ & \mathit{fuse}^{k+s'}_{\delta'} \,; \end{aligned}$$

$$perm^n_{\pi'} \;;$$
$$res^n_n \;;\cdots;\; res^{\ell+1}_{\ell+1}$$

where

- $s' = s'(m) + z' = \sum_{e\in E'} |att'(e)| + z'$ where z' is the number of isolated nodes of G',
- the θ'_i are (injective) functions

  $$\theta'_i \colon \{1,\ldots,|att'(e_i)|\} \to \{k+s'(i-1)+1,\ldots,k+s'(i-1)+|att'(e_i)|\}$$

  where $\theta'_i(j)$ returns the j-th node attached to edge $e_i \in E$ and
- δ' is an equivalence which indicates which nodes must be fused, i. e. δ' is the smallest equivalence containing the pair $\langle v, w\rangle$ if and only one of the following conditions hold:
  - the nodes v and w are the ends of a loop, i. e. let $v = \theta'_i(j)$ be the j-th and $w = \theta'_i(j')$ be the j'-th node of some edge $e_i \in E'$ such that $att'(e_i)[j] = att(e_i)[j']$,
  - the nodes v and w connect two edges, i. e. let $v = \theta'_i(j)$ be the j-th node of some edge $e_i \in E'$ and $w = \theta'_{i'}(j')$ be the j'-th node of some edge $e_{i'} \in E'$ such that $att'(e_i)[j] = att'(e_{i'})[j']$,
  - the node i-th node of the inner interface is mapped to the j-th node of the middle graph, i. e. $d_{\mathrm{L}}(i) = j$, $v = i$ and $w = k + j$.
- π' is a bijection on $\{1,\ldots,n\}$ with $\pi'(j) = d_{\mathrm{R}}(j)$ for $1 \le j \le \ell$ and is arbitrary otherwise.

Since there exists an bijection between the graphs G and G' we can follow that the number of isolated nodes of G and G' is equal, hence $z = z'$. Furthermore, each edge of G can be mapped to exactly one edge of G' with the same label and the same arity (and vice versa). Hence, we have that $s = s'$ and that the number of *vertex*-operations must be equal.

Since both G and G' have the same set of edges (up to isomorphism), the number *connect*-operations must also be equal. Additionally, for each *connect*-operations which occurs in the sequence $\vec{c'}$ there must exist an equivalent *connect*-operation in the sequence $\vec{d'}$, i. e. for every edge $e \in G$, there must exist exactly one operation $connect^{k+s}_{A_i,\theta_i}$ in $\vec{c'}$ and exactly one operation $connect^{k+s}_{A'_j,\theta'_j}$ in $\vec{d'}$, such that $A_i = lab(e) = A'_j$ and $\theta_i(1)\ldots\theta_i(|e|) = att(e) = \theta'_j(1)\ldots\theta'_j(|e|)$. The same must hold for the *connect*-operation in the sequence $\vec{d'}$. Therefore, the sequence of *connect*-operations of both sequences $\vec{c'}$ and $\vec{d'}$ can only differ in the order of the *connect*-operations. Hence, we can repeatedly use equation 6.8 to re-order the *connect*-operations of the sequence $\vec{c'}$ such that $A_i = B_i$ holds for $1 \le i \le m$. Furthermore, we have to construct a permutation $\hat{\pi}$ such that $\theta'_j = \theta_i \;;\hat{\pi}$ where θ'_j and θ_i are given as described above. This permutation can be build by repeated usage of equations A.2 and A.3. Next, we can "shift" the permutation $\hat{\pi}$ (represented by the $perm_{\hat{\pi}}$-operation) over all *connect*-operations by repeatedly use equation 6.10.

Now, we have a sequence $\vec{c''}$ of composable cospans which is equivalent to $\vec{c'}$ of the following form:

$$\begin{aligned}\vec{c''} = {} & vertex^{k}_{k+1} \,;\, vertex^{k+1}_{k+2} \,;\, \cdots \,;\, vertex^{k+s-1}_{k+s} \,; \\ & connect^{k+s}_{A'_1,\theta'_1} \,;\, \cdots \,;\, connect^{k+s}_{A'_m,\theta'_m} \,; \\ & perm^{k+s}_{\hat{\pi}} \,; \\ & fuse^{k+s}_{\delta} \,; \\ & perm^{n}_{\pi} \,; \\ & res^{n}_{n} \,;\, \cdots \,;\, res^{\ell+1}_{\ell+1}\end{aligned}$$

Next, we "shift" $perm_{\hat{\pi}}$-operation over the $fuse_{\delta}$-operation by the equation A.5. This yields the equivalence $\delta' = \hat{\pi}(\delta) = \{\langle \hat{\pi}(v), \hat{\pi}(w)\rangle \mid \langle v, w\rangle \in \delta\}$, since $\vec{c''}$ is equivalent to $\vec{d'}$ and by equation A.5 we obtain a sequence $\vec{c'''}$ which is equivalent to $\vec{c'}$. If $\delta' \neq \hat{\pi}(\delta)$ we could follow that $\vec{c''}$ is not equivalent to $\vec{d'}$, which is a contradiction.

At last, we compose the permutation $\hat{\pi}$ with the permutation π by use of the equation 6.14. This yields the permutation π', i. e. $\pi' = \pi \,;\, \hat{\pi}$, since the $\vec{c'''}$ is equivalent to $\vec{d'}$ and by the same argument as above, we can follow that we obtain the sequence $\vec{d'}$ by equation A.5. If $\pi' \neq \pi \,;\, \hat{\pi}$, we could follow $\vec{c'''}$ is not equivalent to $\vec{d'}$, which is a contradiction.

Hence, we have shown that the sequence $\vec{c}$ can be transformed into the sequence $\vec{d}$ by the equation schemes 6.1–6.17. □

Proposition 6.47. Let $\varphi\colon i \to j$ and $\psi\colon j \to k$ be formulas. Then

$$[\![\forall\varphi\colon \psi]\!] = \{c\colon D_i \nrightarrow D_k \mid \forall c'\colon D_i \nrightarrow D_j, c''\colon D_j \nrightarrow D_k\colon (c = c' \,;\, c'' \wedge c' \models \varphi) \implies c'' \models \psi\}.$$

Proof. The proof can be done by using the basic definitions given above.

$$\begin{aligned}& [\![\forall\varphi\colon \psi]\!] \\ = {} & [\![\neg\,(\exists\varphi\colon \neg\psi)]\!] \\ = {} & \mathcal{M}(C,E) \setminus [\![\exists\varphi\colon \neg\psi]\!] \\ = {} & \mathcal{M}(C,E) \setminus \{c\colon D_i \nrightarrow D_k \mid \exists c'\colon D_i \nrightarrow D_j, c''\colon D_j \nrightarrow D_k\colon \\ & \qquad c = c' \,;\, c'' \wedge c' \models \varphi \wedge c'' \models \neg\psi\} \\ = {} & \{c\colon D_i \nrightarrow D_k \mid \neg\,(\exists c'\colon D_i \nrightarrow D_j, c''\colon D_j \nrightarrow D_k\colon \\ & \qquad c = c' \,;\, c'' \wedge c' \models \varphi \wedge c'' \models \neg\psi)\} \\ = {} & \{c\colon D_i \nrightarrow D_k \mid \forall c'\colon D_i \nrightarrow D_j, c''\colon D_j \nrightarrow D_k\colon \\ & \qquad \neg(c = c' \,;\, c'' \wedge c' \models \varphi) \vee c'' \models \psi\} \\ = {} & \{c\colon D_i \nrightarrow D_k \mid \forall c'\colon D_i \nrightarrow D_j, c''\colon D_j \nrightarrow D_k\colon \\ & \qquad (c = c' \,;\, c'' \wedge c' \models \varphi) \implies c'' \models \psi\}\end{aligned}$$

□

Proposition 6.48. Let $\varphi\colon i \to k$ and $\psi\colon i \to j$ be formulas. Then

$$\left[\!\left[\varphi{\downarrow_\psi^\forall}\right]\!\right] = \{c'\colon D_j \nrightarrow D_k \mid \forall c\colon D_i \nrightarrow D_j\colon c \models \psi \implies c\,;c' \models \varphi\}.$$

Proof. The proof can be done by using the basic definitions given above.

$$\begin{aligned}
&\left[\!\left[\varphi{\downarrow_\psi^\forall}\right]\!\right] \\
&= \left[\!\left[\neg\left(\neg\varphi{\downarrow_\psi^\exists}\right)\right]\!\right] \\
&= \mathcal{M}(D,E) \setminus \left[\!\left[\neg\varphi{\downarrow_\psi^\exists}\right]\!\right] \\
&= \mathcal{M}(D,E) \setminus \{c'\colon D_j \nrightarrow D_k \mid \exists c\colon D_i \nrightarrow D_j\colon c \models \psi \wedge c\,;c' \models \neg\varphi\} \\
&= \{c'\colon D_j \nrightarrow D_k \mid \neg(\exists c\colon D_i \nrightarrow D_j\colon c \models \psi \wedge c\,;c' \models \neg\varphi)\} \\
&= \{c'\colon D_j \nrightarrow D_k \mid \forall c\colon D_i \nrightarrow D_j\colon c \models \psi \implies c\,;c' \not\models \neg\varphi\} \\
&= \{c'\colon D_j \nrightarrow D_k \mid \forall c\colon D_i \nrightarrow D_j\colon c \models \psi \implies c\,;c' \models \varphi\}
\end{aligned}$$

□

Proposition 6.57. Let $n \in \mathbb{N}$, $\psi\colon i \to k$ be a formula and $i,j,k \leq n$. The functions

$$\begin{aligned}
\alpha_\psi^\exists\colon \Phi_{k,j} &\to \Phi_{i,j} & \gamma_\psi^\forall\colon \Phi_{i,j} &\to \Phi_{k,j} \\
\alpha_\psi^\exists(\varphi) &= \exists\psi\colon \varphi & \gamma_\psi^\forall(\varphi') &= \varphi'{\downarrow_\psi^\forall}
\end{aligned}$$

are a Galois connection between $\langle \Phi_{k,j}, \models\rangle$ and $\langle \Phi_{i,j}, \models\rangle$.

Proof.

- We show the monotonicity of $\alpha_\psi^\exists$ as follows. We assume that $\varphi_1 \models \varphi_2$ holds.

$$\begin{aligned}
&[\![\alpha_\psi^\exists(\varphi_1)]\!] \\
&= [\![\exists\psi\colon \varphi_1]\!] \\
&= \{d \mid \exists d', d''\colon d = d'\,;d'' \wedge d' \models \psi \wedge d'' \models \varphi_1\} \\
&\subseteq \{d \mid \exists d', d''\colon d = d'\,;d'' \wedge d' \models \psi \wedge d'' \models \varphi_2\} \\
&= [\![\exists\psi\colon \varphi_2]\!] \\
&= [\![\alpha_\psi^\exists(\varphi_2)]\!]
\end{aligned}$$

  Therefore, $\alpha_\psi^\exists(\phi_1) \models \alpha_\psi^\exists(\phi_2)$ holds.

- We show the monotonicity of $\gamma_\psi^\forall$ as follows. We assume that $\varphi_1 \models \varphi_2$ holds.

$$[\![\gamma_\psi^\forall(\varphi_1)]\!]$$

$$
\begin{aligned}
&= \left[\!\left[\varphi_1{\downarrow}^{\forall}_{\psi}\right]\!\right] \\
&= \left[\!\left[\neg\left(\neg\varphi_1{\downarrow}^{\exists}_{\psi}\right)\right]\!\right] \\
&= \mathcal{M}(k,j) \setminus \left[\!\left[\neg\varphi_1{\downarrow}^{\exists}_{\psi}\right]\!\right] \\
&= \mathcal{M}(k,j) \setminus \{c' \colon D_k \nrightarrow D_j \mid \exists c \colon D_i \nrightarrow D_k \colon c \models \psi \wedge c \,;\, c' \not\models \varphi_1\} \\
&= \{c' \colon D_k \nrightarrow D_j \mid \neg\,(\exists c \colon D_i \nrightarrow D_k \colon c \models \psi \wedge c \,;\, c' \not\models \varphi_1)\} \\
&= \{c' \colon D_k \nrightarrow D_j \mid \forall c \colon D_i \nrightarrow D_k \colon c \not\models \psi \vee c \,;\, c' \models \varphi_1\} \\
&\subseteq \{c' \colon D_k \nrightarrow D_j \mid \forall c \colon D_i \nrightarrow D_k \colon c \not\models \psi \vee c \,;\, c' \models \varphi_2\} \\
&= \{c' \colon D_k \nrightarrow D_j \mid \neg\,(\exists c \colon D_i \nrightarrow D_k \colon c \models \psi \wedge c \,;\, c' \not\models \varphi_2)\} \\
&= \mathcal{M}(k,j) \setminus \{c' \colon D_k \nrightarrow D_j \mid \exists c \colon D_i \nrightarrow D_k \colon c \models \psi \wedge c \,;\, c' \not\models \varphi_2\} \\
&= \mathcal{M}(k,j) \setminus \left[\!\left[\neg\varphi_2{\downarrow}^{\exists}_{\psi}\right]\!\right] \\
&= \left[\!\left[\neg\left(\neg\varphi_2{\downarrow}^{\exists}_{\psi}\right)\right]\!\right] \\
&= \left[\!\left[\varphi_2{\downarrow}^{\forall}_{\psi}\right]\!\right] \\
&= [\![\gamma^{\forall}_{\psi}(\varphi_2)]\!]
\end{aligned}
$$

Therefore, $\gamma^{\forall}_{\psi}(\phi_1) \models \gamma^{\forall}_{\psi}(\phi_2)$ holds.

- We show the extensivity of $\alpha^{\exists}_{\psi} \,;\, \gamma^{\forall}_{\psi}$ as follows. We assume that $\varphi \not\models \gamma^{\forall}_{\psi}(\alpha^{\exists}_{\psi}(\varphi))$. Then there exists a cospan $c \models \varphi$ such that $c \not\models \gamma^{\forall}_{\psi}(\alpha^{\exists}_{\psi}(\varphi))$.

$$
\begin{aligned}
& c \not\models \gamma^{\forall}_{\psi}(\alpha^{\exists}_{\psi}(\varphi)) \\
\iff\ & c \not\models (\exists\psi \colon \varphi){\downarrow}^{\forall}_{\psi} \\
\iff\ & c \models \neg\left((\exists\psi \colon \varphi){\downarrow}^{\forall}_{\psi}\right) \\
\iff\ & c \models \neg\,(\exists\psi \colon \varphi)\,{\downarrow}^{\exists}_{\psi} \\
\iff\ & c \in \{d' \colon D_k \nrightarrow D_j \mid \exists d \colon D_i \nrightarrow D_k \colon d \models \psi \wedge d \,;\, d' \models \neg\,(\exists\psi \colon \varphi)\} \\
\iff\ & c \in \{d' \colon D_k \nrightarrow D_j \mid \exists d \colon D_i \nrightarrow D_k \colon d \models \psi \wedge \\
& \qquad d \,;\, d' \in \{e \colon D_i \nrightarrow D_j \mid \forall e' \colon D_i \nrightarrow D_k, e'' \colon D_k \nrightarrow D_j \colon \\
& \qquad\quad e' \not\models \psi \vee e'' \not\models \varphi \vee e \neq e' \,;\, e''\}
\end{aligned}
$$

Now we can conclude, since $d \models \psi$ and $d \,;\, d' = e$, for some cospan $e \colon D_i \nrightarrow D_j$ satisfying $e' \not\models \psi \vee e'' \not\models \varphi \vee e \neq e' \,;\, e''$, that $d' \not\models \varphi$. But this yields $c \not\models \varphi$, which is a condtradiction to the assumption.

- We show the reduction of $\gamma^{\forall}_{\psi} \,;\, \alpha^{\exists}_{\psi}$ as follows.

$$
\begin{aligned}
& [\![\alpha^{\exists}_{\psi}(\gamma^{\forall}_{\psi}(\varphi))]\!] \\
=\ & \left[\!\left[\exists\psi \colon (\varphi{\downarrow}^{\forall}_{\psi})\right]\!\right]
\end{aligned}
$$

$$
\begin{aligned}
&= \left\{ c\colon D_i \not\hookrightarrow D_j \mid \exists c'\colon D_i \not\hookrightarrow D_k, c''\colon D_k \not\hookrightarrow D_j\colon c = c'\,;c'' \wedge c' \models \psi \wedge c'' \models \varphi{\downarrow_\psi^\forall} \right\} \\
&\subseteq \{ c\colon D_i \not\hookrightarrow D_j \mid \exists c'\colon D_i \not\hookrightarrow D_k, c''\colon D_k \not\hookrightarrow D_j\colon c = c'\,;c'' \wedge c \models \varphi \} \\
&= \{ c\colon D_i \not\hookrightarrow D_j \mid c \models \varphi \} \\
&= [\![\varphi]\!]
\end{aligned}
$$

Therefore, $\alpha_\psi^\exists(\gamma_\psi^\forall(\varphi)) \models \varphi$.

□

Proposition 6.58. Let $n \in \mathbb{N}$, $\psi\colon i \to j$ be a formula and $i, j, k \leq n$. The functions

$$
\begin{aligned}
\alpha_\psi^\forall &\colon \Phi_{j,k} \to \Phi_{i,k} & \qquad \gamma_\psi^\exists &\colon \Phi_{i,k} \to \Phi_{j,k} \\
\alpha_\psi^\forall(\varphi) &= \forall\psi\colon \varphi & \qquad \gamma_\psi^\exists(\varphi') &= \varphi'{\downarrow_\psi^\exists}
\end{aligned}
$$

are a Galois connection between $\langle \Phi_{j,k}, \models \rangle$ and $\langle \Phi_{i,k}, \models \rangle$.

Proof.

- We show the monotonicity of $\alpha_\psi^\forall$ as follows. We assume that $\varphi_1 \models \varphi_2$ holds. Then also $\neg\varphi_2 \models \neg\varphi_1$ holds

$$
\begin{aligned}
&[\![\alpha_\psi^\forall(\varphi_1)]\!] \\
&= [\![\forall\psi\colon \varphi_1]\!] \\
&= [\![\neg\,(\exists\psi\colon \neg\varphi_1)]\!] \\
&= \mathcal{M}(C,E) \setminus [\![(\exists\psi\colon \neg\varphi_1)]\!] \\
&= \mathcal{M}(C,E) \setminus \{ c\colon D_i \not\hookrightarrow D_k \mid \exists c'\colon C \not\hookrightarrow D, \exists c''\colon D_j \not\hookrightarrow D_k\colon \\
&\qquad c = c'\,;c'' \wedge c' \models [\![\psi]\!] \wedge c'' \models [\![\neg\varphi_1]\!] \} \\
&= \{ c\colon D_i \not\hookrightarrow D_k \mid \forall c'\colon D_i \not\hookrightarrow D_j, \forall c''\colon D_j \not\hookrightarrow D_k\colon c \neq c'\,;c'' \vee \\
&\qquad c' \not\models [\![\psi]\!] \vee c'' \models [\![\varphi_1]\!] \} \\
&\subseteq \{ c\colon D_i \not\hookrightarrow D_k \mid \forall c'\colon D_i \not\hookrightarrow D_j, \forall c''\colon D_j \not\hookrightarrow D_k\colon c \neq c'\,;c'' \vee \\
&\qquad c' \not\models [\![\psi]\!] \vee c'' \models [\![\varphi_2]\!] \} \\
&= \mathcal{M}(C,E) \setminus \{ c\colon D_i \not\hookrightarrow D_k \mid \exists c'\colon D_i \not\hookrightarrow D_j, \exists c''\colon D_j \not\hookrightarrow D_k\colon \\
&\qquad c = c'\,;c'' \wedge c' \models [\![\psi]\!] \wedge c'' \models [\![\neg\varphi_2]\!] \} \\
&= \mathcal{M}(C,E) \setminus [\![(\exists\psi\colon \neg\varphi_2)]\!] \\
&= [\![\neg\,(\exists\psi\colon \neg\varphi_2)]\!] \\
&= [\![\forall\psi\colon \varphi_2]\!] \\
&= [\![\alpha_\psi^\forall(\varphi_2)]\!]
\end{aligned}
$$

  Therefore, $\alpha_\psi^\forall(\varphi_1) \models \alpha_\psi^\forall(\varphi_2)$ holds.

- We show the monotonicity of $\gamma_\psi^\exists$ as follows. We assume that $\zeta_1 \models \zeta_2$ holds.

$$
\begin{aligned}
&\llbracket \gamma^{\exists}_{\psi}(\zeta_1) \rrbracket \\
&= \left\llbracket \zeta_1 \downarrow^{\exists}_{\psi} \right\rrbracket \\
&= \{c'' \colon D_j \nrightarrow D_k \mid \exists c' \colon D_i \nrightarrow D_j \colon c \models \llbracket \psi \rrbracket \wedge c' \,;\, c'' \models \llbracket \zeta_1 \rrbracket\} \\
&\subseteq \{c'' \colon D_j \nrightarrow D_k \mid \exists c' \colon D_i \nrightarrow D_j \colon c \models \llbracket \psi \rrbracket \wedge c' \,;\, c'' \models \llbracket \zeta_2 \rrbracket\} \\
&= \left\llbracket \zeta_2 \downarrow^{\exists}_{\psi} \right\rrbracket \\
&= \llbracket \gamma^{\exists}_{\psi}(\zeta_2) \rrbracket
\end{aligned}
$$

- We show the extensivity of $\alpha^{\forall}_{\psi} \,;\, \gamma^{\exists}_{\psi}$ as follows. We assume $\varphi \not\models \gamma^{\exists}_{\psi}\left(\alpha^{\forall}_{\psi}(\varphi)\right)$. Then there exists a cospan c such that $c \models \varphi$ and $c \not\models \gamma^{\exists}_{\psi}\left(\alpha^{\forall}_{\psi}(\varphi)\right)$.

$$
\begin{aligned}
& c \not\models \gamma^{\exists}_{\psi}\left(\alpha^{\forall}_{\psi}(\varphi)\right) \\
\iff\ & c \not\models \forall \psi \colon \varphi \downarrow^{\exists}_{\psi} \\
\iff\ & c \not\models \neg\left(\exists \psi \colon \neg\left(\varphi \downarrow^{\exists}_{\psi}\right)\right) \\
\iff\ & c \models \exists \psi \colon \neg\left(\varphi \downarrow^{\exists}_{\psi}\right) \\
\iff\ & c \in \Big\{ d \colon D_i \nrightarrow D_k \mid \exists d' \colon D_i \nrightarrow D_j, \exists d'' \colon D_j \nrightarrow D_k \colon d = d' \,;\, d'' \wedge \\
& \qquad d' \models \psi \wedge d'' \models \neg\left(\varphi \downarrow^{\exists}_{\psi}\right) \Big\} \\
\iff\ & c \in \Big\{ d \colon D_i \nrightarrow D_k \mid \exists d' \colon D_i \nrightarrow D_j, \exists d'' \colon D_j \nrightarrow D_k \colon d = d' \,;\, d'' \wedge \\
& \qquad d' \models \psi \wedge d'' \in \left\{ e'' \colon D_j \nrightarrow D_k \mid \forall e' \colon D_i \nrightarrow D_j \colon c' \not\models \psi \vee e' \,;\, e'' \not\models \varphi \right\} \Big\}
\end{aligned}
$$

  Now, we have that $c \not\models \gamma^{\exists}_{\psi}\left(\alpha^{\forall}_{\psi}(\varphi)\right)$ holds if and only if there exist two cospans $c' \colon D_i \nrightarrow D_j$ and $c'' \colon D_j \nrightarrow D_k$ such that $c = c' \,;\, c''$ and $c' \models \psi$ and for c'' we have that with every cospan $d' \colon D_i \nrightarrow D_j$ satisfying $d' \models \psi$ it holds $d' \,;\, c''\neg \models \varphi$. Since $c' \models \psi$, we immediately have $c = c' \,;\, c''\neg \models \varphi$, which is a contradiction. Therefore, we have $\varphi \models \gamma^{\exists}_{\psi}\left(\alpha^{\forall}_{\psi}(\varphi)\right)$.

- We show the reduction of $\gamma^{\exists}_{\psi} \,;\, \alpha^{\forall}_{\psi}$ as follows.

$$
\begin{aligned}
&\llbracket \alpha^{\forall}_{\psi}(\gamma^{\exists}_{\psi}(\varphi)) \rrbracket \\
&= \left\llbracket (\forall \psi \colon \varphi) \downarrow^{\exists}_{\psi} \right\rrbracket \\
&= \{c'' \colon D_j \nrightarrow D_k \mid \exists c' \colon D_i \nrightarrow D_j \colon c' \models \psi \wedge c' \,;\, c'' \models \forall \psi \colon \varphi\} \\
&\subseteq \{c'' \colon D_j \nrightarrow D_k \mid c'' \models \varphi\} && \text{(Prop. 6.53)} \\
&= \llbracket \varphi \rrbracket
\end{aligned}
$$

  Therefore, $\alpha^{\forall}_{\psi}(\gamma^{\exists}_{\psi}(\varphi)) \models \varphi$.

□

B
Graph Automata Encoding

In this appendix we exemplarily give the formulas encoding graph automata for the k-colorability language (presented in Examples 6.4 and 6.33) and the k-dominating set language.

In the following we will confuse the states of the graph automata and the bit strings used to encde them. Hence, every bit string is seen as a state.

B.1. Colorability Graph Automaton Encoding

In this section we explain how the n-bounded k-colorability graph automaton $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ can be encoded by propositional formulas. These formulas can in turn be used to obtain BDDs representing the graph automaton.

As already explained in Section 7.1, we encode each state by a bit string of the form:

$$\vec{b}\,\vec{c_1} \dots \vec{c_n} = b_1 \dots b_m (c_{1,1} \dots c_{1,\ell}) \dots (c_{n,1} \dots c_{n,\ell}),$$

where $\vec{b}$ encodes the (outer) interface size of the cospan decomposition seen so far and $\vec{c_j}$ encodes the color of the j-th interface node (for $1 \leq j \leq n$).

For the sake of simplicity we assume that the encoded graph automaton only accepts cospans with empty inner and outer interface. The set of all states Q, the set of initial states I and the set of final states F can then be encoded as follows:

State Set $Q = (Q_i)_{i \leq n}$: The states of the set Q_i are exactly those states

- whose interface size is equal to i,
- whose first i interface nodes are colored valid and
- whose $n - i$ last interface nodes are "uncolored".

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \in Q_i \iff (\vec{b} = i) \wedge \bigwedge_{j=1}^{i} (1 \leq \vec{c_j} \leq k) \wedge \bigwedge_{j=i+1}^{n} (\vec{c_j} = \vec{0})$$

Initial State Set I**:** The initial state set I contains exactly those states

- whose interface size is equal to 0 and
- whose nodes are all "uncolored".

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \in \mathrm{I} \iff (\vec{b} = 0) \wedge \bigwedge_{j=1}^{n} (\vec{c_j} = \vec{0})$$

Final State Set F**:** The set of final states is exactly the set of initial states.

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \in \mathrm{F} \iff (\vec{b} = 0) \wedge \bigwedge_{j=1}^{n} (\vec{c_j} = \vec{0})$$

Next, we show how to encode the transitions for the six types of letters given in Table 6.1.

$connect^i_\diamond$**-Transition:** The $connect^i_\diamond$-transition from one state to another exists if the following conditions are met:

- the arity of the edge is less or equal to the current interface size,
- the interface sizes of the current and the successor state are both equal to i,
- the color of no node is changed,
- the last $n - i$ nodes are "uncolored" in the successor state and
- the nodes incident to the new edge are pairwise colored with different colors.

We set $p = i - ar(A) + 1$.

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \xrightarrow{connect^i_\diamond} \vec{b'}\,\vec{c'_1}\ldots\vec{c'_n} \iff (ar(A) \leq i) \wedge (\vec{b} = i) \wedge (\vec{b} = \vec{b'}) \wedge \bigwedge_{j=1}^{n} (\vec{c_j} = \vec{c'}_j) \wedge \bigwedge_{j=i+1}^{n} (\vec{c_j} = 0) \wedge \bigwedge_{j=p}^{i} \bigwedge_{j'=p}^{i} (j \neq j') \rightarrow (\vec{c_j} \neq \vec{c}_{j'})$$

$fuse^i$**-Transition:** The $fuse^i$-transition from one state to another exists if the following conditions are met:

- the current interface size is between 2 and n (if the interface size is less than 2 there are no nodes to be fused),
- the interface size of the current state is equal to i and the interface size of the successor state is equal to $i - 1$,
- the color of no node is changed (except for the i-th node),
- the color of the i-th node is

  - equal to the color of the $(i-1)$-th node in the current state and
  - "uncolored" in the successor state,
- the last $n-i$ nodes are "uncolored" in the successor state.

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \xrightarrow{fuse^i} \vec{b'}\,\vec{c'_1}\ldots\vec{c'_n} \iff (2 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i-1) \wedge \bigwedge_{\substack{j=1 \\ j \neq i}}^{n} (\vec{c}_j = \vec{c'}_j) \wedge (\vec{c}_i = \vec{c}_{i-1}) \wedge (\vec{c'}_i = \vec{0}) \wedge \bigwedge_{j=i+1}^{n} (\vec{c}_j = \vec{0})$$

shift^i-**Transition:** The *shift*^{i}-transition from one state to another exists if the following conditions are met:

- the current interface size is between 3 and n (if the interface size is less than 2 there are no nodes to be shifted and if the interface is equal to 2 the *shift*-operation is identical to the *trans*-operation),
- the interface size of the current and the successor states are both equal to i,
- the color of the first i nodes is shifted cyclic and
- the color of the last $n-i$ nodes is "uncolored" before and not changed after the application of the shift.

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \xrightarrow{shift^i} \vec{b'}\,\vec{c'_1}\ldots\vec{c'_n} \iff (3 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b} = \vec{b'}) \wedge \bigwedge_{j=1}^{i-1} (\vec{c}_{j+1} = \vec{c'}_j) \wedge (\vec{c}_1 = \vec{c'}_i) \wedge \bigwedge_{j=i+1}^{n} \Big((\vec{c}_j = \vec{0}) \wedge (\vec{c}_j = \vec{c'}_j) \Big)$$

res^i-**Transition:** The *res*^{i}-transition from one state to another exists if the following conditions are met:

- the current interface size is between 1 and n (if the interface size is less than 1 there are no nodes to be restricted),
- the interface size of the current state is equal to i and the interface size of the successor state is equal to $i-1$,
- the color of no node is changed (except for the i-th node),
- the i-th node is "uncolored" in the successor node and
- the last $n-i$ nodes are "uncolored" in the successor state.

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \xrightarrow{res^i} \vec{b'}\,\vec{c'_1}\ldots\vec{c'_n} \iff (1 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i-1) \wedge \bigwedge_{\substack{j=1 \\ j \neq i}}^{n} (\vec{c}_j = \vec{c'}_j) \wedge (\vec{c'}_i = \vec{0}) \wedge \bigwedge_{j=i+1}^{n} (\vec{c}_j = \vec{0})$$

trans^i-**Transition:** The *trans*^{i}-transition from one state to another exists if the following conditions are met:

- the current interface size is between 2 and n (if the interface size is less than 2 there are no nodes to be transposed),
- the interface size of the current and the successor states are both equal to i,
- the color of the first two nodes is transposed and the color of all other nodes is not changed and
- the last $n - i$ nodes are "uncolored" in the successor state

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \xrightarrow{trans^i} \vec{b'}\,\vec{c'_1}\ldots\vec{c'_n} \iff (2 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b} = \vec{b'}) \wedge (\vec{c_1} = \vec{c'}_2) \wedge (\vec{c_2} = \vec{c'}_1) \wedge \bigwedge_{j=3}^{n} (\vec{c_j} = \vec{c'}_j) \wedge \bigwedge_{j=i+1}^{n} (\vec{c_j} = \vec{0})$$

vertex^i-**Transition:** The *vertex*^{i}-transition from one state to another exists if the following conditions are met:

- the current interface size is less than n (otherwise the new node cannot be added to the interface),
- the interface size of the current state is equal to i and the interface size of the successor state is equal to $i + 1$,
- the color of no node is changed (except for the $(i + 1)$-th node),
- the $(i + 1)$-th node is "uncolored" in the current state and has to be colored valid in the successor state and
- the last $n - i$ nodes are "uncolored" in the successor state

$$\vec{b}\,\vec{c_1}\ldots\vec{c_n} \xrightarrow{vertex^i} \vec{b'}\,\vec{c'_1}\ldots\vec{c'_n} \iff (i < n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i + 1) \wedge \bigwedge_{\substack{j=1 \\ j \neq i+1}}^{n} (\vec{c_j} = \vec{c'}_j) \wedge (\vec{c}_{i+1} = 0) \wedge (1 \leq \vec{c'}_{i+1} \leq k) \wedge \bigwedge_{j=i+2}^{n} (\vec{c_j} = \vec{0})$$

B.2. Dominating Set Graph Automaton Encoding

In this section we explain how the n-bounded k-dominating set graph automaton $\mathcal{A} = \langle Q, \Sigma, \delta, \mathrm{I}, \mathrm{F} \rangle$ can be encoded by propositional formulas. These formulas can in turn be used to obtain BDDs representing the graph automaton.

As already explained in Section 7.1, we encode each state by a bit string of the form:

$$\vec{b}\,(m_1\,d_1)\ldots(m_n\,d_n)\,\vec{s}$$

where $\vec{b}$ encodes the (outer) interface size of the cospan decomposition seen so far, the bit m_j encodes whether the j-th node is a member of the dominating set, the bit

d_j encodes whether the j-th nodes is dominated by some node and the bit string $\vec{s}$ encodes the size of the dominating set. Note that the concrete order of the bits not important for the propositional formulas given below. Hence, the formulas can be used for both encodings of the dominating set graph automaton presented in Section 7.1.

For the sake of simplicity we assume that the encoded graph automaton only accepts cospans with empty inner and outer interface. The set of all states Q, the set of initial states I and the set of final states F can then be encoded as follows:

State Set $Q = (Q_i)_{i \leq n}$: The states of the set Q_i are exactly those states

- whose interface size is equal to i,
- whose first i interface nodes may be dominated or not, but if a interface node is a member of the dominating set, it must also be dominated and
- whose size of the dominating set is between 0 and k.

$$\vec{b}\,(m_1\,d_1)\ldots(m_n\,d_n)\,\vec{s} \in Q_i \iff (\vec{b} = i) \wedge \bigwedge_{j=1}^{i} \big((m_j = 1) \to (d_j = 1)\big) \wedge \bigwedge_{j=i+1}^{n} \big((m_j = 0) \wedge (d_j = 0)\big) \wedge \bigvee_{j=0}^{k} (\vec{s} = j)$$

Initial State Set I**:** The initial states set I contains exactly those states

- whose interface size is equal to 0,
- whose interface nodes are neither member of the dominating set nor dominated by some node and
- whose size of the dominating set is between 0.

$$\vec{b}\,(m_1\,d_1)\ldots(m_n\,d_n)\,\vec{s} \in \mathrm{I} \iff (\vec{b} = 0) \wedge \bigwedge_{j=1}^{n} \big((m_j = 0) \wedge (d_j = 0)\big) \wedge (\vec{s} = 0)$$

Final State Set F**:** The set of final states F contains exactly those states

- whose interface size is equal to 0,
- whose interface nodes are neither member of the dominating set nor dominated by some node and
- whose size of the dominating set is between 0 and k.

$$\vec{b}\,(m_1\,d_1)\ldots(m_n\,d_n)\,\vec{s} \in \mathrm{I} \iff (\vec{b} = 0) \wedge \bigwedge_{j=1}^{n} \big((m_j = 0) \wedge (d_j = 0)\big) \wedge \bigvee_{j=0}^{k} (\vec{s} = j)$$

Next, we show how to encode the transitions for the six types of letters given in Table 6.1.

$connect^i_\diamond$-Transition: The $connect^i_\diamond$-transition from one state to another exists if the following conditions are met:

- the arity of the edge is less or equal to the current interface size,
- the interface sizes of the current and the successor state are both equal to i,
- the membership bit of no interface node is changed,
- the dominated bit of an interface node
  - which is not incident to the new edge is not changed
  - which is incident to the new edge
    - is set if at least one node incident to the new edge is a member of the dominating set,
    - is not changed if no node incident to the new edge is a member of the dominating set,
- the size of dominating set is not changed.

$$
\begin{aligned}
q \xrightarrow{connect^i_\diamond} q' \iff\ & (ar(A) \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b} = \vec{b'}) \wedge \\
& \bigwedge_{j=1}^{n} (m_j = m'_j) \wedge \bigwedge_{j=1}^{i-ar(A)} (d_j = d'_j) \wedge \bigwedge_{j=i+1}^{n} (d_j = d'_j) \wedge \\
& \left(\bigvee_{j=i-ar(A)+1}^{i} (m_j = 1) \rightarrow \bigwedge_{j=i-ar(A)+1}^{i} (d'_j = 1) \right) \wedge \\
& \left(\bigwedge_{j=i-ar(A)+1}^{i} (m_j \neq 1) \rightarrow \bigwedge_{j=i-ar(A)+1}^{i} (d_j = d'_j) \right) \wedge (\vec{s} = \vec{s'})
\end{aligned}
$$

$fuse^i$-Transition: The $fuse^i$-transition from one state to another exists if the following conditions are met:

- the current interface size is between 2 and n (if the interface size is less than 2 there are no nodes to be fused),
- the interface size of the current state is equal to i and the interface size of the successor state is equal to $i - 1$,
- the membership bit and the the dominated bit of no interface node is changed (except for the $(i-1)$-th and the i-th node),
- the membership bit of the $(i-1)$-th interface node in the successor state is set to the maximum of the membership bits of the $(i-1)$-th and the i-th interface nodes in current state,
- the membership bit of the i-th interface node in the successor state is cleared,

- the dominated bit of the $(i-1)$-th interface node in the successor state is set to the maximum of the dominated bits of the $(i-1)$-th and the i-th interface nodes in current state,
- the dominated bit of the i-th interface node in the successor state is cleared and
- the size of dominating set
  - is decreased by 1 if both nodes, which were fused, were members of the domating set (in this case the fused node was counted twice) and
  - is not changed if at most one of the two nodes, which are fused, was a member of the domating set.

$$
\begin{aligned}
q \xrightarrow{fuse^i} q' \iff\ & (2 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i-1) \wedge \\
& \bigwedge_{\substack{j=1 \\ j \neq i-1 \\ j \neq i}}^{n} \big((m_j = m'_j) \wedge (d_j = d'_j)\big) \wedge \big(m'_{i-1} = \max\{m_i, m_{i-1}\}\big) \wedge \\
& \big(d'_{i-1} = \max\{d_i, d_{i-1}\}\big) \wedge (m'_i = 0) \wedge (d'_i = 0) \wedge \\
& \Big(\big((m_{i-1} = 1) \wedge (m_i = 1)\big) \rightarrow (\vec{s'} = \vec{s} - 1)\Big) \wedge \\
& \Big(\big((m_{i-1} = 0) \vee (m_i = 0)\big) \rightarrow (\vec{s'} = \vec{s})\Big)
\end{aligned}
$$

shift^i-**Transition:** The *shift*^{i}-transition from one state to another exists if the following conditions are met:

- the current interface size is between 3 and n (if the interface size is less than 2 there are no nodes to be shifted and if the interface is equal to 2 the *shift*-operation is identical to the *trans*-operation),
- the interface sizes of the current and the successor state are both equal to i,
- the membership bits and the dominated bits of the first i nodes are shifted cyclic,
- the membership bits and the the dominated bits of the last $n-i$ nodes are cleared before and not changed after the application of the shift and
- the size of dominating set is not changed

$$
\begin{aligned}
q \xrightarrow{shift^i} q' \iff\ & (3 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b'} = \vec{b}) \wedge \\
& \bigwedge_{j=1}^{i-1} \big((m_j = m'_{j+1}) \wedge (d_j = d'_{j+1})\big) \wedge (m_i = m'_0) \wedge (d_i = d'_0) \wedge \\
& \bigwedge_{j=i+1}^{n} \big((m_j = m'_j) \wedge (d_j = d'_j)\big) \wedge (\vec{s} = \vec{s'})
\end{aligned}
$$

res^i-**Transition:** The res^i-transition from one state to another exists if the following conditions are met:

- the current interface size is between 1 and n (if the interface size is less than 1 there are no nodes to be restricted),
- the interface size of the current state is equal to i and the interface size of the successor state is equal to $i - 1$,
- the membership bit and the the dominated bit of no interface node is changed (except for the the i-th node),
- the i-th membership bit of the successor state is cleared,
- the i-th dominated bit has to be set for the current state (since only dominated nodes may be restricted from the interface) and is cleared in the successor state and
- the size of dominating set is not changed

$$q \xrightarrow{res^i} q' \iff (3 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i - 1) \wedge \bigwedge_{\substack{j=1 \\ j \neq i}}^{n} \big((m_j = m'_j) \wedge (d_j = d'_j)\big) \wedge (m'_i = 0) \wedge (d_i = 1) \wedge (d'_i = 0) \wedge (\vec{s'} = \vec{s})$$

$trans^i$-**Transition:** The $trans^i$-transition from one state to another exists if the following conditions are met:

- the current interface size is between 2 and n (if the interface size is less than 2 there are no nodes to be transposed),
- the interface size of the current and the successor states are both equal to i,
- the membership bits and the the dominated bits of the first two interface nodes are transposed, the membership bits and the the dominated bits of all other interface nodes are not changed and
- the size of dominating set is not changed

$$q \xrightarrow{trans^i} q' \iff (2 \leq i \leq n) \wedge (\vec{b} = i) \wedge (\vec{b'} = \vec{b}) \wedge (m_1 = m'_2) \wedge (m_2 = m'_1) \wedge (d_1 = d'_2) \wedge (d_2 = d'_1) \wedge \bigwedge_{j=3}^{n} \big((m_j = m'_j) \wedge (d_j = d'_j)\big) \wedge (\vec{s} = \vec{s'})$$

$vertex^i$-**Transition:** The $vertex^i$-transition from one state to another exists if the following conditions are met:

- the current interface size is less than n (otherwise the new node cannot be added to the interface),

- the interface size of the current state is equal to i and the interface size of the successor state is equal to $i+1$,
- the membership bits and the dominated bits of the first i nodes are not changed,
- the membership bit and the dominated bit of the $(i+1)$-th node are both unset in the current state and are both either set or unset in the successor state and
- the size of dominating set
  - is increased by 1 if the $(i+1)$-th membership bit is set in the successor state and
  - is not changed if the $(i+1)$-th membership bit is not set in the successor state.

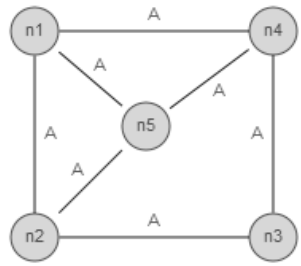
$$
\begin{aligned}
q \xrightarrow{vertex^i} q' \iff\ & (i < n) \wedge (\vec{b} = i) \wedge (\vec{b'} = i+1) \wedge \\
& \bigwedge_{\substack{j=1 \\ j \neq i+1}}^{n} \big((m_j = m'_j) \wedge (d_j = d'_j)\big) \wedge \\
& (m_{i+1} = 0) \wedge (d_{i+1} = 0) \wedge (m'_{i+1} = d'_{i+1}) \wedge \\
& \Big((m'_{i+1} = 1) \rightarrow (\vec{s'} = \vec{s} + 1)\Big) \wedge \Big((m'_{i+1} = 0) \rightarrow (\vec{s'} = \vec{s})\Big)
\end{aligned}
$$

C File Formats

C.1. The GXL Format for Graphs

In this section, we give an example of a graph represented in the GXL-format which is used to save graphs to `.gxl`-files. For a detailed description of the GXL-format see the website at http://www.gupro.de/GXL/.

For our example, we take the following graph:



where the edge from node n_1 to node n_2 is named e_1, the edge from node n_1 to node n_3 is named e_2, and so on. The graph depicted above can then be represented by the following XML-file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">

<gxl>
  <graph id="G" edgeids="true" edgemode="undirected"
         hypergraph="true">
    <node id="n1" />
    <node id="n2" />
```

```
<node id="n3" />
<node id="n4" />
<node id="n5" />

<rel id="e1">
  <attr name="label">
    <string>A</string>
  </attr>
  <relend target="n1" role="vertex" startorder="0" />
  <relend target="n2" role="vertex" startorder="1" />
</rel>

<rel id="e2">
  <attr name="label">
    <string>A</string>
  </attr>
  <relend target="n1" role="vertex" startorder="0" />
  <relend target="n4" role="vertex" startorder="1" />
</rel>

<rel id="e3">
  <attr name="label">
    <string>A</string>
  </attr>
  <relend target="n1" role="vertex" startorder="0" />
  <relend target="n5" role="vertex" startorder="1" />
</rel>

<rel id="e4">
  <attr name="label">
    <string>A</string>
  </attr>
  <relend target="n2" role="vertex" startorder="0" />
  <relend target="n3" role="vertex" startorder="1" />
</rel>

<rel id="e5">
  <attr name="label">
    <string>A</string>
  </attr>
  <relend target="n2" role="vertex" startorder="0" />
  <relend target="n5" role="vertex" startorder="1" />
</rel>

<rel id="e6">
  <attr name="label">
    <string>A</string>
```

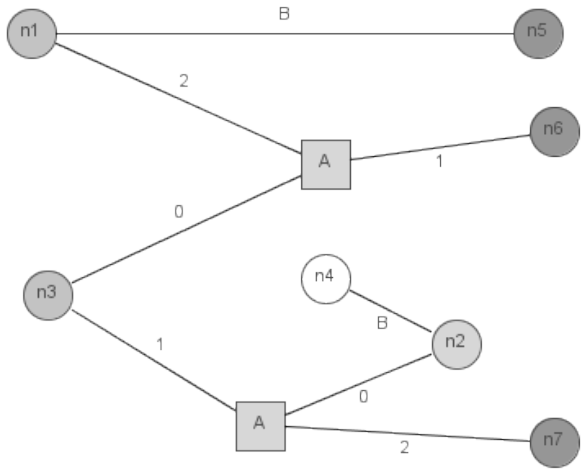
```
      </attr>
      <relend target="n3" role="vertex" startorder="0" />
      <relend target="n4" role="vertex" startorder="1" />
    </rel>

    <rel id="e7">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n4" role="vertex" startorder="0" />
      <relend target="n5" role="vertex" startorder="1" />
    </rel>
  </graph>
</gxl>
```

C.2. The GXL Format for Cospans

In this section, we give an example of a cospan represented in the GXL-format which is used to save cospans to `.cos`-files. The file format is very similar to that used for graphs (see Section C.1). The greatest difference is that we also need to save inner and outer interface of the cospan.

For our example, we take the following cospan:



where the B-edge incident to the nodes n_1 and n_5 is named e_1, the A-edge incident to the nodes n_1, n_3 and n_6 is named e_2, the A-edge incident to the nodes n_2, n_4 and n_7 is named e_3 and the B-edge incident to the nodes n_2 and n_4 is named e_4. Furthermore, the inner interface consists of the nodes $n1$, $n3$ as well as $n4$ and the outer interface consists of the nodes $n4$, $n5$, $n6$ and $n7$.

The graph depicted above can then be represented by the following XML-file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
```

```
<gxl>
  <graph id="innerInterface">
    <attr name="n1">
      <string>"n1"</string>
    </attr>
    <attr name="n3">
      <string>"n3"</string>
    </attr>
    <attr name="n4">
      <string>"n4"</string>
    </attr>
  </graph>

  <graph id="middleGraph" edgeids="true" edgemode="undirected"
         hypergraph="true">
    <node id="n1" />
    <node id="n5" />
    <node id="n4" />
    <node id="n3" />
    <node id="n2" />
    <node id="n7" />
    <node id="n6" />

    <rel id="e3">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n2" role="vertex" startorder="0" />
      <relend target="n3" role="vertex" startorder="1" />
      <relend target="n7" role="vertex" startorder="2" />
    </rel>

    <rel id="e2">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n3" role="vertex" startorder="0" />
      <relend target="n6" role="vertex" startorder="1" />
      <relend target="n1" role="vertex" startorder="2" />
    </rel>

    <rel id="e4">
      <attr name="label">
        <string>B</string>
      </attr>
      <relend target="n2" role="vertex" startorder="0" />
```

```
      <relend target="n4" role="vertex" startorder="1" />
    </rel>

    <rel id="e1">
      <attr name="label">
        <string>B</string>
      </attr>
      <relend target="n1" role="vertex" startorder="0" />
      <relend target="n5" role="vertex" startorder="1" />
    </rel>
  </graph>

  <graph id="outerInterface">
    <attr name="n5">
      <string>"n5"</string>
    </attr>
    <attr name="n7">
      <string>"n7"</string>
    </attr>
    <attr name="n4">
      <string>"n4"</string>
    </attr>
    <attr name="n6">
      <string>"n6"</string>
    </attr>
  </graph>
</gxl>
```

C.3. The Automaton File Format

In this section, we briefly explain the `.aut`-file format which is used to save automata to files. Every `.aut`-file is essentially a ZIP-file consisting of different files depending on the automaton. Each `.aut`-file contains

- a file named `general.obj` which holds the serialized[1] representation of the corresponding automaton object,
- a file named `states.bdd`[2] holding information about the Bdd representing the set of all states,
- a file named `initial.bdd` holding information about the Bdd representing the set of initial states,
- a file named `final.bdd` holding information about the Bdd representing the set of final states,

[1] See http://docs.oracle.com/javase/7/docs/technotes/guides/serialization/ for further information about serialization in Java.

[2] For a description of the file format used in the `.bdd`-files see the manual of the BuDDy-package.

- a file named `nonFinal.bdd` holding information about the BDD representing the set of non-final states
- depending on the input alphabet of the corresponding automaton a `.bdd`-file for each atomic cospan of the input alphabet is contained: these `.bdd`-files hold information about the BDDs representing the transition relations for the particular atomic cospans.

C.4. The Signature File Format

In this section, we briefly explain the `.sig`-file format which is used to save signatures to files. Since a signature is just a set of atomic cospans (used as input alphabet for graph automata) the file format is rather simple. The format is text- and line-oriented, i. e. each line of the file consists of the name of exactly one atomic cospan.

C.5. The Cospan Decomposition File Format

In this section, we briefly explain the `.dec`-file format which is used to save cospan decompositions to files. Since a cospan decomposition is just a sequence of atomic cospans the file format is rather simple. The format is text- and line-oriented, i. e. each line of the file consists of the name of exactly one atomic cospan. With one exception: the first line of the file indicates the size of the inner interface of the resulting cospan. The size of the outer interface of the resulting cospan (and all intermediate interface sizes) can then be computed by the sequence of atomic cospans.

D
Raven Libraries

Library	License	URL
ANTLR	ANTLR 4 License	http://www.antlr.org/
BuDDy	Public Domain	http://buddy.sourceforge.net/
GraphViz	Eclipse Public License 1.0	http://www.graphviz.org/
JAnsi	Apache License 2.0	http://jansi.fusesource.org/
JavaBDD	GNU LGPL 2.0	http://javabdd.sourceforge.net/
JDOM	Apache-style License	http://www.jdom.org/
JGoodies	BSD 2-Clause License	http://www.jdom.org/
JGraphX	BSD 3-Clause License	http://www.jgraph.com/jgraph.html
LibTW	Public Domain	http://www.treewidth.com/

Table D.1.: Libraries on which Raven depends

References

[1] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. "When Simulation Meets Antichains (on Checking Language Inclusion of NFAs)". In: *Proceedings of TACAS '10 (Conference on Tools and Algorithms for the Construction and Analysis of Systems)*. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 158–174.

[2] Jirí Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories – The Joy of Cats*. Dover Publications, 2009, pp. 1–517.

[3] Sheldon B. Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* 100.6 (1978), pp. 509–516.

[4] Stefan Arnborg, Jens Lagergren, and Detlef Seese. "Easy problems for tree-decomposable graphs". In: *Journal of Algorithms* 12.2 (1991), pp. 308–340.

[5] Stefan Arnborg and Andrzej Proskurowski. "Characterization and Recognition of Partial 3-Trees". In: *SIAM Journal on Algebraic Discrete Methods* 7.2 (1986), pp. 305–314.

[6] Adam Bakewell, Detlef Plump, and Colin Runciman. "Checking the Shape Safety of Pointer Manipulations". In: *Proceedings of RelMiCS '03 (Relational and Algebraic Methods in Computer Science)*. Ed. by Rudolf Berghammer, Bernhard Möller, and Georg Struth. Vol. 3051. Lecture Notes in Computer Science. Springer, 2003, pp. 48–61.

[7] Michel Bauderon and Bruno Courcelle. "Graph Expressions and Graph Rewritings". In: *Mathematical Systems Theory* 20.2–3 (1987), pp. 83–127.

[8] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. "Symbolic invariant verification for systems with dynamic structural adaptation". In: *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. Proceedings of ICSE '06. ACM, 2006, pp. 72–81.

[9] Christoph Blume. "Efficient Implementation of Automaton Functors for the Verification of Graph Transformation Systems". In: *Proceedings of ICGT '10 (International Conference on Graph Transformation), Proceedings of the Doctoral Symposium*. Ed. by Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr. Vol. 38. Electronic Communications of the EASST. 2010.

[10] Christoph Blume. "Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen". (in German). MA thesis. Universität Duisburg-Essen, 2008.

[11] Christoph Blume. “Recognizable Graph Languages for the Verification of Dynamic Systems”. In: *Proceedings of ICGT ’10 (International Conference on Graph Transformation), Proceedings of the Doctoral Symposium*. Ed. by Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr. Vol. 6372. Lecture Notes in Computer Science. Springer, 2010, pp. 384–387.

[12] Christoph Blume, H. J. Sander Bruggink, Dominik Engelke, and Barbara König. “Efficient Symbolic Implementation of Graph Automata with Applications to Invariant Checking”. In: *Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 7562. Lecture Notes in Computer Science. Springer, 2012, pp. 264–278.

[13] Christoph Blume, H. J. Sander Bruggink, Martin Friedrich, and Barbara König. “Treewidth, Pathwidth and Cospan Decompositions”. In: *Proceedings of GT-VMT 2011 (Workshop on Graph Transformation and Visual Modeling Techniques)*. Vol. 41. Electronic Communications of the EASST. 2011.

[14] Christoph Blume, H. J. Sander Bruggink, Martin Friedrich, and Barbara König. “Treewidth, Pathwidth and Cospan Decompositions with Applications to Graph-accepting Tree Automata”. In: *Journal of Visual Languages & Computing* 24.3 (2013), pp. 192–206.

[15] Christoph Blume, H. J. Sander Bruggink, and Barbara König. “Recognizable Graph Languages for Checking Invariants”. In: *Proceedings of GT-VMT 2010 (Workshop on Graph Transformation and Visual Modeling Techniques)*. Vol. 29. Electronic Communications of the EASST. 2010.

[16] Gregor von Bochmann. “Finite state description of communication protocols”. In: *Computer Networks (1976)* 2.4—5 (1978), pp. 361–372.

[17] Hans Leo Bodlaender. “A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth”. In: *SIAM Journal on Computation* 25.6 (1996), pp. 1305–1317.

[18] Hans Leo Bodlaender. “Dynamic programming on graphs with bounded treewidth”. In: *Automata, Languages and Programming*. Ed. by Timo Lepistö and Arto Salomaa. Vol. 317. Lecture Notes in Computer Science. Springer, 1988, pp. 105–118.

[19] Hans Leo Bodlaender and Arie Marinus Catharinus Antonius Koster. “Combinatorial Optimization on Graphs of Bounded Treewidth”. In: *The Computer Journal* 51.3 (2008), pp. 255–269.

[20] Hans Leo Bodlaender and Arie Marinus Catharinus Antonius Koster. *Treewidth Computations I. Upper Bounds*. Tech. rep. UU-CS-2008-032. Department of Information and Computing Sciences, Utrecht University, Sept. 2008.

[21] Hans Leo Bodlaender and Arie Marinus Catharinus Antonius Koster. “Treewidth Computations II. Lower bounds”. In: *Information and Computation* 209.7 (2011), pp. 1103–1119.

[22] Hans Leo Bodlaender, Johan M. M. van Rooij, and Peter Rossmanith. “Dynamic Programming on Tree Decompositions Using Generalised Fast Subset Convolution”. In: *Algorithms – ESA 2009 (European Symposium on Algorithms)*. Ed. by Amos Fiat and Peter Sanders. Vol. 5757. Lecture Notes in Computer Science. Springer, 2009, pp. 566–577.

[23] Filippo Bonchi and Damien Pous. “Checking NFA equivalence with bisimulations up to congruence”. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL)*. Proceedings of POPL ’13. ACM, 2013, pp. 457–468.

[24] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. “Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families”. In: *Algorithmica* 7.1–6 (1992), pp. 555–581.

[25] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. “Abstract Regular Tree Model Checking of Complex Dynamic Data Structures”. In: *Procedings of SAS ’06 (International Static Analysis Symposium)*. Vol. 4134. Lecture Notes in Computer Science. Springer, 2006, pp. 52–70.

[26] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. “Regular Model Checking”. In: *Proceedings of CAV ’00 (Conference on Computer Aided Verification)*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 403–418.

[27] Symeon Bozapalidis and Antonios Kalampakas. “Graph automata”. In: *Theoretical Computer Science* 393.1–3 (2008), pp. 147–165.

[28] Symeon Bozapalidis and Antonios Kalampakas. “Recognizability of graph and pattern languages”. In: *Acta Informatica* 42.8/9 (2006), pp. 553–581.

[29] Franz J. Brandenburg and Kostas Skodinis. “Finite graph automata for linear and boundary graph languages”. In: *Theoretical Computer Science* 332.1-–3 (2005), pp. 199–232.

[30] Franz J. Brandenburg and Kostas Skodinis. “Graph automata for linear graph languages”. In: *Graph Grammars and Their Application to Computer Science*. Ed. by Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg. Vol. 1073. Lecture Notes in Computer Science. Springer, 1996, pp. 336–350.

[31] H. J. Sander Bruggink, Raphaël Cauderlier, Barbara König, and Mathias Hülsbusch. “Conditional Reactive Systems”. In: *Proceedings of FSTTCS ’11 (Conference on Foundations of Software Technology and Theoretical Computer Science)*. Vol. 13. LIPIcs. Schloss Dagstuhl – Leibniz Center for Informatics, 2011.

[32] H. J. Sander Bruggink and Barbara König. “A Logic on Subobjects and Recognizability”. In: *Proceedings of IFIP-TCS ’10 (IFIP Theoretical Computer Science)*. Vol. 323. IFIP-AICT. Springer, 2010, pp. 197–212.

[33] H. J. Sander Bruggink and Barbara König. “On the Recognizability of Arrow and Graph Languages”. In: *Proceedings of ICGT ’08 (International Conference on Graph Transformation)*. Vol. 5214. Lecture Notes in Computer Science. Springer, 2008, pp. 336–350.

[34] H. J. Sander Bruggink, Barbara König, and Sebastian Küpper. “Concatenation and other Closure Properties of Recognizable Languages in Adhesive Categories”. In: *Proceedings of GT-VMT 2013 (Workshop on Graph Transformation and Visual Modeling Techniques)*. to appear. 2013.

[35] H. J. Sander Bruggink, Barbara König, and Hans Zantema. “Termination Analysis for Graph Transformation Systems”. To be published.

[36] Randal Everitt Bryant. “Graph-based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* 100.8 (1986), pp. 677–691.

[37] Randal Everitt Bryant. “Symbolic Boolean manipulation with ordered binary-decision diagrams”. In: *ACM Computing Surveys* 24.3 (1992), pp. 293–318.

[38] Julius Richard Büchi. “Weak Second-Order Arithmetic and Finite Automata”. In: *Mathematical Logic Quarterly* 6.1–6 (1960), pp. 66–92.

[39] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. Release October, 12th 2007. 2007. URL: http://www.grappa.univ-lille3.fr/tata.

[40] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reicko Heckel, and Michael Löwe. “Algebraic Approaches to Graph Transformation 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, pp. 163–245.

[41] Bruno Courcelle. “Graph Rewriting: An Algebraic and Logic Approach”. In: *Formal Models and Semantics*. Vol. B. Handbook of Theoretical Computer Science. Elsevier, 1990, pp. 193–242.

[42] Bruno Courcelle. “Recognizable Sets of Graphs: Equivalent Definitions and Closure Properties”. In: *Mathematical Structures in Computer Science* 4 (01 1994), pp. 1–32.

[43] Bruno Courcelle. “The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. Ed. by Grzegorz Rozenberg. World Scientific, 1997. Chap. 5.

[44] Bruno Courcelle. “The Monadic Second-Order Logic of Graphs I. Recognizable Sets of finite Graphs”. In: *Information and Computation* 85 (1990), pp. 12–75.

[45] Bruno Courcelle and Irène Durand. “Verifying monadic second order graph properties with tree automata”. In: *European Lisp Symposium*. May 2010.

[46] Bruno Courcelle and Irène A. Durand. “Automata for the verification of monadic second-order graph properties”. In: *Journal of Applied Logic* 10.4 (2012), pp. 368–409.

[47] Bruno Courcelle and Irène A. Durand. “Fly-Automata, Their Properties and Applications”. In: *Implementation and Application of Automata*. Ed. by Béatrice Bouchou-Markhoff, Pascal Caron, Jean-Marc Champarnaud, and Denis Maurel. Vol. 6807. Lecture Notes in Computer Science. Springer, 2011, pp. 264–272.

[48] Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic. A language-theoretic approach.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2012.

[49] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. "Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width". In: *Theory of Computing Systems* 33.2 (2000), pp. 125–150.

[50] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. "On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic". In: *Discrete Applied Mathematics* 108.1–2 (2001), pp. 23–52.

[51] Bruno Courcelle and Mohamed Mosbah. "Monadic second-order evaluations on tree-decomposable graphs". In: *Theoretical Computer Science* 109.1—2 (1993), pp. 49–82.

[52] Yves Crama and Peter Ladislaw Hammer. *Boolean Functions: Theory, Algorithms, and Applications.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011.

[53] Joseph C. Culberson and Feng Luo. "Exploring the k-colorable Landscape with Iterated Greedy". In: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996, pp. 245–284.

[54] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. *Computing treewidth with LibTW.* (online). Nov. 2006. URL: http://www.treewidth.com/docs/LibTW.pdf.

[55] David L. Dill, Alan John Hu, and Howard Wong-Toi. "Checking for Language Inclusion Using Simulation Preorders". In: *Computer Aided Verification.* Ed. by Kim Guldstrand Larsen and Arne Skou. Vol. 575. Lecture Notes in Computer Science. Springer, 1992, pp. 255–265.

[56] Rodney G. Downey and Michael Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.

[57] Laurent Doyen, Nicolas Maquet, Jean-François Raskin, and Martin De Wulf. "Alaska – Antichains for Logic, Automata and Symbolic Kripke structure Analysis". In: *Automated Technology for Verification and Analysis.* Ed. by Sungdeok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan. Vol. 5311. Lecture Notes in Computer Science. Springer, 2008, pp. 240–245.

[58] Irène A. Durand. "Autowrite: A Tool for Term Rewrite Systems and Tree Automata". In: *Electronic Notes in Computer Science* 124.2 (2005), pp. 29–49.

[59] Irène A. Durand. "Implementing Huge Term Automata". In: *Proceedings of the 4th European Lisp Symposium.* 2011, pp. 17–27.

[60] Andrzej Ehrenfeucht, David Haussler, and Grzegorz Rozenberg. "On Regularity of Context-Free Languages". In: *Theoretical Computer Science* 27 (1983), pp. 311–332.

[61] Hartmut Ehrig. “Introduction to the algebraic theory of graph grammars (a survey)”. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Vol. 73. Lecture Notes in Computer Science. Springer, 1979, pp. 1–69.

[62] Hartmut Ehrig. “Tutorial introduction to the algebraic approach of graph grammars”. In: *Graph-Grammars and Their Application to Computer Science*. Ed. by Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld. Vol. 291. Lecture Notes in Computer Science. Springer, 1987, pp. 1–14.

[63] Hartmut Ehrig, Martin Korff, and Michael Löwe. “Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts”. In: *Graph Grammars and Their Application to Computer Science*. Ed. by Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 532. Lecture Notes in Computer Science. Springer, 1991, pp. 24–37.

[64] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. “Graph-Grammars: An Algebraic Approach”. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*. SWAT ’73. IEEE Computer Society, 1973, pp. 167–180.

[65] Calvin C. Elgot. “Decision problems of finite automata design and related arithmetics”. In: *Transactions of the American Mathematical Society* 98 (1961), pp. 21–52.

[66] Joost Engelfriet and George Leih. “Linear graph grammars: Power and complexity”. In: *Information and Computation* 81.1 (1989), pp. 88–121.

[67] Robert W Floyd and Jeffrey David Ullman. “The Compilation of Regular Expressions into Integrated Circuits”. In: *Journal of the ACM* 29.3 (1982), pp. 603–622.

[68] Jörg Flum, Markus Frick, and Martin Grohe. “Query evaluation via tree-decompositions”. In: *Journal of the ACM* 49.6 (2002), pp. 716–752.

[69] Pascal Fradet and Daniel Le Métayer. “Shape types”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. Proceedings of POPL ’97. ACM, 1997, pp. 27–39.

[70] Pierre Fraigniaud. “Greedy Routing in Tree-Decomposed Graphs”. In: *Algorithms – ESA 2005*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Vol. 3669. Lecture Notes in Computer Science. Springer, 2005, pp. 791–802.

[71] Laurent Fribourg and Hans Olsén. “Reachability sets of parameterized rings as regular languages”. In: *Proceedings of Infinity ’97*. Vol. 9. Electronic Notes in Theoretical Computer Science. Elsevier, 1997.

[72] Markus Frick. “Easy Instances for Model Checking”. PhD thesis. Universität Freiburg, 2001.

[73] Martin Friedrich. “Baumautomaten und Baumzerlegungen für erkennbare Graphsprachen”. (in German). MA thesis. Universität Duisburg-Essen, July 2010.

[74] Paul H.B. Gardiner, Clare E. Martin, and Oege de Moor. “An algebraic construction of predicate transformers”. In: *Science of Computer Programming* 22.1—2 (1994), pp. 21–44.

[75] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[76] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. "Match-bounded string rewriting". In: *Applicable Algebra in Engineering, Communication and Computing* 15.3–4 (2004), pp. 149–171.

[77] Georg Gottlob, Reinhard Pichler, and Fang Wei. "Bounded treewidth as a key to tractability of knowledge representation and reasoning". In: *Journal of Artificial Intelligence* 174.1 (2010), pp. 105–132.

[78] Gary Griffing. "Composition-representative subsets". In: *Theory and Applications of Categories* 11.19 (2003), pp. 420–437.

[79] Weixiang Guan. "Heuristics and Tool Support for Generating Graph Decompositions". MA thesis. Universität Duisburg-Essen, to appear.

[80] Annegret Habel, Hans-Jörg Kreowski, and Clemens Lautemann. "A comparison of compatible, finite, and inductive graph properties". In: *Theoretical Computer Science* 110.1 (1993), pp. 145–168.

[81] Annegret Habel, Hans-Jörg Kreowski, and Walter Vogler. "Metatheorems for decision problems on hyperedge replacement graph languages". In: *Acta Informatica* 26.7 (1989), pp. 657–677.

[82] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. "Weakest Preconditions for High-Level Programs". In: *Proceedings of ICGT '06 (International Conference on Graph Transformation)*. Ed. by Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg. Vol. 4178. Lecture Notes in Computer Science. Springer, 2006, pp. 445–460.

[83] Graham Higman. "Ordering by Divisibility in Abstract Algebras". In: *Proceedings of the London Mathematical Society* s3–2.1 (1952), pp. 326–336.

[84] Richard Craig Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. "GXL: A graph-based standard exchange format for reengineering". In: *Science of Computer Programming* 60.2 (2006), pp. 149–170.

[85] John Edward Hopcroft and Richard Manning Karp. *A Linear Algorithm for Testing Equivalence of Finite Automata*. Tech. rep. Cornell University, 1971.

[86] John Edward Hopcroft and Jeffrey David Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[87] Lauri Karttunen. "Applications of Finite-State Transducers in Natural Language Processing". In: *Implementation and Application of Automata*. Ed. by Shen Yu and Andrei Păun. Vol. 2088. Lecture Notes in Computer Science. Springer, 2001, pp. 34–46.

[88] Gregory Maxwell Kelly. *Basic Concepts of Enriched Category Theory*. Vol. 64. Cambridge University Press, 1982.

[89] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. "MONA Implementation Secrets". In: *International Journal of Foundations of Computer Science* 13.4 (2002), pp. 571–586.

[90] Joachim Kneis and Alexander Langer. “A Practical Approach to Courcelle’s Theorem”. In: *Electronic Notes in Theoretical Computer Science* 251 (2009), pp. 65–81.

[91] Joachim Kneis, Alexander Langer, and Peter Rossmanith. “Courcelle’s Theorem – A Game-Theoretic Approach”. In: *Discrete Optimization* 8.4 (2011), pp. 568–594.

[92] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. “Decidability of Safety in Graph-based Models for Access Control”. In: *Proceedings of the 7th European Symposium on Research in Computer Security*. LNCS 2502. Springer, 2002, pp. 229–243.

[93] Hans-Jörg Kreowski and Grzegorz Rozenberg. “On the constructive description of graph languages accepted by finite automata”. In: *Mathematical Foundations of Computer Science 1981*. Ed. by Jozef Gruska and Michal Chytil. Vol. 118. Lecture Notes in Computer Science. Springer, 1981, pp. 398–409.

[94] Bernt Kullbach, Volker Riediger, and Andreas Winter. “An Overview of the GXL Graph Exchange Language”. In: *Software Visualization*. Ed. by Stephan Diehl. Vol. 2269. Lecture Notes in Computer Science. Springer, 2002, pp. 324–336.

[95] Sebastian Küpper. “Abschlusseigenschaften für Graph-Sprachen mit Anwendungen auf Terminierungsanalyse”. in German. MA thesis. Universität Duisburg-Essen, 2012.

[96] Francis William Lawvere and Robert Rosebrugh. *Sets for Mathematics*. Cambridge University Press, 2003.

[97] Chang-Yeong Lee. “Representation of Switching Circuits by Binary-Decision Programs”. In: *Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[98] James Judi Leifer and Robin Milner. “Deriving Bisimulation Congruences for Reactive Systems”. In: *CONCUR 2000 – Concurrency Theory*. Ed. by Catuscia Palamidessi. Vol. 1877. Lecture Notes in Computer Science. Springer, 2000, pp. 243–258.

[99] Thomas Lengauer and Egon Wanke. “Efficient analysis of graph properties on context-free graph languages”. In: *Automata, Languages and Programming*. Ed. by Timo Lepistö and Arto Salomaa. Vol. 317. Lecture Notes in Computer Science. Springer, 1988, pp. 379–393.

[100] Gary Lewandowski and Anne Condon. “Experiments with Parallel Graph Coloring Heuristics and Applications of Graph Coloring”. In: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996, pp. 309–334.

[101] Jørn Lind-Nielsen. *BuDDy – A Binary Decision Diagram Package*. (online). URL: http://sourceforge.net/projects/buddy.

[102] Aldo de Luca and Stefano Varricchio. “Well-quasi-orders and Regular Languages”. In: *Acta Informatica* 31.6 (1994), pp. 539–557.

[103] Johann A. Makowsky. “Algorithmic uses of the Feferman–Vaught Theorem”. In: *Annals of Pure and Applied Logic* 126.1—3 (2004), pp. 159–213.

[104] Aart Middeldorp. “Approximating Dependency Graphs Using Tree Automata Techniques”. In: *Automated Reasoning*. Ed. by Rajeev Goré, Alexander Leitsch, and Tobias Nipkow. Vol. 2083. Lecture Notes in Computer Science. Springer, 2001, pp. 593–610.

[105] David L. Milgram. “Web automata”. In: *Information and Control* 29.2 (1975), pp. 162–184.

[106] Rolf H. Möhring. “Graph Problems Related to Gate Matrix Layout and PLA Folding”. In: *Computational Graph Theory*. Ed. by Gottfried Tinhofer, Ernst Wilhelm Mayr, Hartmut Noltemeier, and Maciej Marek Sysło. Vol. 7. Computing Supplementum. Springer Vienna, 1990, pp. 17–51.

[107] Craig Morgenstern. “Distributed coloration neighborhood search”. In: *Discrete Mathematics and Theoretical Computer Science* 26 (1996), pp. 335–358.

[108] Karl-Heinz Pennemann. “Development of correct graph transformation systems”. PhD thesis. Department of Computing Science, University of Oldenburg, 2009.

[109] Benjamin Crawford Pierce. *Basic Category Theory for Computer*. Foundations of Computing Series. MIT Press, 1991.

[110] Andreas Potthoff, Sebastian Seibert, and Wolfgang Thomas. “Nondeterminism versus Determinism of Finite Automata Over Directed Acyclic Graphs”. In: *Bulletin of the Belgian Mathematical Society* 1 (1994), pp. 285–298.

[111] Arend Rensink. “Canonical Graph Shapes”. In: *Proceedings of ESOP ’04 (European Symposium on Programming)*. Vol. 2986. Lecture Notes in Computer Science. Springer, 2004, pp. 401–415.

[112] Stefan Rieger. “Verification of Pointer Programs”. PhD thesis. RWTH Aachen, 2009.

[113] Neil Robertson and Paul D. Seymour. “Graph minors. I. Excluding a forest”. In: *Journal of Combinatorial Theory, Series B* 35.1 (1983), pp. 39–61.

[114] Neil Robertson and Paul D. Seymour. “Graph minors. II. Algorithmic aspects of tree width”. In: *Journal of Algorithms* 7 (1986), pp. 309–322.

[115] Neil Robertson and Paul D. Seymour. “Graph Minors. III. Planar treewidth”. In: *Journal of Combinatorial Theory, Series B* 36.1 (1984), pp. 49–64.

[116] Neil Robertson and Paul D. Seymour. “Graph minors—a survey”. In: *Surveys in Combinatorics* 103 (1985), pp. 153–171.

[117] Jan J. M. M. Rutten. “Automata and coinduction (an exercise in coalgebra)”. In: *CONCUR ’98 – Concurrency Theory*. Ed. by Davide Sangiorgi and Robert Simone. Vol. 1466. Lecture Notes in Computer Science. Springer, 1998, pp. 194–218.

[118] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Parametric Shape Analysis via 3-Valued Logic”. In: *TOPLAS (ACM Transactions on Programming Languages and Systems)* 24.3 (2002), pp. 217–298.

[119] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2012.

[120] Vladimiro Sassone and Paweł Sobociński. “Reactive Systems Over Cospans”. In: *Proceedings of LICS ’05 (Logic in Computer Science)*. IEEE, 2005, pp. 311–320.

[121] Claude Elwood Shannon. “The synthesis of two-terminal switching circuits”. In: *Bell Systems Technical Journal* 28 (1949), pp. 59–98.

[122] David Soguet. “Génération automatique d’algorithmes linéaires”. PhD thesis. Université Paris-Sud, 2008.

[123] James W. Thatcher and Jesse B. Wright. “Generalized finite automata theory with an application to a decision problem of second-order logic”. In: *Mathematical Systems Theory* 2.1 (1968), pp. 57–81.

[124] Wolfgang Thomas. “Languages, Automata, and Logic”. In: *Handbook of Formal Languages*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Springer, 1997, pp. 389–455.

[125] Boris Avraamovich Trakhtenbrot. “Finite automata and logic of monadic predicates”. In: *Doklady Akademii Nauk SSSR* 140 (1961). (in Russian), pp. 326–329.

[126] Boris Avraamovich Trakhtenbrot. “Finite automata and the logic of one-place predicates”. In: *Siberian Mathematical Journal* 3 (1962). (in Russian). English translation: American Mathematical Society Translations, Series 2 59, 23—55, 1966, pp. 103–131.

[127] Tanguy Urvoy. “Abstract Families of Graphs”. In: *Developments in Language Theory*. Ed. by Masami Ito and Masafumi Toyama. Vol. 2450. Lecture Notes in Computer Science. Springer, 2003, pp. 381–392.

[128] John Whaley. *JavaBDD – Java Library for Manipulating BDDs*. (online). URL: http://javabdd.sourceforge.net/.

[129] Andreas Winter. “Exchanging Graphs with GXL”. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. Springer, 2002, pp. 485–500.

[130] Kurt-Ulrich Witt. “Finite Graph-Acceptors and Regular Graph-Languages”. In: *Information and Control* 50.3 (1981), pp. 242–258.

[131] Angela Wu and Azriel Rosenfeld. “Cellular graph automata”. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Vol. 73. Lecture Notes in Computer Science. Springer, 1979, pp. 464–475.

[132] Martin De Wulf, Laurent Doyen, Thomas Anton Henzinger, and Jean-François Raskin. “Antichains: A New Algorithm for Checking Universality of finite Automata”. In: *Proceedings of CAV 2006 (Conference on Computer Aided Verification)*. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 17–30.

Nomenclature

Category Symbols

$[G]$	The trivial cospan $\emptyset \to G \leftarrow \emptyset$, page 25
$\sqsubseteq_R$	A (well-)quasi-order on a category, page 19
$\equiv_R$	A equivalence (congruence) on a category, page 19
$f \,;\, g$	The composition of two morphisms $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$, page 18
$J \xrightarrow{c_L} G \xleftarrow{c_R} K$	A cospan from J to K, page 21
$J \looparrowright K$	A cospan from J to K, page 21
$A \xrightarrow{e} B$	An epimorphism e between A and B, page 20
$A \rightarrowtail{i} B$	An isomorphism i between A and B, page 20
$A \rightarrowtail{m} B$	A monomorphism m between A and B, page 20
$\mathbf{C}$	A category, page 18
Graph	The category of graphs and graph morphisms, page 25
$cod(f)$	The codomain of the morphism f, page 18
$Cospan(\mathbf{C})$	The cospan category of $\mathbf{C}$, page 21
$dom(f)$	The domain of the morphism f, page 18
id_A	The identity morphisms on A, page 18
$\mathcal{M}(A, B)$	The class of morphisms from A to B, page 18
OLCG	The category of output-linear cospans of graphs, page 25
$\mathbf{OLCG}_n$	The category of output-linear cospans of graphs of size at most n, page 25
Rel	The category of sets and relations, page 18
Set	The category of sets and functions, page 18

Graph Symbols

$\lvert G\rvert$	The size of the graph G, page 24
$\emptyset$	The empty graph, page 24
$G \overset{\rho,m}{\Longrightarrow} H, G \Longrightarrow H$	A derivation of G to H (by applying ρ via the match m), page 27
$L \leftarrow \ell - I - r \rightarrow R$	A transformation rule rewriting the left-hand side L to the right-hand side R via the interface I, page 26
att	The attachment function of a graph, page 24
D_n	The discrete graph with n nodes, page 24
lab	The labeling function of a graph, page 24

Atomic Cospan Symbols

$connect^n_{A,\theta}$	The atomic cospan for the connection of a single hyperedge, page 28
$fuse^n_{i,j}$	The atomic cospan for the fusion of two nodes, page 28
$perm^n_\pi$	The atomic cospan for the permutation of the outer interface, page 28
res^n_k	The atomic cospan for the restriction of the outer interface, page 28
$shift^n$	The atomic cospan for the shift of the outer interface, page 72
$trans^n$	The atomic cospan for the transposition of the first two nodes of the outer interface, page 72
$vertex^n_k$	The atomic cospan for the disjoint union with a single node, page 28

Language Symbols

$\lvert w\rvert$	The length of the word w, page 13
$\leq_{\mathcal{L}}$	The Myhill-Nerode quasi-order relative to $\mathcal{L}$, page 15
$\approx_{\mathcal{L}}$	The syntactical congruence relative to $\mathcal{L}$, page 15
$\emptyset$	The empty language, page 13
$C_{(k)}$	The language of all k-colorable graphs, page 57
$\mathcal{L}$	A language, page 13
$\mathcal{L}(q)$	The language accepted by some state $q \in Q$, page 14
$\mathcal{L}(X)$	The language accepted by a state set $X \subseteq Q$, page 14
$\overline{\mathcal{L}}$	The complement language of $\mathcal{L}$, page 16

$\mathcal{L}_1 \cap \mathcal{L}_2$	The intersection language of $\mathcal{L}_1$ and $\mathcal{L}_2$, page 16
$\mathcal{L}_1 \cup \mathcal{L}_2$	The union language of $\mathcal{L}_1$ and $\mathcal{L}_2$, page 16
$\mathcal{L}_1 \,;\, \mathcal{L}_2$	The concatenation language of $\mathcal{L}_1$ and $\mathcal{L}_2$, page 16
$V_{(k)}$	The language of all graphs containing a vertex cover of size at most k, page 58
ε	The empty word, page 13
Σ	An alphabet, page 13

Finite Automaton Symbols

F	The set of final states of an automaton $\mathcal{A}$, page 14
I	The set of initial states of an automaton $\mathcal{A}$, page 14
Q	The set of states of an automaton $\mathcal{A}$, page 14
δ	The transition function of an automaton $\mathcal{A}$, page 14
$\hat{\delta}$	The extended transition function of an automaton $\mathcal{A}$, page 14
Σ	The input alphabet of an automaton, page 14

Tree Automaton Symbols

$\square_s$	A hole of type s, page 16
$(\mathrm{F}_\sigma)_{\sigma \in S}$	A family of sets of accepting states, page 17
$(\mathrm{I}_\sigma)_{\sigma \in S}$	A family of sets of initial states, page 17
$\mathcal{O}ps$	The alphabet containing the function symbols $vertex^n_k$, res^n_k, $connect^n_{A,\theta}$, $perm^n_\pi$, $join^n$, page 51
$(Q_\sigma)_{\sigma \in S}$	A family of finite sets of states (indexed by S), page 17
$(\Delta_f)_{f \in \Sigma}$	A family of transition functions indexed by function symbols, page 17
Σ	An S-Signature, page 17

Automaton Functor Symbols

$\mathcal{A}_{C_{(k)}}$	The automaton functor for the language of all k-colorable graphs, page 57
$\mathcal{A}_{V_{(k)}}$	The automaton functor for the language of all graphs containing a vertex cover of size at most k, page 58
$\mathcal{A}(c)$	The transition relation of the automaton functor $\mathcal{A}$ for morphism c, page 56

$\mathcal{A}(X)$	The set of states of the automaton functor $\mathcal{A}$ for the object X, page 56
F	The final states of an automaton functor $\mathcal{A}$, page 56
I	The initial states of an automaton functor $\mathcal{A}$, page 56
$\mathcal{L}(\mathcal{A})$	The language accepted by some automaton functor $\mathcal{A}$, page 56

Graph Automaton Symbols

F	The set of final states of a graph automaton, page 73
I	The set of initial states of a graph automaton, page 73
$(Q_i)_{i\leq n}$	The set of states of a graph automaton, page 73
$(\mathcal{S}ig_i)_{i\leq n}$	The input alphabet of a graph automaton, page 73
$(\delta_{i,j})_{i,j\leq n}$	The transition function of a graph automaton, page 73
$\hat{\delta}_{i,j}$	The extended transition function of a graph automaton from Q_i to Q_j, page 73

Logical Symbols

$\bot$	The symbol for the truth value **false**, page 31
$\top$	The symbol for the truth value **true**, page 31
$\models$	The entailment relation, page 85
$\forall\, x_i(\varphi)$	Universal Quantification of x_i, page 32
$\exists\, x_i(\varphi)$	Existential Quantification of x_i, page 32
$\mathbb{B}$	The set of truth values, page 31
$\mathbb{B}^n$	The set of all bit vectors of length n, page 31
$\eta(x_i), [\![\cdot]\!]_\eta$	The (extended) valuation over some set X, page 32
$\varphi[x_i/\psi]$	The substitution of every occurrence of x_i in a formula φ with ψ, page 32

Other Symbols

$\bigsqcap B$	Greatest lower bound of B, page 12
$\bigsqcup B$	Least upper bound of B, page 12
$\lceil B \rceil$	The set of all maximal elements of B, page 12
$\lfloor B \rfloor$	The set of all minimal elements of B, page 12
$[\![a]\!]_\equiv, [\![a]\!]$	The equivalence class of a w. r. t. $\equiv$, page 12

$\lvert\vec{a}\rvert$	The length of a sequence $\vec{a}$, page 11
$\vec{a}[i]$	The i-th element of $\vec{a}$, page 11
$A/\equiv$	The quotient set of A by $\equiv$, page 12
A^*	The set of all finite sequences over a set A, page 11
A^n	The n-ary cartesian product, page 11
$f(\vec{a})$	Extension of a function $f\colon A \to B$ to a sequence $\vec{a} \in A^*$, page 12
$\mathbb{N}$	The set of natural numbers, i. e. $\{0, 1, 2, \ldots\}$, page 11
$\mathbb{N}_k$	The set of the first k natural numbers, i. e. $\{0, 1, 2, \ldots, k-1\}$, page 11
$\wp(A)$	The powerset of a set A, page 11
R^*	Reflexive and transitive closure of R, page 11
R^+	Transitive closure of R, page 11
R^n	n-th power of R, page 11
$\overline{X}$	The complement of a set X, page 14

Index