Improved verification methods for concurrent systems

Improved Verification Methods for Concurrent Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 27 augustus 2009 om 16.00 uur

door

Sebastiaan Cornelis Willem Ploeger

geboren te Gouda

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. J.F. Groote
en
prof.dr.ir. J.J. van Wijk

Copromotor:
dr.ir. T.A.C. Willemse

IPA dissertation series 2009-20

Typeset using LaTeX
Cover design by Bas van Vlijmen
Printed in the Netherlands by the Eindhoven University of Technology Printservice
Published by the Eindhoven University of Technology

The work in this thesis is partially supported by the Netherlands Organisation for Scientific Research (NWO) under grant number 612.065.410 and has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-1926-2

Contents

Preface **ix**

1 Introduction **1**
1.1 Background and motivation 1
1.2 Overview of the thesis 3
1.3 Origin of the contents 4

2 Model Checking and Equivalence Checking **7**
2.1 Introduction 7
2.2 Preliminaries 8
2.2.1 Partitions and relations 8
2.2.2 Lattices and fixed points 10
2.2.3 Graphs 10
2.2.4 Data 11
2.3 Models 11
2.3.1 Explicit representation 12
2.3.2 Implicit representation 15
2.4 Preorders and equivalences 16
2.4.1 Strong variants 17
2.4.2 Weak and branching variants 18
2.4.3 The linear-time–branching-time spectrum 19
2.5 Temporal logics 20
2.5.1 Syntax 21
2.5.2 Semantics 22
2.5.3 Examples 23
2.6 Equation systems 23
2.6.1 Syntax 24
2.6.2 Semantics 25
2.6.3 Solution techniques 27

3 Determinization **29**
3.1 Introduction . . . 29
3.2 Determinization algorithms . . . 30
3.2.1 Subset construction revisited . . . 30
3.2.2 Subset construction using transition sets . . . 33
3.2.3 Closure with language preorder . . . 36
3.2.4 Closure with simulation preorder . . . 37
3.2.5 Compression on state sets . . . 38
3.2.6 Compression on transition sets . . . 39
3.3 Lattice of algorithms . . . 40
3.4 Implementation and experiments . . . 44
3.4.1 Cellular automaton 110 . . . 44
3.4.2 Random automata . . . 47
3.5 Conclusions . . . 50

4 Simulation Equivalence **55**
4.1 Introduction . . . 55
4.2 Preliminaries . . . 56
4.3 The generalized coarsest partition problem . . . 57
4.3.1 The simulation problem as a GCPP . . . 58
4.4 The original GCPP solution . . . 59
4.5 Incorrectness of the operator σ . . . 60
4.6 An auxiliary fixed-point operator . . . 64
4.7 A correct and efficient algorithm . . . 66
4.7.1 The correction of a minor mistake . . . 68
4.7.2 Correctness of PA . . . 69
4.8 Complexity analysis . . . 74
4.8.1 Space complexity . . . 75
4.8.2 Time complexity . . . 79
4.9 Conclusions . . . 80

5 Equivalence Checking for Infinite-State Systems **81**
5.1 Introduction . . . 81
5.2 Preliminaries . . . 83
5.3 Translation for branching bisimilarity . . . 83
5.4 Examples . . . 88
5.4.1 Two buffers and a queue . . . 89
5.4.2 Unbounded queues . . . 92
5.5 Translations for other equivalences . . . 97
5.5.1 Strong bisimilarity . . . 97
5.5.2 Weak bisimilarity . . . 97
5.5.3 Branching similarity . . . 98
5.6 Conclusions . . . 98

6 Instantiation for Equation Systems **101**
6.1 Introduction . . . 101
6.2 Infinite Boolean equation systems . . . 102
6.3 Instantiation on finite domains . . . 105
6.3.1 Instantiation for a single predicate variable . . . 106
6.3.2 Simultaneous instantiation . . . 111
6.4 Instantiation on countable domains . . . 114
6.5 Examples . . . 118
6.5.1 Model-checking infinite-state systems . . . 118
6.5.2 Automatic verification . . . 120
6.6 Conclusions . . . 122

7 Switching Graphs **125**
7.1 Introduction . . . 125
7.2 Preliminaries . . . 127
7.2.1 Switching graphs . . . 127
7.2.2 The 3SAT problem . . . 128
7.2.3 Boolean equation systems . . . 129
7.3 Switching-graph problems . . . 130
7.3.1 Connection problems . . . 130
7.3.2 Disconnection problems . . . 131
7.3.3 Loop problems . . . 133
7.4 Equivalence to the BES problem . . . 138
7.4.1 Reduction from BES to v-parity loop . . . 139
7.4.2 Reduction from v-parity loop to BES . . . 140
7.5 Conclusions . . . 140

8 Conclusions **143**
8.1 Discussion . . . 143
8.2 Future work . . . 144

Bibliography **147**

Summary **159**

Curriculum Vitae **161**

Preface

When Jan Friso Groote asked me, in March 2005, if I happened to know someone who might be interested in one of his open PhD positions, I experienced an instant flashback. About five years earlier, when I was still a freshman at Eindhoven University, I was invited to join a fraternity in a remarkably similar manner ("do you happen to know someone who. . . ?"). As I am quite sure that there is absolutely no connection between my professor and my fraternity, I have now adopted the theory that this is simply the Eindhoven-way of inviting people to certain positions. Practice has yet to prove me wrong.

After some contemplation, I decided to take on the challenge because I enjoyed working with Jan Friso and felt I would be gaining a lot of valuable knowledge and experience by doing a PhD. Looking back now, I am glad to see that my feelings have not deceived me. I am very grateful to Jan Friso for allowing me to determine my own course of research, for his never-ending enthusiasm, support and faith in me during the past four years, and for showing me a pragmatic approach towards getting things done in general and doing research in particular. I am also grateful to Jack van Wijk for giving me the opportunity to join the VoLTS project and providing valuable feedback and support whenever I had questions regarding visualization topics, even when I had stopped doing research in that area.

Continuing on the professional level, I would like to thank two people who have been important tutors to me. First of all, I am greatly indebted to Rob van Glabbeek who showed me that it is quite rewarding to take the time and do research thoroughly, as it ultimately leads to deeper understanding and higher-quality results. Rob, I shall never forget the inspiring thoughts that you shared with me and the valuable lessons that you taught me, both on a professional and on a personal level. Secondly, I am very grateful to Tim Willemse who, apart from being a pleasant colleague and fun room-mate, has been an excellent sparring partner for me and gave me mental support and guidance whenever I needed it. Tim, it has been a great pleasure and experience for me to work with you.

Though times of joy and happiness certainly prevailed, I occasionally went through times of hopelessness and despair, which seem to be inevitable for many PhD candidates. Through it all, the good times and the bad, several people have given me their unconditional love and support, for which I am very thankful. The most important of these is Hanneke, my love, who never stopped believing in me,

even at times when I myself did. Hanneke, your endless love, patience and support have meant a lot to me and I believe we came out stronger than ever. I love you! Furthermore, I am fortunate to have just about the coolest parents in the world. Ineke and Klaas, mom and dad, your limitless supply of positive energy has lifted my spirit time and time again, and has always encouraged me to keep on going, regardless of what I was doing. I am also grateful to my brother and sister-in-law, Tom and Laura, for their support and warm and inspiring company at many occasions. Thank you all very much!

Special thanks go to Erik de Vink who, apart from being a very kind and helpful mentor to me, also took the effort of reviewing a draft version of this thesis, which clearly improved its quality. Furthermore, I am grateful to Bas van Vlijmen who deeply impressed me with his artistic skills by creating not just one, but a whole series of amazing cover designs within a short period of time.

Finally, I thank all of my co-workers, relatives and friends, in particular my *paranimfen* Daniel and Eric, for their kind support over the past four years, and for making my life more pleasant, interesting and fun in many different ways. You are too numerous to be mentioned here explicitly without my running the risk of unintentionally omitting someone, which I'd rather not. Thanks to you all!

Bas Ploeger
July 2009

Chapter 1

Introduction

1.1 Background and motivation

An *algorithm* is a series of steps or instructions that can be followed in order to determine the answer to a particular question. The construction of algorithms as methods of computation has a long history, going back to the ancient Greeks: one of the oldest known algorithms is Euclid's algorithm for computing the greatest common divisor of two natural numbers. The study of algorithms and computation became more relevant than ever before with the advent of the modern computer in the 1940s and its rapid development over the next decades. Mathematical models of computation, like the *Turing machine* [122], were developed and further refined to allow for sound formal reasoning about computers, algorithms and the nature of computing. Also, formal techniques were developed for the derivation of an algorithm or computer program such that its correctness is guaranteed. These methods of *structured programming* and *programming by derivation* [35], elevated the art of computer programming from a mathematical diversion to a scientific discipline. Along with many other pioneering works, they contributed to the establishment of *computer science* as an academic research field of its own.

In those early days, computer programs were predominantly viewed as *input-output functions*, much like the algorithms that they implemented. Given a certain input, a program performs a predefined sequence of steps that should always terminate and, upon termination, it delivers some output. Indeed, this is still the way in which many programs, most notably *command-line tools*, work today. We call such programs *sequential programs*. A sequential program is typically *monolithic* by design, meaning that the program conceptually consists of a single entity that performs the entire computation.

In the 1980s a completely different view on computer programs gained in popularity: a computer program as a collection of *processes* that run *concurrently* and can *interact* with each other and the environment to perform complex tasks.

Each of the individual processes may be fully sequential or may itself consist of concurrently running processes. Moreover, the program is often assumed to be running indefinitely, without ever terminating. We call such programs *concurrent programs*, or *concurrent systems* to express the fact that they are in general collections of programs. They are *modular* by design, in the sense that multiple processes (or components, or agents), each having its own specific task, cooperate in order to perform the required computation. Today the software for many complex systems is built in this fashion, in particular the software that is embedded into devices like cellular phones, television sets, copying machines, cars and aeroplanes.

The fundamental difference with sequential programs is the presence of concurrency and interaction. Similar to the way in which models of computation (like the Turing machine) were developed to study sequential computations, models of concurrency were proposed to allow for formal reasoning about concurrent computations that involve interaction between the processes. Examples of such models include Petri nets [107, 108] and process algebras, like the *Calculus of Communicating Systems* (CCS) [99], *Communicating Sequential Processes* (CSP) [74], the *Algebra of Communicating Processes* (ACP) [8, 9] and the π-calculus [100]. In these languages a concurrent system can be specified, so that its *behaviour*, being the collection of all computations and interactions that it can perform, can be studied. Some of these languages have been extended for various purposes, *e.g.* for studying computations that involve *real-time* or *probabilistic* aspects.

In addition, a number of techniques has been developed for proving the correctness of concurrent systems, which is usually called *verification* in this context. Among the popular verification techniques are *model checking*, *equivalence checking* and *theorem proving*. The ultimate goal of these techniques is to prove that a concurrent system will always perform the right computations, or never perform wrong computations. In many critical applications, like the on-board software of an aeroplane, a software failure is undesirable. By formally verifying such systems, their reliability can be improved. The verification techniques that we focus on in this thesis are model checking and equivalence checking.

In model checking, a model of a concurrent system is constructed together with a desired property that this system should have. It is then checked whether the model satisfies the property. The model is represented by a finite state machine and the property is formulated in a temporal logic (see *e.g.* [109]). The technique originated from works by Clarke and Emerson [23], and Queille and Sifakis [110]. For finite systems, model checking is effectively decidable, meaning that it can be done fully automatically by a computer program. Indeed, the development of automated model checkers – like EMC [23, 24], CESAR [110], SPIN [77, 78], CWB [27], SMV [96] and CADP [47, 50] – played an important role in the successful application of the technique to real-life sequential systems like hardware circuit designs. However, limitations of the approach were encountered when it was applied to more complex, concurrent software systems. The parallelism between the processes of such systems leads to a combinatorial blow-up of the number of states

that the system can be in, yielding state spaces that are too large to handle for traditional model checkers. This problem is known as the *state (space) explosion problem* and a range of techniques has been developed to address it, including: symbolic representation of state spaces using binary decision diagrams [18, 96], partial order reduction [57, 106, 124], abstract interpretation [32], symmetry reduction [25, 44] and compositional reasoning [1, 101]. Some of these techniques can also be used for model checking infinite-state systems [37, 87, 97] in addition to other approaches, *e.g.* [12, 13, 42].

Equivalence checking is a verification technique that operates on two models of a system: one describing its desired behaviour and the other describing its actual behaviour. It is then checked whether these models are, in a certain sense, behaviourally equivalent. A related technique is *refinement checking*, where it is checked whether one model is behaviourally 'contained' in the other model, meaning that any computation that one model can perform, can also be performed by the other model. The history of equivalence checking can be traced back to the decidability of language equivalence on finite-state automata [102]. Its decidability and complexity has been studied for a variety of computational models. Later, other equivalences were developed for comparing concurrent systems. They serve as the underlying semantics of process algebras: trace semantics [73] and failures semantics [16] for CSP, and (bi)simulation semantics [99, 105] for CCS and ACP. A comparison of these and other equivalences is presented in [55]. For finite-state processes, all of these equivalences are decidable: trace equivalence and failure equivalence require exponential time, while bisimulation equivalence is decidable in polynomial time [83, 104]. Simulation refinement and equivalence are also polynomially decidable [11, 72]. Automated equivalence and refinement checking is provided by a number of tools, including CWB [27], FDR [112] and CADP [47, 50]. For infinite-state systems many equivalences are undecidable, though bisimulation equivalence is still decidable for particular classes of processes [3, 20, 21].

The goal of the work in this thesis is to find improved verification methods for large and infinite-state concurrent systems. This is motivated by the fact that we found existing techniques to be inadequate for the verification of certain large or infinite-state systems. We limit ourselves to pure discrete-event systems, *i.e.* systems that perform computations solely by taking discrete steps, and we do not consider systems that involve real-time, stochastic or probabilistic aspects.

1.2 Overview of the thesis

The thesis has the following structure:

- Chapter 2 defines the relevant concepts for the remainder of the work. Model checking and equivalence checking are described in more detail. We introduce models for concurrent systems, equivalence and refinement relations on such models, a temporal logic in which properties can be expressed, and equation systems in which model-checking problems can be represented.

- Chapter 3 presents five new algorithms for the determinization of automata. Determinization plays an important role in equivalence checking for certain equivalences. The algorithms are based on a standard determinization algorithm and aim to reduce its average-case space requirements and the size of its output. They are implemented and experimentally evaluated.
- Chapter 4 corrects a space-efficient algorithm for checking simulation equivalence and refinement on models of concurrent systems. We show that the theory surrounding the original algorithm was flawed, repair the algorithm, prove its correctness, and analyse its time and space complexities to show that they are unaffected by our corrections.
- Chapter 5 shows how equivalence-checking problems on infinite-state systems can be finitely represented in an equation system. Checking for equivalence then amounts to solving the equation system. This shows that such equation systems can not only be used for model checking but also for equivalence checking. Hence, these equation systems allow for different types of verification problems to be studied within a single, generic framework. We illustrate our approach by two examples.
- Chapter 6 shows how such equation systems can be transformed to simpler ones, for which a solution may be found more easily. More specifically, this technique, called *instantiation*, aims to remove data from an equation system. In an extreme case all data can be removed, by which solving the equation system becomes decidable and can be done fully automatically. We illustrate the efficacy of our technique by two examples and automated experiments.
- Chapter 7 proposes an extension of ordinary graphs, called *switching graphs*, as a novel formalism in which combinatorial problems can be represented and studied. In particular, we show that solving equation systems – and therefore model checking – corresponds with one of the presented switching-graph problems and we investigate the complexity of several closely related problems. Some of these turn out to be polynomial, while others are shown to be NP-complete.

Hence, chapters 3–5 are concerned with equivalence checking. In chapters 6 and 7 we study equation systems, that can be used for both model checking and equivalence checking. All of these chapters depend on chapter 2 for the definitions, notations and theoretical results that it contains. Apart from this, every chapter can be read in isolation.

1.3 Origin of the contents

This thesis is based on the following research papers:

[1] T. Chen, B. Ploeger, J. van de Pol and T.A.C. Willemse (2007): *Equivalence Checking for Infinite Systems using Parameterized Boolean Equation Systems.* In L. Caires and V. Thudichum Vasconcelos, editors: Proc. 18th

International Conference on Concurrency Theory (CONCUR 2007), LNCS 4703, pp. 120–135. Springer.

[2] A. van Dam, B. Ploeger and T.A.C. Willemse (2008): *Instantiation for Parameterised Boolean Equation Systems.* In J.S. Fitzgerald, A.E. Haxthausen and H. Yenigun, editors: Proc. 5th International Colloquium on Theoretical Aspects of Computing (ICTAC 2008), LNCS 5160, pp. 440–454. Springer.

[3] R.J. van Glabbeek and B. Ploeger (2008): *Correcting a Space-Efficient Simulation Algorithm.* In A. Gupta and S. Malik, editors: Proc. 20th International Conference on Computer Aided Verification (CAV 2008), LNCS 5123, pp. 517–529. Springer.

[4] R.J. van Glabbeek and B. Ploeger (2008): *Five Determinisation Algorithms.* In O.H. Ibarra and B. Ravikumar, editors: Proc. 13th International Conference on Implementation and Application of Automata (CIAA 2008), LNCS 5148, pp. 161–170. Springer.

[5] J.F. Groote and B. Ploeger (2008): *Switching Graphs.* In V. Halava and I. Potapov, editors: Proc. 2nd Workshop on Reachability Problems in Computational Models (RP 2008), ENTCS 223, pp. 119–135. Elsevier.

Chapters 3, 4, 5, 6 and 7 are extended and updated versions of [4], [3], [1], [2] and [5], respectively. Chapter 2 consists of both newly written material and excerpts from the above publications.

Chapter 2

Model Checking and Equivalence Checking

2.1 Introduction

Model checking is a technique for the verification of concurrent systems that has its roots in works by Clarke and Emerson [23] and Queille and Sifakis [110] from the early 1980s. We refer to the literature for more information on its origins [22, 39] and for comprehensive introductions to the field [5, 26]. In their book [5], Baier and Katoen define model checking as follows:

> *Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*

In section 2.3 we show how models can be represented and in section 2.5 we introduce a *temporal logic* in which properties can be formulated. Though the definition above is in accordance with the original definition of model checking, it is too narrow for the purpose of this thesis, in two respects.

First of all, model checking need not necessarily be automated. It can equally be done completely by hand or semi-automatically, *i.e.* using manipulations by both computer programs and human beings. Automation is particularly useful for efficiently executing repetitive and clearly defined tasks on large data sets. This type of task is common in model checking. More automation then allows for more complex models or properties to be checked. However, not all tasks can be automated. In model checking it depends on the *decidability* and *complexity* of the modelling languages and logics that are used, whether the process can be fully automated. When the models and properties are expressed in languages and logics that are too rich, model checking can become undecidable or practically infeasible, in which case human ingenuity can help to solve the problem.

This brings us to the second point: a model need not necessarily be finite-state. It can have infinitely many states, in which case model checking is generally undecidable. Infinite-state models cannot be finitely represented by explicitly enumerating all possible states. Finite representations of such models are therefore *symbolic*: they use data sorts and algebraic constructs to specify the set of states implicitly. Similarly, the problem of checking a property on an infinite model can often be finitely and symbolically represented in *equation systems*, which are sequences of fixed-point equations (see section 2.6). Such a representation can be manipulated using both automatic and manual techniques in order to solve the model-checking problem that it encodes.

An alternative way of establishing desirable properties of a model, is by showing that it is *behaviourally related* to another model that has these desirable properties. Depending on the type of relation that is desired, this verification technique is called *refinement* or *equivalence checking*. In section 2.4 we introduce a number of relations between models and show that some are more strict than others, meaning that the models should resemble each other more closely in order for them to be related. Equivalence checking can also be used to reduce the number of states in a model by merging equivalent states. This is particularly profitable when dealing with large models of which the analysis would otherwise consume too much space or time.

Indeed, the most challenging task when applying automated model checking in practice is to conquer the so-called *state explosion problem*: models can easily become very large, on the order of millions or billions of states. This is usually due to parallelism between the processes of the modelled system. When models are too large to fit in a computer's main memory, automated model checking quickly breaks down. In such cases attempts have to be made to reduce the model, for which a number of techniques can be applied, that we do not explore further here.

Before we define the concepts that are related specifically to model checking and equivalence checking, we first introduce some basic mathematical concepts and notations.

2.2 Preliminaries

2.2.1 Partitions and relations

For any set S, the *powerset* of S, denoted $\wp(S)$, is the set of all subsets of S. A *partition* over S is a set $\Sigma \subseteq \wp(S)$ of non-empty, mutually disjoint subsets of S that together cover S, *i.e.* $\bigcup \Sigma = S$ and $\forall \alpha \in \Sigma \,.\, \alpha \neq \emptyset \wedge \forall \beta \in \Sigma \,.\, \alpha \neq \beta \Longrightarrow \alpha \cap \beta = \emptyset$. An element α of a partition Σ is called a *block* and for any $s \in S$ we denote by $[s]_\Sigma$ the block $\alpha \in \Sigma$ such that $s \in \alpha$. Given two partitions Σ and Π we say Π *is finer than* Σ iff for every $\alpha \in \Pi$ there exists an $\alpha' \in \Sigma$ such that $\alpha \subseteq \alpha'$.

For given sets S and T, a *relation* R on S and T is a set of ordered pairs of elements from S and T, *i.e.* $R \subseteq S \times T$. We may use infix notation $s \; R \; t$ to denote

the fact that $(s,t) \in R$. For any $s \in S$ we denote by $R(s)$ the set $\{t \in T \mid s\ R\ t\}$. We say that R is *total* if $R(s) \neq \emptyset$ for all $s \in S$; otherwise we say R is *partial*. The *inverse* R^{-1} of R is the relation that contains the reverse of every pair in R:

$$R^{-1} := \{(t,s) \mid (s,t) \in R\}.$$

A relation on a single set S is a relation on S and S. Let S be a set, R be a relation on S, and R^+ be as defined below. Then R is:

- *reflexive* iff $\forall s \in S\,.\,(s,s) \in R$;
- *symmetric* iff $\forall s,t \in S\,.\,(s,t) \in R \implies (t,s) \in R$;
- *anti-symmetric* iff $\forall s,t \in S\,.\,(s,t) \in R \wedge (t,s) \in R \implies s = t$;
- *acyclic* iff $\forall s,t \in S\,.\,(s,t) \in R^+ \wedge (t,s) \in R^+ \implies s = t$ [1];
- *transitive* iff $\forall s,t,u \in S\,.\,(s,t) \in R \wedge (t,u) \in R \implies (s,u) \in R$.

The *identity relation* $\mathcal{I}$ on S is the smallest reflexive relation on S. The *transitive closure* R^+ of R is the smallest transitive relation that subsumes R. The *reflexive and transitive closure* R^* of R is the smallest reflexive relation that subsumes R^+. Formally, for $i \geq 0$ let R^i be defined recursively as follows:

$$\begin{aligned} R^0 &:= \{(s,s) \mid s \in S\} \\ R^1 &:= R \\ R^{i+2} &:= \{(s,u) \mid \exists t \in S\,.\,(s,t) \in R^{i+1} \wedge (t,u) \in R^{i+1}\}\,. \end{aligned}$$

Then we define:

$$\mathcal{I} := R^0 \qquad R^+ := \textstyle\bigcup_{i>0} R^i \qquad R^* := \textstyle\bigcup_{i\geq 0} R^i\,.$$

Furthermore, we define the following common types of relations:

- a *preorder* is a reflexive and transitive relation;
- a *partial order* is an anti-symmetric preorder;
- an *equivalence* is a symmetric preorder.

For any preorder $\sqsubseteq$, the *equivalence induced by* $\sqsubseteq$ is the equivalence $\equiv$ defined as $\equiv := \sqsubseteq \cap \sqsubseteq^{-1}$. It is not hard to see that $\equiv$ is an equivalence indeed.

For any sets S and T, a *function* f from S to T, denoted $f : S \to T$, is a relation on S and T such that for every $s \in S$ there is precisely one $t \in T$ such that $(s,t) \in f$. Hence, $f(s)$ is a single element of T. In this context, S is called the *domain* and T is called the *codomain* of f. The set of all functions from S to T is denoted by T^S. Given a function $f : S \to T$, $s \in S$ and $t \in T$, we write $f[s \mapsto t]$ to denote the function f in which s is mapped to t, *i.e.* $f[s \mapsto t](s) = t$ and $f[s \mapsto t](s') = f(s')$ for all $s' \in S \setminus \{s\}$. We use λ-notation for function abstraction, *e.g.* $\lambda x\,.\,f(x)$ is the function that maps x to $f(x)$.

[1] Acyclicity is sometimes defined as $(s,t) \in R^+ \implies (t,s) \notin R^+$ which also excludes reflexive pairs. In this thesis, we need the weaker notion defined here to allow for relations to be both reflexive and acyclic.

2.2.2 Lattices and fixed points

A *partially ordered set* or *poset* $(S, \preceq)$ is a set S with a partial order $\preceq$ on S. Let $(S, \preceq)$ be a poset. For any $X \subseteq S$, the set of *upper bounds* of X, $ub(X)$, is defined as $ub(X) := \{u \in S \mid \forall x \in X \,.\, x \preceq u\}$. If it exists, the *least upper bound* or *supremum* of X, denoted $\bigsqcup X$, is the unique $u \in ub(X)$ such that $u \preceq v$ for all $v \in ub(X)$. For any $x, y \in S$ we use the shorthand $x \sqcup y$ to denote $\bigsqcup\{x, y\}$. Similarly, the set of *lower bounds* of X is defined as $lb(X) := \{u \in S \mid \forall x \in X \,.\, u \preceq x\}$; the *greatest lower bound* or *infimum* of X, $\bigsqcap X$, is the unique $u \in lb(X)$ such that $v \preceq u$ for all $v \in lb(X)$; and $x \sqcap y$ denotes $\bigsqcap\{x, y\}$ for all $x, y \in S$.

Now, $(S, \preceq)$ is a *lattice* if $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in S$. Moreover, $(S, \preceq)$ is a *complete lattice* if $\bigsqcup X$ and $\bigsqcap X$ exist for all $X \subseteq S$. It is well known that for any set S the poset $(\wp(S), \subseteq)$ is a complete lattice.

A function $f : S \to S$ is *monotonic* if $x \preceq y$ implies $f(x) \preceq f(y)$ for all $x, y \in S$. Suppose $(S, \preceq)$ is a complete lattice and let $f : S \to S$ be a monotonic function. The set of *fixed points* (or *fixpoints*) of f is defined as $fix(f) := \{x \in S \mid f(x) = x\}$. Tarski's famous fixpoint theorem [121] states that the poset $(fix(f), \preceq)$ is a complete lattice. In particular, this implies the existence of a *least fixed point*, μf, and a *greatest fixed point*, νf, of f, which are defined as follows:

$$\mu f := \bigsqcap\{x \in S \mid x = f(x)\}$$
$$\nu f := \bigsqcup\{x \in S \mid x = f(x)\}.$$

Moreover, $=$ may be replaced by $\succeq$ in the definition of μf and by $\preceq$ in that of νf. In the sequel, we use σ to denote either μ or ν and we abbreviate $\sigma(\lambda x \,.\, f(x))$ to $\sigma x \,.\, f(x)$ for any function f.

2.2.3 Graphs

A *directed graph* (or simply *graph*) is a tuple $(V, \to)$ where V is a set of *vertices* and $\to \subseteq V \times V$ is a relation on V that is the set of directed *edges*. A *rooted graph* $(V, \to, r)$ is a graph $(V, \to)$ in which vertex $r \in V$ is designated as the *root vertex*.

Given a fixed set S, an *edge-labelled graph* $(V, \to)$ is a graph in which an element of S is associated with every edge, *i.e.* $\to \subseteq V \times S \times V$. In this context, the set S is usually called an *alphabet*. Let S^* denote the set of all finite sequences of elements of S and $\varepsilon \in S^*$ denote the empty sequence. For any $v, w \in V$ and $\alpha = \alpha_1 \cdots \alpha_n \in S^*$ for some $\alpha_1, \ldots, \alpha_n \in S$ and $n \geq 0$, we denote by $v \xrightarrow{\alpha} w$ the fact that there exist $v_0, \ldots, v_n \in V$ such that $v_0 = v$, $v_n = w$ and $(v_i, \alpha_{i+1}, v_{i+1}) \in \to$ for all $i, 0 \leq i < n$. Similarly, a *vertex-labelled graph* $(V, \to, L)$ is a graph in which an element of S is associated with every vertex via the *labelling function* $L : V \to S$. A graph can be depicted in the following way:

- a vertex v is represented by a circle $\circ$, optionally containing the name: $\textcircled{v}$;
- a (labelled) edge is represented by a (labelled) arrow, *e.g.* $\circ \xrightarrow{a} \circ$;
- the root vertex of a rooted graph is indicated by a short incoming arrow: $\overset{\downarrow}{\circ}$.

Vertex labels are only drawn in chapter 7. There, a vertex label is placed within the circle and a vertex name is placed outside of it, because the labels are of higher importance to the theory.

2.2.4 Data

In some theoretical considerations in this thesis, abstract data sorts are used to represent data. We have a set $\mathcal{D}$ of *data variables* and we assume that there is some data language that is sufficiently rich to denote all relevant *data terms*, like $3+d_1 \leq d_2$. We assume that every countable sort has a collection of *basic elements* to which every term can be rewritten. For a sort D, we write $v \in D$ to denote that v is a basic element of D and we use set notation to list the basic elements of D, *e.g.* $D = \{v_1, \ldots, v_n\}$. For any terms t and u, and variable d we denote by $t[u/d]$ the *syntactic substitution* of u for d in t.

With every sort D we associate a semantic set $\mathbb{D}$ such that every syntactic term of sort D can be mapped to the element of $\mathbb{D}$ it represents. The set of basic elements of a countable sort D is isomorphic to the semantic set $\mathbb{D}$. For a closed term t of sort D (denoted $t{:}D$), we assume an interpretation function $[\![t]\!]$ that maps t to the data element of $\mathbb{D}$ it represents. For open terms we use a *data environment* $\varepsilon : \mathcal{D} \to \mathbb{D}$ that maps each variable from $\mathcal{D}$ to a data element from the right set $\mathbb{D}$. The interpretation of an open term t, denoted as $[\![t]\!]\varepsilon$ is given by $\varepsilon(t)$ where ε is extended to terms in the standard way.

We assume the existence of a sort $B = \{\top, \bot\}$ representing the Booleans $\mathbb{B}$, and a sort $N = \{0, 1, \ldots\}$ representing the natural numbers $\mathbb{N}$. For these sorts, we assume the usual operators are available and we do not write constants or operators in the syntactic domain any different from their semantic counterparts. For example, we have $\mathbb{B} = \{\top, \bot\}$, and the syntactic operator $\_ \wedge \_ {:} B \times B \to B$ corresponds to the usual semantic conjunction $\_ \wedge \_ {:} \mathbb{B} \times \mathbb{B} \to \mathbb{B}$.

2.3 Models

We are interested in modelling concurrent systems, in which a number of processes operate in parallel. At any time, every process is in a certain *state* from which it may execute an *action* to reach a new state. Several actions may be possible in a state, in which case the action to be executed is chosen *nondeterministically*. Hence, the operation of every process consists in the *sequential* execution of actions, where sequential means that a process cannot execute more than one action at the same time. The execution of an action itself is *atomic* in the sense that it cannot be interrupted. Additionally, in some states a process may have the option of *successful termination* to indicate that its computation has finished.

Concurrency between the processes is interpreted in the following way. In principle, the behaviour of a concurrent system consists of all possible interleavings of the action sequences that its individual processes can perform. However, some

specific pairs of actions, called *communicating actions*, can be executed simultaneously by two separate processes. This allows for *synchronous communication* between the processes, whereby messages are sent and received at the same time. Except for the communicating actions, no two actions can be executed simultaneously. Because the simultaneous execution of two communicating actions must be viewed as a single atomic step, a concurrent system is again a process that executes actions in a purely sequential fashion. Finally, the state of a concurrent system is the combination of the states of its constituent processes, and a concurrent system can terminate successfully only in states where every process can do so.

For modelling the behaviour of concurrent systems, we fix the set Act of actions that a system can perform. Furthermore, we fix a non-empty set AP of *atomic propositions* that can be either valid or invalid in a system's state. The behaviour of a concurrent system can be represented explicitly or implicitly. In both cases, we use the term *model* to refer to the representation of that behaviour.

2.3.1 Explicit representation

In explicit representations, every state and possible action in the modelled system is mapped one-to-one to an entity in the model. The most well-known and common explicit representation is the *transition system*.

Definition 2.1. A *transition system* is a rooted, edge- and vertex-labelled graph $(S, \rightarrow, i, L)$, where $\rightarrow \subseteq S \times Act \times S$, $i \in V$ is the root vertex and $L : S \rightarrow \wp(AP)$.

A transition system models the behaviour of a concurrent system in the following way. The vertices in S are the states of the concurrent system and the root vertex i is its initial state. The labelled edges in $\rightarrow$ are called *transitions* in this context. A transition $s \xrightarrow{a} t$ indicates that when the system is in state s it can execute the action a after which it ends up in the state t. A set of atomic propositions from AP is associated with every state via the labelling function L. For every $s \in S$, $L(s)$ is the set of propositions that hold when the system is in state s; all other propositions do not hold there.

We define $\tau \in Act$ to be a special action, called the *unobservable action*; any action from the set $Act \setminus \{\tau\}$ is called an *observable action*. The τ-action cannot be executed by any real system. It can be used in a model to represent any action that should be considered internal to the modelled system. By renaming all such actions to τ, internal behaviour can be abstracted from the model in order to reduce its complexity. For any transition system $(S, \rightarrow, i, L)$ and $s, t \in S$ we write $s \Rightarrow t$ if there is a sequence of τ-transitions from s to t, *i.e.* $\Rightarrow := (\xrightarrow{\tau})^*$.

Let $\mathbb{T}$ denote the domain of transition systems. For any $M \in \mathbb{T}$, the elements of M are referred to using subscripts, *e.g.* S_M is the set of states of M. The subscript is omitted from the relation $\rightarrow_M$ unless it is necessary to avoid ambiguity. More specific subdomains of $\mathbb{T}$ can be obtained by applying any of the following restrictions.

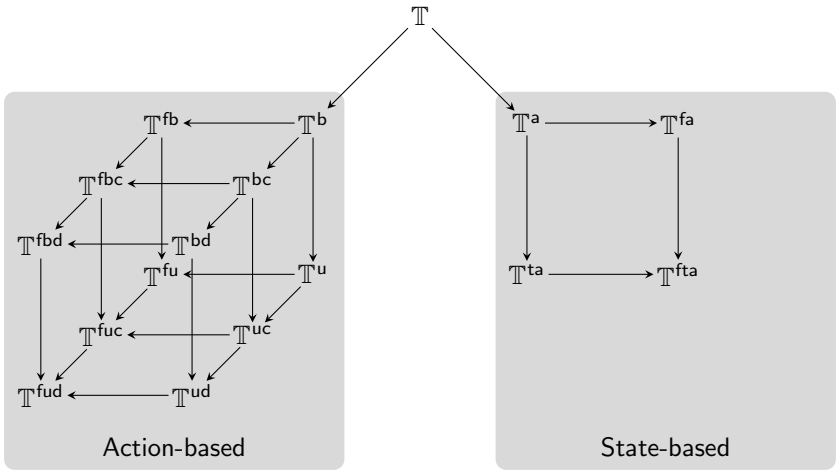


Figure 2.1. An overview of possible domains of transition systems, derived from the general domain $\mathbb{T}$ by applying various restrictions. A coarse distinction can be made between action-based and state-based domains.

Definition 2.2. A transition system $(S, \rightarrow, i, L)$ is called:

1. *finite* if S, $\rightarrow$ and $L(s)$ are finite for all $s \in S$;
2. *total* if $\forall s \in S \,.\, \exists a \in Act, t \in S \,.\, s \xrightarrow{a} t$;
3. *abstract* if $\rightarrow \subseteq S \times \{\tau\} \times S$;
4. *concrete* if $\rightarrow \subseteq S \times (Act \setminus \{\tau\}) \times S$;
5. *deterministic* if it is concrete and $\forall s \in S, a \in Act \,.\, |\{t \in S \mid s \xrightarrow{a} t\}| = 1$; [2]
6. *bipolar* if $\forall s \in S \,.\, L(s) = \emptyset \vee L(s) = AP$;
7. *unipolar* if $\forall s \in S \,.\, L(s) = AP$.

A restricted domain is denoted by appending the first letter of the restriction to the superscript of $\mathbb{T}$, *e.g.* $\mathbb{T}^{bd}$ is the bipolar deterministic domain. An overview of some commonly encountered domains is shown in figure 2.1. More restrictions and subdomains of $\mathbb{T}$ can be defined, but we only consider the ones that are relevant for this thesis. An arrow from domain A to domain B indicates that A subsumes B. The validity of the arrows can be easily verified. The analysis of transition systems typically focuses on either the actions that occur on the transitions, or the labels that are assigned to the states. Hence, there is a natural distinction between *action-based* and *state-based* domains.

[2] Another notion of determinism that is common for concurrent systems, merely demands that $|\{t \in S \mid s \xrightarrow{a} t\}| \leq 1$. In this thesis we shall only use the stronger notion defined here, which has its roots in the classical theory of finite-state acceptors (see also chapter 3).

Action-based domains

Action-based domains are useful when the main focus of the analysis is on the actions that are performed by the modelled concurrent system. In these domains, the state labels are (almost) completely neglected: the labelling function L is either very simple (bipolar domains) or fully trivial (unipolar domains).

The bipolar domains are used for modelling successful termination of a system. The two possible labels that a state s can have, are interpreted as follows:

- if $L(s) = \emptyset$ then s is a *non-final* or *non-accepting* state;
- if $L(s) = AP$ then s is a *final* or *accepting* state.

The system has the option to terminate successfully whenever it is in a final state. As the specific atomic propositions are irrelevant for this purpose, L is usually replaced by a set F of final states, *i.e.* $F = \{s \in S \mid L(s) = AP\}$. Hence, we normally write $(S, \rightarrow, i, F)$ instead of $(S, \rightarrow, i, L)$. In graphical representations of bipolar transition systems, a final state is depicted by a doubly lined circle ◎.

The unipolar domains are used when successful termination plays no role. It is useful to assume that all states are final for reasons that become apparent in section 2.4. Hence, $L(s) = AP$ for all states s, so that $F = S$. As the state labels and the set F have become irrelevant, L and F are usually omitted in practice.

In the diagram of action-based models in figure 2.1, a downwards arrow indicates a restriction from bipolar to unipolar domains. A leftwards arrow indicates a restriction to finite domains. Finally, the diagonal dimension consists of two tiers: the first one indicates a restriction to concrete domains, and the second one indicates a further restriction to deterministic domains.

For action-based analysis of concurrent systems, successful termination is often irrelevant and commonly used domains are the unipolar ($\mathbb{T}^{\mathsf{u}}$) and unipolar concrete ($\mathbb{T}^{\mathsf{uc}}$) domains. These are usually called *labelled transition systems* with or without silent steps, *i.e.* τ, respectively. The bipolar domains are traditionally used for modelling computations of which successful termination is an important aspect. In particular, these include the classical *nondeterministic* and *deterministic finite automaton* which correspond with the finite bipolar concrete ($\mathbb{T}^{\mathsf{fbc}}$) and finite bipolar deterministic[3] ($\mathbb{T}^{\mathsf{fbd}}$) domains, respectively.

State-based domains

In state-based domains, the actions occurring on the transitions are neglected and the focus is on the state labels. The action abstraction is reflected in the model by assuming that all transitions are τ-transitions (abstract domain). These τ-labels are usually omitted, *i.e.* $\rightarrow$ becomes a binary relation on states: $\rightarrow \subseteq S \times S$. For technical reasons, it is often assumed that the system can always perform an action, *i.e.* every state has at least one outgoing transition (total domain).

[3]Note, however, that the notion of determinism is usually more strict in this context: it requires that for every state s and observable action a there is *precisely* one transition $s \xrightarrow{a} t$.

In the diagram of state-based domains in figure 2.1, a restriction to finite domains is shown by a rightward arrow and a restriction to total domains is shown by a downward arrow. In state-based model checking, models are commonly represented by *Kripke structures* (see *e.g.* [26]), which correspond with the finite total abstract domain ($\mathbb{T}^{\mathsf{fta}}$). Sometimes the totality restriction is not necessary and is lifted for generality, yielding the finite abstract domain ($\mathbb{T}^{\mathsf{fa}}$).

Domains used in this thesis

The action-based and state-based domains are equi-expressive and can be used interchangeably (see *e.g.* [34]). Though useful in practice, the general domain $\mathbb{T}$ is too cumbersome for theoretical considerations: it complicates definitions and proofs without adding generality or expressivity over action-based or state-based domains. By default, we work in the context of action-based domains throughout this thesis. This is the case for the remainder of the current chapter, chapter 3 ($\mathbb{T}^{\mathsf{fbc}}$ and $\mathbb{T}^{\mathsf{fbd}}$) and chapter 5 ($\mathbb{T}^{\mathsf{u}}$). An exception is chapter 4 where vertex-labelled graphs ($\mathbb{T}^{\mathsf{fa}}$) are used to allow for easier comparison with the work on which it is based. No models of concurrent systems are used in the theory of chapters 6 and 7.

2.3.2 Implicit representation

Implicit representations allow for a more concise and manageable specification of a system's behaviour than explicit representations. The idea is that a transition system is described implicitly and symbolically in a higher-level specification language. The implicit representation we present here is inspired by the process-algebraic specification language mCRL2 [62, 63] that is based on μCRL [65, 66] and ACP [4, 8, 9]. An mCRL2 specification typically contains a number of sequential processes that are composed in parallel to obtain the entire concurrent system. Apart from parallelism, some other operators may be used, for instance to establish and enforce communication between the processes. We do not deal with these operators here. A central notion in both μCRL and mCRL2 is the *linear process*, which is a process in restricted form. In particular, it contains no parallelism or communication operators; they have been removed in favour of nondeterministic choice and sequential composition. In many cases an mCRL2 specification can be translated automatically to a single linear process by *linearization* [123].

Let $\mathcal{A}$ be a finite set of parameterized actions, *i.e.* every $a \in \mathcal{A}$ can carry a data parameter of some possibly empty data sort D_a. In particular, we have $\tau \in \mathcal{A}$ and D_τ is empty. For brevity, whenever we quantify over all possible actions $a(d)$ for some $a \in \mathcal{A}$ and $d \in D_a$, we also mean to include the action constants a for which D_a is empty.

Definition 2.3. A *linear process equation* (LPE) is an equation of the following

form, for any data sort D:

$$P(d:D) = \sum_{a \in \mathcal{A}} \sum_{e_a : E_a} h_a(d, e_a) \rightarrow a(f_a(d, e_a)) \cdot P(g_a(d, e_a))$$

where for every $a \in \mathcal{A}$, E_a is a data sort, $h_a : D \times E_a \rightarrow B$, $f_a : D \times E_a \rightarrow D_a$ and $g_a : D \times E_a \rightarrow D$.

The LPE defined above consists of a sequence of condition-action-effect rules to specify the behaviour of process P. It specifies that in the current state d of P, for every action $a \in \mathcal{A}$ and value e_a of sort E_a, if condition $h_a(d, e_a)$ holds then P can execute action a with parameter $f_a(d, e_a)$, after which it ends up in the state $g_a(d, e_a)$. The condition, action parameter and next state depend on the current state d and local variable e_a. In mCRL2 the LPE is accompanied by the designation of an initial state $d_0 \in D$ for P.

For every $a \in \mathcal{A}$, the term $\sum_{e_a : E_a} h_a(d, e_a) \rightarrow a(f_a(d, e_a)) \cdot P(g_a(d, e_a))$ is called a *summand*.[4] For convenience and without loss of generality, this summand is unique for every action a. In fact, the summation over the set of actions $\mathcal{A}$ is not part of the mCRL2 language. We have abused notation to abbreviate a finite nondeterministic choice over all possible actions.

For the sake of brevity and clarity of our further considerations, the current state of process P is represented by a single parameter d and an action can carry at most one data parameter. These restrictions do not incur a loss of generality. Also note that the process P cannot terminate successfully. For the expositions in this thesis where LPEs are used, extending the theory to include termination does not pose any theoretical challenges, hence we exclude it for brevity.

The semantics of an LPE is the transition system that it implicitly describes. The relevant domain here is $\mathbb{T}^{\mathsf{u}}$: there is no successful termination, unobservable actions are present and the transition system may be infinitely large due to the possibly infinite data sorts. We assume that for every $a \in \mathcal{A}$ and $d \in D_a$ there is an action $a(\llbracket d \rrbracket) \in Act$.

Definition 2.4. The *labelled transition system* (LTS) of an LPE is a unipolar transition system $(S, \rightarrow, i)$ where:

- $S = \mathbb{D}$;
- $\rightarrow = \{(\llbracket d \rrbracket, a(\llbracket f_a(d, e_a) \rrbracket), \llbracket g_a(d, e_a) \rrbracket) \mid d \in D \wedge a \in \mathcal{A} \wedge e_a \in E_a \wedge \llbracket h_a(d, e_a) \rrbracket\}$;
- $i = \llbracket d_0 \rrbracket$ where $d_0 \in D$ is the initial state of the LPE.

2.4 Preorders and equivalences

In the previous section we introduced the domain of transition systems $\mathbb{T}$ as the universe of discourse when modelling the behaviour of concurrent systems. We now

[4] The use of the term "summand" and of the $\sum$-sign for "summation" over a data sort stem from the fact that, like ACP, mCRL2 uses the $+$-sign to denote nondeterministic choice.

introduce equivalences and preorders on this domain, that allow for proper formal reasoning about transition systems *per se*, and for abstraction from irrelevant differences between them. An equivalence prescribes when two transition systems are to be considered equivalent, and thus when two concurrent systems behave equivalently. Every equivalence partitions the domain of transition systems into *equivalence classes*, such that every equivalence class is a maximal set of transition systems that cannot be distinguished by that equivalence. A preorder specifies when one transition system is a *refinement* of the other, *i.e.* when one system's behaviour is "contained" in that of the other system. For instance, such a relation is often desirable between a system's implementation and its specification. Every preorder induces an equivalence on the domain of transition systems, along with a partial order on the corresponding equivalence classes.

A large number of preorders and equivalences have been defined in the literature and we only consider the ones that are relevant for this thesis. We first focus on *strong* preorders and equivalences, that do not treat τ any differently from observable actions and are typically used in concrete domains. Then we introduce *weak* and *branching* variants, that include special treatment of τ to allow for more proper abstractions in non-concrete domains. Finally, the introduced relations are positioned in a lattice based on their discriminating capabilities.

For the remainder of this section, we work in the bipolar domain $\mathbb{T}^{\mathsf{b}}$, *i.e.* action-based transition systems with successful termination. It is not hard to check that any preorder defined in this section is indeed a preorder and that any equivalence is indeed an equivalence. The relations defined in section 2.4.1 should actually carry the adjective *strong*, but we only use this term explicitly when a clear contrast with weak or branching variants is desired.

2.4.1 Strong variants

For any bipolar transition system, a *trace* of a state is a sequence of actions that can be performed from that state. Moreover, if that trace ends in a final state, the state is said to *accept* the trace. The *language* of a state is the set of traces that it accepts. Formally, for any $M \in \mathbb{T}^{\mathsf{b}}$ and $s \in S_M$ we define the set of traces of s, $\mathcal{T}_M(s)$, and the language of s, $\mathcal{L}_M(s)$, as follows:

$$\mathcal{T}_M(s) := \{\alpha \in Act^* \mid \exists t \in S_M \,.\, s \xrightarrow{\alpha} t\}$$
$$\mathcal{L}_M(s) := \{\alpha \in Act^* \mid \exists t \in F_M \,.\, s \xrightarrow{\alpha} t\}\,.$$

Moreover, the set of traces and the language of M are defined as $\mathcal{T}(M) := \mathcal{T}_M(i_M)$ and $\mathcal{L}(M) := \mathcal{L}_M(i_M)$. We omit the subscripts from $\mathcal{T}_M$ and $\mathcal{L}_M$ if no confusion can arise.

Definition 2.5. Let $M, N \in \mathbb{T}^{\mathsf{b}}$, $s \in S_M$ and $t \in S_N$. We say s is *trace included* in t, denoted $s \sqsubseteq_{\mathcal{T}} t$, if $\mathcal{T}(s) \subseteq \mathcal{T}(t)$, and M is trace included in N, denoted $M \sqsubseteq_{\mathcal{T}} N$, if $\mathcal{T}(M) \subseteq \mathcal{T}(N)$. Trace inclusion $\sqsubseteq_{\mathcal{T}}$ is a preorder on $\mathbb{T}^{\mathsf{b}}$ and *trace equivalence* $\equiv_{\mathcal{T}}$ is its induced equivalence.

Definition 2.6. Let $M, N \in \mathbb{T}^{\mathsf{b}}$, $s \in S_M$ and $t \in S_N$. We say s is *language included* in t, denoted $s \sqsubseteq_{\mathcal{L}} t$, if $\mathcal{L}(s) \subseteq \mathcal{L}(t)$, and M is language included in N, denoted $M \sqsubseteq_{\mathcal{L}} N$, if $\mathcal{L}(M) \subseteq \mathcal{L}(N)$. Language inclusion $\sqsubseteq_{\mathcal{L}}$ is a preorder on $\mathbb{T}^{\mathsf{b}}$ and *language equivalence* $\equiv_{\mathcal{L}}$ is its induced equivalence.

The preorders and equivalences defined above determine whether two transition systems are related by only considering the sequences of actions that are possible from the initial states. More fine-grained comparisons of transition systems take the *branching structure* of the transition systems into account in order to determine whether one transition system is able to faithfully *mimic* or *simulate* another. Starting from the initial states, states of both transition systems are pairwise and iteratively compared based on the action labels and resulting states of the outgoing transitions. The following preorders and equivalences perform this kind of comparison and are due to Milner [99] and Park [105].

Definition 2.7. Let $M, N \in \mathbb{T}^{\mathsf{b}}$. A relation $R \subseteq S_M \times S_N$ is a *simulation* iff for all $(s, t) \in R$:

- $s \in F_M \implies t \in F_N$, and
- $\forall a \in Act, s' \in S_M \,.\, s \xrightarrow{a} s' \implies \exists t' \in S_N \,.\, t \xrightarrow{a} t' \wedge s' \,R\, t'$.

We define $\subseteq$ as the largest simulation and we say s is simulated by t if $s \subseteq t$. Moreover, we define $M \subseteq N$ iff $i_M \subseteq i_N$. The relation $\subseteq$ is a preorder on $\mathbb{T}^{\mathsf{b}}$ called *simulation preorder* and *simulation equivalence* $\rightleftarrows$ is its induced equivalence.

Definition 2.8. Let $M, N \in \mathbb{T}^{\mathsf{b}}$. A relation $R \subseteq S_M \times S_N$ is a *bisimulation* iff both R and R^{-1} are simulations. *Bisimilarity* $\leftrightarrows$ is the largest bisimulation and we define $M \leftrightarrows N$ iff $i_M \leftrightarrows i_N$. Bisimilarity is an equivalence on $\mathbb{T}^{\mathsf{b}}$ that is also called *bisimulation equivalence*.

2.4.2 Weak and branching variants

As mentioned before, an important abstraction mechanism when modelling concurrent systems is to hide internal behaviour by replacing some observable actions by τ. The special meaning of τ as the unobservable action is not reflected in the strong preorders and equivalences of the previous section. In this section we define variants that are weaker: they relate more transition systems than their strong counterparts by relaxing the rules for τ-transitions. We only do so for the simulation-like relations. It is not hard to define weak trace and language preorders, but they are not used in this thesis.

The idea of weak simulation originated from Milner [99]. Like strong simulation, it says that every observable a-step of a state s must be mimicked by a simulating state t. However, t may now perform an arbitrary number of τ-steps before and after performing the a-step. In other words, it may skip any τs in order to meet the requirement of mimicking the a.

Definition 2.9. Let $M, N \in \mathbb{T}^{\mathsf{b}}$. A relation $R \subseteq S_M \times S_N$ is a *weak simulation* iff for all $(s,t) \in R$:

- $s \in F_M \Longrightarrow \exists t' \in F_N \,.\, t \Rightarrow t'$, and
- for all $a \in Act$ and $s' \in S_M$, if $s \xrightarrow{a} s'$ then either:
  - $a = \tau \wedge s' \,R\, t$, or
  - $\exists u, u', t' \in S_N \,.\, t \Rightarrow u \xrightarrow{a} u' \Rightarrow t' \wedge s' \,R\, t'$.

We define $\subseteq_{\mathsf{w}}$ as the largest weak simulation and we say s is weakly simulated by t if $s \subseteq_{\mathsf{w}} t$. Moreover, we define $M \subseteq_{\mathsf{w}} N$ iff $i_M \subseteq_{\mathsf{w}} i_N$. The relation $\subseteq_{\mathsf{w}}$ is a preorder on $\mathbb{T}^{\mathsf{b}}$ called *weak simulation preorder* and *weak simulation equivalence* $\rightleftarrows_{\mathsf{w}}$ is its induced equivalence.

Definition 2.10. Let $M, N \in \mathbb{T}^{\mathsf{b}}$. A relation $R \subseteq S_M \times S_N$ is a *weak bisimulation* iff both R and R^{-1} are weak simulations. *Weak bisimilarity* $\leftrightarrows_{\mathsf{w}}$ is the largest weak bisimulation and we define $M \leftrightarrows_{\mathsf{w}} N$ iff $i_M \leftrightarrows_{\mathsf{w}} i_N$. Weak bisimilarity is an equivalence on $\mathbb{T}^{\mathsf{b}}$ that is also called *weak bisimulation equivalence*.

As argued by Van Glabbeek and Weijland [56], weak bisimulation equivalence does not respect the branching structures of transition systems as well as may be desirable. They introduce a stronger version of weak bisimilarity, called *branching bisimilarity*. The idea is that when mimicking an a-step via arbitrary τ-paths, the intermediately visited states also have to be related to corresponding states in the simulated transition system.

Definition 2.11. Let $M, N \in \mathbb{T}^{\mathsf{b}}$. A relation $R \subseteq S_M \times S_N$ is a *branching simulation* iff for all $(s,t) \in R$:

- $s \in F_M \Longrightarrow \exists t' \in F_N \,.\, t \Rightarrow t' \wedge s \,R\, t'$, and
- for all $a \in Act$ and $s' \in S_M$, if $s \xrightarrow{a} s'$ then either:
  - $a = \tau \wedge s' \,R\, t$, or
  - $\exists u, u', t' \in S_N \,.\, t \Rightarrow u \xrightarrow{a} u' \Rightarrow t' \wedge s \,R\, u \wedge s' \,R\, u' \wedge s' \,R\, t'$.

We define $\subseteq_{\mathsf{b}}$ as the largest branching simulation and we say s is branching simulated by t if $s \subseteq_{\mathsf{b}} t$. Moreover, we define $M \subseteq_{\mathsf{b}} N$ iff $i_M \subseteq_{\mathsf{b}} i_N$. The relation $\subseteq_{\mathsf{b}}$ is a preorder on $\mathbb{T}^{\mathsf{b}}$ called *branching simulation preorder* and *branching simulation equivalence* $\rightleftarrows_{\mathsf{b}}$ is its induced equivalence.

Definition 2.12. Let $M, N \in \mathbb{T}^{\mathsf{b}}$. A relation $R \subseteq S_M \times S_N$ is a *branching bisimulation* iff both R and R^{-1} are branching simulations. *Branching bisimilarity* $\leftrightarrows_{\mathsf{b}}$ is the largest branching bisimulation, and we define $M \leftrightarrows_{\mathsf{b}} N$ iff $i_M \leftrightarrows_{\mathsf{b}} i_N$. Branching bisimilarity $\leftrightarrows_{\mathsf{b}}$ is an equivalence on $\mathbb{T}^{\mathsf{b}}$ that is also called *branching bisimulation equivalence*.

2.4.3 The linear-time–branching-time spectrum

The trace and language preorders and equivalences are typical examples of *linear-time semantics*: transition systems are related by only comparing linear sequences

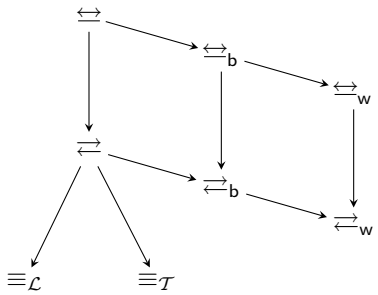


Figure 2.2. The lattice of equivalence relations.

of actions. On the other hand, the simulation-like relations are prime examples of *branching-time semantics*: the branching structure of the transition systems is taken into account. Many more linear-time and branching-time semantics have been defined in the literature. They can be ordered in a lattice based on their capabilities to distinguish between transition systems. This lattice, called the *linear-time–branching-time spectrum*, has been constructed by Van Glabbeek for both concrete [55] and non-concrete [54] domains.

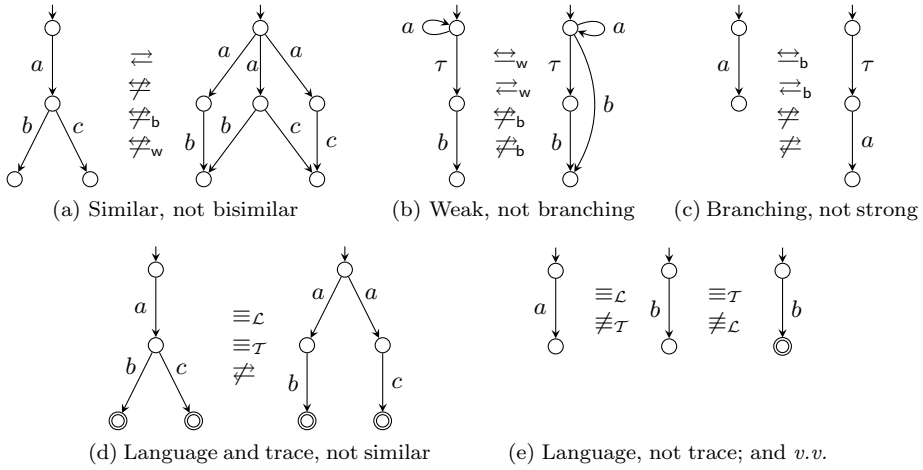
Figure 2.2 shows the lattice for the equivalences introduced in this section. An arrow from A to B means that A is *stronger* than B, *i.e.* A distinguishes more (or identifies less) transition systems than B. Alternatively, we can say that if two transition systems are A-equivalent then they are also B-equivalent. The correctness of each of the arrows follows readily from the definitions. Counterexamples to any other arrows are given in figure 2.3.

For concrete domains it is obvious that $\leftrightarrows = \leftrightarrows_{\mathsf{b}} = \leftrightarrows_{\mathsf{w}}$ and $\rightleftarrows = \rightleftarrows_{\mathsf{b}} = \rightleftarrows_{\mathsf{w}}$, so that each of those groups collapses to a single class in the lattice. Moreover, a further restriction to deterministic domains yields $\leftrightarrows = \rightleftarrows = \equiv_{\mathcal{L}}$ [45, 105], by which those three classes converge as well. Orthogonally to these, for unipolar domains we have $\equiv_{\mathcal{L}} = \equiv_{\mathcal{T}}$ and the first clause in the definition of every simulation-relation holds trivially, by which it is usually omitted.

2.5 Temporal logics

Temporal logics are a special kind of *modal logics*, that extend traditional propositional logic with *modalities* to allow for propositions like "possibly true" (a statement holds in some possible world) or "necessarily true" (a statement holds in all possible worlds). In temporal logics, the modalities relate to time and are represented by *temporal operators*. Typical examples of temporal operators include *eventually* (a statement holds at some time instance, now or in the future) and *globally* (a statement holds at all time instances, now and in the future).

Temporal logics are commonly used for expressing properties about concurrent



(a) Similar, not bisimilar (b) Weak, not branching (c) Branching, not strong

(d) Language and trace, not similar (e) Language, not trace; and *v.v.*

Figure 2.3. Counterexamples to any other arrows in the lattice of figure 2.2.

systems. In this context, time is assumed to progress in a discrete – rather than continuous – manner, along with the computation. Well-known temporal logics include *Linear Temporal Logic* (LTL) [109] and *Computation Tree Logic* (CTL) [23]. LTL assumes a linear model of time and can only be used for specifying linear-time properties. On the other hand, CTL assumes a tree-like model of time, thereby admitting the specification of branching-time properties. LTL and CTL share a common subset of expressible properties but each admits properties that cannot be expressed in the other. They are both proper subsets of CTL* [43]. This logic is state-based, but an action-based version has also been proposed [34].

A more generic temporal logic is the modal *μ-calculus* [85]. It properly contains CTL* and *Hennessy–Milner logic* [71] and can be used for both state- and action-based properties. In this section we introduce a μ-calculus that is action-based and *first-order*: it extends the standard μ-calculus with data (*cf.* [61]). We work in the context of action-based models without termination, *i.e.* the domain $\mathbb{T}^{\mathsf{u}}$.

2.5.1 Syntax

Let $\mathcal{X}$ be a set of *predicate variables*. Every predicate variable $X \in \mathcal{X}$ is a function from a number of data sorts to the Booleans, *i.e.* if X has arity $n \geq 0$ then $X : D_1 \times \ldots \times D_n \to B$ for some data sorts $D_1, \ldots, D_n$. If $n = 0$, we have $X : B$ and we call X a *proposition variable*. In our definitions of the syntax and semantics of the μ-calculus, we only consider variables of arity 1 for the sake of brevity. It is elementary to extend these definitions for variables with arbitrary arities, and we allow ourselves to use such variables in the remainder of this thesis.

Definition 2.13. The syntax of *μ-calculus formulae* φ and *action formulae* α is given by the following grammar:

$$\varphi ::= b \mid X(e) \mid \varphi \oplus \varphi \mid \mathsf{Q}d{:}D \,.\, \varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid (\sigma X(d{:}D) \,.\, \varphi)(e)$$
$$\alpha ::= b \mid a(e) \mid \neg\alpha \mid \alpha \oplus \alpha \mid \mathsf{Q}d{:}D \,.\, \alpha$$

where $\oplus \in \{\vee, \wedge\}$, $\mathsf{Q} \in \{\exists, \forall\}$, $\sigma \in \{\mu, \nu\}$, b and e are data terms of sorts B and D, respectively (possibly containing data variables $d{\in}\mathcal{D}$), $X{\in}\mathcal{X}$ is a predicate variable, and $a \in \mathcal{A}$ is a parameterized action.

Negations occur only at the level of the data terms and we assume that all bound variables are distinct. We shall use the symbols $\oplus$ for either $\vee$ or $\wedge$, Q for either $\exists$ or $\forall$, and σ for either μ or ν when the specific symbol is of lesser importance.

2.5.2 Semantics

The semantics of μ-calculus formulae is given in the context of a unipolar transition system. For any formula φ and transition system $(S, \rightarrow, i)$, the semantics of φ is the set of states in S in which the formula holds. Predicate variables are interpreted in the context of a *predicate environment* $\theta : \mathcal{X} \rightarrow (\mathbb{D} \rightarrow \wp(S))$ that associates a function from $\mathbb{D}$ to $\wp(S)$ with every predicate variable X of sort D.

Definition 2.14. Let $(S, \rightarrow, i) \in \mathbb{T}^{\mathsf{u}}$, ε be a data environment and θ be a predicate environment. The interpretation $[\![\varphi]\!]\theta\varepsilon$ of a μ-calculus formula φ in the context of θ and ε, is a subset of S that is defined inductively as follows:

$$\begin{aligned}
[\![b]\!]\theta\varepsilon &:= \begin{cases} S & \text{if } [\![b]\!]\varepsilon \\ \emptyset & \text{otherwise} \end{cases} \\
[\![X(e)]\!]\theta\varepsilon &:= \theta(X)([\![e]\!]\varepsilon) \\
[\![\varphi \vee \psi]\!]\theta\varepsilon &:= [\![\varphi]\!]\theta\varepsilon \;\cup\; [\![\psi]\!]\theta\varepsilon \\
[\![\varphi \wedge \psi]\!]\theta\varepsilon &:= [\![\varphi]\!]\theta\varepsilon \;\cap\; [\![\psi]\!]\theta\varepsilon \\
[\![\exists d{:}D \,.\, \varphi]\!]\theta\varepsilon &:= \textstyle\bigcup_{v \in \mathbb{D}} [\![\varphi]\!]\theta\varepsilon[d \mapsto v] \\
[\![\forall d{:}D \,.\, \varphi]\!]\theta\varepsilon &:= \textstyle\bigcap_{v \in \mathbb{D}} [\![\varphi]\!]\theta\varepsilon[d \mapsto v] \\
[\![\langle\alpha\rangle\varphi]\!]\theta\varepsilon &:= \{s \in S \mid \exists a{\in}Act, t{\in}S \,.\, s \xrightarrow{a} t \wedge a \in [\![\alpha]\!]\varepsilon \wedge t \in [\![\varphi]\!]\theta\varepsilon\} \\
[\![[\alpha]\varphi]\!]\theta\varepsilon &:= \{s \in S \mid \forall a{\in}Act, t{\in}S \,.\, s \xrightarrow{a} t \wedge a \in [\![\alpha]\!]\varepsilon \Longrightarrow t \in [\![\varphi]\!]\theta\varepsilon\} \\
[\![(\sigma X(d{:}D) \,.\, \varphi)(e)]\!]\theta\varepsilon &:= (\sigma f{:}\mathbb{D} \rightarrow \wp(S) \,.\, \lambda v{:}\mathbb{D} \,.\, [\![\varphi]\!]\theta[X \mapsto f]\varepsilon[d \mapsto v])([\![e]\!]\varepsilon).
\end{aligned}$$

The interpretation $[\![\alpha]\!]\varepsilon$ of an action formula α in the context of ε, is a subset of

Act that is defined inductively as follows:

$$
\begin{array}{lll}
[\![b]\!]\varepsilon & := \begin{cases} \mathit{Act} & \text{if } \varepsilon(b) \\ \emptyset & \text{otherwise} \end{cases} & \\
[\![a(e)]\!]\varepsilon & := \{a([\![e]\!]\varepsilon)\} & [\![\neg\alpha]\!]\varepsilon := \mathit{Act} \setminus [\![\alpha]\!]\varepsilon \\
[\![\alpha \vee \beta]\!]\varepsilon & := [\![\alpha]\!]\varepsilon \cup [\![\beta]\!]\varepsilon & [\![\exists d{:}D \,.\, \alpha]\!]\varepsilon := \bigcup_{v \in \mathbb{D}} [\![\alpha]\!]\varepsilon[d \mapsto v] \\
[\![\alpha \wedge \beta]\!]\varepsilon & := [\![\alpha]\!]\varepsilon \cap [\![\beta]\!]\varepsilon & [\![\forall d{:}D \,.\, \alpha]\!]\varepsilon := \bigcap_{v \in \mathbb{D}} [\![\alpha]\!]\varepsilon[d \mapsto v].
\end{array}
$$

The fixpoint semantics of the μ-calculus is properly defined for the following reasons. Let $\leq$ be the pointwise ordering on the set of functions $\wp(S)^{\mathbb{D}}$, *i.e.* $f \leq g$ iff $\forall d \in \mathbb{D} \,.\, f(d) \subseteq g(d)$. It is not hard to see that the poset $(\wp(S)^{\mathbb{D}}, \leq)$ is a complete lattice. By definition of $[\![\varphi]\!]\theta\varepsilon$, the function $\lambda f \,.\, \lambda v{:}\mathbb{D} \,.\, [\![\varphi]\!]\theta[X \mapsto f]\varepsilon[d \mapsto v]$ over this lattice is monotonic. Tarski's theorem then guarantees the existence of the least and greatest fixpoints of this function.

2.5.3 Examples

Typical examples of properties in the standard μ-calculus (without data) are the following:

- $\nu X.\langle\top\rangle\top \wedge [\top]X$ is *deadlock freedom*: it is always possible to perform an action;
- $\nu X.[a]\bot \wedge [\top]X$ is *safety*: it is never possible to perform an a-action;
- $\mu X.\langle a\rangle\top \vee \langle\top\rangle X$ is *reachability*: a is possible after some finite path;
- $\mu X.\langle\top\rangle\top \wedge [\neg a]X$ is *inevitability*: always eventually a, via finite paths only;
- $\nu X.\mu Y.((\langle a\rangle\top \vee \langle\top\rangle Y) \wedge \langle\top\rangle X)$: a is enabled infinitely often.

The extension with data allows for more involved properties, for example:

- $\nu X \,.\, [\top]X \wedge \forall d{:}D \,.\, [\mathit{read}(d)](\mu Y.\langle\top\rangle\top \wedge [\neg \mathit{store}(d)]Y)$: every datum d that is read, will eventually be stored;
- $(\mu X(n{:}N) \,.\, n \geq K \vee \langle\neg a\rangle X(n) \vee \langle a\rangle X(n+1))(0)$: there is a path containing at least K a-actions, for a given K;
- $\nu X \,.\, [\exists d{:}D \,.\, a(d)]X \wedge [\neg\exists d{:}D \,.\, a(d)]\bot$: the only actions that are ever executed, are a-actions with any parameter.

More examples of first-order μ-calculus properties can be found in [61, 69]. For a more comprehensive introduction to the standard μ-calculus we refer to [14].

2.6 Equation systems

Checking whether a transition system satisfies a standard μ-calculus formula can be done by approximation (see *e.g.* [41]). A different method for solving this model-checking problem is by encoding the transition system and the formula in a sequence of fixpoint equations and subsequently solving these equations. In

general, such sequences are called *fixpoint equation systems*. In this case the underlying lattice of the equation systems is $(\mathbb{B}, \Longrightarrow)$ – *i.e.* the Booleans ordered by implication – by which they are called *Boolean equation systems* (BES). A number of algorithms exist for solving BESs (*e.g.* approximation, Gauß-elimination, tableaux methods). We refer to [91] for a comprehensive study of BESs.

BESs can be extended to *parameterized Boolean equation systems* (PBES) by including data. A PBES is a fixpoint equation system over the lattice of functions from the data domains to the Booleans. This allows for model checking of a first-order μ-calculus formula on a transition system or even an implicit representation thereof, like an LPE. The approach is analogous to the one outlined above: the formula and the model are encoded in a PBES [61], which is subsequently solved to obtain the answer to the model-checking problem. In this way, a PBES can be used for finitely representing the problem of checking a first-order μ-calculus formula on a (possibly) infinite-state system. Such equation systems are also used in [129] for model checking systems with data and time. Solving PBESs is generally undecidable, which is primarily due to the data domains that are involved. However, techniques have been developed that aim to symbolically approximate the solution to a PBES [69], or manipulate a PBES such that a solution may be obtained more easily [70, 103]. We list a number of these techniques in section 2.6.3, after defining the syntax and semantics of PBESs.

2.6.1 Syntax

A PBES is a sequence of fixpoint equations, where each equation is of the form $\sigma X(d{:}D) = \varphi$. The left-hand side consists of a fixpoint symbol σ and a predicate variable $X \in \mathcal{X}$ that depends on the data variable $d \in \mathcal{D}$ of possibly infinite sort D. Again we restrict ourselves to predicate variables of arity 1 in the following definitions. The right-hand side of an equation is a *predicate formula*.

Definition 2.15. *Predicate formulae* φ are defined by the following grammar:

$$\varphi ::= b \mid X(e) \mid \varphi \oplus \varphi \mid \mathsf{Q}d{:}D \,.\, \varphi$$

where b and e are data terms of sorts B and D respectively, $X \in \mathcal{X}$, $\oplus \in \{\vee, \wedge\}$, $\mathsf{Q} \in \{\exists, \forall\}$, and $d \in \mathcal{D}$ is a data variable of sort D.

We allow $b \Longrightarrow \varphi$ in predicate formulae as a shorthand for $\neg b \vee \varphi$, for any data term b of sort B and predicate formula φ. For any predicate formula φ, data term v and data variable d that occurs freely in φ, we denote by $\varphi[v/d]$ the syntactic substitution of v for d in φ.

Definition 2.16. *Parameterized Boolean equation systems* (PBES) $\mathcal{E}$ are defined by the following grammar:

$$\mathcal{E} ::= \epsilon \mid (\sigma X(d{:}D) = \varphi)\ \mathcal{E}$$

where ϵ denotes the empty PBES, $\sigma \in \{\mu, \nu\}$, $X \in \mathcal{X}$ and φ is a predicate formula.

Definition 2.17. A *Boolean equation system* (BES) is a PBES in which every predicate variable is of sort B and every predicate formula φ adheres to the following grammar:

$$\varphi ::= \bot \mid \top \mid X \mid \varphi \oplus \varphi$$

where $\oplus \in \{\vee, \wedge\}$. In this case, X is called a *proposition variable* and φ a *proposition formula*.

We call a predicate formula φ *closed* if no data variable in φ occurs freely. The set of predicate variables that occur in a predicate formula φ, denoted by $\mathsf{occ}(\varphi)$, is defined recursively as follows, for any formulae φ, ψ:

$$\mathsf{occ}(b) := \emptyset \qquad \mathsf{occ}(X(e)) := \{X\}$$
$$\mathsf{occ}(\varphi \oplus \psi) := \mathsf{occ}(\varphi) \cup \mathsf{occ}(\psi) \qquad \mathsf{occ}(\mathsf{Q}d{:}D\,.\,\varphi) := \mathsf{occ}(\varphi).$$

For any PBES $\mathcal{E}$, the set of *binding predicate variables*, $\mathsf{bnd}(\mathcal{E})$, is the set of variables occurring at the left-hand side of some equation in $\mathcal{E}$. The set of *occurring predicate variables*, $\mathsf{occ}(\mathcal{E})$, is the set of variables occurring at the right-hand side of some equation in $\mathcal{E}$. The set of predicate variables occurring anywhere in $\mathcal{E}$ is denoted by $\mathsf{var}(\mathcal{E})$. Formally, we define:

$$\mathsf{bnd}(\epsilon) := \emptyset \qquad \mathsf{bnd}((\sigma X(d{:}D) = \varphi)\ \mathcal{E}) := \{X\} \cup \mathsf{bnd}(\mathcal{E})$$
$$\mathsf{occ}(\epsilon) := \emptyset \qquad \mathsf{occ}((\sigma X(d{:}D) = \varphi)\ \mathcal{E}) := \mathsf{occ}(\varphi) \cup \mathsf{occ}(\mathcal{E})$$
$$\mathsf{var}(\mathcal{E}) := \mathsf{bnd}(\mathcal{E}) \cup \mathsf{occ}(\mathcal{E}).$$

A PBES $\mathcal{E}$ is said to be *well-formed* iff every binding predicate variable occurs at the left-hand side of precisely one equation of $\mathcal{E}$. Thus, $(\nu X = \top)(\mu X = \bot)$ is not a well-formed PBES. We only consider well-formed PBESs in this thesis.

We say a PBES $\mathcal{E}$ is *closed* whenever $\mathsf{occ}(\mathcal{E}) \subseteq \mathsf{bnd}(\mathcal{E})$ and if $\mathcal{E}$ is not closed then $\mathcal{E}$ is called *open*. An equation $\sigma X(d{:}D) = \varphi$ is called *data-closed* if the set of data variables that occur freely in φ is either empty or $\{d\}$. A PBES is called data-closed iff each of its equations is data-closed. We say an equation $\sigma X(d{:}D) = \varphi$ is *solved* if $\mathsf{occ}(\varphi) = \emptyset$, and a PBES $\mathcal{E}$ is *solved* iff each of its equations is solved.

2.6.2 Semantics

Predicate formulae and PBESs are interpreted in the context of a data environment $\varepsilon : \mathcal{D} \to \mathbb{D}$ and a predicate environment $\eta : \mathcal{X} \to (\mathbb{D} \to \mathbb{B})$.

Definition 2.18. Let ε be a data environment and η be a predicate environment. The interpretation $[\![\varphi]\!]\eta\varepsilon$ of a predicate formula φ in the context of η and ε, is a Boolean value that is defined inductively as follows:

$$\begin{aligned}
[\![b]\!]\eta\varepsilon &:= [\![b]\!]\varepsilon \\
[\![X(e)]\!]\eta\varepsilon &:= \eta(X)([\![e]\!]\varepsilon) \\
[\![\varphi \oplus \psi]\!]\eta\varepsilon &:= [\![\varphi]\!]\eta\varepsilon \ \oplus\ [\![\psi]\!]\eta\varepsilon \\
[\![\mathsf{Q}d{:}D.\varphi]\!]\eta\varepsilon &:= \mathsf{Q}v \in \mathbb{D}\,.\, [\![\varphi]\!]\eta\varepsilon[d \mapsto v].
\end{aligned}$$

The predicate formula φ in an equation $\sigma X(d{:}D) = \varphi$ must be interpreted as a fixpoint over the set of functions $\mathbb{B}^{\mathbb{D}}$. The existence of such fixpoints follows from the following observations. The variable d, which may occur freely in φ, is effectively used as a formal, syntactic function parameter. Semantically, this is achieved by associating the interpretation of the predicate formula φ to the functional $(\lambda v \in \mathbb{D}.[\![\varphi]\!]\eta\varepsilon[d \mapsto v])$, which relies on the data environment to assign specific values to variable d. The set $\mathbb{B}^{\mathbb{D}}$ can be equipped with an ordering $\leq$, defined as follows: $f \leq g$ iff $\forall v{\in}\mathbb{D}\,.\,f(v) \Longrightarrow g(v)$. The poset $(\mathbb{B}^{\mathbb{D}}, \leq)$ is a complete lattice. The functional $(\lambda v \in \mathbb{D}.\ [\![\varphi]\!]\eta\varepsilon[d \mapsto v])$ can be turned into a predicate formula transformer by employing the predicate environment η in a similar manner as the data environment is used to turn a predicate formula into a functional. Assuming that the domain of the predicate variable X is of sort D, the functional $(\lambda v \in \mathbb{D}\,.\,[\![\varphi]\!]\eta\varepsilon[d \mapsto v])$ yields the following predicate formula transformer:

$$\lambda f \in \mathbb{B}^{\mathbb{D}}\,.\,(\lambda v \in \mathbb{D}\,.\,[\![\varphi]\!]\eta[X \mapsto f]\varepsilon[d \mapsto v]).$$

The resulting predicate formula transformer is monotonic over the complete lattice $(\mathbb{B}^{\mathbb{D}}, \leq)$. Tarski's theorem then guarantees the existence of least and greatest fixpoints of the predicate formula transformers.

Definition 2.19. Let η be a predicate environment and ε a data environment. The *solution* $[\![\mathcal{E}]\!]\eta\varepsilon$ of a PBES $\mathcal{E}$ in the context of η and ε, is a predicate environment that is defined inductively as follows:

$$\begin{aligned}
[\![\epsilon]\!]\eta\varepsilon &:= \eta \\
[\![(\sigma X(d{:}D) = \varphi)\ \mathcal{E}]\!]\eta\varepsilon &:= [\![\mathcal{E}]\!]\eta[X \mapsto \sigma\Phi]\varepsilon
\end{aligned}$$

where $\Phi = \lambda f \in \mathbb{B}^{\mathbb{D}}\,.\,\lambda v \in \mathbb{D}\,.\,[\![\varphi]\!]([\![\mathcal{E}]\!]\eta[X \mapsto f]\varepsilon)\varepsilon[d \mapsto v]$.

The solution of a PBES prioritizes the fixpoint signs of equations that come first over those of equations that follow. In that sense, the solution of a PBES is sensitive to the order of the equations of that PBES. Moreover, the solution $[\![\mathcal{E}]\!]\eta\varepsilon$ of a PBES $\mathcal{E}$ only modifies η for the *binding* variables of $\mathcal{E}$; η is left unmodified for other predicate variables, as is shown by the following lemma.

Lemma 2.20. *Let $\mathcal{E}$ be an arbitrary PBES. Then:*

$$\forall X \in \mathcal{X}\,.\,\forall \eta, \varepsilon\,.\,X \notin \mathsf{bnd}(\mathcal{E}) \Longrightarrow [\![\mathcal{E}]\!]\eta\varepsilon(X) = \eta(X).$$

Proof. The proof is by induction on the length of equation system $\mathcal{E}$. If $\mathcal{E}$ is of length 0 then the equivalence holds trivially. Suppose $\mathcal{E}$ is of length $m+1$ and for any PBES $\mathcal{F}$ of length m we have:

(IH) $$\forall X \in \mathcal{X}\,.\,\forall \eta, \varepsilon\,.\,X \notin \mathsf{bnd}(\mathcal{F}) \Longrightarrow [\![\mathcal{F}]\!]\eta\varepsilon(X) = \eta(X).$$

Necessarily, $\mathcal{E}$ is of the form $(\sigma Y(d{:}D) = \varphi)\ \mathcal{E}'$, where $\mathcal{E}'$ is of length m. Let $X{\in}\mathcal{X}$ such that $X \notin \mathsf{bnd}((\sigma Y(d{:}D) = \varphi)\ \mathcal{E}')$, and let η, ε be arbitrary environments. We derive:

$$
\begin{aligned}
&[\![(\sigma Y(d{:}D) = \varphi)\ \mathcal{E}']\!]\eta\varepsilon(X) \\
&\quad = \quad ([\![\mathcal{E}']\!]\eta[Y \mapsto (\sigma f \in \mathbb{B}^{\mathbb{D}} \,.\, \lambda v{\in}\mathbb{D} \,.\, [\![\varphi]\!]([\![\mathcal{E}']\!]\eta[Y \mapsto f]\varepsilon)\varepsilon[d \mapsto v])]\varepsilon)(X) \\
&\quad \overset{\text{(IH)}}{=} \quad (\eta[Y \mapsto (\sigma f \in \mathbb{B}^{\mathbb{D}} \,.\, \lambda v{\in}\mathbb{D} \,.\, [\![\varphi]\!]([\![\mathcal{E}']\!]\eta[Y \mapsto f]\varepsilon)\varepsilon[d \mapsto v])])(X) \\
&\quad \overset{X \neq Y}{=} \quad \eta(X)\,. \qquad \square
\end{aligned}
$$

For a closed PBES $\mathcal{E}$, the solution $[\![\mathcal{E}]\!]\eta\varepsilon(X)$ for $X \in \mathsf{bnd}(\mathcal{E})$ does not depend on the environment η (see theorem 6 of [70]). Similarly, if $\mathcal{E}$ is data-closed, the solution $[\![\mathcal{E}]\!]\eta\varepsilon$ does not depend on the environment ε (lemma 4 of [70]). In these cases, we may omit η and ε, respectively, to indicate that the specific environment is irrelevant. Another property of the solution of a PBES is that it is a monotonic function over the set of predicate environments. This is expressed by lemma 5 of [70], which we repeat here.

Lemma 2.21. *Let η, η' be predicate environments. Then for any PBES $\mathcal{E}$, $\eta \leq \eta'$ implies $[\![\mathcal{E}]\!]\eta \leq [\![\mathcal{E}]\!]\eta'$.*

2.6.3 Solution techniques

A number of solution techniques for PBESs have been developed in the literature. Below we describe the ones that are used most frequently throughout this thesis.

- *Approximation.* For any equation $\sigma X(d{:}D) = \varphi$, we can iteratively approximate the fixpoint solution of X. The first approximant X_0 depends solely on the fixpoint symbol σ: if $\sigma = \mu$ then $X_0(d{:}D) := \bot$, and if $\sigma = \nu$ then $X_0(d{:}D) := \top$. For any $i > 0$ we iteratively compute the ith approximant by substituting the $(i-1)$th approximant for X in φ, *i.e.* $X_i(d{:}D) := \varphi[X_{i-1}/X]$. If for some $k > 0$ we have $X_k = X_{k-1}$ then X_k is the solution of X and the approximation terminates. Note that it may not terminate in general.
- *Patterns.* If an equation is of a particular shape, we can immediately substitute its solution thereby avoiding a tedious, or even non-terminating approximation. In particular, if an equation is of the following form:

  $$\sigma X(d{:}D) = \varphi(d) \wedge (\psi(d) \vee X(f(d)))$$

  where $f : D \to D$ is an arbitrary function, and φ and ψ are predicate formulae in which X does not occur, then the right-hand side can be replaced by:

  $$
  \begin{array}{ll}
  \forall j{:}N \,.\, ((\forall i{:}N \,.\, i < j \implies \neg\psi(f^i(d))) \implies \varphi(f^j(d))) & \text{if } \sigma = \nu, \text{ and} \\
  \exists i{:}N \,.\, \psi(f^i(d)) \wedge \forall j{:}N \,.\, (j \leq i \implies \varphi(f^j(d))) & \text{if } \sigma = \mu.
  \end{array}
  $$

- *Substitution.* For any equation $\sigma X(d{:}D) = \varphi$ of a PBES $\mathcal{E}$, we can substitute φ for X in all equations that precede the equation for X in $\mathcal{E}$. This technique is part of the Gauß-elimination procedure described in [91]. To indicate that it only works in one direction, it is sometimes called *backwards substitution.*

- *Migration.* For any solved equation $\sigma X(d{:}D) = \varphi$ of a PBES $\mathcal{E}$, we are allowed to move the equation for X to any position within $\mathcal{E}$. Recall that an equation is solved if $\mathsf{occ}(\varphi) = \emptyset$. This technique is useful in combination with the substitution technique above: once an equation for X has been solved, we can migrate it towards the end of the PBES and substitute its right-hand side for X in all other equations.

More information on these techniques can be found in [69] and [70]. Other techniques used in this thesis include strengthening [70] and invariants [103]. We explain these techniques in more detail where they are used.

Chapter 3

Determinization

3.1 Introduction

Given a finite-state, *nondeterministic* transition system M, the *determinization problem* asks to give a *deterministic* transition system that is language equivalent to M. Determinization plays an important role in deciding language inclusion (or language equivalence) on nondeterministic transition systems. This can be done by determinizing both transition systems and deciding simulation preorder or (bi)simulation equivalence on the resulting structures, as language and (bi)simulation semantics coincide for deterministic systems (see section 2.4.3). A similar strategy can be applied for deciding trace inclusion (or trace equivalence) by omitting the special treatment of final states.

We study the determinization problem in its original context, *viz.* that of *nondeterministic finite automata* (NFA). These are models of terminating, sequential computations, that coincide with finite concrete transition systems with termination ($\mathbb{T}^{\mathsf{fbc}}$). Traditionally, typical applications of NFAs involve checking whether some sequence of symbols meets some syntactic criterion, like displaying a prescribed pattern or being correct input for a given program. Such a problem can often be recast as checking whether that sequence is accepted by a given NFA.

A *deterministic finite automaton* (DFA) is an NFA that is deterministic (see definition 2.2). In graphical representations of DFAs we also allow states that have *at most* one outgoing a-transition for some action a. Formally speaking, such a DFA abbreviates the one obtained by adding a new, non-accepting *sink* state ς and transitions $s \xrightarrow{a} \varsigma$ for all states s that do not already have an outgoing a-transition. Note that these additions preserve language equivalence.

For every NFA there exists a language equivalent DFA. Contrary to NFAs, for any DFA there is a trivial linear-time, constant-space, online algorithm to check whether an input sequence is accepted or not. For this reason, in many applications it pays off to determinize NFAs, even though the problem is EXPTIME-hard in

general. Once a language equivalent DFA of an NFA has been found, it is usually minimized to obtain the smallest such DFA. This minimal DFA is unique and the problem of finding it for a given NFA is called the *canonization problem*.

The standard algorithm for determinization is called *subset construction* (see *e.g.* [80]) and is renowned for its good performance in practice. For DFA minimization a number of algorithms have been proposed, of which Watson presents a taxonomy and performance analyses [126]. The algorithm with the best time complexity is by Hopcroft [79]: $\mathcal{O}(N \log N)$ where N is the number of states in the input DFA.

Another algorithm for canonization is by Brzozowski [17]. It generates the minimal DFA directly by repeating the process of "reversing" and determinizing the input NFA twice. Tabakov and Vardi compare both approaches to canonization experimentally by running them on randomly generated automata [118]. They show that the subset–Hopcroft approach performs best overall and for smaller transition densities, but for larger transition densities Brzozowski's algorithm is faster.

On some NFAs, the exponential blow-up by subset construction is unavoidable. However, we have encountered NFAs for which subset construction consumes a substantial amount of memory and generates a DFA that is much larger than the minimal DFA. Therefore, our main goal is to find algorithms that are more memory efficient and produce smaller DFAs than subset construction.

In this chapter we present five determinization algorithms based on subset construction. For all of them we prove correctness. One algorithm generates the minimal DFA directly and hence is a *canonization algorithm*. However, it calculates language inclusion as a subroutine; as deciding language inclusion is PSPACE-complete, it is unattractive to use in an implementation. The other four algorithms produce a DFA that is not necessarily minimal but is usually smaller than the DFA produced by subset construction.

We have implemented subset construction and these four new algorithms. We have performed experiments with these implementations by running them on NFAs that describe patterns on the lines of a cellular automaton's evolution and on randomly generated automata. We compare the implementations on the time and memory needed for the complete canonization process (*i.e.* including minimization) and the size of the DFA after determinization.

3.2 Determinization algorithms

3.2.1 Subset construction revisited

We fix a finite alphabet $\Sigma \subseteq Act$ of transition labels for all automata in this chapter. We define the language of a set of states P of an NFA N as $\mathcal{L}_N(P) := \bigcup_{p \in P} \mathcal{L}_N(p)$. Language inclusion and equivalence can now be defined on any combination of states and sets of states, in terms of set inclusion and equivalence.

Algorithm 3.1. The SUBSET(f) determinization algorithm.

Pre: N is an NFA and $f(P) \equiv_{\mathcal{L}} P$ for all $P \subseteq S_N$.
Post: D is a DFA such that $N \equiv_{\mathcal{L}} D$.

1: $\rightarrow_D, F_D, i_D := \emptyset, \emptyset, f(\{i_N\})$;
2: S_D, *todo*, *done* $:= \{i_D\}, \{i_D\}, \emptyset$;
3: **while** *todo* $\neq \emptyset$ **do**
4: pick a $P \in$ *todo*;
5: **for all** $a \in \Sigma$ **do**
6: $P' := f(\{p' \in S_N \mid \exists p \in P \,.\, p \xrightarrow{a}_N p'\})$;
7: $S_D := S_D \cup \{P'\}$;
8: $\rightarrow_D := \rightarrow_D \cup \{(P, a, P')\}$;
9: *todo* := *todo* $\cup$ $(\{P'\} \setminus$ *done*$)$
10: **end for**;
11: **if** $\exists p \in P \,.\, p \in F_N$ **then**
12: $F_D := F_D \cup \{P\}$
13: **end if**;
14: *todo*, *done* $:=$ *todo* $\setminus \{P\}$, *done* $\cup \{P\}$
15: **end while**

For instance, for $p \in S_N$ and $P \subseteq S_N$, $p \sqsubseteq_{\mathcal{L}} P$ holds if $\mathcal{L}_N(p) \subseteq \mathcal{L}_N(P)$.

For reasons that will become apparent in the next sections, we slightly generalize the standard subset construction algorithm by augmenting it with a function f on sets of states of the input NFA, which is applied to every generated set. The algorithm is algorithm 3.1 and shall be referred to as SUBSET(f). It takes an NFA N and generates a DFA D. Of course, it should be the case that $N \equiv_{\mathcal{L}} D$, which depends strongly on the function f. For standard subset construction, SUBSET($\mathcal{I}$) where $\mathcal{I}$ is the identity function, it is known that the language of N is indeed preserved. We now prove that the same holds for SUBSET(f), for any function f that satisfies $f(P) \equiv_{\mathcal{L}} P$ for every $P \subseteq S_N$. We first establish the following lemma, for which it is important to understand that a set of states P of the NFA N must be regarded as only a single state in the DFA D.

Lemma 3.1. *Let N be an NFA and $f : \wp(S_N) \rightarrow \wp(S_N)$ such that $\mathcal{L}_N(f(Q)) = \mathcal{L}_N(Q)$ for any $Q \subseteq S_N$. Let D be obtained by applying* SUBSET(f) *to N. Then for any $P \in S_D$ it holds that $\mathcal{L}_N(P) = \mathcal{L}_D(P)$.*

Proof. We show set inclusion both ways.

- We prove that $\sigma \in \mathcal{L}_N(P)$ implies $\sigma \in \mathcal{L}_D(P)$ for any $\sigma \in \Sigma^*$ and $P \in S_D$ by induction on the length of σ.

  Base: $\sigma = \varepsilon$. Let $P \in S_D$, and assume $\varepsilon \in \mathcal{L}_N(P)$. Then there exists a $p \in P$ such that $\varepsilon \in \mathcal{L}_N(p)$ and hence $p \in F_N$. By line 12 of algorithm 3.1 we have $P \in F_D$ and thus $\varepsilon \in \mathcal{L}_D(P)$.

Step: $\sigma = a\rho$ for some $a \in \Sigma$ and $\rho \in \Sigma^*$. Let $P \in S_D$ and assume $\sigma \in \mathcal{L}_N(P)$. Then there exists a $p \in P$ such that $\sigma \in \mathcal{L}_N(p)$. Hence there is a $q \in S_N$ such that $p \xrightarrow{a}_N q$ and $\rho \in \mathcal{L}_N(q)$. Since $p \in P$ and $p \xrightarrow{a}_N q$, SUBSET(f) (lines 6–8) ensures that $P \xrightarrow{a}_D f(Q)$ for some $Q \subseteq S_N$ with $q \in Q$. We have $\rho \in \mathcal{L}_N(q) \subseteq \mathcal{L}_N(Q) = \mathcal{L}_N(f(Q))$, and thus, by induction, $\rho \in \mathcal{L}_D(f(Q))$. This implies $a\rho \in \mathcal{L}_D(P)$.

- We prove that $\sigma \in \mathcal{L}_D(P)$ implies $\sigma \in \mathcal{L}_N(P)$ for any $\sigma \in \Sigma^*$ and $P \in S_D$ by induction on the length of σ.

  Base: $\sigma = \varepsilon$. Assume $\varepsilon \in \mathcal{L}_D(P)$. Then $P \in F_D$, so there exists a $p \in P$ such that $p \in F_N$. For this p, it holds that $\varepsilon \in \mathcal{L}_N(p)$. As $\mathcal{L}_N(p) \subseteq \mathcal{L}_N(P)$ it follows that $\varepsilon \in \mathcal{L}_N(P)$.

  Step: $\sigma = a\rho$ for some $a \in \Sigma$ and $\rho \in \Sigma^*$. Let $P \in S_D$ and assume $\sigma \in \mathcal{L}_D(P)$. Then there is a $P' \in S_D$ such that $P \xrightarrow{a}_D P'$ and $\rho \in \mathcal{L}_D(P')$. By induction $\rho \in \mathcal{L}_N(P')$. Let $Q = \{q \in S_N \mid \exists p \in P \,.\, p \xrightarrow{a}_N q\}$, then by construction $P' = f(Q)$. As $\mathcal{L}_N(f(Q)) = \mathcal{L}_N(Q)$, we have $\rho \in \mathcal{L}_N(Q)$. Because $\mathcal{L}_N(Q) = \bigcup_{q \in Q} \mathcal{L}_N(q)$, there is a $q \in Q$ such that $\rho \in \mathcal{L}_N(q)$. For that q, there is a $p \in P$ such that $p \xrightarrow{a}_N q$ and thus $a\rho \in \mathcal{L}_N(p)$. Hence $\sigma \in \mathcal{L}_N(P)$. □

By lemma 3.1 we are now allowed to use language equivalence and inclusion without specifying whether we mean language equivalence (resp. inclusion) between sets of states in an NFA or single states in the generated DFA.

Theorem 3.2. *Let N be an NFA and $f : \wp(S_N) \to \wp(S_N)$ such that $\mathcal{L}_N(f(Q)) = \mathcal{L}_N(Q)$ for any $Q \subseteq S_N$. Let D be obtained by applying* SUBSET(f) *to N. Then D is deterministic and language equivalent to N.*

Proof. It can be easily seen from the algorithm that for each combination of $P \in S_D$ and $a \in \Sigma$, precisely one tuple (P, a, P') for some $P' \in S_D$ is added to $\to_D$. Hence D is deterministic. We derive $D \equiv_{\mathcal{L}} N$, using lemma 3.1:

$$\mathcal{L}(D) = \mathcal{L}_D(i_D) = \mathcal{L}_D(f(\{i_N\})) \overset{3.1}{=} \mathcal{L}_N(f(\{i_N\})) = \mathcal{L}_N(\{i_N\}) = \mathcal{L}_N(i_N) = \mathcal{L}(N). \quad \square$$

In the sequel, whenever we use the term "subset construction" we mean the standard algorithm, *i.e.* SUBSET($\mathcal{I}$). It is known that in the worst case, determinization yields a DFA that is exponentially larger than the input NFA. An example of an NFA that gives rise to such an exponential blow-up is the NFA that accepts the language specified by the regular expression $\Sigma^* x \Sigma^n$ for some alphabet Σ, $x \in \Sigma$ and $n \geq 0$. Figure 3.1(a) shows the NFA for $\Sigma = \{a, b\}$ and $x = a$. This NFA has $n+2$ states; the corresponding DFA has 2^{n+1} states and is already minimal. Note that if the initial state were accepting (figure 3.1(b)), the minimal DFA would consist of only one state with an a, b-loop: the accepted language has become Σ^*. However, subset construction still produces the exponentially larger DFA first, which may then be reduced to obtain the single-state, minimal DFA.

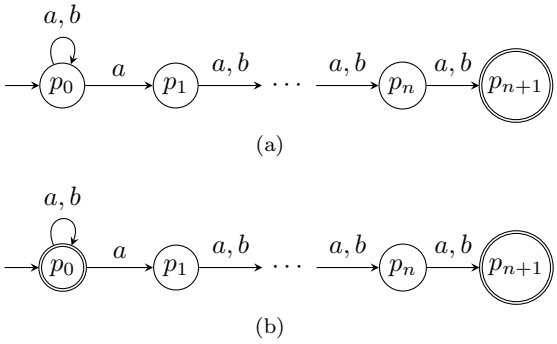


Figure 3.1. Two NFAs of size $\mathcal{O}(n)$ for which subset construction produces a DFA of size $\mathcal{O}(2^n)$. In case (a) this DFA is already minimal; in case (b) the minimal DFA has size 1.

3.2.2 Subset construction using transition sets

In this section we show that subset construction can just as well be done on sets of transitions as on sets of states. Observe that the contribution of an NFA state p to the behaviour of a DFA state P consists entirely of p's outgoing transitions. We therefore consider a DFA state as a set of NFA *transitions*, rather than a set of NFA *states*.

Definition 3.3. Given an NFA N, a *transition tuple* is a pair (T, b) where $T \in \wp(\Sigma \times S_N)$ is a set of transitions and $b \in \mathbb{B}$ is a Boolean.

For every transition tuple (T, b) we define the projection functions set and fin as: $\mathsf{set}(T, b) := T$ and $\mathsf{fin}(T, b) := b$. Given NFA N, for every $p \in S_N$ and $P \subseteq S_N$ we define the corresponding set of outgoing transitions and transition tuple as follows:

$$\mathsf{trans}(p) := \{(a, q) \in \Sigma \times S_N \mid p \xrightarrow{a}_N q\} \qquad \mathsf{tuple}(p) := (\mathsf{trans}(p), p \in F_N)$$
$$\mathsf{trans}(P) := \textstyle\bigcup_{p \in P} \mathsf{trans}(p) \qquad \mathsf{tuple}(P) := (\mathsf{trans}(P), \exists p \in P \,.\, p \in F_N).$$

We need the Boolean b in a transition tuple (T, b) to indicate whether the DFA state is final as this cannot be determined from the elements of T. Only the labels and target states of the transitions are stored because the source states are irrelevant and would only make the sets unnecessarily large.

Given NFA N, for any set of transitions $T \subseteq \Sigma \times S_N$ and $a \in \Sigma$, $\mathsf{img}_a(T)$ is the set of target states of the a-transitions in T:

$$\mathsf{img}_a(T) := \{p \in S_N \mid (a, p) \in T\}\,.$$

The language of a transition $(a, p) \in \Sigma \times S_N$, a set of transitions $T \subseteq \Sigma \times S_N$ and

Algorithm 3.2. The TRANSSET(f) determinization algorithm.

Pre: N is an NFA and $f(P) \equiv_{\mathcal{L}} P$ for all transition tuples P.
Post: D is a DFA such that $N \equiv_{\mathcal{L}} D$.

```
1: →_D, F_D, i_D := ∅, ∅, f(tuple(i_N));
2: S_D, todo, done := {i_D}, {i_D}, ∅;
3: while todo ≠ ∅ do
4:     pick a P ∈ todo;
5:     for all a ∈ Σ do
6:         P' := f(tuple(img_a(set(P))));
7:         S_D := S_D ∪ {P'};
8:         →_D := →_D ∪ {(P, a, P')};
9:         todo := todo ∪ ({P'} \ done)
10:    end for;
11:    if fin(P) then
12:        F_D := F_D ∪ {P}
13:    end if;
14:    todo, done := todo \ {P}, done ∪ {P}
15: end while
```

a transition tuple $(T', b) \in \wp(\Sigma \times S_N) \times \mathbb{B}$ are defined as follows:

$$\begin{aligned}
\mathcal{L}_N(a,p) &:= \{a\sigma \in \Sigma^* \mid \sigma \in \mathcal{L}_N(p)\} \\
\mathcal{L}_N(T) &:= \textstyle\bigcup_{t\in T} \mathcal{L}_N(t) \\
\mathcal{L}_N(T',b) &:= \mathcal{L}_N(T') \cup \begin{cases} \{\varepsilon\} & \text{if } b \\ \emptyset & \text{otherwise.} \end{cases}
\end{aligned}$$

Language inclusion and equivalence for transitions and transition tuples can now be defined in the usual way by means of set inclusion and equality.

The determinization algorithm that uses transition tuples is algorithm 3.2. We shall refer to it as TRANSSET(f) where f is a function on transition tuples. Again, language preservation depends on the specific function f being used. We now prove that this is the case when f satisfies $f(P) \equiv_{\mathcal{L}} P$ for each transition tuple P.

Proposition 3.4. *Given NFA N, for any $p \in S_N$: $p \equiv_{\mathcal{L}} \mathsf{tuple}(p)$.*

Proof. Let $p \in S_N$ and $S = \{\varepsilon\}$ if $p \in F_N$ and $S = \emptyset$ otherwise. We derive:

$$\begin{aligned}
\mathcal{L}_N(p) &= \{\sigma \in \Sigma^* \mid \exists q \in F_N \,.\, p \xrightarrow{\sigma}_N q\} \\
&= \{a\sigma \in \Sigma^* \mid \exists q \in F_N \,.\, p \xrightarrow{a\sigma}_N q\} \cup S \\
&= \{a\sigma \in \Sigma^* \mid \exists q \in S_N \,.\, p \xrightarrow{a}_N q \wedge \sigma \in \mathcal{L}_N(q)\} \cup S \\
&= \{a\sigma \in \Sigma^* \mid \exists q \in S_N \,.\, (a,q) \in \mathsf{trans}(p) \wedge \sigma \in \mathcal{L}_N(q)\} \cup S \\
&= \textstyle\bigcup_{(a,q)\in\mathsf{trans}(p)} \{a\sigma \in \Sigma^* \mid \sigma \in \mathcal{L}_N(q)\} \cup S \\
&= \textstyle\bigcup_{t\in\mathsf{trans}(p)} \mathcal{L}_N(t) \cup S \;=\; \mathcal{L}_N(\mathsf{trans}(p)) \cup S \;=\; \mathcal{L}_N(\mathsf{tuple}(p))\,.
\end{aligned}$$

□

Note that proposition 3.4 implies that $P \equiv_{\mathcal{L}} \mathsf{tuple}(P)$ for all $P \subseteq S_N$. In order to prove the correctness of $\textsc{Transset}(f)$ for suitable functions f, we need the following lemma, in which a transition tuple (T, b) is regarded as such in the input NFA N, but is regarded as a single state in the output NFA D.

Lemma 3.5. *Let N be an NFA and $f : \wp(\Sigma \times S_N) \times \mathbb{B} \to \wp(\Sigma \times S_N) \times \mathbb{B}$ such that $\mathcal{L}_N(f(P)) = \mathcal{L}_N(P)$ for all transition tuples P. Let D be obtained by applying $\textsc{Transset}(f)$ to N. Then for any $(T, b) \in S_D$, it holds that $\mathcal{L}_N(T, b) = \mathcal{L}_D(T, b)$.*

Proof. We show that $\sigma \in \mathcal{L}_N(T, b) \Leftrightarrow \sigma \in \mathcal{L}_D(T, b)$ for any $\sigma \in \Sigma^*$ and $(T, b) \in S_D$ by induction on the length of σ.

Base: $\sigma = \varepsilon$. Let $(T, b) \in S_D$. Then $\varepsilon \in \mathcal{L}_N(T, b) \Leftrightarrow b \overset{*}{\Leftrightarrow} (T, b) \in F_D \Leftrightarrow \varepsilon \in \mathcal{L}_D(T, b)$, where $\overset{*}{\Leftrightarrow}$ follows from lines 11–13 of $\textsc{Transset}(f)$.

Step: $\sigma = a\rho$ for some $a \in \Sigma$ and $\rho \in \Sigma^*$. We assume for any $(T', b') \in S_D$:

$$\text{(IH)} \qquad \rho \in \mathcal{L}_N(T', b') \Leftrightarrow \rho \in \mathcal{L}_D(T', b')$$

and derive, for any $(T, b) \in S_D$:

$$\begin{aligned} &a\rho \in \mathcal{L}_N(T, b) \Leftrightarrow a\rho \in \mathcal{L}_N(T) \Leftrightarrow a\rho \in \textstyle\bigcup_{t \in T} \mathcal{L}_N(t) \Leftrightarrow a\rho \in \bigcup_{p \in \mathsf{img}_a(T)} \mathcal{L}_N(a, p) \\ &\Leftrightarrow \rho \in \textstyle\bigcup_{p \in \mathsf{img}_a(T)} \mathcal{L}_N(p) \Leftrightarrow \rho \in \mathcal{L}_N(\mathsf{img}_a(T)) \overset{*}{\Leftrightarrow} \rho \in \mathcal{L}_N(\mathsf{tuple}(\mathsf{img}_a(T))) \\ &\Leftrightarrow \rho \in \mathcal{L}_N(f(\mathsf{tuple}(\mathsf{img}_a(T)))) \overset{\dagger}{\Leftrightarrow} \rho \in \mathcal{L}_D(f(\mathsf{tuple}(\mathsf{img}_a(T)))) \overset{\ddagger}{\Leftrightarrow} a\rho \in \mathcal{L}_D(T, b) \end{aligned}$$

where at $*$ we used that $P \equiv_{\mathcal{L}} \mathsf{tuple}(P)$ for all $P \subseteq S_N$, which follows from proposition 3.4; at $\dagger$ we used (IH) and the fact that lines 6–7 of $\textsc{Transset}(f)$ ensure that $f(\mathsf{tuple}(\mathsf{img}_a(T))) \in S_D$; and, finally, at $\ddagger$ we used that line 8 of $\textsc{Transset}(f)$ ensures that $(T, b) \xrightarrow{a}_D f(\mathsf{tuple}(\mathsf{img}_a(T)))$. □

Using this lemma we can prove the main theorem, which states correctness of $\textsc{Transset}(f)$ for functions f satisfying $f(P) \equiv_{\mathcal{L}} P$ for all transition tuples P.

Theorem 3.6. *Let N be an NFA and $f : \wp(\Sigma \times S_N) \times \mathbb{B} \to \wp(\Sigma \times S_N) \times \mathbb{B}$ such that $\mathcal{L}_N(f(P)) = \mathcal{L}_N(P)$ for all transition tuples P. Let D be obtained by applying $\textsc{Transset}(f)$ to N. Then D is deterministic and language equivalent to N.*

Proof. It can be easily seen from the algorithm that for each combination of $P \in S_D$ and $a \in \Sigma$, precisely one tuple (P, a, P') for some $P' \in S_D$ is added to $\to_D$. Hence D is deterministic. We derive $D \equiv_{\mathcal{L}} N$ using lemma 3.5 and proposition 3.4 as follows:

$$\begin{aligned} \mathcal{L}(D) \;&=\; \mathcal{L}_D(i_D) = \mathcal{L}_D(f(\mathsf{tuple}(i_N))) \overset{3.5}{=} \mathcal{L}_N(f(\mathsf{tuple}(i_N))) = \mathcal{L}_N(\mathsf{tuple}(i_N)) \\ &\overset{3.4}{=} \mathcal{L}_N(i_N) = \mathcal{L}(N)\,. \end{aligned}$$

□

As the identity function $\mathcal{I}$ trivially satisfies $\mathcal{L}_N(P) = \mathcal{L}_N(\mathcal{I}(P))$ for all P, the following result follows immediately from theorem 3.6.

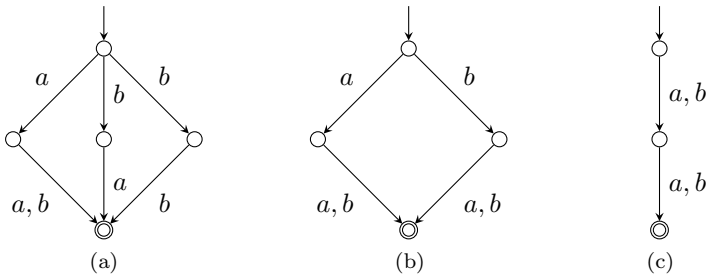


Figure 3.2. NFA (a) for which the DFA produced by Subset($\mathcal{I}$) (b) is larger than the (minimal) DFA produced by Transset($\mathcal{I}$) (c).

Corollary 3.7. *Let D be the automaton obtained by applying* Transset($\mathcal{I}$) *to an NFA N. Then D is deterministic and language equivalent to N.* □

Using Transset($\mathcal{I}$) for determinization can give a smaller DFA than Subset($\mathcal{I}$) as is shown by the example in figure 3.2. Here, Transset($\mathcal{I}$) happens to produce the minimal DFA directly. However, this is generally not the case: when applied to the NFA of figure 3.1(b), Transset($\mathcal{I}$) generates a DFA of size 2^{n+1}, while the minimal DFA has size 1.

3.2.3 Closure with language preorder

We introduce a *closure* operation that can be used in the Subset algorithm instead of the identity function $\mathcal{I}$. It aims to add NFA states to a given DFA state (*i.e.* a set of NFA states) without affecting its language. The set of states that are added is determined by a relation $\sqsubseteq$. We shall instantiate $\sqsubseteq$ by the language preorder and the simulation preorder, resulting in algorithms that generate smaller DFAs than Subset($\mathcal{I}$). In particular, we show that if the language preorder is used for $\sqsubseteq$, Subset with closure is an algorithm that produces the minimal DFA directly.

Definition 3.8. For any set of states $P \subseteq S_N$ of an NFA N and relation $\sqsubseteq \subseteq S_N \times \wp(S_N)$, the *closure* of P under $\sqsubseteq$, $\mathsf{close}_{\sqsubseteq}(P)$, is defined as:

$$\mathsf{close}_{\sqsubseteq}(P) := \{p \in S_N \mid p \sqsubseteq P\}.$$

Applying this, the algorithm Subset($\mathsf{close}_{\sqsubseteq_{\mathcal{L}}}$) generates the minimal DFA that is language equivalent to the input NFA, which we prove below. For any set of states P in an NFA N, $\mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(P)$ contains all states that are language included in P. In particular, because $p \sqsubseteq_{\mathcal{L}} P$ for all $p \in P$, we have that $P \subseteq \mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(P)$.

Proposition 3.9. *For any NFA N and $P \subseteq S_N$, it holds that $P \equiv_{\mathcal{L}} \mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(P)$.*

Proof. Let $Q := \mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(P)$. We show that $P \sqsubseteq_{\mathcal{L}} Q$ and $Q \sqsubseteq_{\mathcal{L}} P$.

- $P \sqsubseteq_{\mathcal{L}} Q$: using the fact that $P \subseteq Q$, we derive: $\mathcal{L}_N(P) = \bigcup_{p \in P} \mathcal{L}_N(p) \subseteq \bigcup_{p \in Q} \mathcal{L}_N(p) = \mathcal{L}_N(Q)$, hence $P \sqsubseteq_{\mathcal{L}} Q$;
- $Q \sqsubseteq_{\mathcal{L}} P$: for all $p \in Q$ we know (by definition) that $p \sqsubseteq_{\mathcal{L}} P$, *i.e.* $\mathcal{L}_N(p) \subseteq \mathcal{L}_N(P)$. Thus: $\mathcal{L}_N(Q) = \bigcup_{p \in Q} \mathcal{L}_N(p) \subseteq \mathcal{L}_N(P)$, hence $Q \sqsubseteq_{\mathcal{L}} P$. □

Theorem 3.10. *Given an NFA N,* Subset($\mathsf{close}_{\sqsubseteq_{\mathcal{L}}}$) *constructs the minimal DFA that is language equivalent to N.*

Proof. It follows immediately from theorem 3.2 and proposition 3.9 that the automaton D constructed by Subset($\mathsf{close}_{\sqsubseteq_{\mathcal{L}}}$) is a DFA that is language equivalent to N. D is minimal if there is no DFA that is language equivalent to D and has fewer states than D. It is known that this follows directly if there is no pair of different states in D that are language equivalent (see for instance corollary 4.24 of [48]). Suppose D contains states P and Q such that $P \equiv_{\mathcal{L}} Q$ and for some $T, U \subseteq S_N$, $P = \mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(T)$ and $Q = \mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(U)$. Then for all $p \in P$ we have $p \sqsubseteq_{\mathcal{L}} P \equiv_{\mathcal{L}} Q = \mathsf{close}_{\sqsubseteq_{\mathcal{L}}}(U) \equiv_{\mathcal{L}} U$, by proposition 3.9. Because $Q = \{q \in S_N \mid q \sqsubseteq_{\mathcal{L}} U\}$ we see that $p \in Q$ and thus that $P \subseteq Q$. By symmetry we also have that $Q \subseteq P$. Hence $P = Q$. □

3.2.4 Closure with simulation preorder

Although it ensures that the output DFA of Subset($\mathsf{close}_{\sqsubseteq_{\mathcal{L}}}$) is minimal, language inclusion is an unattractive preorder to use. Deciding language inclusion is PSPACE-complete [115] which implies that known algorithms have an exponential time complexity. Moreover, most algorithms involve a determinization step which would render our optimization useless.

As mentioned in section 2.4.3 the simulation preorder is finer than language inclusion on NFAs, meaning that it relates fewer NFAs. However, considering its PTIME complexity (see *e.g.* [11, 72] and chapter 4), it is an attractive way to "approximate" language inclusion (see also [36]). Hence, as a more practical alternative to Subset($\mathsf{close}_{\sqsubseteq_{\mathcal{L}}}$) we define the algorithm Subset($\mathsf{close}_{\subseteq}$). The required lifting of $\subseteq$ to states and sets of states is as follows. For any state $p \in S_N$ and set of states $P \subseteq S_N$ of an NFA N, we have $p \subseteq P$ iff:

- if $p \in F_N$ then $\exists q \in P \,.\, q \in F_N$, and
- if $p \xrightarrow{a}_N p'$ then $\exists q \in P, q' \in S_N \,.\, q \xrightarrow{a}_N q' \wedge p' \subseteq q'$.

The following property states that $\mathsf{close}_{\subseteq}$ maintains the language of a set of states.

Proposition 3.11. *For any NFA N and $P \subseteq S_N$, it holds that $P \equiv_{\mathcal{L}} \mathsf{close}_{\subseteq}(P)$.*

Proof. Let $Q := \mathsf{close}_{\subseteq}(P) = \{q \in S_N \mid q \subseteq P\}$. We show that $P \equiv_{\mathcal{L}} Q$:

- $P \sqsubseteq_{\mathcal{L}} Q$: this follows immediately from $P \subseteq Q$;
- $Q \sqsubseteq_{\mathcal{L}} P$: we show that $\sigma \in \mathcal{L}_N(Q)$ implies $\sigma \in \mathcal{L}_N(P)$ by induction on the length of $\sigma \in \Sigma^*$.

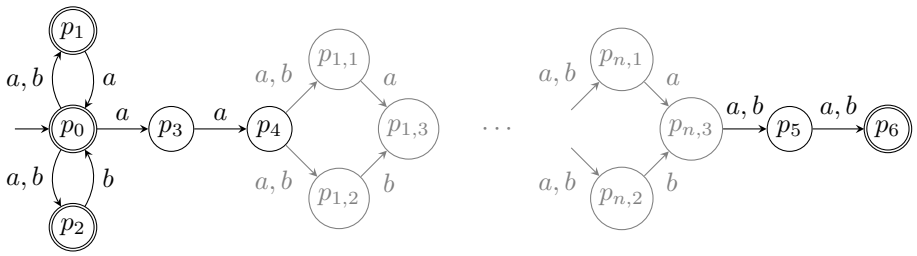


Figure 3.3. NFA of size $\mathcal{O}(n)$ for which Subset($\mathsf{close}_{\subseteq}$) generates a DFA of size $\mathcal{O}(2^n)$ for any $n \geq 1$. The minimal DFA has 1 state.

Base: $\sigma = \varepsilon$. If $\varepsilon \in \mathcal{L}_N(Q)$ then $\varepsilon \in \mathcal{L}_N(q)$ for some $q \in Q$. Hence $q \in F_N$. It must be that $q \subseteq P$. Thus $\exists p \in P.\ p \in F_N$ and therefore $\varepsilon \in \mathcal{L}_N(p) \subseteq \mathcal{L}_N(P)$.
Step: $\sigma = a\rho$. If $a\rho \in \mathcal{L}_N(Q)$ then there is a $q \in Q$ and $q' \in S_N$ such that $q \xrightarrow{a}_N q'$ and $\rho \in \mathcal{L}_N(q')$. It must be that $q \subseteq P$, so there exists a simulation $R \subseteq S_N \times S_N$ such that $\exists p \in P, p' \in S_N.\ p \xrightarrow{a}_N p' \wedge q'\ R\ p'$. Therefore, $q' \subseteq p'$ and hence $q' \sqsubseteq_{\mathcal{L}} p'$, so $\rho \in \mathcal{L}_N(p')$. It follows that $a\rho \in \mathcal{L}_N(P)$. □

Theorem 3.12. *Given an NFA N,* Subset($\mathsf{close}_{\subseteq}$) *constructs a DFA that is language equivalent to N.*

Proof. Immediate from theorem 3.2 and proposition 3.11. □

The example in figure 3.3 shows not only that the resulting DFA is no longer minimal, but moreover that it can be exponentially larger than the minimal DFA. The example NFA contains a pattern that repeats itself n times for any $n \geq 1$. It is based on the NFA of figure 3.1(b) interwoven with a pattern that prevents Subset($\mathsf{close}_{\subseteq}$) from merging states that in the end will turn out to be equivalent. The NFA accepts the language given by the regular expression $(a + b)^*$.

3.2.5 Compression on state sets

Compared to Subset($\mathcal{I}$), Subset($\mathsf{close}_{\subseteq}$) adds all simulated states to every generated set of states. Another option would be to remove all redundant states from such a set. More specifically, we remove every state that is simulated by another state in the set. For this operation to be well-defined, it is essential that no two different states in the set are simulation equivalent. This can be achieved by minimizing the input NFA using simulation equivalence prior to determinization. In turn, this amounts to computing the simulation preorder that was already necessary in the first place.

Definition 3.13. Given a set P such that $\neg\exists p, q \in P\,.\,p \neq q \wedge p \rightleftarrows q$. Then $\mathsf{compress}_{\subseteq}(P)$ denotes the *compression* of P under $\subseteq$ and is defined as:

$$\mathsf{compress}_{\subseteq}(P) := \{p \in P \mid \forall q \in P\,.\,p \neq q \Longrightarrow p \not\subseteq q\}.$$

This way, we obtain the determinization algorithm SUBSET($\mathsf{compress}_{\subseteq}$). We prove its correctness assuming that the input NFA N is minimal under simulation equivalence. The correctness follows readily from the following property.

Proposition 3.14. *For any $P \subseteq S_N$, it holds that $P \equiv_{\mathcal{L}} \mathsf{compress}_{\subseteq}(P)$.*

Proof. Let $Q := \mathsf{compress}_{\subseteq}(P)$. We show that $P \sqsubseteq_{\mathcal{L}} Q$ and $Q \sqsubseteq_{\mathcal{L}} P$:

- $P \sqsubseteq_{\mathcal{L}} Q$: for any $p \in P$ there is a $q \in Q$ with $p \subseteq q$ and hence $p \sqsubseteq_{\mathcal{L}} q$. Thus $\mathcal{L}_N(P) = \bigcup_{p \in P} \mathcal{L}_N(p) \subseteq \bigcup_{q \in Q} \mathcal{L}_N(q) = \mathcal{L}_N(Q)$, so $P \sqsubseteq_{\mathcal{L}} Q$;
- $Q \sqsubseteq_{\mathcal{L}} P$: this follows immediately from $Q \subseteq P$. □

Theorem 3.15. *When applied to N, SUBSET($\mathsf{compress}_{\subseteq}$) constructs a DFA that is language equivalent to N.*

Proof. Immediate from theorem 3.2 and proposition 3.14. □

3.2.6 Compression on transition sets

The function $\mathsf{compress}_{\subseteq}$ can be used not only for sets of states but also for transition tuples. For that, we first define $\subseteq$ on the transitions of an NFA N as follows. For any $(a,p),(b,q) \in \Sigma \times S_N$, we have $(a,p) \subseteq (b,q)$ iff $a = b$ and $p \subseteq q$. By definition 3.13, $\mathsf{compress}_{\subseteq}$ is now properly defined on sets of transitions and it can be extended to transition tuples in a straightforward manner: $\mathsf{compress}_{\subseteq}(T,b) = (\mathsf{compress}_{\subseteq}(T), b)$. This way, we obtain the determinization algorithm TRANSSET($\mathsf{compress}_{\subseteq}$). For proving correctness, we again assume that the input NFA N is minimal under simulation equivalence.

Proposition 3.16. *For any transition tuple (T,b): $(T,b) \equiv_{\mathcal{L}} \mathsf{compress}_{\subseteq}(T,b)$.*

Proof. Let $U := \mathsf{compress}_{\subseteq}(T)$. We have to show that:

$$\mathcal{L}_N(T) \cup \begin{cases} \{\varepsilon\} & \text{if } b \\ \emptyset & \text{if } \neg b \end{cases} \quad = \quad \mathcal{L}_N(U) \cup \begin{cases} \{\varepsilon\} & \text{if } b \\ \emptyset & \text{if } \neg b \end{cases}$$

which follows naturally if $\mathcal{L}_N(T) = \mathcal{L}_N(U)$. For any $t \in T$ there is a $u \in U$ with $t \subseteq u$ and hence $t \sqsubseteq_{\mathcal{L}} u$. Thus: $\mathcal{L}_N(T) = \bigcup_{t \in T} \mathcal{L}_N(t) \subseteq \bigcup_{t \in U} \mathcal{L}_N(t) = \mathcal{L}_N(U)$. Because $U \subseteq T$, we also have that $\mathcal{L}_N(U) \subseteq \mathcal{L}_N(T)$. Hence $\mathcal{L}_N(T) = \mathcal{L}_N(U)$. □

Theorem 3.17. *Let D be obtained by applying TRANSSET($\mathsf{compress}_{\subseteq}$) to N. Then D is deterministic and language equivalent to N.*

Proof. Immediate from theorem 3.6 and proposition 3.16. □

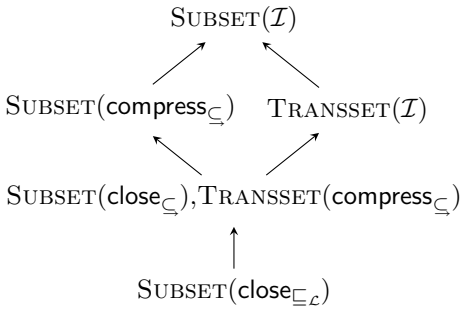


Figure 3.4. The lattice of algorithms presented in the previous sections.

3.3 Lattice of algorithms

We order the algorithms described in the previous section in a lattice based on the number of states in the resulting DFA. More precisely, for any algorithm A and NFA N, let $A(N)$ be the DFA produced by A on input N. We define the ordering $\preceq$ on the algorithms as follows: $A \preceq B$ iff for every NFA N, $|S_{A(N)}| \leq |S_{B(N)}|$. Figure 3.4 shows the lattice, where an arrow from A to B denotes that $A \preceq B$.

Subset(close$_{\subseteq}$) and Transset(compress$_{\subseteq}$) are in the same class of the lattice, because these algorithms always yield isomorphic DFAs. This statement is substantiated below, as well as the validity of the other $\preceq$-relations of figure 3.4. The following shows that there are no further $\preceq$-relations between our algorithms:

- Subset(close$_{\sqsubseteq_{\mathcal{L}}}$) is the unique $\preceq$-smallest algorithm because it is the only one that always generates the minimal DFA;
- Subset(compress$_{\subseteq}$) $\not\preceq$ Transset($\mathcal{I}$) by the example in figure 3.2;
- Transset($\mathcal{I}$) $\not\preceq$ Subset(compress$_{\subseteq}$) by the example in figure 3.1(b).

For the remainder of this section, we fix an NFA N that is minimal under simulation equivalence. For $P \subseteq S_N$ and $a \in \Sigma$ let

$$P/a := \{q \in S_N \mid \exists p \in P.\ p \xrightarrow{a}_N q\}\,,$$

so that line 6 of algorithm 3.1 can be rewritten as $P' := f(P/a)$. It follows that $S_{\text{Subset}(f,N)}$ is the smallest set S satisfying, for all $P \in \wp(S_N)$:

$$f(\{i_N\}) \in S \ \wedge\ (P \in S \Longrightarrow \forall a \in \Sigma\,.\, f(P/a) \in S)\,. \tag{3.1}$$

Likewise, for any transition tuple P and $a \in \Sigma$, let

$$P /\!\!/ a := \mathsf{tuple}(\mathsf{img}_a(\mathsf{set}(P)))\,,$$

so that line 6 of algorithm 3.2 can be rewritten as $P' := f(P /\!\!/ a)$. It follows that

$S_{\text{TRANSSET}(f,N)}$ is the smallest set S satisfying, for all transition tuples P:

$$f(\mathsf{tuple}(i_N)) \in S \ \wedge \ (P \in S \Longrightarrow \forall a \in \Sigma \,.\, f(P/\!\!/a) \in S)\,. \tag{3.2}$$

We extend the operator $\mathsf{close}_{\subseteq}$, originally defined on sets of states, to transition tuples. The result of applying the operator remains a set of states. For any state $p \in S_N$ and transition tuple (T, b) we write $p \subseteq (T, b)$ iff:

- if $p \in F_N$ then b, and
- if $p \xrightarrow{a}_N p'$ then $\exists q' \in S_N \,.\, (a, q') \in T \wedge p' \subseteq q'$.

For any transition tuple P we define $\mathsf{close}_{\subseteq}(P) := \{p \in S_N \mid p \subseteq P\}$.

Lemma 3.18. *Let $a \in \Sigma$ and $P \subseteq S_N$. Then*

1. $\mathsf{tuple}(P)/\!\!/a = \mathsf{tuple}(P/a)$,
2. $\mathsf{compress}_{\subseteq}(P/a) \subseteq \mathsf{compress}_{\subseteq}(P)/a$,
3. $\mathsf{compress}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)/a) = \mathsf{compress}_{\subseteq}(P/a)$,
4. $\mathsf{close}_{\subseteq}(P/a) \supseteq \mathsf{close}_{\subseteq}(P)/a$,
5. $\mathsf{close}_{\subseteq}(\mathsf{close}_{\subseteq}(P)/a) = \mathsf{close}_{\subseteq}(P/a)$,
6. $\mathsf{close}_{\subseteq}(P) = \mathsf{close}_{\subseteq}(\mathsf{compress}_{\subseteq}(P))$,
7. $\mathsf{close}_{\subseteq}(P) = \mathsf{close}_{\subseteq}(\mathsf{tuple}(P))$,
8. $\mathsf{compress}_{\subseteq}(\mathsf{tuple}(P)) = \mathsf{compress}_{\subseteq}(\mathsf{tuple}(\mathsf{close}_{\subseteq}(P)))$.

Proof.

1. We derive:
$$\begin{aligned}\mathsf{tuple}(P)/\!\!/a &= \mathsf{tuple}(\mathsf{img}_a(\mathsf{set}(\mathsf{tuple}(P)))) \ = \ \mathsf{tuple}(\mathsf{img}_a(\mathsf{trans}(P)))\\ &= \mathsf{tuple}(\{q \in S_N \mid (a,q) \in \mathsf{trans}(P)\})\\ &= \mathsf{tuple}(\{q \in S_N \mid (a,q) \in \textstyle\bigcup_{p\in P} \mathsf{trans}(p)\})\\ &= \mathsf{tuple}(\{q \in S_N \mid \exists p \in P \,.\, p \xrightarrow{a}_N q\}) \ = \ \mathsf{tuple}(P/a)\,.\end{aligned}$$
2. Let $p' \in \mathsf{compress}_{\subseteq}(P/a)$. Then $p' \in P/a$, so $\exists p \in P.\ p \xrightarrow{a}_N p'$. In fact, the set of all $p \in P$ with $p \xrightarrow{a}_N p'$ is ordered by $\subseteq$, and we take a $\subseteq$-maximal p within that set. Suppose, by contradiction, that $\exists q \in P.\ q \neq p \wedge p \subseteq q$. Take that q. Then, by definition of simulation, $\exists q' \in S_N.\ q \xrightarrow{a}_N q' \wedge p' \subseteq q'$, so $q' \in P/a$. By construction, $q' \neq p'$. Hence $p' \notin \mathsf{compress}_{\subseteq}(P/a)$. So $\neg\exists q \in P.\ q \neq p \wedge p \subseteq q$, and thus $p \in \mathsf{compress}_{\subseteq}(P)$. Hence $p' \in \mathsf{compress}_{\subseteq}(P)/a$.
3. By lemma 3.18.2 we have: $\mathsf{compress}_{\subseteq}(P/a) = \mathsf{compress}_{\subseteq}(\mathsf{compress}_{\subseteq}(P/a)) \subseteq \mathsf{compress}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)/a)$. As $\mathsf{compress}_{\subseteq}(P) \subseteq P$ we have $\mathsf{compress}_{\subseteq}(P)/a \subseteq P/a$. Hence $\mathsf{compress}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)/a) \subseteq \mathsf{compress}_{\subseteq}(P/a)$.
4. Let $p' \in \mathsf{close}_{\subseteq}(P)/a$, then $\exists p \in \mathsf{close}_{\subseteq}(P).\ p \xrightarrow{a}_N p'$. Take that p and observe that $p \subseteq P$. Hence, $\exists q \in P, q' \in S_N \,.\, q \xrightarrow{a}_N q' \wedge p' \subseteq q'$. Therefore $q' \in P/a$, which combined with $p' \subseteq q'$ gives $p' \subseteq P/a$ and thus $p' \in \mathsf{close}_{\subseteq}(P/a)$.

5. By lemma 3.18.4 we have:

$$\mathsf{close}_{\subseteq}(P/a) = \mathsf{close}_{\subseteq}(\mathsf{close}_{\subseteq}(P/a)) \supseteq \mathsf{close}_{\subseteq}(\mathsf{close}_{\subseteq}(P)/a).$$

   Moreover, $P \subseteq \mathsf{close}_{\subseteq}(P)$ so $P/a \subseteq \mathsf{close}_{\subseteq}(P)/a$ and thus $\mathsf{close}_{\subseteq}(P/a) \subseteq \mathsf{close}_{\subseteq}(\mathsf{close}_{\subseteq}(P)/a)$.
6. As $\mathsf{compress}_{\subseteq}(P) \subseteq P$, we have $\mathsf{close}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)) \subseteq \mathsf{close}_{\subseteq}(P)$. For the reverse inclusion, note that $p \in P$ implies $\exists q \in \mathsf{compress}_{\subseteq}(P) \,.\, p \subseteq q$. From this it easily follows that $p \subseteq P$ implies $p \subseteq \mathsf{compress}_{\subseteq}(P)$, which in turn yields $\mathsf{close}_{\subseteq}(P) \subseteq \mathsf{close}_{\subseteq}(\mathsf{compress}_{\subseteq}(P))$.
7. This can be restated as $p \subseteq P \Leftrightarrow p \subseteq \mathsf{tuple}(P)$, which follows directly from the definitions.
8. Let $T := \mathsf{set}(\mathsf{tuple}(P))$ and $T_c := \mathsf{set}(\mathsf{tuple}(\mathsf{close}_{\subseteq}(P)))$. We have to prove:

   (i) $\exists p \in P.\ p \in F_N \;\Leftrightarrow\; \exists p \in \mathsf{close}_{\subseteq}(P).\ p \in F_N$

   (ii) $\mathsf{compress}_{\subseteq}(T) = \mathsf{compress}_{\subseteq}(T_c)$.

   Statement (i) follows directly from $P \subseteq \mathsf{close}_{\subseteq}(P)$ and the definitions. Regarding (ii), $\mathsf{compress}_{\subseteq}(T) \subseteq \mathsf{compress}_{\subseteq}(T_c)$ because $T \subseteq T_c$. Take $(a, p') \in T_c \setminus T$. Then $p \xrightarrow{a}_N p'$ for some $p \subsetneq P$. Hence $\exists q \in P, q' \in S_N \,.\, q \xrightarrow{a}_N q' \wedge p' \subseteq q'$. Therefore $(a, q') \in T$, so $(a, p') \neq (a, q')$ and $(a, p') \subseteq (a, q')$. Hence $(a, p') \notin \mathsf{compress}_{\subseteq}(T_c)$. This implies that $\mathsf{compress}_{\subseteq}(T_c) \subseteq \mathsf{compress}_{\subseteq}(T)$. □

We now establish some of the $\preceq$-relations between the algorithms.

Theorem 3.19. $\textsc{Transset}(\mathcal{I}) \preceq \textsc{Subset}(\mathcal{I})$.

Proof. A straightforward induction on $S_{\textsc{Subset}(\mathcal{I},N)}$ (see (3.1) for the inductive characterization) using lemma 3.18.1 at the induction step yields:

$$S_{\textsc{Transset}(\mathcal{I},N)} = \{\mathsf{tuple}(P) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\}\,. \tag{3.3}$$

This immediately implies that $|S_{\textsc{Transset}(\mathcal{I},N)}| \leq |S_{\textsc{Subset}(\mathcal{I},N)}|$. (Note that we do not have an equality here, because it might be that $\mathsf{tuple}(P) = \mathsf{tuple}(Q)$ for different $P, Q \in S_{\textsc{Subset}(\mathcal{I},N)}$.) Hence $\textsc{Transset}(\mathcal{I}) \preceq \textsc{Subset}(\mathcal{I})$. □

Theorem 3.20. $\textsc{Subset}(\mathsf{compress}_{\subseteq}) \preceq \textsc{Subset}(\mathcal{I})$.

Proof. An induction on $S_{\textsc{Subset}(\mathcal{I},N)}$ (see (3.1)) using lemma 3.18.3 yields:

$$S_{\textsc{Subset}(\mathsf{compress}_{\subseteq},N)} = \{\mathsf{compress}_{\subseteq}(P) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\}\,. \tag{3.4}$$

Hence $|S_{\textsc{Subset}(\mathsf{compress}_{\subseteq},N)}| \leq |S_{\textsc{Subset}(\mathcal{I},N)}|$. □

Theorem 3.21. $\textsc{Subset}(\mathsf{close}_{\subseteq}) \preceq \textsc{Subset}(\mathsf{compress}_{\subseteq})$.

Proof. An induction on $S_{\textsc{Subset}(\mathcal{I},N)}$ (see (3.1)) using lemma 3.18.5 yields:

$$S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)} = \{\mathsf{close}_{\subseteq}(P) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\}\,. \tag{3.5}$$

We derive using (3.4) and lemma 3.18.6:

$$\begin{aligned} S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)} &\overset{(3.5)}{=} \{\mathsf{close}_{\subseteq}(P) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\} \\ &\overset{3.18.6}{=} \{\mathsf{close}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\} \\ &\overset{(3.4)}{=} \{\mathsf{close}_{\subseteq}(P) \mid P \in S_{\textsc{Subset}(\mathsf{compress}_{\subseteq},N)}\}\,. \end{aligned}$$

Hence $|S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)}| \leq |S_{\textsc{Subset}(\mathsf{compress}_{\subseteq},N)}|$. □

In the following lemma we employ a relation $\leq$ on transition tuples defined by $(T,b) \leq (T',b')$ iff $T \subseteq T'$ and $b \Longrightarrow b'$.

Lemma 3.22. *Let $a \in \Sigma$ and P be a transition tuple. Then*

1. $\mathsf{compress}_{\subseteq}(P/\!\!/a) \leq \mathsf{compress}_{\subseteq}(P)/\!\!/a$,
2. $\mathsf{compress}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)/\!\!/a) = \mathsf{compress}_{\subseteq}(P/\!\!/a)$,
3. $\mathsf{close}_{\subseteq}(P) = \mathsf{close}_{\subseteq}(\mathsf{compress}_{\subseteq}(P))$.

Proof. Similar to the proof of lemmas 3.18.2, 3.18.3 and 3.18.6, respectively. □

We can now establish the remaining $\preceq$-relations between the algorithms, which completes the correctness proof of the lattice of figure 3.4.

Theorem 3.23. $\textsc{Transset}(\mathsf{compress}_{\subseteq}) \preceq \textsc{Transset}(\mathcal{I})$.

Proof. An induction on $S_{\textsc{Transset}(\mathcal{I},N)}$ (see (3.2)) using lemma 3.22.2 yields:

$$S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)} = \{\mathsf{compress}_{\subseteq}(P) \mid P \in S_{\textsc{Transset}(\mathcal{I},N)}\}. \tag{3.6}$$

Hence $|S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)}| \leq |S_{\textsc{Transset}(\mathcal{I},N)}|$. □

Theorem 3.24. $\textsc{Transset}(\mathsf{compress}_{\subseteq}) \preceq \textsc{Subset}(\mathsf{close}_{\subseteq})$.

Proof. From (3.3), (3.5), (3.6) and lemma 3.18.8 we obtain:

$$\begin{aligned} S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)} &\overset{(3.6)}{=} \{\mathsf{compress}_{\subseteq}(P) \mid P \in S_{\textsc{Transset}(\mathcal{I},N)}\} \\ &\overset{(3.3)}{=} \{\mathsf{compress}_{\subseteq}(\mathsf{tuple}(P)) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\} \\ &\overset{3.18.8}{=} \{\mathsf{compress}_{\subseteq}(\mathsf{tuple}(\mathsf{close}_{\subseteq}(P))) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\} \\ &\overset{(3.5)}{=} \{\mathsf{compress}_{\subseteq}(\mathsf{tuple}(P)) \mid P \in S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)}\}\,. \end{aligned}$$

Hence $|S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)}| \leq |S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)}|$. □

Theorem 3.25. $\textsc{Subset}(\mathsf{close}_{\subseteq}) \preceq \textsc{Transset}(\mathsf{compress}_{\subseteq})$.

Proof. From (3.3), (3.5), (3.6) and lemmas 3.18.7 and 3.22.3 we obtain:

$$\begin{aligned} S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)} &\overset{(3.5)}{=} \{\mathsf{close}_{\subseteq}(P) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\} \\ &\overset{3.18.7}{=} \{\mathsf{close}_{\subseteq}(\mathsf{tuple}(P)) \mid P \in S_{\textsc{Subset}(\mathcal{I},N)}\} \\ &\overset{(3.3)}{=} \{\mathsf{close}_{\subseteq}(P) \mid P \in S_{\textsc{Transset}(\mathcal{I},N)}\} \\ &\overset{3.22.3}{=} \{\mathsf{close}_{\subseteq}(\mathsf{compress}_{\subseteq}(P)) \mid P \in S_{\textsc{Transset}(\mathcal{I},N)}\} \\ &\overset{(3.6)}{=} \{\mathsf{close}_{\subseteq}(P) \mid P \in S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)}\}. \end{aligned}$$

Hence $|S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)}| \leq |S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)}|$. □

By the last two theorems we have $|S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)}| = |S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)}|$ for any input NFA N. Thus, the surjective function $\mathsf{close}_{\subseteq}$ from $S_{\textsc{Transset}(\mathsf{compress}_{\subseteq},N)}$ to $S_{\textsc{Subset}(\mathsf{close}_{\subseteq},N)}$ constructed in the last proof must be a bijection. It is not hard to see that it therefore is an isomorphism.

3.4 Implementation and experiments

We have implemented the algorithms $\textsc{Subset}(\mathcal{I})$, $\textsc{Transset}(\mathcal{I})$, $\textsc{Subset}(\mathsf{close}_{\subseteq})$, $\textsc{Subset}(\mathsf{compress}_{\subseteq})$ and $\textsc{Transset}(\mathsf{compress}_{\subseteq})$ in the C++ programming language [116]. A set of states or transitions is stored as a tree with the elements in the leaves. All subtrees are shared among the sets to improve memory efficiency. A hash table provides a fast and efficient lookup of existing subtrees.

The experiments are performed on a 64-bits architecture computer having an Intel Core2 Quad 2.40 GHz CPU and 4 GB of RAM. It runs Fedora Core 8 Linux, kernel 2.6.26. The code is compiled using the GNU C++ compiler, version 4.1.2.

Every experiment starts off by minimizing the NFA using simulation equivalence. For this we have implemented our partitioning algorithm (see chapter 4) which is based on [51] and also computes the simulation preorder on the states of the resulting NFA. Every determinization algorithm is applied to this minimized NFA, after which the resulting DFA is minimized by the tool ltsmin of the μCRL toolset[1], version 2.18.3.

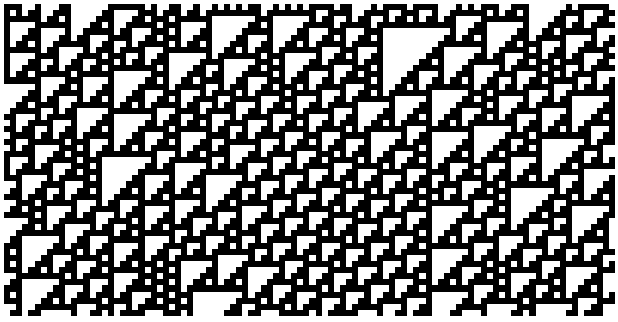
3.4.1 Cellular automaton 110

In his book [128], Wolfram studies cellular automata as a model of computation. A *cellular automaton* (CA) consists of a line of white or black cells[2] of which

[1] Available at http://www.cwi.nl/~mcrl.

[2] Actually, a wide variety of cellular automata can be defined. We consider a basic type here with two colours and a *neighbourhood* of 1.

(a) Rule 110



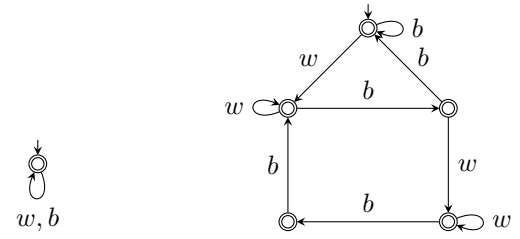
(b) Evolution

Figure 3.5. The rule 110 cellular automaton (a) and a possible evolution (b).

the colours are changed in every step of the automaton. The colour of a cell in the next step of the automaton's computation depends on its current colour and those of its left- and right-hand neighbours, as specified by a so-called *rule*. An example is given in figure 3.5. The rule is depicted in figure 3.5(a). For example, it specifies that if a cell is black and both of its neighbours are black, then that cell becomes white in the next step of the automaton's evolution (*cf.* the leftmost part of the rule). It is easy to see that there are 256 such rules. The rules can be numbered uniquely in a straightforward way by taking the bottom row and reading 0 for a white cell and 1 for a black cell. This gives the number for that rule in binary notation. For example, the rule in figure 3.5(a) has number 110 in decimal notation (01101110 in binary).

Figure 3.5(b) shows the *evolution* of this automaton. The first line is the initial state, for which the colours of the cells have been chosen randomly. Every successive line shows the next step in the evolution and is computed by applying the rule to every subsequence of length 3 on the previous line. The line of white or black cells on which the rule operates can be chosen to be two-way infinite or cyclic. Figure 3.5(b) is an example of the latter: the lines are assumed to "wrap", meaning that the left-hand neighbour of a cell in the leftmost column is the cell in the rightmost column of the same line, and vice versa. Here, we have chosen a line width of 100 cells and the first 50 steps of the evolution are depicted.

In general, a one-dimensional cellular automaton can be formally represented by a function $\rho : \Sigma^w \to \Sigma$, the rule, where Σ is an alphabet and $w \geq 1$ is the *width* of the automaton. Given an infinite sequence $\sigma \in \Sigma^\omega$, a *step* of a CA is an application of ρ to every w-length subsequence of σ, which produces a new infinite sequence. In the example of figure 3.5(a) we have $\Sigma = \{\text{white}, \text{black}\}$, $w = 3$ and



(a) 0 steps (b) 1 step

Figure 3.6. Minimal DFAs describing the possible sequences of white and black cells that can occur after 0 steps (a) and 1 step (b) of cellular automaton 110.

ρ is as depicted. In figure 3.5(b) we assume that the infinite sequences $\sigma \in \Sigma^\omega$ are periodic with a period of 100 cells, and only one period is displayed. Periodicity of the input sequence guarantees that the successive sequences are also periodic, with a period of the same size.

Wolfram classifies cellular automata based on the complexity of the patterns that emerge in their evolutions. Four classes are distinguished of which class 1 contains the simplest automata and class 4 the most complex ones. An example of a class-1 automaton is the one with rule 0, which simply colours every cell white in the first step and retains this state in subsequent steps. The complex pattern of figure 3.5(b) identifies automaton 110 as a class-4 automaton. Moreover, Wolfram has shown that the 110 automaton is *universal* or *Turing-complete*, which means it can perform exactly the same computations a Turing machine can. To be precise: given a (possibly universal) Turing machine M, there are finite sequences $\rho, \nu \in \{w, b\}^*$ as well as an encoding of any (finite) input sequence σ of M as a finite sequence $\sigma' \in \{w, b\}^*$, such that the behaviour of M on the input σ is in some sense mimicked (through a complicated encoding) by the evolution of cellular automaton 110 on the infinite input sequence composed of σ', flanked on the left and right by infinitely many repetitions of ρ and ν, respectively.

As described in [127], the possible finite sequences appearing as a continuous subsequence of the infinite sequence obtained after n steps of a given cellular automaton (starting from a random input sequence) constitute a language that can be described by a DFA. For example, the DFA that describes the possible sequences after 0 steps of cellular automaton 110 is depicted in figure 3.6(a): any sequence of white or black cells is allowed. The DFA after 1 step is shown in figure 3.6(b). Both DFAs are minimal and all states are final, except for omitted sink states. It is known that for some rules, the size of these DFAs increases exponentially in n (*cf.* [117]). Rule 110 exhibits this phenomenon.

We have generated the minimal DFAs for steps 1 through 6 of this cellular automaton using the implemented algorithms. Given the NFA N for any of these

steps, we minimize N modulo simulation equivalence to yield the NFA $N_{\rightleftarrows}$. The simulation preorder on $N_{\rightleftarrows}$ is also known after this minimization. Next, each of the determinization algorithms is applied to $N_{\rightleftarrows}$ to produce a DFA D, which is then minimized to yield the DFA D_m. The algorithms $\textsc{Subset}(\mathcal{I})$ and $\textsc{Transset}(\mathcal{I})$ are applied to N as well in order to assess the benefit of minimization prior to determinization. This gives a total of seven DFAs for every step.

For every generated automaton, we measure the amount of time and memory needed to produce it, and record the number of states. The most interesting results are those for steps 4 through 6, which are shown in table 3.1. It turns out that the costs in time and memory of minimization under $\rightleftarrows$ is very small compared to the costs of the subsequent determinization process, and that it makes the overall algorithm much more efficient. For step 6, the †-marks indicate that $\textsc{Subset}(\mathcal{I})$ and $\textsc{Transset}(\mathcal{I})$ ran out of the four gigabytes of memory, both on input N and on input $N_{\rightleftarrows}$. The ‡-marks mean that we terminated $\textsc{Subset}(\mathsf{close}_{\subseteq})$ prematurely after roughly 44 hours of computation without measuring its memory consumption. Consequently, the data for minimization could not be collected in these cases, as indicated by the question marks. The algorithms that use $\mathsf{compress}_{\subseteq}$ clearly outperform the others, in both memory and time efficiency. Every algorithm that uses a function other than $\mathcal{I}$ generates a DFA that is an order of magnitude smaller than that of its $\mathcal{I}$-counterpart.

3.4.2 Random automata

In [118], Tabakov and Vardi experimentally evaluate the performance of several automata algorithms by running them on randomly generated automata. In their model, the randomly generated NFAs have an alphabet $\Sigma = \{0, 1\}$. The parameters that can be set by the user are the number of states N, the *transition density* r, and the *final state density* f. The transition density indicates the ratio of the number of transitions to the number of states N for a given label $a \in \Sigma$. The final state density indicates the ratio of the number of final states $|F|$ to N. For example, if we choose $N = 20$, $r = 2.0$ and $f = 0.4$, the resulting NFA will have twenty states, forty 0-transitions, forty 1-transitions and eight final states.

The generated automaton need not be connected: for every label $a \in \Sigma$ transitions are added by repeatedly choosing two states s, t at random and adding the transition (s, a, t) only if it does not already exist, until the number of a-transitions equals (or exceeds) $r \cdot N$. So, if the transitions are chosen poorly or the transition density is not high enough, not all states are reachable from the initial state.

For our experiments, N ranges from 10 to 100 with steps of 10, r ranges from 0.25 to 4.0 with steps of 0.25, and f ranges from 0.0 to 1.0 with steps of 0.2. For every combination of parameter values, we generate 100 random NFAs, giving a grand total of 96 000 automata. Every automaton is first minimized under simulation equivalence; this step is not included in our measurements. Each of the resulting automata is determinized in turn by each of the algorithms and subsequently minimized. We measure the time, memory and size of the intermediate

Table 3.1. Results for canonizing the NFAs of steps 4, 5 and 6 of CA 110.

Sizes of N, $N_{\rightleftarrows}$ and D_m, and costs of $\rightleftarrows$-minimization per step.

				Minimization under $\rightleftarrows$	
Step	$\lvert S_N\rvert$	$\lvert S_{N_{\rightleftarrows}}\rvert$	$\lvert S_{D_m}\rvert$	Time (sec.)	Memory (MB)
4	800	228	1 357	<0.1	0.4
5	5 224	1 421	18 824	1.3	5.9
6	73 905	18 934	136 401	490.0	425.2

Results for step 4.

	Time (sec.)		Memory (MB)		
Algorithm	Det.	Min.	Det.	Min.	$\lvert S_D\rvert$
Subset$(\mathcal{I}, N)$	1.1	0.2	15.1	8.2	152 805
Transset$(\mathcal{I}, N)$	0.8	0.1	15.8	5.1	94 474
Subset$(\mathcal{I}, N_{\rightleftarrows})$	0.2	<0.1	5.3	3.1	58 371
Transset$(\mathcal{I}, N_{\rightleftarrows})$	0.4	<0.1	8.8	3.1	58 095
Subset$(\mathsf{close}_{\subseteq}, N_{\rightleftarrows})$	1.0	<0.1	2.1	0.3	4 721
Subset$(\mathsf{compress}_{\subseteq}, N_{\rightleftarrows})$	<0.1	<0.1	0.6	0.3	4 746
Transset$(\mathsf{compress}_{\subseteq}, N_{\rightleftarrows})$	<0.1	<0.1	0.8	0.3	4 721

Results for step 5.

	Time (sec.)		Memory (MB)		
Algorithm	Det.	Min.	Det.	Min.	$\lvert S_D\rvert$
Subset$(\mathcal{I}, N)$	611	55	1 924	960	17 960 609
Transset$(\mathcal{I}, N)$	253	36	2 242	646	12 083 654
Subset$(\mathcal{I}, N_{\rightleftarrows})$	93	24	673	409	7 663 166
Transset$(\mathcal{I}, N_{\rightleftarrows})$	130	24	1 121	403	7 541 249
Subset$(\mathsf{close}_{\subseteq}, N_{\rightleftarrows})$	1 285	<1	120	10	176 009
Subset$(\mathsf{compress}_{\subseteq}, N_{\rightleftarrows})$	2	<1	16	10	179 147
Transset$(\mathsf{compress}_{\subseteq}, N_{\rightleftarrows})$	2	<1	36	10	176 009

Results for step 6.

	Time (sec.)		Memory (MB)		
Algorithm	Det.	Min.	Det.	Min.	$\lvert S_D\rvert$
Subset$(\mathcal{I}, N)$	†	?	†	?	†
Transset$(\mathcal{I}, N)$	†	?	†	?	†
Subset$(\mathcal{I}, N_{\rightleftarrows})$	†	?	†	?	†
Transset$(\mathcal{I}, N_{\rightleftarrows})$	†	?	†	?	†
Subset$(\mathsf{close}_{\subseteq}, N_{\rightleftarrows})$	‡	?	‡	?	‡
Subset$(\mathsf{compress}_{\subseteq}, N_{\rightleftarrows})$	794	30	754	380	7 100 550
Transset$(\mathsf{compress}_{\subseteq}, N_{\rightleftarrows})$	170	29	1 202	363	6 770 156

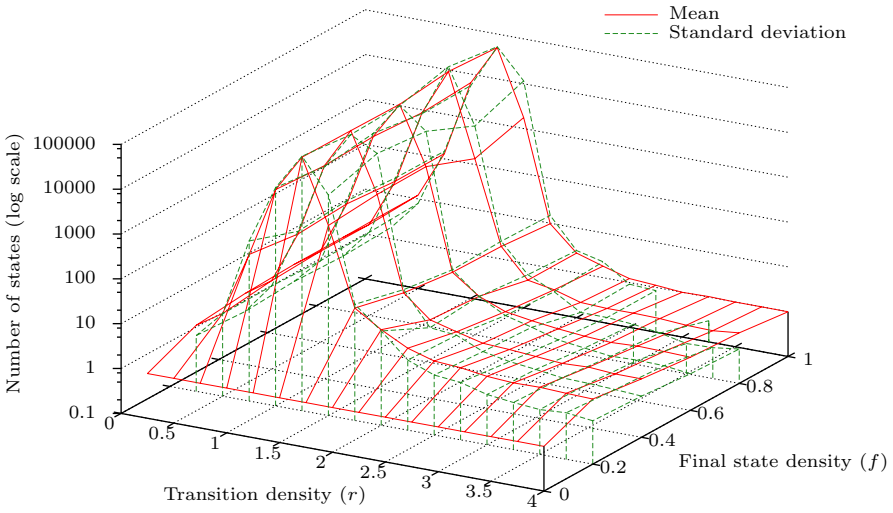


Figure 3.7. Mean and standard deviation of size of minimal DFA ($N = 100$).

DFA of each run of an algorithm. For each combination of parameter values and an algorithm we take the mean of the 100 measurements.

The size of the minimal DFA for $N = 100$ is plotted in figure 3.7. Every data point is the mean (solid line) or standard deviation (dashed line) of the 100 minimal DFAs that were generated for that point. We see the same phenomena as noticed in [118]: apart from the case $f = 0$, the final state density does not have a noteworthy effect on the size of the minimal DFA, but the transition density does. The mean size shows a clear peak at $r = 1.25$. For this value, the standard deviation is about as large as the mean, indicating that there is quite some variance in the data. For example, for $r = 1.25$ and $f = 0.40$ the mean is 44 213, the standard deviation is 36 182 and the sizes of the minimal DFAs range from 296 to 179 757. For other values of N the plot has the same shape as that of figure 3.7.

Because the final state density does not seem to influence the results that much, we fix $f = 0.40$. In plate I we have plotted the mean size of the minimal DFA with the NFA size (N) along the y-axis. We see that the NFA size really matters for values of r that are in the range 0.75–1.50. Outside of this range, the size of the minimal DFA remains almost constant as the size of the NFA increases. At $r = 1.25$ the effect of the NFA size is most dramatic: the mean size of the minimal DFA at $N = 100$ is more than 2 000 times as large as the mean size at $N = 10$.

To compare the determinization algorithms, we fix $f = 0.40$ and $N = 100$. In plates II–IV we have plotted the size of the DFA after determinization, and the amount of time and memory needed for determinization and minimization against

the transition density r, for each of the implemented algorithms. Again, every data point is the mean of 100 experiments.

In plate II we see that $\textsc{Subset}(\mathcal{I})$ and $\textsc{Subset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ generate significantly larger DFAs than the other algorithms for all values of r. For $r \leq 1.00$ these DFAs are very close (if not equal) to the minimal DFAs. We also see that for $r > 3$ the DFA generated by $\textsc{Transset}(\mathcal{I})$ is slightly larger than the ones generated by $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$ and $\textsc{Transset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$. Note that the measurements are in accordance with our lattice.

It is clear from the time plot of plate III that $\textsc{Subset}(\mathcal{I})$ is the fastest algorithm. A mean value of 0.001 indicates that the runs were too fast to allow proper measuring. $\textsc{Subset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ is as fast as $\textsc{Subset}(\mathcal{I})$ for $0.25 \leq r \leq 1$, but for larger r-values it becomes slower, ending up in fourth place for $r > 2$. Overall, $\textsc{Transset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ can be considered the slowest algorithm.

Regarding memory consumption (plate IV), the $\textsc{Subset}(f)$ algorithms overall outperform the $\textsc{Transset}(f)$ algorithms, with $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$ being the most memory efficient. The curve for $\textsc{Subset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ closely follows the one for $\textsc{Subset}(\mathcal{I})$, but $\textsc{Subset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ performs slightly better for $r < 2$. Given the fact that $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$ and $\textsc{Transset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ produce the same DFA, the former is preferable as it has better time and space performance. Overall, $\textsc{Subset}(\mathcal{I})$ and $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$ have the best performance; $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$ appears to use around 20% less memory overall, but $\textsc{Subset}(\mathcal{I})$ is significantly faster.

3.5 Conclusions

Determinization plays an important role in deciding language and trace preorder and equivalence. We have presented a schematic generalization of the well-known determinization algorithm, subset construction, that allows for a function to be applied to every generated set of states. We have given a similar scheme for a variant of subset construction that operates on sets of transitions rather than states. Next, we instantiated these schemes with several set-expanding or set-reducing functions to obtain various determinization algorithms. One of these algorithms even produces the minimal DFA directly, but its use of the PSPACE-hard language preorder renders it impractical. As our aim is to reduce the average-case workload in practice, we instead use the PTIME-decidable simulation preorder in the other algorithms. We have classified all presented algorithms in a lattice, based on the sizes of the DFAs they produce. This is a natural criterion, as the worst-case complexities are the same for all algorithms.

To assess their performance, we have implemented and experimentally evaluated the algorithms. The experiments comprised NFAs describing patterns in the elementary cellular automaton with rule number 110 and randomly generated NFAs. On the cellular automaton examples, the algorithms that use a function to reduce the computed sets, convincingly outperformed the others. On the random

automata, three of the algorithms generated smaller DFAs than subset construction, which led to less memory consumption in some cases. In particular, our algorithm SUBSET($\mathsf{close}_{\subseteq}$) systematically outperforms the standard subset construction in memory consumption. However, the gain is relatively small, and goes at the expense of the speed of the algorithm.

Clearly many more algorithms can be constructed based on our algorithm schemes by substituting various functions, depending on the specific needs and applications. Moreover, the functions we defined here could be equipped with any suitable preorder or partial order, *e.g.* from the linear-time–branching-time spectrum [55]. We believe that our alternatives to subset construction are particularly beneficial in cases where standard subset construction is known to leave a large gap between the generated DFA and the minimal one. The cellular-automaton experiments deal with such a situation and supports this theory.

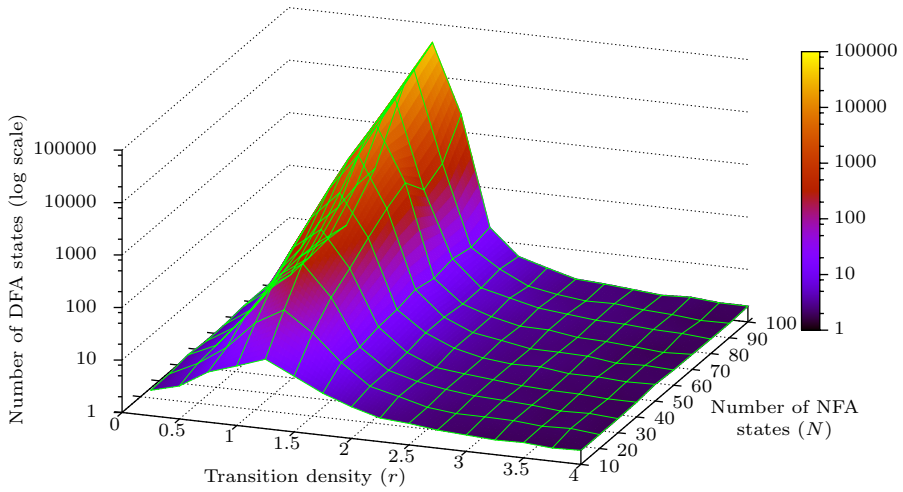


Plate I. Mean size of minimal DFA ($f = 0.40$).

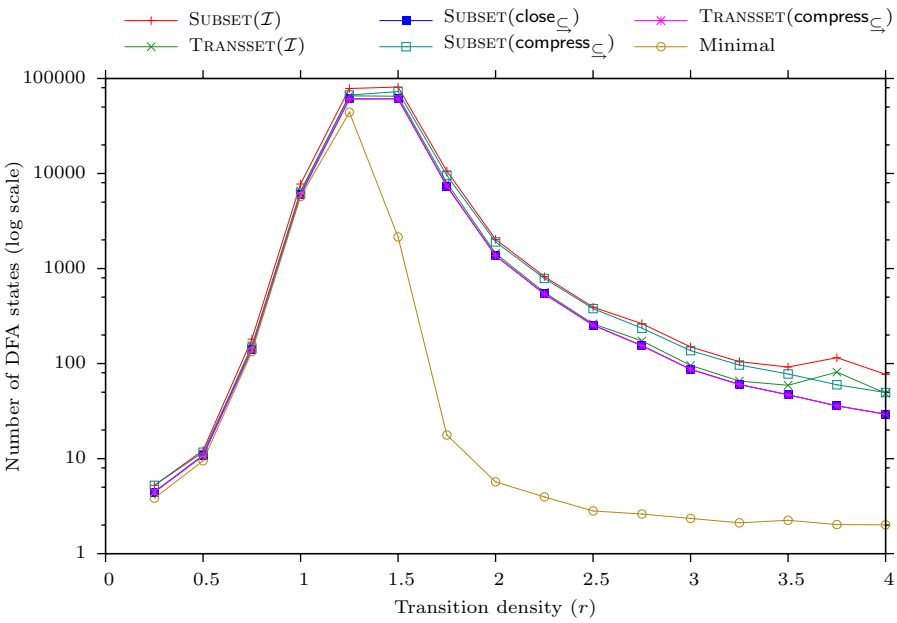


Plate II. Mean size of intermediate and minimal DFAs ($N = 100$, $f = 0.40$).

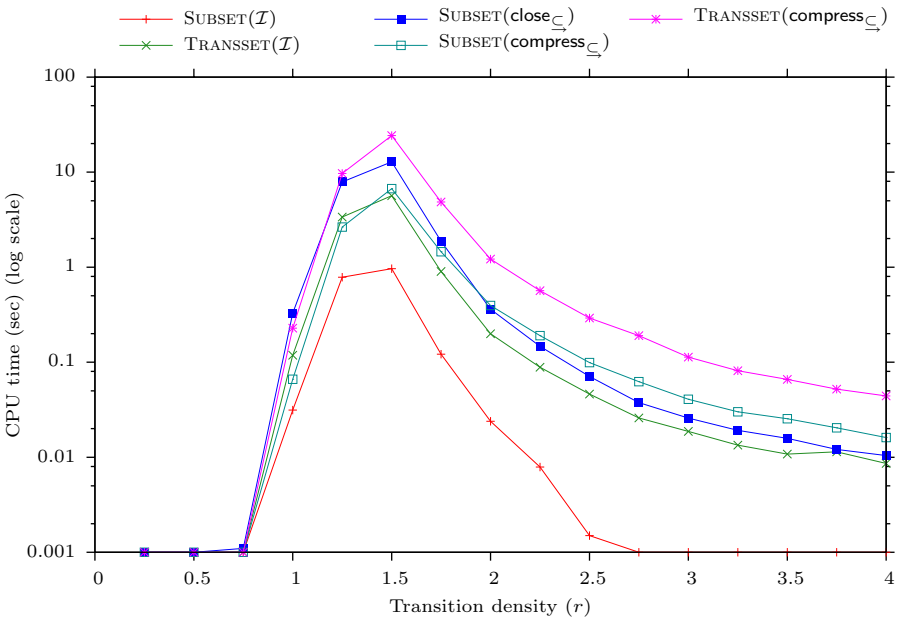


Plate III. Mean time needed for canonization ($N = 100$, $f = 0.40$).

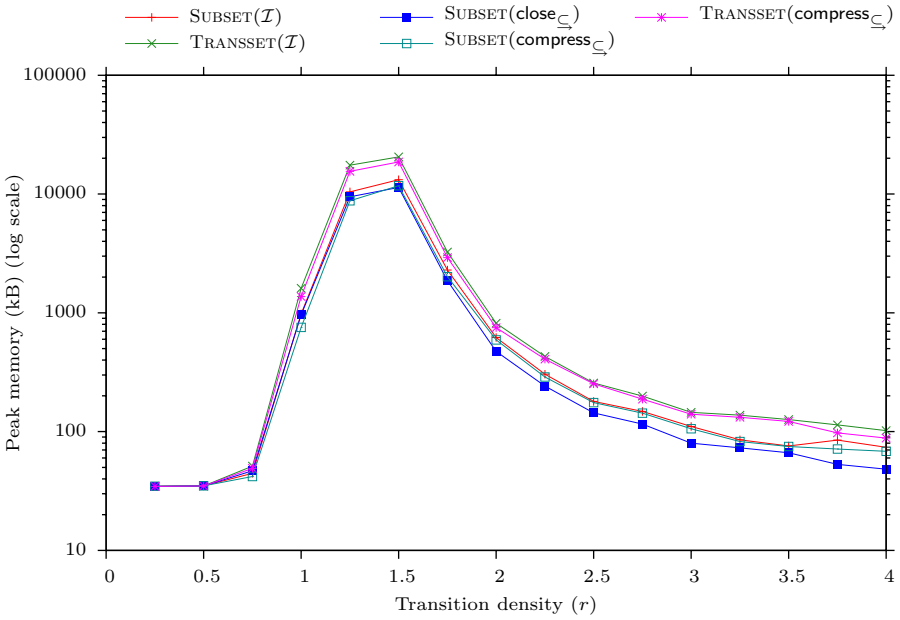


Plate IV. Mean peak memory needed for canonization ($N = 100$, $f = 0.40$).

Chapter 4

Simulation Equivalence

4.1 Introduction

The simulation preorder plays a crucial role in compositional verification and model checking. It is a natural preorder to use for matching an implementation with a specification when preservation of the branching structure is important. Also, deciding the simulation preorder on processes is often an appropriate method of showing that two systems are related by a weaker preorder, that may be suitable for the task at hand. In applications where deadlock behaviour is crucial, the *ready simulation preorder* [10] is widely regarded to be an appropriate behavioural refinement relation. Via a straightforward reduction (the computation of the initial partition ER_1 in [11]), finding a ready simulation between two processes is as hard as finding a plain simulation. In applications where deadlock behaviour plays no role, trace inclusion is often proposed as an appropriate refinement relation. However, deciding trace inclusion on finite-state processes is PSPACE-hard [115], and as the simulation preorder is the coarsest preorder included in trace inclusion that is PTIME-decidable [11, 19, 51, 72, 111, 119], establishing a simulation between two processes is a favourite way of showing that they are related by trace inclusion.

Simulation equivalence can be used directly in equivalence checking of finite-state processes. As shown in [31] and [90], respectively, it preserves the existential and universal fragments of CTL*, as well as the standard modal μ-calculus. This makes it possible to combat the state explosion problem in model checking by minimizing the state space of a given system modulo simulation equivalence before checking the validity of relevant properties within that fragment. Given that simulation equivalence is a congruence for parallel composition [68], components in parallel compositions can even be minimized individually.

In many practical verifications, space rather than time becomes the bottleneck as the input graph grows [30, 46, 51, 76]. Hence, simulation algorithms with minimal space complexity are of particular interest. These are the ones by Bustan

and Grumberg [19] and by Gentilini, Piazza and Policriti [51]. For an input graph with N states, T transitions and S simulation equivalence classes, the space complexity of both algorithms is $\mathcal{O}(S^2 + N \log S)$. This can be considered minimal: $\mathcal{O}(S^2)$ space is needed for storing the simulation preorder as a partial order on simulation equivalence classes and $\mathcal{O}(N \log S)$ space is needed to store for every state, the equivalence class to which it belongs. Of these algorithms, the one by Gentilini *et al.* has a better time complexity: $\mathcal{O}(S^2 \cdot T)$. A more time-efficient algorithm is the one by Ranzato and Tapparo [111] ($\mathcal{O}(S \cdot T)$), but this algorithm is less space-efficient ($\mathcal{O}(S \cdot N \log N)$).

The approach of Gentilini *et al.* represents the simulation problem as a generalized coarsest partition problem (GCPP), see section 4.3. According to the authors, this problem can be solved by approximating the greatest fixed point of a decreasing operator on partition pairs that they define in their paper. They give a partitioning algorithm to compute this fixed point for any legal input. We recall this definition and a part of the algorithm in section 4.4. In section 4.5 we show that the operator is flawed because it is not uniquely defined for all partition pairs. We give an instance of the GCPP for which repeated application of the operator does not lead to a unique fixed point. We also show that on this example the partitioning algorithm irrevocably allocates two simulation-equivalent states to different simulation-equivalence classes, and subsequently deadlocks.

In section 4.6 we define a simple, yet inefficient fixed-point operator for which we prove correctness. This operator is not meant to be an improvement over the original one, but merely serves as an expedient for establishing correctness of the algorithm that we present in section 4.7. This algorithm is obtained from that of Gentilini *et al.* by means of a few simple corrections. Yet its correctness proof requires entirely new techniques and is surprisingly intricate. In section 4.8 an implementation of our algorithm is described in order to show that it has the same time and space complexities as the original algorithm.

The original paper [51] studies the simulation problem in the context of finite vertex-labelled graphs. For the sake of clarity and easy comparison, we work in the same context in this chapter. Finite vertex-labelled graphs correspond with the finite abstract domain ($\mathbb{T}^{\mathsf{fa}}$) except for the absence of a root vertex, which does not make any difference for the problem of determining the simulation preorder on all vertices of a given graph. We define the simulation problem on finite vertex-labelled graphs in section 4.2.

4.2 Preliminaries

Finite vertex-labelled graphs are simply called *labelled graphs* in this chapter. For a labelled graph $(V, \rightarrow, L)$ and $a \in V$ the specific label $L(a)$ is irrelevant for our purposes. Therefore, we represent the labelling function L as the partition Σ over V that it induces: for all $a, b \in V$ we have $[a]_\Sigma = [b]_\Sigma$ iff $L(a) = L(b)$. A labelled graph is then written as $(V, \rightarrow, \Sigma)$ where V is finite. For a graph $(V, \rightarrow)$,

$a \in V$ and $\beta \subseteq V$, we write $a \to \beta$ if $\exists b \in \beta \,.\, a \to b$. Moreover, we define the relations $\to_\exists$ and $\to_\forall$ on $\wp(V)$ as follows, for any $\alpha, \beta \subseteq V$:

$$\alpha \to_\exists \beta \;\Leftrightarrow\; \exists a \in \alpha \,.\, a \to \beta \qquad\qquad \alpha \to_\forall \beta \;\Leftrightarrow\; \forall a \in \alpha \,.\, a \to \beta \,.$$

For any labelled graph $(V, \to, \Sigma)$ a relation $R \subseteq V \times V$ is a simulation iff for any $a, b \in V$, $a \; R \; b$ implies:

- $[a]_\Sigma = [b]_\Sigma$, and
- $\forall c \in V \,.\, a \to c \implies \exists d \in V \,.\, b \to d \wedge c \; R \; d$.

The simulation preorder $\subseteq$ on V is now defined as the largest simulation and simulation equivalence $\rightleftarrows$ is the induced equivalence (*cf.* definition 2.7).

Given a labelled graph G, the *simulation problem* over G consists in finding the simulation preorder $\subseteq$ on G. A variant of the simulation problem asks, given a labelled graph $(V, \to, \Sigma)$ and two vertices $a, b \in V$, whether $a \subseteq b$. In general, no methods to solve this problem are known that are more efficient than computing the entire relation $\subseteq \; \subseteq V \times V$ and looking up whether $(a, b) \in \subseteq$. Another variant of the simulation problem merely asks to find the simulation equivalence relation $\rightleftarrows$ rather than the preorder $\subseteq$. Again, no methods to solve that problem are known that do not amount to finding $\subseteq$ as well.

4.3 The generalized coarsest partition problem

Given a graph $G = (V, \to)$, a *partition pair* over G is a pair $\langle \Sigma, P \rangle$ where Σ is a partition over V and $P \subseteq \Sigma \times \Sigma$ is a reflexive, acyclic relation on Σ (see section 2.2.1 for the notion of acyclicity that we use here). A partition pair $\langle \Sigma, P \rangle$ is called transitive if P is transitive, and hence a partial order. Given a partition Σ, a partition Π finer than Σ, and a relation P on Σ, we denote by $P(\Pi)$ the *induced relation* of P on Π:

$$P(\Pi) := \{(\alpha, \beta) \in \Pi \times \Pi \mid \exists (\alpha', \beta') \in P \,.\, \alpha \subseteq \alpha' \wedge \beta \subseteq \beta'\}.$$

We define a partial order $\leq$ on partition pairs as follows: $\langle \Pi, Q \rangle \leq \langle \Sigma, P \rangle$ iff Π is finer than Σ and $Q \subseteq P(\Pi)$. Given a graph $G = (V, \to)$, a partition pair $\langle \Sigma, P \rangle$ over G is *stable with respect to* $\to$ [51] iff:

$$\forall \alpha, \beta, \gamma \in \Sigma \,.\, ((\alpha, \beta) \in P \wedge \alpha \to_\exists \gamma) \implies \exists \delta \in \Sigma \,.\, (\gamma, \delta) \in P \wedge \beta \to_\forall \delta.$$

In words, stability means that if a block α is related to β in P and some state in α can perform a transition to a state in a block γ, then *all* states in β must be able to mimic this transition: there must be a block δ that is larger than γ in P, and to which every state in β has a transition.

Given a graph $G = (V, \to)$ and a partition pair $\langle \Sigma, P \rangle$ over G, the *generalized coarsest partition problem* (GCPP) [51] consists in finding a $\leq$-maximal partition pair $\langle \Xi, \preceq \rangle$ such that $\langle \Xi, \preceq \rangle \leq \langle \Sigma, P^+ \rangle$ and $\langle \Xi, \preceq \rangle$ is stable with respect to $\to$.

4.3.1 The simulation problem as a GCPP

We now represent the simulation problem as a GCPP, in the following way. It is well known that there is a bijective correspondence between preorders on a set S and partitions over S that are equipped with a partial order on the blocks – *i.e.* *partition pairs* over S. In particular, as we prove below, simulations on a labelled graph $G = (V, \to, \Sigma)$ correspond bijectively with partition pairs over $(V, \to)$ that are stable with respect to $\to$ and at most as large as $\langle \Sigma, \mathcal{I} \rangle$. It then follows that the simulation preorder (being the largest simulation) on G corresponds with the $\leq$-largest stable partition pair over $(V, \to)$ that is at most as large as $\langle \Sigma, \mathcal{I} \rangle$.

Formally, let $G = (V, \to, \Sigma)$ be a labelled graph. For any preorder $\sqsubseteq$ on V we define the partition pair $\mathsf{PP}(\sqsubseteq) := \langle \Pi, \preceq \rangle$ as follows: Π is the set of equivalence classes of V with respect to the equivalence relation $\equiv$ induced by $\sqsubseteq$, and $\preceq$ is given by $[a]_\Pi \preceq [b]_\Pi$ iff $a \sqsubseteq b$. Note that $\preceq$ is a partial order. Conversely, for any partition pair $\langle \Pi, Q \rangle$ over the graph $(V, \to)$, let the relation $R_{\langle \Pi, Q \rangle} \subseteq V \times V$ be defined as follows: $R_{\langle \Pi, Q \rangle} := \{(a, b) \mid \exists (\alpha, \beta) \in Q \,.\, a \in \alpha \wedge b \in \beta\}$. Observe that for any preorder $\sqsubseteq$ we have $R_{\mathsf{PP}(\sqsubseteq)} = \sqsubseteq$.

Lemma 4.1. *For any partition pairs $\langle \Pi, Q \rangle$ and $\langle \Pi', Q' \rangle$, it holds that $\langle \Pi, Q \rangle \leq \langle \Pi', Q' \rangle$ if and only if $R_{\langle \Pi, Q \rangle} \subseteq R_{\langle \Pi', Q' \rangle}$.*

Proof. Suppose $\langle \Pi, Q \rangle \leq \langle \Pi', Q' \rangle$ and take $(a, b) \in R_{\langle \Pi, Q \rangle}$. Then we have $(\alpha, \beta) \in Q$ such that $a \in \alpha \wedge b \in \beta$. Observe that $(\alpha, \beta) \in Q'(\Pi)$. Then also $\exists (\alpha, \beta) \in Q' \,.\, a \in \alpha \wedge b \in \beta$, and thus $(a, b) \in R_{\langle \Pi', Q' \rangle}$.

Suppose $R_{\langle \Pi, Q \rangle} \subseteq R_{\langle \Pi', Q' \rangle}$. Take $(\alpha, \beta) \in Q$, $a \in \alpha$ and $b \in \beta$. Then $(a, b) \in R_{\langle \Pi', Q' \rangle}$. In the case that $\alpha = \beta$ we also have $(b, a) \in R_{\langle \Pi', Q' \rangle}$ from which it follows that Π is finer than Π'. In the general case we have $(\alpha', \beta') \in Q'$ such that $a \in \alpha'$ and $b \in \beta'$. As Π is finer than Π' we have $\alpha \subseteq \alpha'$ and $\beta \subseteq \beta'$, hence $(\alpha, \beta) \in Q'(\Pi)$ and thus $Q \subseteq Q'(\Pi)$. □

Lemma 4.2. *Let $G = (V, \to, \Sigma)$ be a labelled graph and $\sqsubseteq$ be a preorder on V. If $\sqsubseteq$ is a simulation then $\mathsf{PP}(\sqsubseteq)$ is stable with respect to $\to$ and $\mathsf{PP}(\sqsubseteq) \leq \langle \Sigma, \mathcal{I} \rangle$.*

Proof. Suppose that $\sqsubseteq$ is a simulation. Let $\langle \Pi, \preceq \rangle := \mathsf{PP}(\sqsubseteq)$ and $\equiv := \sqsubseteq \cap \sqsubseteq^{-1}$. Obviously, for any $a, b \in V$ we have $a \sqsubseteq b \Longrightarrow [a]_\Sigma = [b]_\Sigma$, hence $\langle \Pi, \preceq \rangle \leq \langle \Sigma, \mathcal{I} \rangle$.

Take $\alpha, \beta, \gamma \in \Pi$ such that $\alpha \preceq \beta$ and $\alpha \to_\exists \gamma$. Then $\exists a \in \alpha \,.\, a \to \gamma$. Take that a, and a $b' \in \beta$. As $a \sqsubseteq b'$, we have $\exists \delta' \in \Pi \,.\, \gamma \preceq \delta' \wedge b' \to \delta'$. Hence $\beta \to_\exists \delta'$. As $\preceq$ is a partial order on a finite set, let δ be a $\preceq$-maximal element of Π larger than δ' such that $\beta \to_\exists \delta$, *i.e.* $\delta' \preceq \delta$ and $\forall \varepsilon \in \Pi \,.\, \delta \preceq \varepsilon \wedge \beta \to_\exists \varepsilon \Longrightarrow \varepsilon = \delta$. Note that $\gamma \preceq \delta$. As $\beta \to_\exists \delta$, $\exists b_0 \in \beta \,.\, b_0 \to \delta$. For any $b \in \beta$ we have $b_0 \sqsubseteq b$, so $\exists \varepsilon_b \in \Pi \,.\, \delta \preceq \varepsilon_b \wedge b \to \varepsilon_b$. It must be that $\varepsilon_b = \delta$. Hence $\beta \to_\forall \delta$. □

Lemma 4.3. *Let $G = (V, \to, \Sigma)$ be a labelled graph and $\langle \Pi, Q \rangle$ be a partition pair over $(V, \to)$. If $\langle \Pi, Q \rangle$ is stable with respect to $\to$ and $\langle \Pi, Q \rangle \leq \langle \Sigma, \mathcal{I} \rangle$ then $R_{\langle \Pi, Q \rangle}$ is a simulation.*

Proof. Suppose that $\langle \Pi, Q\rangle$ is stable with respect to $\rightarrow$ and $\langle \Pi, Q\rangle \leq \langle \Sigma, \mathcal{I}\rangle$. Take $(a, b) \in R_{\langle \Pi, Q\rangle}$, and let $\alpha = [a]_\Pi$ and $\beta = [b]_\Pi$. As $(\alpha, \beta) \in Q$ and $\langle \Pi, Q\rangle \leq \langle \Sigma, \mathcal{I}\rangle$ we have that $(\alpha, \beta) \in \mathcal{I}(\Pi)$. Hence $([a]_\Sigma, [b]_\Sigma) \in \mathcal{I}$ which implies that $[a]_\Sigma = [b]_\Sigma$. Now suppose that $a \rightarrow c$ for some $c \in V$, and let $\gamma = [c]_\Pi$. Then $\alpha \rightarrow_\exists \gamma$ and, by stability of $\langle \Pi, Q\rangle$, $\exists \delta \in \Pi \,.\, (\gamma, \delta) \in Q \wedge \beta \rightarrow_\forall \delta$. Hence there is a $d \in \delta$ such that $b \rightarrow d$ and, by definition, $(c, d) \in R_{\langle \Pi, Q\rangle}$. Hence, $R_{\langle \Pi, Q\rangle}$ is a simulation. □

Theorem 4.4. *For any labelled graph $(V, \rightarrow, \Sigma)$, $\mathsf{PP}(\subseteq)$ is the solution of the GCPP on $(V, \rightarrow)$ and $\langle \Sigma, \mathcal{I}\rangle$.*

Proof. By lemma 4.2 $\mathsf{PP}(\subseteq)$ is stable with respect to $\rightarrow$ and $\mathsf{PP}(\subseteq) \leq \langle \Sigma, \mathcal{I}\rangle$. Let $\langle \Pi, Q\rangle \leq \langle \Sigma, \mathcal{I}\rangle$ be another partition pair that is stable with respect to $\rightarrow$. Then by lemma 4.3, $R_{\langle \Pi, Q\rangle}$ is a simulation, hence $R_{\langle \Pi, Q\rangle} \subseteq \subseteq$. Then by lemma 4.1, $\langle \Pi, Q\rangle \leq \mathsf{PP}(\subseteq)$, so $\mathsf{PP}(\subseteq)$ is the $\leq$-largest partition pair that is stable with respect to $\rightarrow$ and smaller than $\langle \Sigma, \mathcal{I}\rangle$. □

In particular, theorem 4.4 implies that the GCPP, when applied to partition pairs of the form $\langle \Sigma, \mathcal{I}\rangle$ (plain partitions), always has a unique solution $\langle \Xi, \preceq\rangle$, in which moreover $\preceq$ is always a partial order.[1]

4.4 The original GCPP solution

To solve the GCPP, Gentilini *et al.* [51] introduce the following operator, of which we show that it is not well-defined in section 4.5.

Definition 4.5 (definition 4.11 of [51])**.** Let $G = (V, \rightarrow)$ be a graph and $\langle \Sigma, P\rangle$ be a partition pair over G. The partition pair $\langle \Pi, Q\rangle = \sigma(\langle \Sigma, P\rangle)$ is defined as follows:

(1σ) Π is the coarsest partition finer than Σ such that

(a) $\forall \alpha \in \Pi \,.\, \forall \gamma \in \Sigma \,.\, \alpha \rightarrow_\exists \gamma \Longrightarrow \exists \delta \in \Sigma \,.\, ((\gamma, \delta) \in P \wedge \alpha \rightarrow_\forall \delta)$;

(2σ) Q is maximal such that $Q \subseteq P(\Pi)$ and if $(\alpha, \beta) \in Q$, then

(b) $\forall \gamma \in \Sigma \,.\, \alpha \rightarrow_\forall \gamma \Longrightarrow \exists \gamma' \in \Sigma \,.\, ((\gamma, \gamma') \in P \wedge \beta \rightarrow_\exists \gamma')$ and

(c) $\forall \gamma \in \Pi \,.\, \alpha \rightarrow_\forall \gamma \Longrightarrow \exists \gamma' \in \Pi \,.\, ((\gamma, \gamma') \in Q \wedge \beta \rightarrow_\exists \gamma')$.

Condition (a) expresses that any block α of the finer partition Π has to be stable with respect to itself. Conditions (b) and (c) restrict the pairs (α, β) that may occur in Q: if every state of α has a transition to a block γ then there must be a state in β that can go to a block γ' that is larger than γ. Otherwise, it is certain that no sub-block of α will ever be stable with respect to any sub-block of β, so (α, β) has to be removed from Q. Condition (b) applies this restriction with respect to $\langle \Sigma, P\rangle$ and condition (c) applies it with respect to $\langle \Pi, Q\rangle$ itself.

[1] The same reasoning extends to the GCPP applied to any partition pairs, but this requires considering simulations on structures of the form $(V, \rightarrow, \Sigma, \preceq)$ with $(V, \rightarrow, \Sigma)$ a labelled graph, and $\preceq$ a partial order on Σ; the first clause in the definition of simulation then becomes $[a]_\Sigma \preceq [b]_\Sigma$.

Gentilini *et al.* argue that applying σ iteratively to an initial partition pair $\langle \Sigma_0, P_0 \rangle$ yields a sequence $\{\langle \Sigma_i, P_i \rangle\}_{i \geq 0}$ with $\langle \Sigma_{i+1}, P_{i+1} \rangle = \sigma(\langle \Sigma_i, P_i \rangle)$. By construction, this sequence is decreasing, *i.e.* $\langle \Sigma_{i+1}, P_{i+1} \rangle \leq \langle \Sigma_i, P_i \rangle$. Hence it will reach a fixed point $\langle \Sigma_k, P_k \rangle = \sigma(\langle \Sigma_k, P_k \rangle)$, which is the solution to the GCPP.

Applying this, they give a partitioning algorithm to solve the GCPP. We call it PA_{GPP} and have included it here as algorithm 4.1. It takes a graph $(V, \rightarrow)$ and a transitive partition pair $\langle \Sigma, P \rangle$ as input and repeatedly calls the following functions until a fixed point is reached: $\text{REFINE}_{\text{GPP}}$, which computes the partition Π of (1σ), and $\text{UPDATE}_{\text{GPP}}$, which computes the relation Q of (2σ). The Boolean variable *change* is set to $\top$ by $\text{REFINE}_{\text{GPP}}$ iff its output partition differs from its input partition. We have included the $\text{REFINE}_{\text{GPP}}$ function as algorithm 4.2. In line 4 of this algorithm, a "reverse topological sorting of Σ_i with respect to P_i" is an ordered list of the elements of Σ_i such that if $(\gamma, \delta) \in P_i$ then δ is before γ in the list. The $\text{UPDATE}_{\text{GPP}}$ function is included as algorithm 4.3. It uses the NEW_HHK function (algorithm 4.4) to refine the induced relation $P_i(\Sigma_{i+1})$ in two steps, yielding the relation P_{i+1}. In line 2 of $\text{UPDATE}_{\text{GPP}}$, the structure $(\Sigma_{i+1}, \rightarrow_{\exists}^{ind}, \rightarrow_{\forall}^{ind})$ is called the $\exists\forall$-*induced quotient structure* over Σ_{i+1}, and the relations $\rightarrow_{\exists}^{ind}$ and $\rightarrow_{\forall}^{ind}$ on Σ_{i+1} are defined as follows [51]:

$$\rightarrow_{\exists}^{ind} := \rightarrow_{\exists}$$
$$\rightarrow_{\forall}^{ind} := \{(\alpha, \beta) \in \rightarrow_{\exists}^{ind} \mid \exists \beta' \in \Sigma_i \,.\, \beta \subseteq \beta' \wedge \alpha \rightarrow_{\forall} \beta'\}\,.$$

4.5 Incorrectness of the operator σ

Following the definition of σ, it is claimed in [51] that for any partition pair $\langle \Sigma, P \rangle$, if $\langle \Pi, Q \rangle = \sigma(\langle \Sigma, P \rangle)$ then Q is acyclic. We give a counterexample to this claim.

Counterexample 4.6. Consider the graph in figure 4.1(a) and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{\alpha, \beta, \gamma, \delta\}$ as depicted and $P = \mathcal{I} \cup \{(\beta, \delta), (\delta, \gamma)\}$. Let $\langle \Pi, Q \rangle = \sigma(\langle \Sigma, P \rangle)$, then

$$\Pi = \{\alpha_1, \alpha_2, \beta, \gamma, \delta\} \qquad Q = \mathcal{I} \cup \{(\alpha_1, \alpha_2), (\alpha_2, \alpha_1), (\beta, \delta), (\delta, \gamma)\}$$

where $\alpha_1 = \{a_1\}$ and $\alpha_2 = \{a_2\}$. Q is not acyclic, which counters the claim. □

This counterexample shows that applying σ to a given partition pair does not necessarily yield another partition pair, because the resulting relation need not be acyclic. However, a more fundamental property of σ turns out not to hold. Theorem 4.13 of [51] states that for every partition pair $\langle \Sigma, P \rangle$ there exists a unique $\leq$-maximal partition pair $\langle \Pi, Q \rangle \leq \langle \Sigma, P \rangle$ satisfying conditions (a), (b) and (c) of definition 4.5, *i.e.* the σ operator is well-defined, and a function. This statement is refuted by the following example.

Counterexample 4.7. Consider the graph in figure 4.1(b) and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{\alpha, \beta, \gamma, \delta\}$ as depicted and $P = \mathcal{I} \cup \{(\beta, \gamma), (\gamma, \delta)\}$. Let $\langle \Pi, Q \rangle$

Algorithm 4.1. The partitioning algorithm of [51]: $\mathrm{PA}_{\mathrm{GPP}}((V, \to), \langle \Sigma, P \rangle)$

$\Sigma_0 := \Sigma$; $P_0 := P$;
change $:= \top$; $i := 0$;
while *change* **do**
    change $:= \bot$;
    $\Sigma_{i+1} := \textsc{Refine}_{\mathrm{GPP}}(\Sigma_i, P_i, \mathit{change})$;
    $P_{i+1} := \textsc{Update}_{\mathrm{GPP}}(\Sigma_i, P_i, \Sigma_{i+1})$;
    $i := i + 1$
end while

Algorithm 4.2. The refine function of [51]: $\textsc{Refine}_{\mathrm{GPP}}(\Sigma_i, P_i, \mathit{change})$

$\Sigma_{i+1} := \Sigma_i$;
for all $\alpha \in \Sigma_{i+1}$ **do** $\mathit{Stable}(\alpha) := \emptyset$ **end for**;
for all $\gamma \in \Sigma_i$ **do** $\mathit{Row}(\gamma) := \{\gamma' \mid (\gamma, \gamma') \in P_i\}$ **end for**;
Let *Sort* be a reverse topological sorting of Σ_i with respect to P_i;
while *Sort* $\neq \emptyset$ **do**
    $\gamma := \mathit{dequeue}(\mathit{Sort})$;
    $A := \emptyset$;
    for all $\alpha \in \Sigma_{i+1}$, $\alpha \to_\exists \gamma$, $\mathit{Stable}(\alpha) \cap \mathit{Row}(\gamma) = \emptyset$ **do**
        $\alpha_1 := \alpha \cap \to^{-1}(\gamma)$;
        $\alpha_2 := \alpha \setminus \alpha_1$;
        if $\alpha_2 \neq \emptyset$ **then**
            change $:= \top$
        end if;
        $\Sigma_{i+1} := \Sigma_{i+1} \setminus \{\alpha\}$;
        $A := A \cup \{\alpha_1, \alpha_2\}$;
        $\mathit{Stable}(\alpha_1) := \mathit{Stable}(\alpha) \cup \{\gamma\}$;
        $\mathit{Stable}(\alpha_2) := \mathit{Stable}(\alpha)$
    end for;
    $\Sigma_{i+1} := \Sigma_{i+1} \cup A$;
    Sort $:=$ *Sort* $\setminus \{\gamma\}$
end while;
return Σ_{i+1}

Algorithm 4.3. The update function of [51]: $\textsc{Update}_{\text{GPP}}(\Sigma_i, P_i, \Sigma_{i+1})$

1: $Ind_{i+1} := \{(\alpha_1, \beta_1) \in \Sigma_{i+1} \times \Sigma_{i+1} \mid \exists (\alpha, \beta) \in P_i \,.\, \alpha_1 \subseteq \alpha \wedge \beta_1 \subseteq \beta\}$;
2: $Ref_{i+1} := \textsc{New\_HHK}(\Sigma_{i+1}, \rightarrow_{\exists}^{ind}, \rightarrow_{\forall}^{ind}, Ind_{i+1}, \bot)$;
3: $P_{i+1} := \textsc{New\_HHK}(\Sigma_{i+1}, \rightarrow_{\exists}, \rightarrow_{\forall}, Ref_{i+1}, \top)$;
4: **return** P_{i+1}

Algorithm 4.4. The New_HHK function of [51]: $\textsc{New\_HHK}(T, R_1, R_2, K, U)$

1: $P := K$;
2: **for all** $c \in T$ **do**
3: $\quad sim(c) := \{e \mid (c, e) \in K\}$;
4: $\quad rem(c) := \{b \in T \mid \neg \exists d \in sim(c) \,.\, (b, d) \in R_1\}$
5: **end for**;
6: **while** $\{c \mid rem(c) \neq \emptyset\} \neq \emptyset$ **do**
7: $\quad$ let $c \in \{c \mid rem(c) \neq \emptyset\}$;
8: $\quad$ **while** $rem(c) \neq \emptyset$ **do**
9: $\quad\quad$ let $b \in rem(c)$;
10: $\quad\quad rem(c) := rem(c) \setminus \{b\}$;
11: $\quad\quad$ **for all** $a \in T, (a, c) \in R_2$ **do**
12: $\quad\quad\quad$ **if** $b \in sim(a)$ **then**
13: $\quad\quad\quad\quad sim(a) := sim(a) \setminus \{b\}$;
14: $\quad\quad\quad\quad P := P \setminus \{(a, b)\}$;
15: $\quad\quad\quad\quad$ **if** U **then**
16: $\quad\quad\quad\quad\quad$ **for all** $b_1 \in T, (b_1, b) \in R_1$ **do**
17: $\quad\quad\quad\quad\quad\quad$ **if** $\neg \exists d \in sim(a) \,.\, (b_1, d) \in R_1$ **then**
18: $\quad\quad\quad\quad\quad\quad\quad rem(a) := rem(a) \cup \{b_1\}$
19: $\quad\quad\quad\quad\quad\quad$ **end if**
20: $\quad\quad\quad\quad\quad$ **end for**
21: $\quad\quad\quad\quad$ **end if**
22: $\quad\quad\quad$ **end if**
23: $\quad\quad$ **end for**
24: $\quad$ **end while**
25: **end while**;
26: **return** P

Figure 4.1. Examples for which (a) σ produces a partition pair with a non-acyclic relation and (b) σ is not well-defined.

and $\langle \Pi', Q' \rangle$ be partition pairs such that:

$$\Pi = \{\alpha_0, \alpha_1, \beta, \gamma, \delta\} \qquad Q = \mathcal{I} \cup \{(\alpha_0, \alpha_1), (\alpha_1, \alpha_0), (\beta, \gamma), (\gamma, \delta)\}$$
$$\Pi' = \{\alpha'_0, \alpha'_1, \beta, \gamma, \delta\} \qquad Q' = \mathcal{I} \cup \{(\alpha'_0, \alpha'_1), (\alpha'_1, \alpha'_0), (\beta, \gamma), (\gamma, \delta)\}$$

where $\alpha_0 = \{a_0, a_1\}$, $\alpha_1 = \{a_2\}$, $\alpha'_0 = \{a_0\}$ and $\alpha'_1 = \{a_1, a_2\}$. Both $\langle \Pi, Q \rangle$ and $\langle \Pi', Q' \rangle$ satisfy conditions (a), (b) and (c) of definition 4.5, but neither is the $\leq$-largest. The only partition pair greater than both $\langle \Pi, Q \rangle$ and $\langle \Pi', Q' \rangle$ and at most as large as $\langle \Sigma, P \rangle$, is $\langle \Sigma, P \rangle$ itself, but $\langle \Sigma, P \rangle$ does not satisfy (a). Hence, this example counters theorem 4.13 and shows that σ is not well-defined. □

Following theorem 4.13, the main fixed-point theorem of [51] states that the solution of the GCPP over a graph G and partition pair $\langle \Sigma, P \rangle$ can be computed by applying σ to $\langle \Sigma, P \rangle$ finitely many times until a fixed point is reached (theorem 4.14). In this theorem, it is required that P be transitive. One might expect that counterexample 4.7 does not affect this theorem, as we used a non-transitive P. We now show that this is not the case: the main theorem indeed loses its meaning due to our counterexample. To do so, we first give an example in which the application of σ to a transitive partition pair produces a non-transitive partition pair.

Example 4.8. Consider the graph in figure 4.2(a) and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{\alpha, \beta, \gamma\}$ as depicted and $P = \mathcal{I}$. Let $\langle \Pi, Q \rangle = \sigma(\langle \Sigma, P \rangle)$, then:

$$\Pi = \{\alpha_1, \alpha_2, \alpha_3, \beta, \gamma\} \qquad Q = \mathcal{I} \cup \{(\alpha_3, \alpha_1), (\alpha_1, \alpha_2)\}$$

where $\alpha_1 = \{a_0, a_1\}$, $\alpha_2 = \{a_2\}$ and $\alpha_3 = \{a_3\}$. □

Our final counterexample shows that σ is not suitable for computing the solution to the GCPP, and is constructed by embedding counterexample 4.7 in example 4.8, such that the first application of σ produces a non-transitive partition pair on which σ is not well-defined.

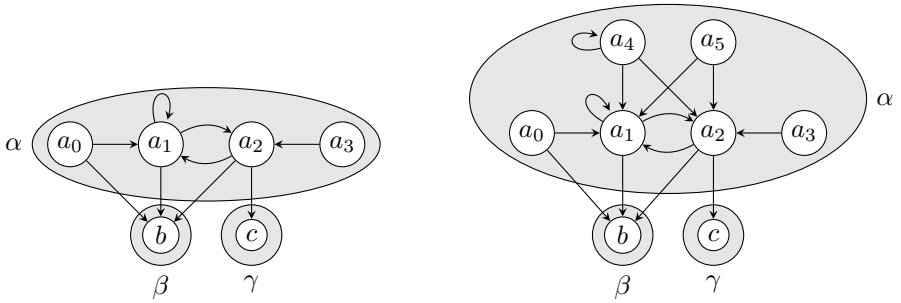


Figure 4.2. (a) Example for which σ produces a partition pair with a non-transitive relation and (b) counterexample for correctness of σ.

Counterexample 4.9. Consider the graph in figure 4.2(b) and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{\alpha, \beta, \gamma\}$ as depicted and $P = \mathcal{I}$. Let $\langle \Pi, Q \rangle = \sigma(\langle \Sigma, P \rangle)$, then:

$$\Pi = \{\alpha_1, \alpha_2, \alpha_3, \beta, \gamma\} \qquad Q = \mathcal{I} \cup \{(\alpha_3, \alpha_1), (\alpha_1, \alpha_2)\}$$

where $\alpha_1 = \{a_0, a_1\}$, $\alpha_2 = \{a_2\}$ and $\alpha_3 = \{a_3, a_4, a_5\}$. Now, in $\langle \Pi, Q \rangle$ the block α_3 has to be split, because $\alpha_3 \rightarrow_\exists \alpha_3$ but $\neg\exists \delta \in \Pi \,.\, ((\alpha_3, \delta) \in Q \wedge \alpha_3 \rightarrow_\forall \delta))$. There are two candidate partition pairs for $\sigma(\langle \Pi, Q \rangle)$: α_3 can be split into either $\alpha_{3,0} = \{a_4\}$ and $\alpha_{3,1} = \{a_3, a_5\}$ or $\alpha'_{3,0} = \{a_4, a_5\}$ and $\alpha'_{3,1} = \{a_3\}$. However, neither of these is greater than the other, so a unique $\leq$-maximal partition pair does not exist. □

When splitting α_3 in counterexample 4.9, the $\textsc{Refine}_{\text{GPP}}$ function of algorithm PA_{GPP} splits the block into $\alpha_{3,0}$ and $\alpha_{3,1}$. Observe that this is wrong: a_4 and a_5 should not end up in different equivalence classes because $a_4 \rightleftarrows a_5$. This split also results in $\textsc{Update}_{\text{GPP}}$'s returning a cyclic relation. In the subsequent iteration of PA_{GPP}, the execution of $\textsc{Refine}_{\text{GPP}}$ then fails because there is no reverse topological sorting of the partition with respect to the cyclic relation (line 4).

4.6 An auxiliary fixed-point operator

In this section we introduce a fixed-point operator ρ to solve the GCPP and prove its correctness. The definition of ρ is straightforward: it is based directly on the stability condition of section 4.3. We emphasize that ρ is not intended to be an improvement over the σ operator of section 4.4 in any way: it is a less advanced operator than σ aimed to be. The purpose of σ is to compute the solution to the GCPP efficiently, while ρ gives rise to an algorithm that has an inferior time-complexity of $\mathcal{O}(S^3T)$ where S is the number of equivalence classes of the GCPP solution and T the number of transitions of the input graph.

The complexity analysis of [51] uses that, as long as no fixed point is reached, in each refine–update step the refinement of the partition will be non-trivial, *i.e.* the number of blocks increases. As a consequence, there will be at most S refine–update steps before the algorithm terminates. Such an analysis is not appropriate for ρ: applying ρ repeatedly may involve many steps in which the partition does not change. Consequently, the number of iterations of the algorithm is bounded merely by the size of a relation on the eventual partition, *i.e.* by S^2.

The sole purpose of ρ is to serve as an auxiliary operator for establishing the correctness of the algorithm that we present in section 4.7. That algorithm has the same time complexity as $\mathrm{PA}_{\mathrm{GPP}}$ and involves a non-functional refinement step, as we show in the same section.

Definition 4.10. Let $\langle \Sigma, P\rangle$ be a transitive partition pair over a graph $(V, \rightarrow)$. Then $\rho(\langle \Sigma, P\rangle)$ is the $\leq$-largest partition pair $\langle \Pi, Q\rangle \leq \langle \Sigma, P\rangle$ that satisfies

$$(4.1)\quad \forall \alpha, \beta \in \Pi \,.\, \forall \gamma \in \Sigma \,.\, ((\alpha, \beta) \in Q \wedge \alpha \rightarrow_\exists \gamma \Longrightarrow \exists \delta \in \Sigma \,.\, ((\gamma, \delta) \in P \wedge \beta \rightarrow_\forall \delta)).$$

Alternatively, ρ could be defined just like σ of definition 4.5, but insisting that its input partition pair is transitive, and omitting clause (c). It is not hard to check that this definition is equivalent to the one above. We now prove that ρ is properly defined, is monotonic and has a fixed point.

Proposition 4.11. *Let $\langle \Sigma, P\rangle$ be a transitive partition pair over a graph $(V, \rightarrow)$. Then there exists a $\leq$-largest partition pair $\langle \Pi, Q\rangle \leq \langle \Sigma, P\rangle$ that satisfies* (4.1)*. Moreover, Q is transitive.*

Proof. Define the relation $\sqsubseteq \subseteq V \times V$ by $a \sqsubseteq b$ iff

- $\exists (\alpha, \beta) \in P \,.\, a \in \alpha \wedge b \in \beta$ and
- $\forall \gamma \in \Sigma \,.\, (a \rightarrow \gamma \Longrightarrow \exists \delta \in \Sigma \,.\, ((\gamma, \delta) \in P \wedge b \rightarrow \delta))$.

Using the reflexivity and transitivity of P, this relation is a preorder. Take $\langle \Pi, Q\rangle := \mathsf{PP}(\sqsubseteq)$, as defined in section 4.3.1. So Q is transitive. By construction, $\langle \Pi, Q\rangle \leq \langle \Sigma, P\rangle$. It is not hard to check that Π satisfies (4.1); the argument is similar to the proof of lemma 4.2.

Now let $\langle \Pi', Q'\rangle$ be another partition pair with $\langle \Pi', Q'\rangle \leq \langle \Sigma, P\rangle$ that satisfies (4.1). Suppose $(\alpha, \beta) \in Q'$, $a \in \alpha$ and $b \in \beta$. Using (4.1) we find $a \sqsubseteq b$. Applying this to the case $\alpha = \beta$ we find that Π' is finer than Π. Applying it in general yields $Q' \subseteq Q(\Pi')$. Hence $\langle \Pi', Q'\rangle \leq \langle \Pi, Q\rangle$. □

Proposition 4.12. *The operator ρ is monotonic with respect to $\leq$: if $\langle \Pi, Q\rangle$ and $\langle \Sigma, P\rangle$ are transitive partition pairs with $\langle \Pi, Q\rangle \leq \langle \Sigma, P\rangle$, then $\rho(\langle \Pi, Q\rangle) \leq \rho(\langle \Sigma, P\rangle)$.*

Proof. As $\rho(\langle \Pi, Q\rangle)$ satisfies (4.1) with respect to $\langle \Pi, Q\rangle$, it certainly satisfies (4.1) with respect to $\langle \Sigma, P\rangle$. As $\rho(\langle \Pi, Q\rangle) \leq \langle \Pi, Q\rangle \leq \langle \Sigma, P\rangle$ and $\rho(\langle \Sigma, P\rangle)$ is the $\leq$-largest partition pair with $\rho(\langle \Sigma, P\rangle) \leq \langle \Sigma, P\rangle$ that satisfies (4.1), it follows that $\rho(\langle \Pi, Q\rangle) \leq \rho(\langle \Sigma, P\rangle)$. □

Since $\rho(\langle\Sigma, P\rangle) \leq \langle\Sigma, P\rangle$ and $\leq$ is a partial order on a finite set, we obtain:

Proposition 4.13. *Let $\langle\Sigma, P\rangle$ be a transitive partition pair over a graph. Then for some $n \geq 0$, $\rho^{n+1}(\langle\Sigma, P\rangle) = \rho^n(\langle\Sigma, P\rangle)$, i.e. repeated application of ρ leads to a fixed point.* □

The solution to the GCPP over an input graph G and an initial partition pair $\langle\Sigma, P\rangle$ over G can be obtained by repeatedly applying ρ to $\langle\Sigma, P^+\rangle$. The following lemmas say that as soon as a fixed point is reached, the resulting partition pair is stable. Moreover, each of the intermediate partition pairs is larger than or equal to the solution of the GCPP. It then follows that the obtained fixed point is in fact the solution to the GCPP.

Lemma 4.14. *Let $\langle\Sigma, P\rangle$ be a transitive partition pair over a graph $(V, \rightarrow)$. Then $\rho(\langle\Sigma, P\rangle) = \langle\Sigma, P\rangle$ if and only if $\langle\Sigma, P\rangle$ is stable with respect to $\rightarrow$.*

Proof. Because $\rho(\langle\Sigma, P\rangle)$ is the $\leq$-largest partition pair satisfying (4.1), we have that $\rho(\langle\Sigma, P\rangle) = \langle\Sigma, P\rangle$ if and only if $\langle\Sigma, P\rangle$ satisfies (4.1) with respect to itself, which is equivalent to stability with respect to $\rightarrow$. □

Lemma 4.15. *Let $\langle\Sigma, P\rangle$ and $\langle\Pi, Q\rangle$ be partition pairs over a graph G, with Q transitive, and let $\langle\Xi, \preceq\rangle$ be the solution of the GCPP over G and $\langle\Sigma, P\rangle$. If $\langle\Xi, \preceq\rangle \leq \langle\Pi, Q\rangle$ then $\langle\Xi, \preceq\rangle \leq \rho(\langle\Pi, Q\rangle)$.*

Proof. By lemma 4.14 $\rho(\langle\Xi, \preceq\rangle) = \langle\Xi, \preceq\rangle$. Assuming that $\langle\Xi, \preceq\rangle \leq \langle\Pi, Q\rangle$, the statement now follows from proposition 4.12. □

Theorem 4.16. *Let $\langle\Sigma, P\rangle$ be a partition pair over a graph $G = (V, \rightarrow)$ and $\langle\Xi, \preceq\rangle$ be the solution of the GCPP over G and $\langle\Sigma, P\rangle$. Let $n \geq 0$ be such that $\rho^{n+1}(\langle\Sigma, P^+\rangle) = \rho^n(\langle\Sigma, P^+\rangle)$. Then $\rho^n(\langle\Sigma, P^+\rangle) = \langle\Xi, \preceq\rangle$.*

Proof. Note that n exists by proposition 4.13. We prove that $\langle\Xi, \preceq\rangle \leq \rho^n(\langle\Sigma, P^+\rangle)$ and $\rho^n(\langle\Sigma, P^+\rangle) \leq \langle\Xi, \preceq\rangle$.

- $\langle\Xi, \preceq\rangle \leq \rho^n(\langle\Sigma, P^+\rangle)$: By definition $\langle\Xi, \preceq\rangle \leq \langle\Sigma, P^+\rangle$. Applying lemma 4.15 n times gives us $\langle\Xi, \preceq\rangle \leq \rho^n(\langle\Sigma, P^+\rangle)$.
- $\rho^n(\langle\Sigma, P^+\rangle) \leq \langle\Xi, \preceq\rangle$: Obviously $\rho^n(\langle\Sigma, P^+\rangle) \leq \langle\Sigma, P^+\rangle$ and by lemma 4.14 $\rho^n(\langle\Sigma, P^+\rangle)$ is stable with respect to $\rightarrow$. By definition $\langle\Xi, \preceq\rangle$ is the $\leq$-largest partition pair that has these properties. Hence $\rho^n(\langle\Sigma, P^+\rangle) \leq \langle\Xi, \preceq\rangle$. □

The operator ρ and its established properties allow us to prove correctness of the repaired algorithm that we present in the next section.

4.7 A correct and efficient algorithm

The repaired partitioning algorithm is called PA, see algorithm 4.5. The variable *change* and the input graph $(V, \rightarrow)$ have global scope: they can be accessed within any function. Note however, that UPDATE$_{\text{GPP}}$ does not access *change*.

Algorithm 4.5. The repaired partitioning algorithm: $\mathrm{PA}((V, \to), \langle \Sigma, P \rangle)$

$\Sigma_1 := \textsc{Refine}(\Sigma, P)$;
$P_1 := \textsc{Update}_{\mathrm{GPP}}(\Sigma, P, \Sigma_1)$;
change $:= \top$; $i := 1$;
while *change* **do**
    change $:= \bot$;
    $\Sigma_{i+1} := \textsc{Refine}(\Sigma_i, P_i)$;
    $P_{i+1} := \textsc{Update}_{\mathrm{GPP}}(\Sigma_i, P_i, \Sigma_{i+1})$;
    $i := i + 1$
end while

Algorithm 4.6. The repaired refine function: $\textsc{Refine}(\Sigma, P)$

$\Pi := \Sigma$;
for all $\alpha \in \Pi$ **do** $\mathit{Stable}(\alpha) := \emptyset$ **end for**;
for all $\gamma \in \Sigma$ **do** $\mathit{Row}(\gamma) := \{\gamma' \mid (\gamma, \gamma') \in P\}$ **end for**;
Let *Sort* be a reverse topological sorting of Σ with respect to P;
while $\mathit{Sort} \neq [\,]$ **do**
    $\gamma := \mathit{head}(\mathit{Sort})$;
    $A := \emptyset$;
    for all $\alpha \in \Pi$, $\alpha \to_\exists \gamma$ **do**
        if $\mathit{Stable}(\alpha) \cap \mathit{Row}(\gamma) = \emptyset$ **then**
            $\alpha_1 := \alpha \cap \to^{-1}(\gamma)$;
            $\alpha_2 := \alpha \setminus \alpha_1$;
            $\Pi := \Pi \setminus \{\alpha\}$;
            $A := A \cup \{\alpha_1\}$;
            $\mathit{Stable}(\alpha_1) := \mathit{Stable}(\alpha) \cup \{\gamma\}$;
            if $\alpha_2 \neq \emptyset$ **then**
                change $:= \top$;
                $A := A \cup \{\alpha_2\}$;
                $\mathit{Stable}(\alpha_2) := \mathit{Stable}(\alpha)$
            end if
        else
            $\mathit{Stable}(\alpha) := \mathit{Stable}(\alpha) \cup \{\gamma\}$
        end if
    end for;
    $\Pi := \Pi \cup A$;
    Sort $:= \mathit{tail}(\mathit{Sort})$
end while;
return Π

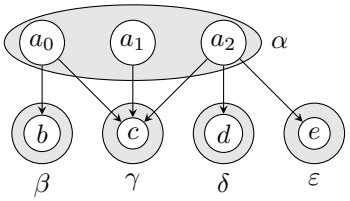


Figure 4.3. Example on which the result of REFINE is not uniquely defined.

We apply two corrections to the algorithm. Firstly, it is ensured that at least two refine–update steps are taken before the algorithm terminates (lines 1 and 2). The necessity of this (minor) correction is explained in section 4.7.1. Secondly, the correction of the operator σ is incorporated in the new REFINE function, algorithm 4.6. It contains a few minor improvements over $\text{REFINE}_{\text{GPP}}$: using list notations for variable *Sort* and preventing empty blocks from being added to Π. The actual correction is in line 21: if for some $\gamma \in \Sigma$ and $\alpha \in \Pi$ with $\alpha \rightarrow_\exists \gamma$ we have $\mathit{Stable}(\alpha) \cap \mathit{Row}(\gamma) \neq \emptyset$ then we add γ to $\mathit{Stable}(\alpha)$.

Note that REFINE is not functional in the sense that its returned partition is not uniquely defined; it depends on the particular reverse topological sorting that is chosen in line 4, as demonstrated by the following example.

Example 4.17. Consider the graph $G = (V, \rightarrow)$ of figure 4.3 and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{\alpha, \beta, \gamma, \delta, \varepsilon\}$ as depicted and $P = \mathcal{I} \cup \{(\beta, \delta), (\delta, \gamma)\}$. Then $S = [\varepsilon, \gamma, \delta, \beta, \alpha]$ and $S' = [\gamma, \delta, \beta, \varepsilon, \alpha]$ are reverse topological sortings of Σ with respect to P. Let Π and Π' be the partitions returned by $\text{REFINE}(\Sigma, P)$ on sortings S and S' respectively. Then $\Pi = \{\{a_0\}, \{a_1\}, \{a_2\}\}$ and $\Pi' = \{\{a_0, a_1\}, \{a_2\}\}$. □

Similar to the construction of counterexample 4.9, this example can be embedded in example 4.8 to obtain an example with a transitive relation for which the outcome of the second refinement depends on the reverse topological sorting. Because of this non-functionality of REFINE, we use the operator ρ of section 4.6 to prove correctness of PA in section 4.7.2. We show in section 4.8 that the space and time complexities of PA_{GPP} have been maintained.

4.7.1 The correction of a minor mistake

Apart from the error in PA_{GPP} that results from the incorrect σ operator, we found another, minor mistake in the algorithm. We describe it in this section and propose a solution. The mistake is shown by the following example.

Example 4.18. Consider the graph $G = (V, \rightarrow)$ in figure 4.4 and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{\alpha, \beta\}$ as depicted and $P = \mathcal{I} \cup \{(\alpha, \beta)\}$. Observe that the solution to the GCPP over G and $\langle \Sigma, P \rangle$ is $\langle \Xi, \preceq \rangle$ with $\Xi = \{\alpha_0, \alpha_1, \beta\}$ and $\preceq = \mathcal{I} \cup \{(\alpha_1, \alpha_0)\}$ where $\alpha_i = \{a_i\}$. After the first iteration of $\text{PA}_{\text{GPP}}(G, \langle \Sigma, P \rangle)$,

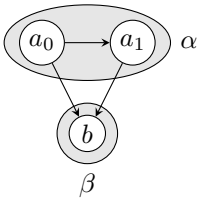


Figure 4.4. Example for which the algorithm PA_{GPP} terminates prematurely.

we have $\Sigma_1 = \Sigma_0 = \Sigma$ and $P_1 = \mathcal{I}$. The algorithm then terminates because $\mathit{change} = \bot$, and $\langle \Sigma_1, P_1 \rangle$ is its answer to the GCPP over G and $\langle \Sigma, P \rangle$. Obviously $\langle \Sigma_1, P_1 \rangle \neq \langle \Xi, \preceq \rangle$, so this answer is wrong. □

The correctness of the algorithm PA_{GPP} hinges on the assumption that whenever $\textsc{Refine}_{\text{GPP}}(\Pi, Q, \mathit{change})$ returns the input partition Π, then also the relation Q will be unaffected by $\textsc{Update}_{\text{GPP}}$, *i.e.* $\textsc{Update}_{\text{GPP}}(\Pi, Q, \Pi)$ returns Q. This is the upshot of theorem 4.15 of [51] and it is essential in the complexity analysis of the algorithm. However, the above example shows that it does not hold in general. In the next section we show that the assumption does hold under the condition that Q itself is obtained as output of $\textsc{Update}_{\text{GPP}}$ (proposition 4.20). Therefore, this error in PA_{GPP} can be fixed, without violating the complexity analysis, by insisting that at least two refine–update steps are performed prior to termination.

4.7.2 Correctness of PA

From here on we will use the correctness of the function $\textsc{Update}_{\text{GPP}}$, as established by Gentilini *et al.* [52]. This correctness can be summarized as follows:

Proposition 4.19. *Let $\langle \Sigma, P \rangle$ be a partition pair over a graph $(V, \to)$, and Π be a partition over V that is finer than Σ. Then there exists a unique relation $Q \subseteq P(\Pi)$ satisfying condition (2σ) of definition 4.5. Moreover, this relation is returned by $\textsc{Update}_{\text{GPP}}(\Sigma, P, \Pi)$.*

Proof. The union of all relations $Q \subseteq P(\Pi)$ such that (b) and (c) hold for all $(\alpha, \beta) \in Q$ is itself a relation with these properties. The last claim has been established in [52]. □

Using proposition 4.19, we obtain the result announced in section 4.7.1: the following proposition implies that if a call to Refine in the while-loop of PA does not split any blocks, then the subsequent call to $\textsc{Update}_{\text{GPP}}$ will return its input relation. The requirement that this relation has been computed by a previous call to $\textsc{Update}_{\text{GPP}}$ is guaranteed by line 2.

Proposition 4.20. *Let $\langle \Sigma, P \rangle$ and $\langle \Pi, Q \rangle$ be partition pairs over a graph such that Π is finer than Σ and $\textsc{Update}_{\text{GPP}}(\Sigma, P, \Pi)$ returns Q. Then $\textsc{Update}_{\text{GPP}}(\Pi, Q, \Pi)$ also returns Q.*

Proof. By proposition 4.19, $\textsc{Update}_{\text{GPP}}(\Pi, Q, \Pi)$ returns the largest relation $Q' \subseteq Q(\Pi)$ satisfying conditions (b) and (c) of definition 4.5 with respect to Π, Q and Π (*i.e.* substituting Q', Π, Q and Π for Q, Σ, P and Π in these conditions, respectively). We have to prove that $Q' = Q$. As $Q = Q(\Pi)$ it suffices to show that Q satisfies (b) and (c) with Π substituted for Σ and Q for P. Under these substitutions (b) becomes equal to (c). By proposition 4.19 applied to $\textsc{Update}_{\text{GPP}}(\Sigma, P, \Pi)$, Q satisfies this condition. □

Let $\langle \Sigma_i, P_i \rangle_{1 \leq i \leq k}$ be the sequence of partition pairs that PA produces. The following proposition says that every P_i is acyclic and that the sequence is decreasing. The former implies that PA will never deadlock due to the inability to find a reverse topological sorting (see line 4 of Refine). The latter implies that the algorithm terminates.

Proposition 4.21. *Let $\langle \Sigma, P \rangle$ be a partition pair over a graph $(V, \rightarrow)$. Suppose* $\textsc{Refine}(\Sigma, P)$ *returns* Π *and* $\textsc{Update}_{\text{GPP}}(\Sigma, P, \Pi)$ *returns* Q*. Then $\langle \Pi, Q \rangle$ is a partition pair with $\langle \Pi, Q \rangle \leq \langle \Sigma, P \rangle$.*

Proof. From algorithm 4.6 and the fact that Σ is a partition, it is not hard to see that Π is a partition that is, moreover, finer than Σ. Also, by proposition 4.19 we have that $Q \subseteq P(\Pi)$. Hence $\langle \Pi, Q \rangle \leq \langle \Sigma, P \rangle$. To prove that the pair $\langle \Pi, Q \rangle$ is a partition pair, we need to prove reflexivity and acyclicity of Q. Using reflexivity of P and $P(\Pi)$, the identity relation $\mathcal{I}$ trivially satisfies conditions (b) and (c) of definition 4.5. Hence proposition 4.19 implies that $\mathcal{I} \subseteq Q$, *i.e.* Q is reflexive.

Suppose Q contains a cycle: there are pairwise distinct $\alpha_0, \ldots, \alpha_{n-1} \in \Pi$ for $n > 1$ such that $(\alpha_i, \alpha_{i+1 \bmod n}) \in Q$ for $0 \leq i < n$. By acyclicity of P, it must be that these α_i are all subsets of the same block $\alpha \in \Sigma$. Let $\gamma \in \Sigma$ and $\alpha' \subseteq \alpha$ be the first blocks considered in an iteration of Refine's main for-loop (line 8) such that γ splits α' into an α'_1 and an α'_2 such that $\alpha_i \subseteq \alpha'_1$ and $\alpha_j \subseteq \alpha'_2$ for some $0 \leq i, j < n$. Then $\mathit{Stable}(\alpha') \cap \mathit{Row}(\gamma) = \emptyset$. For any $0 \leq k < n$ we have either $\alpha_k \rightarrow_\forall \gamma$ or $\alpha_k \not\rightarrow_\exists \gamma$, and both possibilities occur. Take $0 \leq i < n$ such that $\alpha_{i-1 \bmod n} \rightarrow_\forall \gamma$ and $\alpha_i \not\rightarrow_\exists \gamma$. By proposition 4.19, Q satisfies (b) of definition 4.5. Hence $\exists \gamma' \in \Sigma \,.\, ((\gamma, \gamma') \in P \wedge \alpha_i \rightarrow_\exists \gamma'))$. As $(\gamma, \gamma') \in P$, in Refine's while-loop γ' is considered prior to γ. Consider the unique iteration of Refine's main for-loop (line 8) involving γ' and an α'' with $\alpha' \subseteq \alpha'' \subseteq \alpha$ — observe that $\alpha'' \rightarrow_\exists \gamma'$. At the end of that iteration we have obtained a block α''' with $\alpha' \subseteq \alpha''' \subseteq \alpha''$ and $\gamma' \in \mathit{Stable}(\alpha''')$. It follows that at the later iteration involving γ and α' we have $\gamma' \in \mathit{Stable}(\alpha') \cap \mathit{Row}(\gamma)$, which is a contradiction. □

Corollary 4.22. *For any graph G and any partition pair $\langle \Sigma, P \rangle$ over G, the algorithm* $\text{PA}(G, \langle \Sigma, P \rangle)$ *terminates.* □

Having established termination of PA, we prove its correctness by establishing a connection with the operator ρ of definition 4.10. More precisely, we prove that a

single refine–update step by PA yields a partition pair that is (1) at most as large as the one produced by ρ (lemma 4.26), and (2) at least as large as the GCPP solution (lemma 4.27). This then implies that PA computes the GCPP solution (theorem 4.28). We first establish three lemmas that are necessary for proving the above results (lemmas 4.23–4.25).

Lemma 4.23. *The following predicate is an invariant for the while-loop of algorithm 4.6:*

$$\forall \beta \in \Pi \cup A \,.\, \forall \varepsilon \in \mathit{Stable}(\beta) \,.\, \exists \delta \in \Sigma \,.\, ((\varepsilon, \delta) \in P^+ \wedge \beta \rightarrow_\forall \delta) \,.$$

Proof. The predicate holds trivially after the initialization of the *Stable*-sets in line 2. The only points where it could be influenced are at lines 14, 18 and 21. For $\varepsilon \neq \gamma$ lines 14 and 18 are harmless because if $\alpha_i \subseteq \alpha$ and $\alpha \rightarrow_\forall \delta$ then certainly $\alpha_i \rightarrow_\forall \delta$. For $\varepsilon = \gamma$ and $\beta = \alpha_1$, at line 14 the predicate holds by construction of α_1, taking $\delta := \gamma$. Finally, line 21 is only executed when there is an $\varepsilon \in \mathit{Stable}(\alpha) \cap \mathit{Row}(\gamma)$. As $(\gamma, \varepsilon) \in P$, the predicate holds for γ and α because it held already for ε and α. □

Lemma 4.24. *Let $\langle \Sigma, P \rangle$ be a partition pair over a graph $(V, \rightarrow)$ and suppose that* Refine(Σ, P) *returns Π. Then:*

$$\forall \alpha \in \Pi \,.\, \forall \gamma \in \Sigma \,.\, (\alpha \rightarrow_\exists \gamma \implies \exists \delta \in \Sigma \,.\, ((\gamma, \delta) \in P^+ \wedge \alpha \rightarrow_\forall \delta)) \,.$$

Proof. Let $\alpha \in \Pi$ and $\gamma \in \Sigma$ such that $\alpha \rightarrow_\exists \gamma$. In the computation of Π, take the unique iteration of Refine's main for-loop (line 8) in which γ and an α' are considered with $\alpha \subseteq \alpha'$. Then $\alpha' \rightarrow_\exists \gamma$ and there are two cases:

- $\mathit{Stable}(\alpha') \cap \mathit{Row}(\gamma) = \emptyset$: Then α' is split into α_1 and α_2 such that $\alpha_1 \rightarrow_\forall \gamma$ and $\alpha_2 \not\rightarrow_\exists \gamma$. It must be that $\alpha \subseteq \alpha_1$. Then $\alpha \rightarrow_\forall \gamma$ and $(\gamma, \gamma) \in P^+$.
- $\mathit{Stable}(\alpha') \cap \mathit{Row}(\gamma) \neq \emptyset$: Then γ is added to $\mathit{Stable}(\alpha')$. lemma 4.23 gives us $\exists \delta \in \Sigma \,.\, ((\gamma, \delta) \in P^+ \wedge \alpha' \rightarrow_\forall \delta)$. As $\alpha \subseteq \alpha'$ we have $\alpha \rightarrow_\forall \delta$. □

Lemma 4.25. *Let $\langle \Sigma, P \rangle$ and $\langle \Pi, Q \rangle$ be partition pairs over a graph $(V, \rightarrow)$ such that $\langle \Pi, Q \rangle \leq \langle \Sigma, P \rangle$ and let P be transitive. If $\langle \Pi, Q \rangle$ satisfies (4.1) of definition 4.10 with respect to $\langle \Sigma, P \rangle$ then so does $\langle \Pi, Q^+ \rangle$.*

Proof. Suppose $\langle \Pi, Q \rangle$ satisfies (4.1) with respect to $\langle \Sigma, P \rangle$ and take $(\alpha, \beta) \in Q^+$ and $\gamma \in \Sigma$ such that $\alpha \rightarrow_\exists \gamma$. There are $\alpha_0, \ldots, \alpha_n \in \Pi$ for $n \geq 0$ such that $\alpha = \alpha_0$, $\beta = \alpha_n$ and $(\alpha_i, \alpha_{i+1}) \in Q$ for $0 \leq i < n$. Applying (4.1) n times we obtain $\delta_1, \ldots, \delta_n \in \Sigma$ such that $\alpha_i \rightarrow_\forall \delta_i$ (and thus $\alpha_i \rightarrow_\exists \delta_i$) for $1 \leq i \leq n$, $(\gamma, \delta_1) \in P$ and $(\delta_i, \delta_{i+1}) \in P$ for $1 \leq i < n$. Hence $\beta \rightarrow_\forall \delta_n$ and $(\gamma, \delta_n) \in P$ by transitivity of P. □

As described earlier, the following lemmas state that Refine and Update$_{\text{GPP}}$ converge towards a fixed point at least as fast as ρ without ever diverging from the path towards the GCPP solution.

Lemma 4.26. *Let $\langle \Sigma, P\rangle$ be a partition pair over a graph $(V, \to)$,* Refine(Σ, P) *return* Π*, and* UpdateGPP(Σ, P, Π) *return* Q*. Then* $\langle \Pi, Q^+\rangle \leq \rho(\langle \Sigma, P^+\rangle)$*.*

Proof. By proposition 4.21, $\langle \Pi, Q\rangle$ is a partition pair with $\langle \Pi, Q\rangle \leq \langle \Sigma, P\rangle \leq \langle \Sigma, P^+\rangle$. By definition 4.10, $\rho(\langle \Sigma, P^+\rangle)$ is the $\leq$-largest partition pair smaller than $\langle \Sigma, P^+\rangle$ that satisfies (4.1) with respect to $\langle \Sigma, P^+\rangle$. So the statement follows if we prove that $\langle \Pi, Q^+\rangle$ satisfies (4.1) with respect to $\langle \Sigma, P^+\rangle$. By lemma 4.25 it suffices to show that $\langle \Pi, Q\rangle$ satisfies (4.1) with respect to $\langle \Sigma, P^+\rangle$. Let $(\alpha, \beta) \in Q$ and $\alpha \to_\exists \gamma$ for $\gamma \in \Sigma$. Using lemma 4.24, take $\delta \in \Sigma$ such that $(\gamma, \delta) \in P^+$ and $\alpha \to_\forall \delta$. By proposition 4.19, $\exists \delta' \in \Sigma \,.\, (\delta, \delta') \in P \wedge \beta \to_\exists \delta'$. For that δ', by lemma 4.24 it holds that $\exists \gamma' \in \Sigma \,.\, (\delta', \gamma') \in P^+ \wedge \beta \to_\forall \gamma'$. For this γ' it holds that $(\gamma, \gamma') \in P^+$. Hence $\langle \Pi, Q\rangle$ satisfies (4.1) with respect to $\langle \Sigma, P^+\rangle$. □

Lemma 4.27. *Let $\langle \Sigma, P\rangle$ and $\langle \Pi, Q\rangle$ be partition pairs over a graph $G = (V, \to)$, $\langle \Xi, \preceq\rangle$ be the solution of the GCPP over G and $\langle \Sigma, P\rangle$, and $\langle \Xi, \preceq\rangle \leq \langle \Pi, Q\rangle$. Suppose that* Refine$(\Pi, Q)$ *returns* Π' *and* UpdateGPP(Π, Q, Π') *returns* Q'*. Then* $\langle \Xi, \preceq\rangle \leq \langle \Pi', Q'\rangle$*.*

Proof. We have to prove that (i) Ξ is finer than Π' and (ii) $\preceq \subseteq Q'(\Xi)$.

Ad (i). Let $\alpha \in \Xi$ and $\alpha_\Pi \in \Pi$ such that $\alpha \subseteq \alpha_\Pi$. By contradiction, suppose there is no $\alpha' \in \Pi'$ such that $\alpha \subseteq \alpha'$. Hence, there are $a_1, a_2 \in \alpha$ such that Refine at some point separates $a_1 \in \alpha_\Pi$ from $a_2 \in \alpha_\Pi$. Let $\alpha'_\Pi \subseteq \alpha_\Pi$ be such that $a_1, a_2 \in \alpha'_\Pi$ and a_1 and a_2 got separated when α'_Π was split by a block $\gamma_\Pi \in \Pi$. Hence $\mathit{Stable}(\alpha'_\Pi) \cap \mathit{Row}(\gamma_\Pi) = \emptyset$. Consider the case where $a_1 \to \gamma_\Pi$ and $a_2 \not\to \gamma_\Pi$. The case with $a_1 \not\to \gamma_\Pi$ and $a_2 \to \gamma_\Pi$ is fully symmetrical. Let $\gamma \in \Xi$ be such that $\gamma \subseteq \gamma_\Pi$ and $a_1 \to \gamma$. As $\alpha \to_\exists \gamma$ and $\langle \Xi, \preceq\rangle$ is stable with respect to $\to$, there must be a $\delta \in \Xi$ with $\gamma \preceq \delta$ and $\alpha \to_\forall \delta$. Let $\delta_\Pi \in \Pi$ be such that $\delta \subseteq \delta_\Pi$. Then $(\gamma_\Pi, \delta_\Pi) \in Q$, using that $\langle \Xi, \preceq\rangle \leq \langle \Pi, Q\rangle$. So $\delta_\Pi \in \mathit{Row}(\gamma_\Pi)$ and δ_Π is before γ_Π in the reverse topological sorting of Π with respect to Q. As $a_2 \not\to \gamma_\Pi$ we have $\alpha \not\to_\forall \gamma_\Pi$, yet $\alpha \to_\forall \delta_\Pi$, hence $\gamma_\Pi \neq \delta_\Pi$. Let $\alpha''_\Pi \subseteq \alpha_\Pi$ be the block containing a_1 and a_2 when blocks were split with respect to δ_Π by Refine. Observe that $\alpha''_\Pi \to_\exists \delta_\Pi$, so there were two cases:

- $\mathit{Stable}(\alpha''_\Pi) \cap \mathit{Row}(\delta_\Pi) = \emptyset$: Then α''_Π may have been split, but this did not separate a_1 and a_2. Then $\alpha'_\Pi \subseteq (\alpha''_\Pi \cap \to^{-1}(\delta_\Pi))$ and hence $\delta_\Pi \in \mathit{Stable}(\alpha'_\Pi)$.
- $\mathit{Stable}(\alpha''_\Pi) \cap \mathit{Row}(\delta_\Pi) \neq \emptyset$: Then δ_Π was added to $\mathit{Stable}(\alpha''_\Pi)$ (line 21) and because $\alpha'_\Pi \subseteq \alpha''_\Pi$ we have $\delta_\Pi \in \mathit{Stable}(\alpha'_\Pi)$.

In both cases we have that $\delta_\Pi \in \mathit{Stable}(\alpha'_\Pi) \cap \mathit{Row}(\gamma_\Pi)$, which contradicts the fact that $\mathit{Stable}(\alpha'_\Pi) \cap \mathit{Row}(\gamma_\Pi) = \emptyset$.

Ad (ii). Let $Q'_\preceq := \{(\alpha, \beta) \in \Pi' \times \Pi' \mid \exists (\alpha_\Xi, \beta_\Xi) \in \preceq \,.\, \alpha_\Xi \subseteq \alpha \wedge \beta_\Xi \subseteq \beta\}$. We will show that $Q'_\preceq \subseteq Q'$, which immediately yields $\preceq \subseteq Q'_\preceq(\Xi) \subseteq Q'(\Xi)$. To this end, using proposition 4.19, we establish that $Q'_\preceq \subseteq Q(\Pi')$ and any pair $(\alpha, \beta) \in Q'_\preceq$ satisfies conditions (b) and (c) of definition 4.5, reading Π, Q, Π' and $Q'_\preceq$ for Σ, P, Π and Q, respectively.

- $Q'_{\preceq} \subseteq Q(\Pi')$: Let $(\alpha, \beta) \in Q'_{\preceq}$. Take $\alpha_\Pi, \beta_\Pi \in \Pi$ such that $\alpha \subseteq \alpha_\Pi$ and $\beta \subseteq \beta_\Pi$. Because $\preceq \subseteq Q(\Xi)$ we have $(\alpha_\Pi, \beta_\Pi) \in Q$, and hence $(\alpha, \beta) \in Q(\Pi')$.
- *Condition* (b): Let $(\alpha, \beta) \in Q'_{\preceq}$ and $\gamma \in \Pi$ such that $\alpha \rightarrow_\forall \gamma$. Take $\alpha_\Xi, \beta_\Xi \in \Xi$ such that $\alpha_\Xi \subseteq \alpha$, $\beta_\Xi \subseteq \beta$ and $\alpha_\Xi \preceq \beta_\Xi$. Also take $\gamma_\Xi \in \Xi$ such that $\gamma_\Xi \subseteq \gamma$ and $\alpha_\Xi \rightarrow_\exists \gamma_\Xi$. Because $\langle \Xi, \preceq \rangle$ is stable with respect to $\rightarrow$ we obtain a $\delta_\Xi \in \Xi$ such that $\gamma_\Xi \preceq \delta_\Xi$ and $\beta_\Xi \rightarrow_\forall \delta_\Xi$. Take $\delta \in \Pi$ such that $\delta_\Xi \subseteq \delta$. As $\preceq \subseteq Q(\Xi)$ we have $(\gamma, \delta) \in Q$. We also obtain $\beta \rightarrow_\exists \delta$.
- *Condition* (c): Let $(\alpha, \beta) \in Q'_{\preceq}$ and $\gamma \in \Pi'$ such that $\alpha \rightarrow_\forall \gamma$. Take $\alpha_\Xi, \beta_\Xi, \gamma_\Xi \in \Xi$ and obtain $\delta_\Xi \in \Xi$ exactly as above. Take $\delta \in \Pi'$ such that $\delta_\Xi \subseteq \delta$. We have $(\gamma, \delta) \in Q'_{\preceq}$ by construction. Again we obtain $\beta \rightarrow_\exists \delta$. □

In combination with the monotonicity of ρ (proposition 4.12), lemmas 4.26 and 4.27 imply the correctness of PA, as stated by the following theorem.

Theorem 4.28. *Let $\langle \Sigma, P \rangle$ be a partition pair over a graph $G = (V, \rightarrow)$. Let k be the value of variable i upon termination of* $\mathrm{PA}(G, \langle \Sigma, P^+ \rangle)$. *Then $\langle \Sigma_k, P_k \rangle$ is the solution of the GCPP over G and $\langle \Sigma, P \rangle$.*

Proof. Let the sequence of partition pairs $\langle \Sigma_i, P_i \rangle_{1 \leq i \leq k}$ be obtained by running $\mathrm{PA}(G, \langle \Sigma, P^+ \rangle)$ until it terminates, implying that $k \geq 2$ and $\Sigma_k = \Sigma_{k-1}$. Proposition 4.20 yields $P_k = P_{k-1}$. Extend this sequence by defining $\langle \Sigma_0, P_0 \rangle := \langle \Sigma, P^+ \rangle$ and $\langle \Sigma_i, P_i \rangle := \langle \Sigma_k, P_k \rangle$ for $i > k$. Now for all $i \geq 0$ we have that $\textsc{Refine}(\Sigma_i, P_i)$ returns Σ_{i+1} and $\textsc{Update}_{\mathrm{GPP}}(\Sigma_i, P_i, \Sigma_{i+1})$ returns P_{i+1}. Let $\langle \Xi, \preceq \rangle$ be the solution of the GCPP over G and $\langle \Sigma, P \rangle$. We need to show that $\langle \Sigma_k, P_k \rangle = \langle \Xi, \preceq \rangle$, for which we require the following properties:

$$\text{(P1)} \qquad \langle \Xi, \preceq \rangle \leq \langle \Sigma_i, P_i \rangle \quad \text{for all } i \geq 0$$

$$\text{(P2)} \qquad \langle \Sigma_i, P_i^+ \rangle \leq \rho^i(\langle \Sigma, P^+ \rangle) \quad \text{for all } i \geq 0.$$

Proof of (P1): By definition $\langle \Xi, \preceq \rangle \leq \langle \Sigma, P^+ \rangle = \langle \Sigma_0, P_0 \rangle$, and lemma 4.27 yields $\langle \Xi, \preceq \rangle \leq \langle \Sigma_i, P_i \rangle$ for all $i > 0$, by induction on i.

Proof of (P2): By induction on i. If $i = 0$ then $\langle \Sigma_0, P_0^+ \rangle = \langle \Sigma, P^+ \rangle = \rho^0(\langle \Sigma, P^+ \rangle)$. For the inductive step, suppose $\langle \Sigma_i, P_i^+ \rangle \leq \rho^i(\langle \Sigma, P^+ \rangle)$. Then, using lemma 4.26 and proposition 4.12: $\langle \Sigma_{i+1}, P_{i+1}^+ \rangle \leq \rho(\langle \Sigma_i, P_i^+ \rangle) \leq \rho^{i+1}(\langle \Sigma, P^+ \rangle)$.

Applying (P1) *and* (P2): By proposition 4.13 and theorem 4.16 there is an $n > 0$ such that $\rho^n(\langle \Sigma, P^+ \rangle) = \langle \Xi, \preceq \rangle$, so, using proposition 4.21:

$$\langle \Xi, \preceq \rangle \overset{\text{(P1)}}{\leq} \langle \Sigma_{n+1}, P_{n+1} \rangle \overset{4.21}{\leq} \langle \Sigma_n, P_n \rangle \leq \langle \Sigma_n, P_n^+ \rangle \overset{\text{(P2)}}{\leq} \rho^n(\langle \Sigma, P^+ \rangle) = \langle \Xi, \preceq \rangle .$$

Thus $\langle \Sigma_{n+1}, P_{n+1} \rangle = \langle \Sigma_n, P_n \rangle$, so $k \leq n+1$, and $\langle \Sigma_k, P_k \rangle = \langle \Sigma_n, P_n \rangle = \langle \Xi, \preceq \rangle$. □

Procedure 4.1. Partitioning$(V, \rightarrow, \Sigma, P \mid \Pi, Q)$

```
1: Refine(V, →, Σ, P | Π, parent, change);
2: Update(V, →, Σ, P, Π, parent | Q);
3: change := ⊤;
4: while change do
5:     change := ⊥;
6:     Σ, P := Π, Q;
7:     Refine(V, →, Σ, P | Π, parent, change);
8:     Update(V, →, Σ, P, Π, parent | Q)
9: end while
```

4.8 Complexity analysis

We implement PA in sufficient detail to determine its space and time complexities, see procedures 4.1–4.5. In the parameter list of a procedure, we use a vertical bar to separate the input parameters (on the left) from the output parameters (on the right). Algorithm PA is implemented by the Partitioning procedure (procedure 4.1). At any time it stores only two of the k partition pairs constructed by PA: $\langle \Sigma, P \rangle$ and $\langle \Pi, Q \rangle$ which correspond with $\langle \Sigma_i, P_i \rangle$ and $\langle \Sigma_{i+1}, P_{i+1} \rangle$ in iteration i of PA, respectively. Let $N := |V|$ be the number of vertices and $T := |\rightarrow|$ be the number of transitions of the input graph. We expect that $T < N^2$ in general. Any partition Σ is stored as a set of block names $\widehat{\Sigma}$, which are the first $S_\Sigma := |\widehat{\Sigma}|$ natural numbers, together with a function $\mathit{block}_\Sigma : V \rightarrow \widehat{\Sigma}$ that associates a block with every vertex. We define $S := S_\Pi$ where Π is the output partition of the algorithm. Note that $S \leq N$. A relation P on a partition Σ is stored as a function $P : \widehat{\Sigma} \times \widehat{\Sigma} \rightarrow \mathbb{B}$. The following auxiliary data structures are used:

- $\mathit{parent} : \widehat{\Pi} \rightarrow \widehat{\Sigma}$ stores for any Π-block the name of its parent block in Σ.
- $\mathit{stable} : \widehat{\Pi} \times \widehat{\Sigma} \rightarrow \mathbb{B}$ stores *Stable* of Refine as follows: we have $\mathit{stable}(\alpha, \gamma)$ iff $\gamma \in \mathit{Stable}(\alpha)$.
- $\mathit{split} : \widehat{\Pi} \rightarrow \widehat{\Pi}$ stores for every $\alpha \in \Pi$ that is split into non-empty blocks α_1 and α_2, the name of α_2, which is the first available block name at the time of splitting (*i.e.* S_Π). The name of α is reused for α_1. For all $\alpha \in \Pi$ that are not split into two non-empty blocks, $\mathit{split}(\alpha)$ is simply set to α.
- $\mathit{splitoff} : V \rightarrow \mathbb{B}$ indicates for every vertex whether it has to be moved from its block α to the new block $\mathit{split}(\alpha)$.
- $\mathit{match} : \widehat{\Pi} \times \widehat{\Sigma} \rightarrow \mathbb{B}$ indicates for every $\beta \in \Pi$ and $\gamma \in \Sigma$ whether there is a $\delta \in \Sigma$ such that $(\gamma, \delta) \in P$ and $\beta \rightarrow_\exists \delta$.

The Refine function is implemented by the Refine and Initialize procedures (procedures 4.2 and 4.3) in the following way. The auxiliary set A of Refine has been removed: we add new blocks to Π directly (line 29 of Refine) and use variable *last* to ensure that, in the loop of lines 12–34, we only iterate over Π-blocks that

were already in Π at the start of that loop. When a block α is split by a block γ into α_1 and α_2, the actual transfer of vertices from α to α_2 is postponed until all blocks α have been considered (lines 35–39). Consequently, the condition $\alpha_2 \neq \emptyset$ of Refine has been replaced by the equivalent $\alpha \not\to_\forall \gamma$ (line 20). The data structures needed to facilitate this postponed splitting are *split*, which is computed in lines 21 and 24, and *splitoff* and $\to_\forall$, which are computed by Initialize in line 10 in addition to $\to_\exists$. The *Row*-sets of Refine have been removed: lines 14–18 compute the negation of the condition $\mathit{Stable}(\alpha) \cap \mathit{Row}(\gamma) = \emptyset$ of Refine. The result is stored in $\mathit{stable}(\alpha, \gamma)$, thereby implementing line 21 of Refine. Finally, Refine also computes the function *parent* (lines 5 and 25), which is needed by Update.

The $\text{Update}_{\text{GPP}}$ function is implemented by the procedures Update, Filter and Cleanup (procedures 4.4–4.6) in the following way. The New_HHK function corresponds with Filter, where *rem* has been replaced by *match*. This is because storing *rem* would require too much space as explained in section 4.8.1. For every $\gamma \in \Sigma$, $\mathit{rem}(\gamma)$ gives the set of $\beta \in \Pi$ for which there is no $\delta \in \Sigma$ such that $(\gamma, \delta) \in P$ and $\beta \to_\exists \delta$. Observe that *match* stores the complement of *rem* in the sense that $\mathit{match}(\beta, \gamma)$ iff $\beta \notin \mathit{rem}(\gamma)$. In the second call to Filter (line 11 of Update), the parameters Σ and P of Filter are set to Π and Q, respectively. In this case, whenever a pair (α, β) is removed from Q (line 13 of Filter) it may be that for some $\beta_1 \in \Pi$ with $\beta_1 \to_\exists \beta$, there no longer is a $\delta \in \Pi$ such that $(\alpha, \delta) \in Q$ and $\beta_1 \to_\exists \delta$. For those β_1 Cleanup sets $\mathit{match}(\beta_1, \alpha)$ to $\bot$ (lines 3–8). In turn, this may lead to the removal of more pairs from Q (line 12) upon which Cleanup has to be called again (line 13).

4.8.1 Space complexity

The following data structures constitute the input of the algorithm:

- V is represented as the first N natural numbers. Hence, storing V amounts to storing N, which takes space $\mathcal{O}(\log N)$.
- $\to \subseteq V \times V$ is stored as a list of pairs of vertices, which takes space $\mathcal{O}(T \log N)$.
- Σ is stored as the number S_Σ and the function block_Σ. This requires $\mathcal{O}(\log S_\Sigma + N \log S_\Sigma)$ space.
- P is stored as function of type $\widehat{\Sigma} \times \widehat{\Sigma} \to \mathbb{B}$, which takes space $\mathcal{O}(S_\Sigma^2)$.

As V and $\to$ are read-only input, they are not counted in determining the space complexity; Σ and P are modified while the algorithm is running. The output consists of a partition Π of size S and a relation $Q \subseteq \widehat{\Pi} \times \widehat{\Pi}$. Thus, storing the output takes space $\mathcal{O}(\log S + N \log S + S^2)$. As the output partition is finer than all intermediately computed partitions, the total space used for storing Σ, P, Π and Q is bounded by $2 \cdot (S^2 + (N+1) \log S)$. Assuming that $N > 0$, this amounts to $\mathcal{O}(S^2 + N \log S)$. The other data structures do not require more space: *splitoff* requires $\mathcal{O}(N)$, *parent* and *split* require $\mathcal{O}(S \log S)$ each, and *stable*, $\to_\exists$, $\to_\forall$ and *match* require $\mathcal{O}(S^2)$ each.

Procedure 4.2. Refine$(V, \rightarrow, \Sigma, P \mid \Pi, \mathit{parent}, \mathit{change})$

```
1: S_Π := S_Σ;
2: for all a ∈ V do block_Π(a) := block_Σ(a) end for;
3: for all α ∈ Π̂ do
4:     for all γ ∈ Σ̂ do stable(α, γ) := ⊥ end for;
5:     parent(α) := α
6: end for;
7: ReverseTopologicalSort(Σ̂, P | Sort);
8: while Sort ≠ [ ] do
9:     γ := head(Sort);
10:    Initialize(V, →, γ, block_Σ, Π | splitoff, →_∃, →_∀);
11:    last := S_Π − 1;
12:    for all α ∈ {0, ..., last} do
13:        if α →_∃ γ then
14:            for all δ ∈ Σ̂ do
15:                if stable(α, δ) ∧ P(γ, δ) then
16:                    stable(α, γ) := ⊤
17:                end if
18:            end for;
19:            if ¬stable(α, γ) then
20:                if α →_∀ γ then
21:                    split(α) := α
22:                else
23:                    change := ⊤;
24:                    split(α) := S_Π;
25:                    parent(S_Π) := parent(α);
26:                    for all δ ∈ Σ̂ do
27:                        stable(S_Π, δ) := stable(α, δ)
28:                    end for;
29:                    S_Π := S_Π + 1
30:                end if;
31:                stable(α, γ) := ⊤
32:            end if
33:        end if
34:    end for;
35:    for all p ∈ V do
36:        if splitoff(p) then
37:            block_Π(p) := split(block_Π(p))
38:        end if
39:    end for;
40:    Sort := tail(Sort)
41: end while
```

Procedure 4.3. Initialize$(V, \to, \gamma, \mathit{block}_\Sigma, \Pi \mid \mathit{splitoff}, \to_\exists, \to_\forall)$

for all $a \in V$ **do** $\mathit{splitoff}(a) := \top$ **end for**;
for all $\alpha \in \widehat{\Pi}$ **do**
    $\alpha \to_\exists \gamma := \bot$;
    $\alpha \to_\forall \gamma := \top$
end for;
for all $(a, c) \in \to$ **do**
    if $\mathit{block}_\Sigma(c) = \gamma$ **then**
        $\mathit{splitoff}(a) := \bot$;
        $\mathit{block}_\Pi(a) \to_\exists \gamma := \top$
    end if
end for;
for all $a \in V$ **do**
    if $\mathit{splitoff}(a)$ **then**
        $\mathit{block}_\Pi(a) \to_\forall \gamma := \bot$
    end if
end for

Procedure 4.4. Update$(V, \to, \Sigma, P, \Pi, \mathit{parent} \mid Q)$

for all $\alpha, \beta \in \widehat{\Pi}$ **do**
    $Q(\alpha, \beta) := P(\mathit{parent}(\alpha), \mathit{parent}(\beta))$
end for;
for all $\gamma \in \widehat{\Sigma}$ **do**
    Initialize$(V, \to, \gamma, \mathit{block}_\Sigma, \Pi \mid \mathit{splitoff}, \to_\exists, \to_\forall)$
end for;
Filter$(\Sigma, P, \Pi, Q, \to_\exists, \to_\forall, \bot \mid Q)$;
for all $\gamma \in \widehat{\Pi}$ **do**
    Initialize$(V, \to, \gamma, \mathit{block}_\Pi, \Pi \mid \mathit{splitoff}, \to_\exists, \to_\forall)$
end for;
Filter$(\Pi, Q, \Pi, Q, \to_\exists, \to_\forall, \top \mid Q)$;

Procedure 4.5. Filter$(\Sigma, P, \Pi, Q, \rightarrow_\exists, \rightarrow_\forall, B \mid Q)$

```
 1: for all β ∈ Π̂, γ ∈ Σ̂ do  match(β, γ) := ⊥  end for;
 2: for all β ∈ Π̂, δ ∈ Σ̂ do
 3:     if β →∃ δ then
 4:         for all γ ∈ Σ̂ do
 5:             if P(γ, δ) then  match(β, γ) := ⊤  end if
 6:         end for
 7:     end if
 8: end for;
 9: for all α ∈ Π̂, γ ∈ Σ̂ do
10:     if α →∀ γ then
11:         for all β ∈ Π̂ do
12:             if Q(α, β) ∧ ¬match(β, γ) then
13:                 Q(α, β) := ⊥;
14:                 if B then
15:                     Cleanup(α, β, Π, Q, →∃, →∀ | Q)
16:                 end if
17:             end if
18:         end for
19:     end if
20: end for
```

Procedure 4.6. Cleanup$(\alpha, \beta, \Pi, Q, \rightarrow_\exists, \rightarrow_\forall \mid Q)$

```
 1: for all β1 ∈ Π̂ do
 2:     if β1 →∃ β then
 3:         match(β1, α) := ⊥;
 4:         for all δ ∈ Π̂ do
 5:             if β1 →∃ δ ∧ Q(α, δ) then
 6:                 match(β1, α) := ⊤
 7:             end if
 8:         end for;
 9:         if ¬match(β1, α) then
10:             for all α1 ∈ Π̂ do
11:                 if Q(α1, β1) ∧ α1 →∀ α then
12:                     Q(α1, β1) := ⊥;
13:                     Cleanup(α1, β1, Π, Q, →∃, →∀ | Q)
14:                 end if
15:             end for
16:         end if
17:     end if
18: end for
```

Hence, the space complexity of PA is $\mathcal{O}(S^2 + N \log S)$. Note that the function *rem* used in NEW_HHK would increase this space complexity: it associates a set of blocks with every block, which requires $\mathcal{O}(S^2 \log S)$ space.

4.8.2 Time complexity

Regarding the Refine procedure:

- Lines 1–6 initialize Π, *stable* and *parent*. This takes $\mathcal{O}(N + S^2)$ time.
- It is well known that computing a (reverse) topological sorting with respect to a relation is linear in the size of that relation. Hence, line 7 takes time $\mathcal{O}(S^2)$.
- Regarding the procedure Initialize (procedure 4.3): lines 1–5 require $\mathcal{O}(N+S)$ time, lines 6–11 require $\mathcal{O}(T)$ time, and lines 12–16 require $\mathcal{O}(N)$ time. Hence, using that $S < N$ and assuming that $N < T$, every call to Initialize requires $\mathcal{O}(T)$ time, which amounts to $\mathcal{O}(S \cdot T)$ in total.
- Line 13 is executed $S_\Sigma \cdot S_\Pi$ times. However, lines 14–32 are executed only $|\rightarrow_\exists|$ times, which is bounded by T (see lines 6–11 of Initialize). The loops on lines 14–18 and 26–28 take time $\mathcal{O}(S)$ each. Hence, the loop of lines 12–34 takes $\mathcal{O}(S \cdot T)$ time in total.
- Finally, lines 35–39 take $\mathcal{O}(N)$ time, which amounts to $\mathcal{O}(S \cdot N)$ in total.

Hence, the time complexity of Refine is $\mathcal{O}(S \cdot T)$. Regarding Update:

- Lines 1–3, 4–6 and 8–10 cost $\mathcal{O}(S^2 + S \cdot T + S \cdot T) = \mathcal{O}(S \cdot T)$ time.
- Line 1 of Filter costs $\mathcal{O}(S^2)$. Lines 4–6 are executed $|\rightarrow_\exists| \leq T$ times and take $\mathcal{O}(S)$ time. Hence, the time complexity of lines 2–8 is $\mathcal{O}(S \cdot T)$. The same reasoning applies to lines 9–20, not counting the call to Cleanup.

Hence, the time complexity of Update is $\mathcal{O}(S \cdot T)$ without counting the calls to Cleanup. Let Ξ be the partition that PA produces as part of its output. For every pair of blocks $\alpha, \beta \in \Xi$, it will happen at most once during the course of the algorithm that a pair (α', β') with $\alpha \subseteq \alpha'$ and $\beta \subseteq \beta'$ is removed from of the relation Q, and only in that circumstance is Cleanup called. For every transition $\beta_1 \rightarrow_\exists \beta$ and every block $\alpha \in \Xi$, there is at most one triple $(\alpha', \beta_1', \beta')$ with $\alpha \subseteq \alpha'$, $\beta_1 \subseteq \beta_1'$ and $\beta \subseteq \beta'$, such that Cleanup(α', β') is executed and $\beta_1' \rightarrow_\exists \beta'$. Thus lines 3–16 of Cleanup are executed at most $S \cdot T$ times during the course of the entire algorithm. The complexity of these lines, not counting line 13, is $\mathcal{O}(S)$. Hence, the total cost of all Cleanups executed during a run of Partitioning is $\mathcal{O}(S^2 \cdot T)$.

In summary, as the body of the main loop of Partitioning is executed S times, the total cost of Refine, Update and Cleanup over the entire algorithm is $\mathcal{O}(S^2 \cdot T)$. The time complexity of PA therefore is $\mathcal{O}(S^2 \cdot T)$.

4.9 Conclusions

The correspondence between the simulation problem for finite, labelled graphs and the generalized coarsest partition problem (GCPP) for unlabelled graphs can be easily established. We have shown that the operator σ defined by Gentilini *et al.* [51] to solve the GCPP is flawed. In particular, when applied to a partition pair, the result is not necessarily another partition pair or even well-defined. Moreover, when applied repeatedly to a transitive partition pair, convergence towards a unique fixed point is not guaranteed. Thereby we have shown that σ is not suitable for solving the GCPP.

On the counterexample for correctness of σ, the partitioning algorithm of [51] produces an incorrect answer: two simulation-equivalent states end up in different equivalence classes. We have repaired this algorithm such that it correctly computes the solution of the GCPP. Apart from correcting the error that results from the flaws in the operator σ, we also corrected a minor mistake that caused premature termination of the algorithm on certain input. We have proven the correctness of our algorithm using an auxiliary operator ρ of which we have shown that it solves the GCPP, though inefficiently. Finally, we have implemented our algorithm to show that it has the same space and time complexities as the original partitioning algorithm.

Another way to repair the algorithm of [51] may be to use the relation P^+ instead of P in the $\textsc{Refine}_{\text{GPP}}$ function. The thusly obtained algorithm would converge to a fixed point slightly slower than ours. More importantly, due to the cost of computing the transitive closure in each iteration, the time complexity would not match that of the original algorithm.

The implementation presented in section 4.8 is primarily meant for determining the complexity of the repaired algorithm, but can also serve as a starting point for an implementation on a computer. We have created an efficient implementation of our algorithm in C++, that employs advanced data structures like hash tables for efficiency. It is distributed as part of the mCRL2 toolset, that is publicly available at http://mcrl2.org. This implementation operates on labelled transition systems (domain $\mathbb{T}^{\mathsf{fu}}$) rather than vertex-labelled graphs. It can be used to minimize an LTS under simulation equivalence, via the tool ltsconvert, or to determine whether two LTSs are related by simulation preorder or simulation equivalence, via the tool ltscompare. Because the toolset already included an implementation of the subset construction algorithm for the determinization of LTSs (see also chapter 3), our contribution also enabled checking for trace preorder.

Chapter 5

Equivalence Checking for Infinite-State Systems

5.1 Introduction

As described in chapter 2, equivalence checking is a standard approach for verifying the correctness of concurrent systems. It aims to establish a behavioural equivalence or preorder between two models, usually an implementation and a specification. For finite-state systems, we already explored algorithms for checking language (and trace) preorder and equivalence through determinization (chapter 3), and for checking simulation preorder and equivalence (chapter 4). Checking strong bisimilarity of finite systems can be done very efficiently using the well-known *partition refinement* algorithm [104]. This algorithm can be lifted to weak bisimilarity by computing the transitive closure of τ-transitions. This is viable but costly, since it may incur a quadratic blow-up with respect to the original LTSs. Instead, one could employ the more efficient solution of [67] for checking branching bisimilarity. Branching and weak bisimilarity often coincide, especially when one of the models does not contain τ-transitions [56].

As an alternative to equivalence-checking directly on the LTSs, one can transform several equivalence-checking problems into solving Boolean equation systems (BES). As the standard μ-calculus model checking problem for finite systems can also be transformed to the problem of solving a BES [2, 91], a BES solver, *e.g.* the one of [94], provides a uniform engine for verification by model checking and equivalence checking for finite systems. Various encodings of behavioural equivalences have been proposed in the literature [2, 28, 94], leading to efficient tools. In [2] it is shown that the BESs obtained from equivalence relations have a special format; the encodings of [94] even yield alternation-free BESs (*cf.* the definition of alternation depth in [91]) for up to five different behavioural equivalences. Solving alternation-free BESs can be done very efficiently. However, finiteness of the

graphs is crucial for the encodings yielding alternation-free BESs.

Generally for concurrent systems with data, the induced LTS is no longer finite, and the traditional algorithms fail for infinite transition systems. In this chapter, we propose to check branching bisimilarity of infinite systems by solving recursive equations. In particular, in section 5.3 we show how to generate a PBES from two LPEs (see section 2.3.2). The resulting PBES has an alternation depth of two. We prove that the PBES has a positive solution if and only if the two (infinite) systems are branching bisimilar. Moreover, we illustrate the technique by two examples on queues (section 5.4), and show similar transformations for strong and weak bisimilarity and for branching simulation equivalence (section 5.5).

It is advantageous to translate the branching bisimilarity problem for infinite systems to solving PBESs, even though both problems are undecidable. The main reason is that solving PBESs is a more fundamental problem, as it boils down to solving equations between predicates. The other reason is that model checking μ-calculus with data has already been mapped to PBESs. Hence, all techniques for solving PBESs (see section 2.6.3) can now also be applied to the branching bisimilarity problem, as demonstrated by the examples in section 5.4.

A method related to ours is the cones and foci method [49] which rephrases the question whether two LPEs are bisimilar in terms of proof obligations on data objects. There the user must first identify invariants, a focus condition and a state mapping. In contrast, generating a PBES requires no human ingenuity, although solving the PBES still may. Furthermore, our solution is considerably more general, because it lifts two severe limitations of the cones and foci method. The first limitation is that the cones and foci method only works in case the branching bisimulation is functional, meaning that a state in the implementation can only be related to a unique state in the specification. Another severe limitation of the cones and foci method is that it cannot handle specifications that contain τ-transitions. In some protocols (*e.g.* the bounded retransmission protocol [64]) this condition is not met and thus the cones and foci method fails. In our example on unbounded queues (see section 5.4.2), both systems perform τ-steps, and their bisimulation is not functional.

Our work can be seen as the generalization of [89] to weak and branching equivalences. In [89], Lin proposes Symbolic Transition Graphs with Assignments (STGA) as a new model for message-passing processes. An algorithm is also presented which computes bisimilarity formulae for finite state STGAs, in terms of the greatest solutions of a *predicate equation system*. This corresponds with an alternation-free PBES, and thus it can only deal with strong bisimilarity.

The extension of Lin's work to weak and branching equivalences is not straightforward. This is testified by the encoding of weak bisimilarity in predicate equation systems by Kwak *et al.* [86]. However, their encoding is not generally correct for STGA, as they use a conjunction over the complete τ-closure of a state. This only works in case that the τ-closure of every state is finite, which is generally not the case for STGA, nor for LPEs. Alternation depth 2 seems unavoidable but does not occur in [86]. Note that for finite LTSs a conjunction over the τ-closure is

possible [94], but leads to a quadratic blow-up of the BES in the worst case.

5.2 Preliminaries

In this chapter we do not distinguish syntactic data sorts and terms from their semantic counterparts, *i.e.* we write both D and $\mathbb{D}$ as D, and both t and $[\![t]\!]$ as t.

Given an LPE P, we mark its elements using superscripts to avoid ambiguity:

$$P(d{:}D^P) = \sum_{a \in \mathcal{A}} \sum_{e_a : E_a^P} h_a^P(d, e_a) \rightarrow a(f_a^P(d, e_a)) \cdot P(g_a^P(d, e_a)) \,.$$

Moreover, we write $d \xrightarrow{a(d')}_P d''$ to denote the fact that $(d, a(d'), d'')$ is in the transition relation of the LTS of P (see definition 2.4).

For any transition system $(S, \rightarrow, i) \in \mathbb{T}^{\mathsf{u}}$ and $a \in Act$ we define $\xrightarrow{\overline{a}}$ as $\xrightarrow{\tau} \cup \mathcal{I}$ if $a = \tau$, and $\xrightarrow{a}$ otherwise. Hence, $s \xrightarrow{\overline{a}} t$ is an abbreviation for $s \xrightarrow{a} t \vee (a = \tau \wedge s = t)$. For this chapter, the concept of semi-branching bisimilarity is more convenient than that of branching bisimilarity as it allows for shorter and clearer proofs.

Definition 5.1. Let $M, N \in \mathbb{T}^{\mathsf{u}}$. A relation $R \subseteq S_M \times S_N$ is a *semi-branching simulation* iff for all $(s, t) \in R$:

$$\forall a \in Act, s' \in S_M \,.\, s \xrightarrow{a} s' \Longrightarrow \exists u, t' \in S_N \,.\, t \Rightarrow u \xrightarrow{\overline{a}} t' \wedge s \; R \; u \wedge s' \; R \; t'.$$

A relation R is a *semi-branching bisimulation* if both R and R^{-1} are semi-branching simulations, and *semi-branching bisimilarity* $\leftrightarroweq_{\mathsf{sb}}$ is the largest semi-branching bisimulation.

Although a semi-branching simulation is not necessarily a branching simulation, semi-branching bisimilarity coincides with branching bisimilarity [6], *i.e.*:

$$\leftrightarroweq_{\mathsf{sb}} = \leftrightarroweq_{\mathsf{b}} \,. \tag{5.1}$$

Therefore, in the sequel we take the liberty to use *semi-branching bisimilarity* and *branching bisimilarity* interchangeably.

5.3 Translation for branching bisimilarity

We encode the problem of finding the largest branching bisimulation in the problem of solving an equation system. More precisely, for any LPEs P and Q, algorithm 5.1 constructs the equation system that encodes branching bisimilarity (actually, semi-branching bisimilarity) between the states of P and those of Q. The main function $\textsc{Bisim}_b$ returns a closed equation system of the form $\mathcal{E}_1 \; \mathcal{E}_2$ where the bound predicate variables in $\mathcal{E}_1$ are called X and those in $\mathcal{E}_2$ are called Y. Intuitively, $\mathcal{E}_1$ is used to characterize branching bisimilarity while $\mathcal{E}_2$ is used to absorb the τ-transitions. The equation system's predicate formulae are constructed from the syntactic elements of P and Q. Note that P and Q are treated

Algorithm 5.1. Construction of a PBES for branching bisimilarity on P and Q. $\textsc{Bisim}_b(P,Q) = \mathcal{E}_1\ \mathcal{E}_2$, where:

$$\begin{aligned}
\mathcal{E}_1 &:= (\nu X^{P,Q}(d{:}D^P, d'{:}D^Q) = \mathsf{match}^{P,Q}(d,d') \wedge \mathsf{match}^{Q,P}(d',d)) \\
&\quad\ (\nu X^{Q,P}(d'{:}D^Q, d{:}D^P) = X^{P,Q}(d,d'))\,, \\
\mathcal{E}_2 &:= \{(\mu Y_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) = \mathsf{close}_a^{x,y}(d,d',e)) \mid a \in \mathcal{A} \wedge \\
&\qquad (x,y) \in \{(P,Q),(Q,P)\}\}
\end{aligned}$$

and we use the following abbreviations, for all $a \in \mathcal{A} \wedge (x,y) \in \{(P,Q),(Q,P)\}$:

$$\begin{aligned}
\mathsf{match}^{x,y}(d{:}D^x, d'{:}D^y) &= \textstyle\bigwedge_{a\in\mathcal{A}} \forall e{:}E_a^x\,.\,(h_a^x(d,e) \Longrightarrow Y_a^{x,y}(d,d',e)) \\
\mathsf{close}_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) &= \exists e'{:}E_\tau^y\,.\,(h_\tau^y(d',e') \wedge Y_a^{x,y}(d, g_\tau^y(d',e'),e)) \vee \\
&\qquad (X^{x,y}(d,d') \wedge \mathsf{step}_a^{x,y}(d,d',e)) \\
\mathsf{step}_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) &= (a = \tau \wedge X^{x,y}(g_\tau^x(d,e),d')) \vee \\
&\qquad \exists e'{:}E_a^y\,.\,h_a^y(d',e') \wedge f_a^x(d,e) = f_a^y(d',e') \\
&\qquad\quad \wedge X^{x,y}(g_a^x(d,e), g_a^y(d',e'))\,.
\end{aligned}$$

completely symmetrically. As we will show in theorem 5.5, the solution for $X^{P,Q}$ in the resulting equation system gives the largest branching bisimulation between P and Q as a predicate on $D^P \times D^Q$. In particular, if $d{:}D^P$ and $d'{:}D^Q$ are the initial states of P and Q, respectively, then P and Q are branching bisimilar if and only if the solution for $X^{P,Q}$ in the equation system yields $\top$ on (d,d'). We illustrate the translation by a small example.

Example 5.2. Consider the following LPEs:

$$\begin{aligned}
P(b{:}\mathbb{B}) &= b \to \tau \cdot P(\bot) + \\
&\quad\ a \cdot P(\top)
\end{aligned}
\qquad\qquad
\begin{aligned}
Q(c{:}\mathbb{B}) = &\quad c \to \tau \cdot Q(\bot) + \\
&\neg c \to a \cdot Q(\top)\,.
\end{aligned}$$

We describe the PBES that we obtain by applying algorithm 5.1 to P and Q. The first equations are as follows:

$$\begin{aligned}
\nu X^{P,Q}(b{:}\mathbb{B}, c{:}\mathbb{B}) &= (b \implies Y_\tau^{P,Q}(b,c)) \wedge Y_a^{P,Q}(b,c) \wedge \\
&\qquad (c \implies Y_\tau^{Q,P}(c,b)) \wedge (\neg c \implies Y_a^{Q,P}(c,b)) \\
\nu X^{Q,P}(c{:}\mathbb{B}, b{:}\mathbb{B}) &= X^{P,Q}(b,c)\,.
\end{aligned}$$

The idea of the equation for $X^{P,Q}$ is that any of its solutions is a branching bisimulation on the states of P and Q. Because branching bisimilarity is the *largest* branching bisimulation, we are interested in the *largest* solution to this equation, hence the fixpoint sign is ν. The right-hand side of the equation encodes

that any action a that is possible from the state b of P should be matched by the state c of Q: if the guard for a in LPE P holds then c should match this a, which is encoded by the equation for $Y_a^{P,Q}(b,c)$ as explained below. Additionally, the converse should hold for actions that are possible from the state c of Q. The second equation expresses that a branching bisimulation is symmetrical.

The next pair of equations in the PBES is the following:

$$\begin{aligned}\mu Y_a^{P,Q}(b{:}\mathbb{B}, c{:}\mathbb{B}) &= (c \wedge Y_a^{P,Q}(b,\bot)) \vee (X^{P,Q}(b,c) \wedge \neg c \wedge X^{P,Q}(\top,\top))\\ \mu Y_\tau^{P,Q}(b{:}\mathbb{B}, c{:}\mathbb{B}) &= (c \wedge Y_\tau^{P,Q}(b,\bot)) \vee\\ &\qquad (X^{P,Q}(b,c) \wedge (X^{P,Q}(\bot,c) \vee (c \wedge X^{P,Q}(\bot,\bot))))\,.\end{aligned}$$

For any states b of P and c of Q, these equations encode c's matching an a-step and τ-step by b, respectively. Before matching an a-step, c is allowed to perform any number of τ-steps. The first disjunct of the right-hand side expresses this τ-closure: the guard of Q's τ-summand holds (c) and we inductively require $Y_a^{P,Q}(b,\bot)$ where $\bot$ is the state of Q after the τ-step. Because only *finite* τ-sequences can be skipped, the fixpoint sign of $Y_a^{P,Q}$ is μ. The second disjunct expresses the possibility that c performs a matching a-step: b and c are in the bisimulation ($X^{P,Q}(b,c)$), the guard of Q's a-summand holds ($\neg c$), and the states of P and Q after the a-steps are in the bisimulation ($X^{P,Q}(\top,\top)$). The equation for $Y_\tau^{P,Q}$ has a similar structure, but in the second disjunct c does not have to be able to perform a matching τ-step; it is also allowed for c to be in the bisimulation with b's τ-successor (*cf.* definition 5.1).

The remaining equations generated by algorithm 5.1 are the following:

$$\begin{aligned}\mu Y_a^{Q,P}(c{:}\mathbb{B}, b{:}\mathbb{B}) &= (b \wedge Y_a^{Q,P}(c,\bot)) \vee (X^{Q,P}(c,b) \wedge X^{Q,P}(\top,\top))\\ \mu Y_\tau^{Q,P}(c{:}\mathbb{B}, b{:}\mathbb{B}) &= (b \wedge Y_\tau^{Q,P}(c,\bot)) \vee\\ &\qquad (X^{Q,P}(c,b) \wedge (X^{Q,P}(\bot,b) \vee (b \wedge X^{Q,P}(\bot,\bot))))\,.\end{aligned}$$

These are the symmetrical counterparts of the previous equations. □

Before establishing the correctness of algorithm 5.1 we first provide a fixpoint characterization of semi-branching bisimilarity, which we exploit in the correctness proof of our algorithm. Given LPEs P and Q, and a relation $R \subseteq D^P \times D^Q$, we define the function F as follows:

$$\begin{aligned}F(R) \;:=\; \{(d,d') \mid\; &(\forall a \in \mathcal{A}, e_a \in E_a^P \,.\, h_a^P(d,e_a) \Longrightarrow\\ &\quad \exists d'_2, d'_3 \,.\, d' \Rightarrow_Q d'_2 \wedge d'_2 \xrightarrow{\overline{a(f_a^P(d,e_a))}}_Q d'_3 \wedge (d,d'_2) \in R \wedge (g_a^P(d,e_a), d'_3) \in R)\\ \wedge\; &(\forall a \in \mathcal{A}, e'_a \in E_a^Q \,.\, h_a^Q(d',e'_a) \Longrightarrow\\ &\quad \exists d_2, d_3 \,.\, d \Rightarrow_P d_2 \wedge d_2 \xrightarrow{\overline{a(f_a^Q(d',e'_a))}}_P d_3 \wedge (d_2,d') \in R \wedge (d_3, g_a^Q(d',e'_a)) \in R)\}\,.\end{aligned}$$

We now prove that semi-branching bisimilarity is the greatest fixpoint of F.

Lemma 5.3. $\leftrightarrows_{\mathsf{sb}} = \nu R \,.\, F(R)$.

Proof. The following two facts follow directly from the definitions:

(i) F is monotonic;

(ii) R is a semi-branching bisimulation if and only if $F(R) = R$, hence $\leftrightarrows_{\mathsf{sb}} = \bigcup\{R \mid F(R) = R\}$.

Therefore, it holds that:

(1) for any R, $F(R) = R$ implies $R \subseteq \leftrightarrows_{\mathsf{sb}}$, by (ii).

(2) $\leftrightarrows_{\mathsf{sb}} \subseteq F(\leftrightarrows_{\mathsf{sb}})$. Namely, for any R with $F(R) = R$, we have $F(R) \subseteq F(\leftrightarrows_{\mathsf{sb}})$ using (1) and (i), and thus $R \subseteq F(\leftrightarrows_{\mathsf{sb}})$. Then by (ii), $\leftrightarrows_{\mathsf{sb}} \subseteq F(\leftrightarrows_{\mathsf{sb}})$.

By Tarski's theorem and (i) we have $\nu F = \bigcup\{R \mid R \subseteq F(R)\}$. Then by (2), $\leftrightarrows_{\mathsf{sb}} \subseteq \nu R \,.\, F(R)$. Also, by (1), $\nu R \,.\, F(R) \subseteq \leftrightarrows_{\mathsf{sb}}$. Hence $\leftrightarrows_{\mathsf{sb}} = \nu R \,.\, F(R)$. □

For proving the correctness of our translation, we first show that $\mathcal{E}_2$ correctly encodes the requirement for matching a-steps between branching bisimilar states.

Proposition 5.4. *For any LPEs P and Q, let $\mathcal{E}_2$ be as defined by algorithm 5.1. Let η be an arbitrary predicate environment and $\theta := [\![\mathcal{E}_2]\!]\eta$. Then for any action a and any d, d' and e, we have $\theta(Y_a^{P,Q})(d, d', e)$ if and only if:*

$$\exists d_2, d_3 \,.\, d' \Rightarrow_Q d_2 \wedge d_2 \xrightarrow{\overline{a(f_a^P(d,e))}}_Q d_3 \wedge \eta(X^{P,Q})(d, d_2) \wedge \eta(X^{P,Q})(g_a^P(d,e), d_3) \,.$$

Proof. We omit the superscript P, Q when no confusion can arise. We define sets $\mathcal{R}_i^{a,d,e} \subseteq D^Q$, for any $a \in \mathcal{A}$, d, e and $i \geq 0$, and depending on $\eta(X)$, as follows:

$$\begin{aligned}
\mathcal{R}_0^{a,d,e} &:= \{d' \mid \exists d_3 \,.\, d' \xrightarrow{\overline{a(f_a^P(d,e))}}_Q d_3 \wedge \eta(X)(d, d') \wedge \eta(X)(g_a^P(d,e), d_3)\} \\
\mathcal{R}_{i+1}^{a,d,e} &:= \{d' \mid \exists d_2 \,.\, d' \xrightarrow{\tau}_Q d_2 \wedge d_2 \in R_i^{a,d,e}\} \,.
\end{aligned}$$

Let $\mathcal{R}^{a,d,e} = \bigcup_{i \geq 0} \mathcal{R}_i^{a,d,e}$. Obviously, by definition of $\Rightarrow$ we have:

$$\begin{aligned}
\mathcal{R}^{a,d,e} = \{d' \mid \exists d_2, d_3 \,.\, d' \Rightarrow_Q d_2 \wedge d_2 &\xrightarrow{\overline{a(f_a^P(d,e))}}_Q d_3 \wedge \eta(X)(d, d_2) \\
&\wedge \eta(X)(g_a^P(d,e), d_3)\} \,.
\end{aligned}$$

We will prove, using an approximation method, that this coincides with the solution for $Y_a^{P,Q}$ in $\mathcal{E}_2$. More precisely, we claim:

$$\theta(Y_a^{P,Q})(d, d', e) \;\Leftrightarrow\; d' \in \mathcal{R}^{a,d,e} \,.$$

By algorithm 5.1, the equation for Y_a is of the form

$$Y_a(d, d', e) = (X(d, d') \wedge \varphi) \vee \exists e'_\tau \,.\, (h_\tau^Q(d', e'_\tau) \wedge Y_a(d, g_\tau^Q(d', e'_\tau), e)) \tag{5.2}$$

where φ (generated by step) stands for the formula

$$\begin{aligned}
&(a = \tau \wedge X(g_\tau^P(d,e), d')) \vee \\
&\quad \exists e' \,.\, h_a^Q(d', e') \wedge f_a^P(d,e) = f_a^Q(d', e') \wedge X(g_a^P(d,e), g_a^Q(d', e')) \,.
\end{aligned}$$

Because the fixpoint symbol of Y_a is μ, we have to prove that the smallest solution of (5.2) coincides with $\mathcal{R}^{a,d,e}$. We do so by showing that the first infinitary approximation Y_a^ω of (5.2) coincides with $\mathcal{R}^{a,d,e}$ and is a solution of (5.2). We first prove the following facts:

(a) $[\![X(d,d') \wedge \varphi]\!]\eta \;\Leftrightarrow\; d' \in \mathcal{R}_0^{a,d,e}$;

(b) $Y_a^n(d,d',e) = (d' \in \bigcup_{0\le i<n} \mathcal{R}_i^{a,d,e})$, for any finite approximation Y_a^n of (5.2).

Ad (a). This statement is equivalent to

$$[\![X(d,d') \wedge \varphi]\!]\eta = \exists d''\,.\, d' \xrightarrow{\overline{a(f_a^P(d,e))}}_Q d'' \wedge \eta(X)(d,d') \wedge \eta(X)(g_a^P(d,e),d'')$$

which follows readily if $[\![\varphi]\!]\eta = \exists d''\,.\, d' \xrightarrow{\overline{a(f_a^P(d,e))}}_Q d'' \wedge \eta(X)(g_a^P(d,e),d'')$. We derive, using definition 2.4 at step $*$ and the definition of $\xrightarrow{\overline{a}}$ at step †:

$$\begin{aligned}
[\![\varphi]\!]\eta &= (a=\tau \wedge \eta(X)(g_\tau^P(d,e),d')) \vee \\
&\quad \exists e'\,.\,(h_a^Q(d',e') \wedge f_a^P(d,e) = f_a^Q(d',e') \wedge \eta(X)(g_a^P(d,e),g_a^Q(d',e'))) \\
&= \exists d''\,.\,(a=\tau \wedge d'=d'' \wedge \eta(X)(g_a^P(d,e),d'')) \vee \\
&\quad \exists d'',e'\,.\,(h_a^Q(d',e') \wedge g_a^Q(d',e') = d'' \wedge f_a^P(d,e) = f_a^Q(d',e') \\
&\qquad \wedge \eta(X)(g_a^P(d,e),d'')) \\
&\overset{*}{=} \exists d''\,.\,(a=\tau \wedge d'=d'' \wedge \eta(X)(g_a^P(d,e),d'')) \vee \\
&\quad \exists d''\,.\,(d' \xrightarrow{a(f_a^P(d,e))}_Q d'' \wedge \eta(X)(g_a^P(d,e),d'')) \\
&= \exists d''\,.\,((a=\tau \wedge d'=d'') \vee d' \xrightarrow{a(f_a^P(d,e))}_Q d'') \wedge \eta(X)(g_a^P(d,e),d'') \\
&\overset{\dagger}{=} \exists d''\,.\, d' \xrightarrow{\overline{a(f_a^P(d,e))}}_Q d'' \wedge \eta(X)(g_a^P(d,e),d'')\,.
\end{aligned}$$

Ad (b). We prove this statement by induction on n.

Base: Trivially, $Y_a^0(d,d',e) = \bot = d' \in \emptyset = d' \in \bigcup_{0\le i<0} \mathcal{R}_i^{a,d,e}$.

Step: We derive, using the induction hypothesis at step $*$, and (a), definition 2.4 and the definition of $\mathcal{R}_i^{a,d,e}$ at step †:

$$\begin{aligned}
&\{d' \mid Y_a^{n+1}(d,d',e)\} \\
&\overset{*}{=} \{d' \mid (\eta(X)(d,d') \wedge [\![\varphi]\!]\eta) \vee \exists e'_\tau\,.\,(h_\tau^Q(d',e'_\tau) \wedge g_\tau^Q(d',e'_\tau) \in \textstyle\bigcup_{0\le i<n} \mathcal{R}_i^{a,d,e})\} \\
&= \{d' \mid [\![X(d,d') \wedge \varphi]\!]\eta\} \cup \textstyle\bigcup_{0\le i<n} \{d' \mid \exists e'_\tau\,.\,h_\tau^Q(d',e'_\tau) \wedge g_\tau^Q(d',e'_\tau) \in \mathcal{R}_i^{a,d,e}\} \\
&\overset{\dagger}{=} \mathcal{R}_0^{a,d,e} \cup \textstyle\bigcup_{0\le i<n} \mathcal{R}_{i+1}^{a,d,e} \;=\; \bigcup_{0\le i<n+1} \mathcal{R}_i^{a,d,e}\,.
\end{aligned}$$

Using (b), we can now prove that Y_a^ω coincides with $\mathcal{R}^{a,d,e}$:

$$\begin{aligned}
\{d' \mid Y_a^\omega(d,d',e)\} &= \textstyle\bigcup_{n\ge 0} \{d' \mid Y_a^n(d,d',e)\} \overset{\text{(b)}}{=} \bigcup_{n\ge 0} \bigcup_{0\le i<n} \mathcal{R}_i^{a,d,e} \\
&= \textstyle\bigcup_{i\ge 0} \mathcal{R}_i^{a,d,e} \;=\; \mathcal{R}^{a,d,e}.
\end{aligned}$$

Finally, we show that Y_a^ω is a fixed point and, hence, a solution of (5.2):

$\{d' \mid (\eta(X)(d,d') \wedge [\![\varphi]\!]\eta) \vee \exists e'_\tau \,.\, (h^Q_\tau(d',e'_\tau) \wedge g^Q_\tau(d',e'_\tau) \in \mathcal{R}^{a,d,e})\}$
$= \mathcal{R}^{a,d,e}_0 \cup \bigcup_{i\geq 1} \mathcal{R}^{a,d,e}_i = \mathcal{R}^{a,d,e}$. □

The correctness of algorithm 5.1 is established by the following theorem.

Theorem 5.5. *Let $\mathcal{E}_1$ $\mathcal{E}_2$ be the equation system generated by algorithm 5.1 on LPEs P and Q, and let $\theta := [\![\mathcal{E}_1\ \mathcal{E}_2]\!]$. Then for all d and d', we have $P(d) \leftrightarrows_{\mathsf{b}} Q(d')$ if and only if $\theta(X^{P,Q})(d,d')$.*

Proof. By algorithm 5.1, $X^{P,Q}$ is of the form:

$$X^{P,Q}(d,d') = \bigwedge_{a\in\mathcal{A}} \forall e_a \,.\, (h^P_a(d,e_a) \Longrightarrow Y^{P,Q}_a(d,d',e_a)) \ \wedge \\ \bigwedge_{a\in\mathcal{A}} \forall e'_a \,.\, (h^Q_a(d',e'_a) \Longrightarrow Y^{Q,P}_a(d',d,e'_a)) \,.$$

By symmetry, without loss of generality we only consider $\bigwedge_{a\in\mathcal{A}} \forall e_a \,.\, h^P_a(d,e_a) \Longrightarrow Y^{P,Q}_a(d,d',e_a)$. We redefine the function F correspondingly:

$$F(R) := \{(d,d') \mid \forall a \in \mathcal{A}, e_a \in E^P_a \,.\, h^P_a(d,e_a) \Longrightarrow \\ \exists d_2, d_3 \,.\, d' \Rightarrow_Q d_2 \wedge d_2 \xrightarrow{\overline{a(f^P_a(d,e_a))}}_Q d_3 \wedge (d,d_2) \in R \wedge (g^P_a(d,e_a),d_3) \in R\} \,.$$

We define $G : \wp(D^P \times D^Q) \to \wp(D^P \times D^Q)$ as

$$G(R) := \{(d,d') \mid \textstyle\bigwedge_{a\in\mathcal{A}} \forall e_a \,.\, h^P_a(d,e_a) \Longrightarrow \eta(Y^{P,Q}_a)(d,d',e_a)\} \,.$$

where $\eta = [\![\mathcal{E}_2]\!][X^{P,Q} \mapsto (\lambda x,y \,.\, (x,y) \in R)]$. By proposition 5.4, we have $F(R) = G(R)$ for any R. Note that by lemma 2.21, G is a monotonic function over a complete lattice, and thus its greatest fixpoint exists. By definition 2.19 and using the isomorphism between $\wp(D^P \times D^Q)$ and $B^{D^P \times D^Q}$, we obtain

$$\nu R \,.\, G(R) = \{(d,d') \mid \theta(X^{P,Q})(d,d')\} \,.$$

Hence, using (5.1) and lemma 5.3:

$$\leftrightarrows_{\mathsf{b}} = \leftrightarrows_{\mathsf{sb}} = \nu R \,.\, F(R) = \nu R \,.\, G(R) = \{(d,d') \mid \theta(X^{P,Q})(d,d')\}$$

from which the theorem readily follows. □

5.4 Examples

In this section we demonstrate the technique outlined in the previous section by applying it to two examples. In both examples we prove branching bisimilarity between two systems that have infinite state spaces. To compute the solutions to the resulting PBESs, we rely on techniques for solving and manipulating PBESs like adding invariants, symbolic approximations and strengthening equations.

Let D be an arbitrary data sort (possibly of infinite size) equipped with an equality relation, and let Q denote the data sort of queues over D with infinite capacity. We use list enumeration to denote queues containing specific elements, *e.g.* $[]$ and $[d, e]$ denote the empty queue and the queue containing d and e for any $d, e \in D$, respectively. Operations on queues include $q +\!\!+ r$, denoting the concatenation of queues q and r, $|q|$ denoting the length of queue q (where $|q| > n$ for all $n \in \mathbb{N}$ if q is infinite), and functions $hd : Q \to D$ and $tl : Q \to Q$ which yield the head and tail of a queue, respectively.

Our processes can communicate with their environments via parameterized actions $r(d)$ (read d from the environment) and $w(d)$ (write d to the environment). The τ-steps represent the internal communication of data from one queue or buffer to the next.

5.4.1 Two buffers and a queue

In this example, we show that two one-place buffers in sequence behave branching bisimilar to a queue of capacity two. The behaviour of these systems is given by the LPEs P and R, respectively:

$$\begin{aligned} P(b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) \;=\; & \textstyle\sum_{d:D} \neg b_1 \to r(d) \cdot P(\top, d, b_2, d_2) + \\ & b_1 \wedge \neg b_2 \to \tau \cdot P(\bot, d_1, \top, d_1) + \\ & b_2 \to w(d_2) \cdot P(b_1, d_1, \bot, d_2) \end{aligned}$$

$$\begin{aligned} R(q{:}Q) \;=\; & \textstyle\sum_{d:D} |q| < 2 \to r(d) \cdot R([d] +\!\!+ q) + \\ & |q| > 0 \to w(hd(q)) \cdot R(tl(q)) \end{aligned}$$

To check whether these processes are branching bisimilar, they are translated to the following PBES using algorithm 5.1:

$$\begin{aligned} & \nu X^{P,R}(b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, q{:}Q) = \\ & \quad (\neg b_1 \Longrightarrow \forall d{:}D \,.\, Y_r^{P,R}(b_1, d_1, b_2, d_2, q, d)) \wedge \\ & \quad ((b_1 \wedge \neg b_2) \Longrightarrow Y_\tau^{P,R}(b_1, d_1, b_2, d_2, q)) \wedge \\ & \quad (b_2 \Longrightarrow Y_w^{P,R}(b_1, d_1, b_2, d_2, q)) \wedge \\ & \quad (|q|{<}2 \Longrightarrow \forall d{:}D \,.\, Y_r^{R,P}(q, b_1, d_1, b_2, d_2, d)) \wedge \\ & \quad (|q|{>}0 \Longrightarrow Y_w^{R,P}(q, b_1, d_1, b_2, d_2)) \end{aligned}$$

$$\begin{aligned} & \nu X^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) = \\ & \quad X^{P,R}(b_1, d_1, b_2, d_2, q) \end{aligned}$$

$$\begin{aligned} & \mu Y_r^{P,R}(b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, q{:}Q, e{:}D) = \\ & \quad X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge |q| < 2 \wedge X^{P,R}(\top, e, b_2, d_2, [e] +\!\!+ q) \end{aligned}$$

$$\mu Y_w^{P,R}(b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, q{:}Q) =$$
$$\quad X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge |q| > 0 \wedge d_2 = hd(q) \wedge X^{P,R}(b_1, d_1, \bot, d_2, tl(q))$$

$$\mu Y_\tau^{P,R}(b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, q{:}Q) =$$
$$\quad X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge X^{P,R}(\bot, d_1, \top, d_1, q)$$

$$\mu Y_r^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, e{:}D) =$$
$$\quad (b_1 \wedge \neg b_2 \wedge Y_r^{R,P}(q, \bot, d_1, \top, d_1, e)) \vee$$
$$\quad (X^{R,P}(q, b_1, d_1, b_2, d_2) \wedge \neg b_1 \wedge X^{R,P}([e]{+\!\!+}q, \top, e, b_2, d_2))$$

$$\mu Y_w^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) =$$
$$\quad (b_1 \wedge \neg b_2 \wedge Y_w^{R,P}(q, \bot, d_1, \top, d_1)) \vee$$
$$\quad (X^{R,P}(q, b_1, d_1, b_2, d_2) \wedge b_2 \wedge hd(q) = d_2 \wedge X^{R,P}(tl(q), b_1, d_1, \bot, d_2))\,.$$

Each of the variables $X^{R,P}$, $Y_r^{P,R}$, $Y_w^{P,R}$ and $Y_\tau^{P,R}$ does not occur in the right-hand side of its own equation. We eliminate such self-references from the equations for $Y_r^{R,P}$ and $Y_w^{R,P}$ by approximation:

$$Y_{r,0}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, e{:}D) = \bot$$
$$Y_{r,1}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, e{:}D) =$$
$$\quad X^{R,P}(q, b_1, d_1, b_2, d_2) \wedge \neg b_1 \wedge X^{R,P}([e]{+\!\!+}q, \top, e, b_2, d_2)$$
$$Y_{r,2}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, e{:}D) =$$
$$\quad (b_1 \wedge \neg b_2 \wedge X^{R,P}(q, \bot, d_1, \top, d_1) \wedge X^{R,P}([e]{+\!\!+}q, \top, e, \top, d_1)) \vee$$
$$\quad (X^{R,P}(q, b_1, d_1, b_2, d_2) \wedge \neg b_1 \wedge X^{R,P}([e]{+\!\!+}q, \top, e, b_2, d_2))$$
$$Y_{r,3}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, e{:}D) = Y_{r,2}^{R,P}(q, b_1, d_1, b_2, d_2, e)$$

$$Y_{w,0}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) = \bot$$
$$Y_{w,1}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) =$$
$$\quad X^{R,P}(q, b_1, d_1, b_2, d_2) \wedge b_2 \wedge hd(q) = d_2 \wedge X^{R,P}(tl(q), b_1, d_1, \bot, d_2)$$
$$Y_{w,2}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) =$$
$$\quad (b_1 \wedge \neg b_2 \wedge X^{R,P}(q, \bot, d_1, \top, d_1) \wedge hd(q) = d_1 \wedge X^{R,P}(tl(q), \bot, d_1, \bot, d_2)) \vee$$
$$\quad (X^{R,P}(q, b_1, d_1, b_2, d_2) \wedge b_2 \wedge hd(q) = d_2 \wedge X^{R,P}(tl(q), b_1, d_1, \bot, d_2))$$
$$Y_{w,3}^{R,P}(q{:}Q, b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D) = Y_{w,2}^{R,P}(q, b_1, d_1, b_2, d_2)\,.$$

We apply substitution to obtain the following equation for $X^{P,R}$:

$$\begin{aligned}
&\nu X^{P,R}(b_1{:}B, d_1{:}D, b_2{:}B, d_2{:}D, q{:}Q) = \\
&\quad (\neg b_1 \Longrightarrow (X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge |q| < 2 \\
&\qquad\qquad \wedge \forall d \in D \,.\, X^{P,R}(\top, d, b_2, d_2, [d] \mathbin{+\!\!+} q))) \\
&\wedge ((b_1 \wedge \neg b_2) \Longrightarrow (X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge X^{P,R}(\bot, d_1, \top, d_1, q))) \\
&\wedge (b_2 \Longrightarrow (X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge |q| > 0 \wedge d_2 = hd(q) \\
&\qquad\qquad \wedge X^{P,R}(b_1, d_1, \bot, d_2, tl(q)))) \\
&\wedge (|q| < 2 \Longrightarrow \forall d \in D \,.\, ((b_1 \wedge \neg b_2 \wedge X^{P,R}(\bot, d_1, \top, d_1, q) \\
&\qquad\qquad \wedge X^{P,R}(\top, d, \top, d_1, [d] \mathbin{+\!\!+} q)) \vee (X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge \neg b_1 \\
&\qquad\qquad \wedge X^{P,R}(\top, d, b_2, d_2, [d] \mathbin{+\!\!+} q)))) \\
&\wedge (|q| > 0 \Longrightarrow ((b_1 \wedge \neg b_2 \wedge X^{P,R}(\bot, d_1, \top, d_1, q) \wedge hd(q) = d_1 \\
&\qquad\qquad \wedge X^{P,R}(\bot, d_1, \bot, d_2, tl(q))) \vee (X^{P,R}(b_1, d_1, b_2, d_2, q) \wedge b_2 \\
&\qquad\qquad \wedge hd(q) = d_2 \wedge X^{P,R}(b_1, d_1, \bot, d_2, tl(q))))) \,.
\end{aligned}$$

Note that we can safely omit every occurrence of $X^{P,R}(b_1, d_1, b_2, d_2, q)$ from this equation. To solve this equation we *instantiate* the Boolean parameters b_1 and b_2. Instantiation is a technique in which an equation $\sigma X(d{:}D, e{:}E) = \varphi$ with $D = \{d_1, \ldots, d_n\}$ is replaced by n equations $(\sigma X_{d_1}(e{:}E) = \varphi[d_1/d]) \cdots (\sigma X_{d_n}(e{:}E) = \varphi[d_n/d])$ where occurrences of $X(d_i, \ldots)$ in the right-hand sides are replaced by $X_{d_i}(\ldots)$. Instantiation can be done for multiple parameters. The technique is described more formally and proven correct in chapter 6. We obtain the following four equations when instantiating for parameters b_1 and b_2, omitting the superscripts P, R for the sake of readability:

$$\begin{aligned}
\nu X_{\top,\top}(d_1{:}D, d_2{:}D, q{:}Q) &= |q| \geq 2 \wedge hd(q) = d_2 \wedge X_{\top,\bot}(d_1, d_2, tl(q)) \\
\nu X_{\top,\bot}(d_1{:}D, d_2{:}D, q{:}Q) &= X_{\bot,\top}(d_1, d_1, q) \\
&\qquad \wedge (|q| < 2 \Longrightarrow \forall d \in D \,.\, X_{\top,\top}(d, d_1, [d] \mathbin{+\!\!+} q)) \\
&\qquad \wedge (|q| > 0 \Longrightarrow hd(q) = d_1 \wedge X_{\bot,\bot}(d_1, d_2, tl(q))) \\
\nu X_{\bot,\top}(d_1{:}D, d_2{:}D, q{:}Q) &= q = [d_2] \wedge (\forall d \in D \,.\, X_{\top,\top}(d, d_2, [d] \mathbin{+\!\!+} q)) \\
&\qquad \wedge X_{\bot,\bot}(d_1, d_2, tl(q)) \\
\nu X_{\bot,\bot}(d_1{:}D, d_2{:}D, q{:}Q) &= |q| = 0 \wedge (\forall d \in D \,.\, X_{\top,\bot}(d, d_2, [d] \mathbin{+\!\!+} q)) \,.
\end{aligned}$$

By substituting the equations for $X_{\bot,\top}$, $X_{\bot,\bot}$ and $X_{\top,\top}$ (in that order) in the equation for $X_{\top,\bot}$, we obtain for $X_{\top,\bot}$:

$$\nu X_{\top,\bot}(d_1{:}D, d_2{:}D, q{:}Q) \;=\; q = [d_1] \wedge (\forall d \in D \,.\, X_{\top,\bot}(d, d_2, [d])) \,.$$

By approximation we find that the solution for $X_{\top,\bot}$ is $q = [d_1]$. Substituting this solution in the other equations gives us the solutions for all $X_{x,y}$:

$$\begin{aligned}
\nu X_{\top,\top}(d_1{:}D, d_2{:}D, q{:}Q) &= q = [d_1, d_2] \\
\nu X_{\top,\bot}(d_1{:}D, d_2{:}D, q{:}Q) &= q = [d_1] \\
\nu X_{\bot,\top}(d_1{:}D, d_2{:}D, q{:}Q) &= q = [d_2] \\
\nu X_{\bot,\bot}(d_1{:}D, d_2{:}D, q{:}Q) &= q = [] \,.
\end{aligned}$$

Evaluating $X^{P,R}$ for the initial state $(\bot, d, \bot, e, [])$ for any $d, e \in D$, amounts to evaluating $X_{\bot,\bot}$ for $(d, e, [])$ which yields *true*. Hence, the processes $P(\bot, d, \bot, e)$ and $R([])$ are branching bisimilar for all $d, e \in D$. Using standard manipulation and solution techniques for PBESs, we have proven this statement without making any assumption on the data sort D. In fact, if D is an infinite sort and, hence, the state spaces of both processes are infinite, our proof remains unchanged.

5.4.2 Unbounded queues

The finiteness of the processes of the previous section depends solely on the data sort D. One could use abstraction techniques to abstract away from D entirely and obtain finite state spaces on which branching bisimilarity can be checked using conventional methods. By contrast, for the problem that we consider in this section, it is inherent that the involved processes are infinite. In other words, we cannot abstract away from their being infinite without changing the problem in a fundamental way.

The capacity of a bounded queue is doubled by connecting a queue of the same size, which means that a composition of bounded queues is behaviourally different from the constituent queues. In this respect, a composition of queues with infinite capacity does not change the behaviour, as this again yields an unbounded queue. The LPEs S and T defined below model the composition of two unbounded queues and three unbounded queues, respectively. We prove that these processes are branching bisimilar using our translation to a PBES.

$$\begin{aligned}
S(s_0, s_1{:}Q) \;=\; & \textstyle\sum_{v:D} r(v) \cdot S([v] {+\!\!+} s_0, s_1) + \\
& s_1 \neq [] \rightarrow w(hd(s_1)) \cdot S(s_0, tl(s_1)) + \\
& s_0 \neq [] \rightarrow \tau \cdot S(tl(s_0), [hd(s_0)] {+\!\!+} s_1)
\end{aligned}$$

$$\begin{aligned}
T(t_0, t_1, t_2{:}Q) \;=\; & \textstyle\sum_{u:D} r(u) \cdot T([u] {+\!\!+} t_0, t_1, t_2) + \\
& t_2 \neq [] \rightarrow w(hd(t_2)) \cdot T(t_0, t_1, tl(t_2)) + \\
& t_0 \neq [] \rightarrow \tau \cdot T(tl(t_0), [hd(t_0)] {+\!\!+} t_1, t_2) + \\
& t_1 \neq [] \rightarrow \tau \cdot T(t_0, tl(t_1), [hd(t_1)] {+\!\!+} t_2)
\end{aligned}$$

By applying algorithm 5.1 to LPEs S and T, we obtain the following PBES:

$$
\begin{aligned}
&\nu X^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad (\forall v{:}D\,.\, Y_r^{S,T}(s_0, s_1, t_0, t_1, t_2, v)) \wedge (s_1 \neq [] \Longrightarrow Y_w^{S,T}(s_0, s_1, t_0, t_1, t_2)) \wedge \\
&\quad (s_0 \neq [] \Longrightarrow Y_\tau^{S,T}(s_0, s_1, t_0, t_1, t_2)) \wedge \\
&\quad (\forall u{:}D\,.\, Y_r^{T,S}(t_0, t_1, t_2, s_0, s_1, u)) \wedge (t_2 \neq [] \Longrightarrow Y_w^{T,S}(t_0, t_1, t_2, s_0, s_1)) \wedge \\
&\quad ((t_1 \neq [] \vee t_0 \neq []) \Longrightarrow Y_\tau^{T,S}(t_0, t_1, t_2, s_0, s_1)) \\
&\nu X^{T,S}(t_0, t_1, t_2, s_0, s_1{:}Q) = X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\mu Y_r^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q, v{:}D) = \\
&\quad (t_0 \neq [] \wedge Y_r^{S,T}(s_0, s_1, tl(t_0), [hd(t_0)] +\!\!+ t_1, t_2, v)) \vee \\
&\quad (t_1 \neq [] \wedge Y_r^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)] +\!\!+ t_2, v)) \vee \\
&\quad (X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge X^{S,T}([v] +\!\!+ s_0, s_1, [v] +\!\!+ t_0, t_1, t_2)) \\
&\mu Y_w^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad (t_0 \neq [] \wedge Y_w^{S,T}(s_0, s_1, tl(t_0), [hd(t_0)] +\!\!+ t_1, t_2)) \vee \\
&\quad (t_1 \neq [] \wedge Y_w^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)] +\!\!+ t_2)) \vee \\
&\quad (X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge t_2 \neq [] \wedge hd(t_2) = hd(s_1) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2))) \\
&\mu Y_\tau^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad (t_0 \neq [] \wedge Y_\tau^{S,T}(s_0, s_1, tl(t_0), [hd(t_0)] +\!\!+ t_1, t_2)) \vee \\
&\quad (t_1 \neq [] \wedge Y_\tau^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)] +\!\!+ t_2)) \vee \\
&\quad (X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge (X^{S,T}(tl(s_0), [hd(s_0)] +\!\!+ s_1, t_0, t_1, t_2) \vee \\
&\qquad (t_0 \neq [] \wedge X^{S,T}(tl(s_0), [hd(s_0)] +\!\!+ s_1, tl(t_0), [hd(t_0)] +\!\!+ t_1, t_2)) \vee \\
&\qquad (t_1 \neq [] \wedge X^{S,T}(tl(s_0), [hd(s_0)] +\!\!+ s_1, t_0, tl(t_1), [hd(t_1)] +\!\!+ t_2)))) \\
&\mu Y_r^{T,S}(t_0, t_1, t_2, s_0, s_1{:}Q, u{:}D) = \\
&\quad (s_0 \neq [] \wedge Y_r^{T,S}(t_0, t_1, t_2, tl(s_0), [hd(s_0)] +\!\!+ s_1, u)) \vee \\
&\quad (X^{T,S}(t_0, t_1, t_2, s_0, s_1) \wedge X^{T,S}([u] +\!\!+ t_0, t_1, t_2, [u] +\!\!+ s_0, s_1)) \\
&\mu Y_w^{T,S}(t_0, t_1, t_2, s_0, s_1{:}Q) = \\
&\quad (s_0 \neq [] \wedge Y_w^{T,S}(t_0, t_1, t_2, tl(s_0), [hd(s_0)] +\!\!+ s_1)) \vee \\
&\quad (X^{T,S}(t_0, t_1, t_2, s_0, s_1) \wedge s_1 \neq [] \wedge hd(s_1) = hd(t_2) \\
&\qquad \wedge X^{T,S}(t_0, t_1, tl(t_2), s_0, tl(s_1))) \\
&\mu Y_\tau^{T,S}(t_0, t_1, t_2, s_0, s_1{:}Q) = \\
&\quad (s_0 \neq [] \wedge Y_\tau^{T,S}(t_0, t_1, t_2, tl(s_0), [hd(s_0)] +\!\!+ s_1)) \vee \\
&\quad (X^{T,S}(t_0, t_1, t_2, s_0, s_1) \wedge \\
&\quad (X^{T,S}(tl(t_0), [hd(t_0)] +\!\!+ t_1, t_2, s_0, s_1) \vee X^{T,S}(t_0, tl(t_1), [hd(t_1)] +\!\!+ t_2, s_0, s_1) \\
&\qquad \vee (s_0 \neq [] \wedge (X^{T,S}(tl(t_0), [hd(t_0)] +\!\!+ t_1, t_2, tl(s_0), [hd(s_0)] +\!\!+ s_1) \\
&\qquad\quad \vee X^{T,S}(t_0, tl(t_1), [hd(t_1)] +\!\!+ t_2, tl(s_0), [hd(s_0)] +\!\!+ s_1)))))
\end{aligned}
$$

Consider the equation for $Y_w^{S,T}$. It represents the case where process T has to

simulate a $w(hd(s_1))$ action of process S by possibly executing a finite number of τ-steps before executing action $w(hd(t_2))$. Inspired by the scenario that captures the minimal amount of τ-steps that are needed (two steps when $t_1 = [] \wedge t_2 = []$, one when $t_1 \neq [] \wedge t_2 = []$ and none otherwise), we strengthen the equation for $Y_w^{S,T}$ as follows:

$$
\begin{aligned}
&\mu Y_w^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad (t_0 \neq [] \wedge \underline{t_1 = [] \wedge t_2 = [] \wedge}\ Y_w^{S,T}(s_0, s_1, tl(t_0), [hd(t_0)] \mathbin{+\!\!+} t_1, t_2)) \vee \\
&\quad (t_1 \neq [] \wedge \underline{t_2 = [] \wedge}\ Y_w^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)] \mathbin{+\!\!+} t_2)) \vee \\
&\quad (t_2 \neq [] \wedge hd(t_2) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2)))
\end{aligned}
\tag{5.3}
$$

The solution to this equation is at most as large as the solution to the original equation. Therefore, if the solution to the strengthened PBES yields *true* on the initial state, then the solution to the original PBES would have done so as well, thereby justifying our modification. We approximate the solution to equation (5.3) as follows:

$$
\begin{aligned}
&Y_{w,0}^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \bot \\
&Y_{w,1}^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad t_2 \neq [] \wedge hd(t_2) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\quad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2)) \\
&Y_{w,2}^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad (t_1 \neq [] \wedge t_2 = [] \wedge hd(t_1) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)]) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, tl(t_1), [])) \vee \\
&\quad (t_2 \neq [] \wedge hd(t_2) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2))) \\
&Y_{w,3}^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad (t_0 \neq [] \wedge t_1 = [] \wedge t_2 = [] \wedge hd(t_0) = hd(s_1) \wedge X^{S,T}(s_0, s_1, tl(t_0), [], [hd(t_0)]) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), tl(t_0), [], [])) \vee \\
&\quad (t_1 \neq [] \wedge t_2 = [] \wedge hd(t_1) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)]) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, tl(t_1), [])) \vee \\
&\quad (t_2 \neq [] \wedge hd(t_2) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2))) \\
&Y_{w,4}^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = Y_{w,3}^{S,T}(s_0, s_1, t_0, t_1, t_2)
\end{aligned}
$$

Because $Y_{w,4}^{S,T} = Y_{w,3}^{S,T}$, the solution to equation (5.3) is $Y_{w,3}^{S,T}$. Similarly, we obtain a solution for $Y_w^{T,S}$ by strengthening the first disjunct. Regarding the equations

for the Y_τs and Y_rs, we note that the mimicking of a τ-step of one process by the other can be postponed. This means that we can strengthen each of the equations for the Y_τs and Y_rs by removing all but the last disjunct.

For every predicate variable except $X^{S,T}$ we have now obtained an equation in which that variable does not occur in the right-hand side. These solutions can be substituted in the equation for $X^{S,T}$ without affecting its solution, yielding the following closed equation for $X^{S,T}$:

$$
\begin{aligned}
&\nu X^{S,T}(s_0, s_1, t_0, t_1, t_2{:}Q) = \\
&\quad X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge (\forall v{:}D \,.\, X^{S,T}([v]{+\!\!+}s_0, s_1, [v]{+\!\!+}t_0, t_1, t_2)) \\
&\wedge\ (s_1 \neq [] \Longrightarrow ((t_0 \neq [] \wedge t_1 = [] \wedge t_2 = [] \wedge hd(t_0) = hd(s_1) \\
&\qquad \wedge X^{S,T}(s_0, s_1, tl(t_0), [], [hd(t_0)]) \wedge X^{S,T}(s_0, tl(s_1), tl(t_0), [], [])) \vee \\
&\qquad (t_1 \neq [] \wedge t_2 = [] \wedge hd(t_1) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)]) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, tl(t_1), [])) \vee \\
&\qquad (t_2 \neq [] \wedge hd(t_2) = hd(s_1) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2))))) \\
&\wedge\ (s_0 \neq [] \Longrightarrow (X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge (X^{S,T}(tl(s_0), [hd(s_0)]{+\!\!+}s_1, t_0, t_1, t_2) \vee \\
&\qquad (t_0 \neq [] \wedge X^{S,T}(tl(s_0), [hd(s_0)]{+\!\!+}s_1, tl(t_0), [hd(t_0)]{+\!\!+}t_1, t_2)) \vee \\
&\qquad (t_1 \neq [] \wedge X^{S,T}(tl(s_0), [hd(s_0)]{+\!\!+}s_1, t_0, tl(t_1), [hd(t_1)]{+\!\!+}t_2))))) \\
&\wedge\ (t_2 \neq [] \Longrightarrow ((s_0 \neq [] \wedge s_1 = [] \wedge hd(s_0) = hd(t_2) \\
&\qquad \wedge X^{S,T}(tl(s_0), [hd(s_0)], t_0, t_1, t_2) \wedge X^{S,T}(tl(s_0), [], t_0, t_1, t_2)) \vee \\
&\qquad (s_1 \neq [] \wedge hd(s_1) = hd(t_2) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \\
&\qquad \wedge X^{S,T}(s_0, tl(s_1), t_0, t_1, tl(t_2))))) \\
&\wedge\ ((t_0 \neq [] \vee t_1 \neq []) \Longrightarrow (X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge \\
&\qquad (X^{S,T}(s_0, s_1, tl(t_0), [hd(t_0)]{+\!\!+}t_1, t_2) \vee X^{S,T}(s_0, s_1, t_0, tl(t_1), [hd(t_1)]{+\!\!+}t_2) \\
&\qquad \vee (s_0 \neq [] \wedge (X^{S,T}(tl(s_0), [hd(s_0)]{+\!\!+}s_1, tl(t_0), [hd(t_0)]{+\!\!+}t_1, t_2) \\
&\qquad\qquad \vee X^{S,T}(tl(s_0), [hd(s_0)]{+\!\!+}s_1, t_0, tl(t_1), [hd(t_1)]{+\!\!+}t_2))))))
\end{aligned}
$$

The formula $s_0{+\!\!+}s_1 = t_0{+\!\!+}t_1{+\!\!+}t_2$ can be shown to be an invariant for this equation (see [103] for a treatment of PBES invariants). This implies that we can add the invariant to this equation without restricting the solution for $X^{S,T}$ for those values of s_0, s_1, t_0, t_1, t_2 that satisfy the invariant:

$$
\begin{aligned}
&\nu X^{S,T}(s_0, s_1, t_0, t_1, t_2 : Q) = \\
&\quad (s_0{+\!\!+}s_1 = t_0{+\!\!+}t_1{+\!\!+}t_2) \wedge X^{S,T}(s_0, s_1, t_0, t_1, t_2) \wedge \ldots
\end{aligned}
$$

The addition of the invariant $s_0{+\!\!+}s_1 = t_0{+\!\!+}t_1{+\!\!+}t_2$ accelerates and simplifies the

approximation of $X^{S,T}$, which now stabilizes at the third approximation:

$$X_0^{S,T}(s_0, s_1, t_0, t_1, t_2 : Q) = \top$$

$$\begin{aligned}
&X_1^{S,T}(s_0, s_1, t_0, t_1, t_2 : Q)\\
&= (s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2) \wedge\\
&\quad (s_1 \neq [] \Longrightarrow ((t_0 \neq [] \wedge t_1 = [] \wedge t_2 = [] \wedge hd(t_0) = hd(s_1))\\
&\qquad \vee (t_1 \neq [] \wedge t_2 = [] \wedge hd(t_1) = hd(s_1)) \vee (t_2 \neq [] \wedge hd(t_2) = hd(s_1))) \wedge\\
&\quad (t_2 \neq [] \Longrightarrow ((s_0 \neq [] \wedge s_1 = [] \wedge hd(s_0) = hd(t_2))\\
&\qquad \vee (s_1 \neq [] \wedge hd(s_1) = hd(t_2)))\\
&= (s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2)
\end{aligned}$$

$$\begin{aligned}
&X_2^{S,T}(s_0, s_1, t_0, t_1, t_2 : Q)\\
&= (s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2) \wedge \forall v{:}D\,.\,([v] \mathbin{+\!\!+} s_0) \mathbin{+\!\!+} s_1 = ([v] \mathbin{+\!\!+} t_0) \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2 \wedge\\
&\quad (s_1 \neq [] \Longrightarrow ((t_0 \neq [] \wedge t_1 = [] \wedge t_2 = [] \wedge s_0 \mathbin{+\!\!+} s_1 = t_0 \wedge s_0 \mathbin{+\!\!+} tl(s_1) = tl(t_0))\\
&\qquad \vee (t_1 \neq [] \wedge t_2 = [] \wedge s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \wedge s_0 \mathbin{+\!\!+} tl(s_1) = t_0 \mathbin{+\!\!+} tl(t_1))\\
&\qquad \vee (t_2 \neq [] \wedge s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2 \wedge s_0 \mathbin{+\!\!+} tl(s_1) = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} tl(t_2)))) \wedge\\
&\quad (s_0 \neq [] \Longrightarrow s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2) \wedge\\
&\quad (t_2 \neq [] \Longrightarrow ((s_0 \neq [] \wedge s_1 = [] \wedge s_0 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2 \wedge tl(s_0) = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} tl(t_2))\\
&\qquad \vee (s_1 \neq [] \wedge s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2 \wedge s_0 \mathbin{+\!\!+} tl(s_1) = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} tl(t_2)))) \wedge\\
&\quad ((t_0 \neq [] \vee t_1 \neq []) \Longrightarrow s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2)\\
&= (s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2)
\end{aligned}$$

Assuming that the invariant holds, the solution for $X^{S,T}$ is $s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2$. By substituting the initial states of S and T we obtain $[] \mathbin{+\!\!+} [] = [] \mathbin{+\!\!+} [] \mathbin{+\!\!+} []$, which clearly holds. Hence $S([], [])$ and $T([], [], [])$ are branching bisimilar. More generally, we have established that all processes $S(s_0, s_1)$ and $T(t_0, t_1, t_2)$ satisfying the condition $s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2$ are branching bisimilar.

Again, this result is independent of the data sort D. The branching bisimilarity problem on these infinite-state processes was translated to a PBES in a straightforward way. Solving the PBES required some ingenuity (*viz.* strengthening equations and adding an invariant), but this is not surprising given the problem's inherent complexity. Moreover, we note that this example cannot be solved directly using the cones and foci method (without introducing a third process), because both processes contain τ-steps and the branching bisimilarity on S and T is not functional. For instance, if $s_0 \mathbin{+\!\!+} s_1 = t_0 \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2$, then $S(s_0, s_1)$ is branching bisimilar to both $T(t_0, t_1, t_2)$ and $T([], t_0 \mathbin{+\!\!+} t_1, t_2)$ which are in general not identical. In turn, these two processes are also branching bisimilar to $S([], s_0 \mathbin{+\!\!+} s_1)$, which is generally not identical to $S(s_0, s_1)$.

5.5 Translations for other equivalences

We adapt algorithm 5.1 to obtain translations for strong bisimilarity, weak bisimilarity and branching similarity. First, let us recall the conditions in the definitions of strong, weak and semi-branching simulation (definitions 2.7, 2.9 and 5.1, respectively). In order for a relation R to be one of these simulations, it should satisfy the corresponding condition below, whenever $s\ R\ t$ and $s \xrightarrow{a} s'$:

- for strong: $\exists t' \,.\, t \xrightarrow{a} t' \wedge s'\ R\ t'$;
- for weak: $(a = \tau \wedge s'\ R\ t) \vee \exists u, u', t' \,.\, t \Rightarrow u \xrightarrow{a} u' \Rightarrow t' \wedge s'\ R\ t'$;
- for semi-branching: $\exists u, t' \,.\, t \Rightarrow u \xrightarrow{\overline{a}} t' \wedge s\ R\ u \wedge s'\ R\ t'$.

Below we derive translations $\textsc{Bisim}_s$ and $\textsc{Bisim}_w$ for strong and weak bisimilarity, respectively, from the translation $\textsc{Bisim}_b$ for branching bisimilarity based on the differences between these conditions. We also derive a translation $\textsc{Sim}_b$ for branching similarity from $\textsc{Bisim}_b$. The correctness proofs of these translations are similar to the one for $\textsc{Bisim}_b$. The translation for strong bisimilarity is known in [89] modulo different formalisms; the translations for weak bisimilarity and branching similarity are novel. Finally, we note that translations for strong and weak similarity can be obtained by applying similar modifications to $\textsc{Bisim}_s$ and $\textsc{Bisim}_w$, respectively, as we apply to $\textsc{Bisim}_b$ to obtain $\textsc{Sim}_b$.

5.5.1 Strong bisimilarity

For the strong case, the differences with semi-branching are the following:

(a) There is no intermediate u such that $t \Rightarrow u \wedge s\ R\ u$. More precisely, the condition has been strengthened by adding a conjunct $t = u$.

(b) The requirement $u \xrightarrow{\overline{a}} t'$ has been strengthened to $u \xrightarrow{a} t'$.

Compared to $\textsc{Bisim}_b$, these differences are reflected in the algorithm $\textsc{Bisim}_s$ for strong bisimilarity (algorithm 5.2) in the following ways, respectively:

(a) $\mathcal{E}_2$ has been removed along with the variables $Y_a^{x,y}$ and formulae $\mathsf{close}_a^{x,y}$; every occurrence of $Y_a^{x,y}$ in $\mathsf{match}^{x,y}$ has been replaced by $\mathsf{step}_a^{x,y}$.

(b) The disjunct $(a = \tau \wedge X^{x,y}(g_\tau^x(d,e), d')$ has been removed from $\mathsf{step}_a^{x,y}$.

5.5.2 Weak bisimilarity

For the weak case, the differences with semi-branching are the following:

(a) The requirement $s\ R\ u$ has been removed.

(b) An intermediate u' such that $u' \Rightarrow t'$ has been inserted.

(c) The requirement $u \xrightarrow{\overline{a}} t'$ has been replaced by $u \xrightarrow{a} u'$, and a disjunct $(a = \tau \wedge s'\ R\ t)$ has been added; this is analogous to the difference between semi-branching simulation and branching simulation.

Algorithm 5.2. Construction of a PBES for strong bisimilarity on P and Q.

$\textsc{Bisim}_s(P, Q) = \mathcal{E}_1$, where:

$$\begin{aligned}\mathcal{E}_1 := (&\nu X^{P,Q}(d{:}D^P, d'{:}D^Q) = \mathsf{match}^{P,Q}(d, d') \wedge \mathsf{match}^{Q,P}(d', d)) \\ (&\nu X^{Q,P}(d'{:}D^Q, d{:}D^P) = X^{P,Q}(d, d'))\end{aligned}$$

and we use the following abbreviations, for all $a \in \mathcal{A} \wedge (x, y) \in \{(P, Q), (Q, P)\}$:

$$\begin{aligned}\mathsf{match}^{x,y}(d{:}D^x, d'{:}D^y) &= \textstyle\bigwedge_{a \in \mathcal{A}} \forall e{:}E^x_a \,.\, (h^x_a(d, e) \Longrightarrow \mathsf{step}^{x,y}_a(d, d', e)) \\ \mathsf{step}^{x,y}_a(d{:}D^x, d'{:}D^y, e{:}E^x_a) &= \exists e'{:}E^y_a \,.\, h^y_a(d', e') \wedge f^x_a(d, e) = f^y_a(d', e') \\ &\quad \wedge X^{x,y}(g^x_a(d, e), g^y_a(d', e'))\,.\end{aligned}$$

The algorithm $\textsc{Bisim}_w$ for weak bisimilarity (algorithm 5.3) reflects these differences in the following ways, respectively:

(a) In the second disjunct of $\mathsf{close}^{x,y}_a$, the conjunct $X^{x,y}(d, d')$ has been removed.

(b) An equation system $\mathcal{E}_3$, consisting of two equations, has been added with variables $Y^{x,y}$ and right-hand side formulae $\mathsf{close}^{x,y}$. This equation system serves to skip any number of τ-steps from u' to t', similarly to the way in which $\mathcal{E}_2$ is used to skip τ-steps from t to u. The second occurrence of $X^{x,y}$ in $\mathsf{step}^{x,y}_a$ has been replaced by $Y^{x,y}$.

(c) The disjunct $(a = \tau \wedge X^{x,y}(g^x_\tau(d, e), d'))$ has been moved from $\mathsf{step}^{x,y}_a$ to the right-hand side of the implication in $\mathsf{match}^{x,y}$.

5.5.3 Branching similarity

Regarding branching similarity, the only difference with branching bisimilarity is that there must be branching simulations in both directions, instead of a single (symmetric) branching bisimulation. This is reflected in the equation system $\mathcal{E}_1$ of the algorithm $\textsc{Sim}_b$ for branching similarity (algorithm 5.4). There, the equation for $X^{P,Q}$ encodes the branching simulation preorder that relates process P with process Q, and vice versa for $X^{Q,P}$. The equation for X then encodes branching similarity on P and Q. The equation system $\mathcal{E}_2$ is the same as for branching bisimilarity.

5.6 Conclusions

We have shown how to translate the problems of checking strong, weak and branching (bi)similarity for infinite systems to the problem of solving parameterized Boolean equation systems. For the translation for branching bisimilarity and its correctness proof, we used the equivalent notion of semi-branching bisimilarity for

Algorithm 5.3. Construction of a PBES for weak bisimilarity on P and Q.
$\textsc{Bisim}_w(P,Q) = \mathcal{E}_1\ \mathcal{E}_2\ \mathcal{E}_3$, where:

$$\begin{aligned}
\mathcal{E}_1 &:= (\nu X^{P,Q}(d{:}D^P, d'{:}D^Q) = \mathsf{match}^{P,Q}(d,d') \wedge \mathsf{match}^{Q,P}(d',d)) \\
&\quad\ (\nu X^{Q,P}(d'{:}D^Q, d{:}D^P) = X^{P,Q}(d,d'))\,, \\
\mathcal{E}_2 &:= \{(\mu Y_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) = \mathsf{close}_a^{x,y}(d,d',e)) \mid a \in \mathcal{A} \wedge \\
&\qquad (x,y) \in \{(P,Q),(Q,P)\}\} \\
\mathcal{E}_3 &:= \{(\mu Y^{x,y}(d{:}D^x, d'{:}D^y) = \mathsf{close}^{x,y}(d,d')) \mid (x,y) \in \{(P,Q),(Q,P)\}\}
\end{aligned}$$

and we use the following abbreviations, for all $a \in \mathcal{A} \wedge (x,y) \in \{(P,Q),(Q,P)\}$:

$$\begin{aligned}
\mathsf{match}^{x,y}(d{:}D^x, d'{:}D^y) &= \textstyle\bigwedge_{a\in\mathcal{A}} \forall e{:}E_a^x\,.\,(h_a^x(d,e) \Longrightarrow (Y_a^{x,y}(d,d',e) \vee \\
&\qquad (a = \tau \wedge X^{x,y}(g_\tau^x(d,e), d')))) \\
\mathsf{close}_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) &= \exists e'{:}E_\tau^y\,.\,(h_\tau^y(d',e') \wedge Y_a^{x,y}(d, g_\tau^y(d',e'), e)) \vee \\
&\qquad \mathsf{step}_a^{x,y}(d,d',e) \\
\mathsf{step}_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) &= \exists e'{:}E_a^y\,.\,h_a^y(d',e') \wedge f_a^x(d,e) = f_a^y(d',e') \\
&\qquad \wedge Y^{x,y}(g_a^x(d,e), g_a^y(d',e')) \\
\mathsf{close}^{x,y}(d{:}D^x, d'{:}D^y) &= \exists e'{:}E_\tau^y\,.\,(h_\tau^y(d',e') \wedge Y^{x,y}(d, g_\tau^y(d',e'))) \vee \\
&\qquad X^{x,y}(d,d')\,.
\end{aligned}$$

Algorithm 5.4. Construction of a PBES for branching similarity on P and Q.
$\textsc{Sim}_b(P,Q) = \mathcal{E}_1\ \mathcal{E}_2$, where:

$$\begin{aligned}
\mathcal{E}_1 &:= (\nu X(d{:}D^P, d'{:}D^Q) = X^{P,Q}(d,d') \wedge X^{Q,P}(d',d)) \\
&\quad\ (\nu X^{P,Q}(d{:}D^P, d'{:}D^Q) = \mathsf{match}^{P,Q}(d,d')) \\
&\quad\ (\nu X^{Q,P}(d'{:}D^Q, d{:}D^P) = \mathsf{match}^{Q,P}(d',d))\,, \\
\mathcal{E}_2 &:= \{(\mu Y_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) = \mathsf{close}_a^{x,y}(d,d',e)) \mid a \in \mathcal{A} \wedge \\
&\qquad (x,y) \in \{(P,Q),(Q,P)\}\}
\end{aligned}$$

and we use the following abbreviations, for all $a \in \mathcal{A} \wedge (x,y) \in \{(P,Q),(Q,P)\}$:

$$\begin{aligned}
\mathsf{match}^{x,y}(d{:}D^x, d'{:}D^y) &= \textstyle\bigwedge_{a\in\mathcal{A}} \forall e{:}E_a^x\,.\,(h_a^x(d,e) \Longrightarrow Y_a^{x,y}(d,d',e)) \\
\mathsf{close}_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) &= \exists e'{:}E_\tau^y\,.\,(h_\tau^y(d',e') \wedge Y_a^{x,y}(d, g_\tau^y(d',e'), e)) \vee \\
&\qquad (X^{x,y}(d,d') \wedge \mathsf{step}_a^{x,y}(d,d',e)) \\
\mathsf{step}_a^{x,y}(d{:}D^x, d'{:}D^y, e{:}E_a^x) &= (a = \tau \wedge X^{x,y}(g_\tau^x(d,e), d')) \vee \\
&\qquad \exists e'{:}E_a^y\,.\,h_a^y(d',e') \wedge f_a^x(d,e) = f_a^y(d',e') \\
&\qquad\ \wedge X^{x,y}(g_a^x(d,e), g_a^y(d',e'))\,.
\end{aligned}$$

simplicity. We have demonstrated our method for branching bisimilarity on two examples, showing that the concatenation of two one-place buffers behave branching bisimilar to a queue of size 2, and that the concatenation of two unbounded queues is branching bisimilar to the concatenation of three unbounded queues. The latter example could not be solved directly with the cones and foci method.

Our solution is a symbolic verification algorithm. Compared to the previously known algorithms, it has the advantage that the solution of the PBES indicates exactly which states of the two input processes are bisimilar. This provides some positive feedback in case the initial states of the two systems are not bisimilar. For infinite systems it is essential that the PBES has alternation depth two. In the finite case an alternation depth of one can be obtained by exploring τ-paths on-the-fly. This explicitly enumerates the τ-closures in sub-formulae on the right-hand side, instead of implicitly encoding them in least-fixpoint equations and leaving their exploration to PBES solution techniques like approximation.

Note that we have introduced a *generic* scheme that can be applied to other weak equivalences and preorders in the branching-time spectrum [54], and also to other formalisms of concurrency. Other possible extensions of our method are the various equivalences for mobile processes – in particular the π-calculus [100] – such as weak early, late and open bisimilarity.

By encoding equivalence-checking problems for infinite-state systems in equation systems, we have demonstrated the generality of the PBES-framework, in which the model-checking problem for infinite-state systems and the first-order μ-calculus could already be encoded. The advantage of using a single formalism for representing these problems is evident: research can be focused on studying PBESs, and any developed tool or technique that aids in solving PBESs, readily aids in solving any of the encoded problems. The next chapter reports on such a technique.

Chapter 6

Instantiation for Equation Systems

6.1 Introduction

As described in section 2.6 and chapter 5, PBESs are useful for representing various verification problems on possibly infinite-state systems. The problem is then shifted to solving the PBES, which is generally undecidable. Nevertheless, an array of techniques is available to manipulate a PBES such that a solution may still be obtained (see section 2.6.3). In this chapter, we extend this collection with a technique called *instantiation* that aims to eliminate data from a PBES. When this is done *partially*, as described in section 6.3, it results in a new PBES. On the other hand, *full* instantiation (section 6.4) yields a BES when all involved data sorts are finite, and an *infinite BES* (IBES) [92] when all data sorts are countable. The concept of IBESs is introduced in section 6.2.

From a theoretical viewpoint, our transformations firmly relate the aforementioned formalisms. Moreover, the transformations help in understanding the intricate theory underlying PBESs. In particular, they confirm the intuition that the inherent computational complexity in PBESs is due to the complexity of the data sorts that appear in the equations.

On a more practical level, our technique for partially instantiating a PBES leads to a wider applicability of existing solution techniques – such as the use of a pattern – even in the presence of countably and uncountably infinite data sorts. A full instantiation of a PBES to a BES allows for complete automation: BESs form a subset of PBESs for which computing the solution is effectively decidable (see also chapter 7). As such, both types of instantiation are welcome additions to the growing library of PBES manipulation methods.

We illustrate the efficacy of our instantiation techniques by several examples in section 6.5. Partial instantiation is demonstrated to be very effective on two

model-checking problems that involve infinite-state systems and have been taken from the literature. These examples contain both finite and infinite data sorts, and partial instantiation enables the use of pattern matching to solve the problem. The effectiveness of fully instantiating a PBES to an (I)BES is illustrated by a prototype tool that implements this instantiation. We use it to solve equivalence-checking problems (see chapter 5) and local model-checking problems that are encoded in PBESs.

Our transformations take inspiration from the algorithm that is proposed in [93] to construct *and* solve an alternation-free PBES – encoding the local model-checking problem for the alternation-free modal μ-calculus – in an on-the-fly manner. The experiments with our prototype implementation suggest that the approach is practically feasible. This is in accordance with the results reported in [95] where another implementation of this on-the-fly approach is described.

6.2 Infinite Boolean equation systems

Mader [91] introduces *infinite* Boolean equation systems (IBESs) as a vehicle for solving a model-checking problem for infinite-state systems. IBESs resemble BESs but differ in the following aspects: (1) finite *and* countably infinite conjunction and disjunction over proposition variables are allowed, and (2) finite *and* countably infinite sequences of equations, called *blocks*, are allowed (but still only finitely many blocks).

Definition 6.1. *Infinite proposition formulae* ω are defined by the following grammar, for any countable sorts I and $J \subseteq I$:

$$\omega ::= \top \mid \bot \mid X_i \mid \omega \oplus \omega \mid \bigoplus_{j \in J} \omega$$

where $\oplus \in \{\wedge, \vee\}$, $\bigoplus \in \{\bigwedge, \bigvee\}$ and $X_i{:}B$ is a proposition variable for any $i \in I$.

Here, $\bigwedge_{j \in J}$ and $\bigvee_{j \in J}$ denote the infinite conjunction and disjunction over basic elements of a countable sort J, respectively.

Definition 6.2. *Infinite Boolean equation systems* (IBES) $\mathcal{E}$ are defined by the following grammar:

$$\mathcal{E} ::= \epsilon \mid \sigma\mathcal{B}\ \mathcal{E}$$

where $\sigma \in \{\mu, \nu\}$ and $\sigma\mathcal{B} = \{\sigma X_j = \omega_j \mid j \in J\}$ is a *block* for some countable sort J, proposition variables $X_j{:}B$ and infinite proposition formulae ω_j.

Notice that BESs are, syntactically, exactly in the intersection of PBESs and IBESs. The notions of binding and occurring variables, and the derived notions of *open*, *closed* and *well-formedness* that are defined for PBESs in section 2.6, transfer to IBESs without problems. We restrict to IBESs that are well-formed.

The semantics of infinite proposition formulae is defined in the context of a proposition environment $\eta : \mathcal{X} \to \mathbb{B}$. For any countable sort I, environment η

and function $f : I \to \mathbb{B}$ we denote by X_I the set of variables $\{X_i \mid i \in I\}$ and by $\eta[X_I \mapsto f]$ the simultaneous assignment of $f(i)$ to X_i in η for all $i \in I$, such that $\eta[X_I \mapsto f](X_i) = f(i)$ if $i \in I$ and $\eta(X_i)$ otherwise.

Definition 6.3. Let η be a proposition environment. The interpretation $[\![\omega]\!]\eta$ of an infinite proposition formula ω in the context of η is a Boolean value that is defined inductively as follows:

$$\begin{aligned}
[\![\top]\!]\eta &:= \top \\
[\![\bot]\!]\eta &:= \bot \\
[\![X_i]\!]\eta &:= \eta(X_i) \\
[\![\omega_1 \oplus \omega_2]\!]\eta &:= [\![\omega_1]\!]\eta \oplus [\![\omega_2]\!]\eta \\
[\![\textstyle\bigoplus_{j\in J} \omega]\!]\eta &:= \mathsf{Q} v \in J \,.\, [\![\omega[v/j]]\!]\eta
\end{aligned}$$

where $\mathsf{Q} = \forall$ if $\bigoplus = \bigwedge$, and $\mathsf{Q} = \exists$ if $\bigoplus = \bigvee$.

For (countable) sort I, the poset $(\mathbb{B}^I, \leq)$ is a complete lattice. Let $\Omega = \{\omega_i \mid i \in I\}$ be a countable set of infinite proposition formulae. The functional induced by the interpretation of Ω is written $(\lambda i \in I \,.\, [\![\omega_i]\!]\eta)$, with $\omega_i \in \Omega$. This leads to the following transformer on infinite proposition formulae:

$$\lambda g \in \mathbb{B}^I \,.\, (\lambda i \in I \,.\, [\![\omega_i]\!]\eta[X_I \mapsto g]).$$

This transformer is a monotonic operator on $(\mathbb{B}^I, \leq)$. Hence, the existence of its least and greatest fixpoints is guaranteed.

Definition 6.4. Let η be a proposition environment, $\mathcal{E}$ be an IBES and $\sigma\mathcal{B} = \{\sigma X_i = \omega_i \mid i \in I\}$ be a block for some countable sort I. The *solution of an IBES* is inductively defined as follows:

$$\begin{aligned}
[\![\epsilon]\!]\eta &:= \eta \\
[\![\sigma\mathcal{B}\ \mathcal{E}]\!]\eta &:= [\![\mathcal{E}]\!]\eta[X_I \mapsto \sigma f \in \mathbb{B}^I \,.\, \lambda i \in I \,.\, [\![\omega_i]\!]([\![\mathcal{E}]\!]\eta[X_I \mapsto f])].
\end{aligned}$$

The solution of an IBES assigns a value to *every* proposition variable that occurs as a binding variable in the IBES. Often, only the value for specific proposition variables is sought, *e.g.* in local model checking. In such a case, equations that are unimportant to the solution of that variable can be pruned, yielding a smaller IBES, or even a BES. This follows from the following results.

Lemma 6.5. *Let $\mathcal{F}$ and $\mathcal{G}$ be IBESs. Let η be an arbitrary environment. Then* $\mathsf{occ}(\mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}) = \emptyset \implies \forall X \notin \mathsf{bnd}(\mathcal{F}) \,.\, [\![\mathcal{F}\ \mathcal{G}]\!]\eta(X) = [\![\mathcal{G}]\!]\eta(X)$.

Proof. We use induction on the number of blocks in $\mathcal{F}$.

Base: $\mathcal{F} = \epsilon$. Let η be an arbitrary environment. Then $[\![\mathcal{F}\ \mathcal{G}]\!]\eta(X) = [\![\epsilon\ \mathcal{G}]\!]\eta(X) = [\![\mathcal{G}]\!]\eta(X)$ for all proposition variables X.

Step: Assume that for some IBES $\mathcal{F}'$ we have for all θ:

(IH) $\quad \mathsf{occ}(\mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}') = \emptyset$ implies $\forall X \notin \mathsf{bnd}(\mathcal{F}) \,.\, [\![\mathcal{F}'\ \mathcal{G}]\!]\theta(X) = [\![\mathcal{G}]\!]\theta(X)$.

Assume that $\mathsf{occ}(\mathcal{G}) \cap \mathsf{bnd}(\sigma\mathcal{B}\ \mathcal{F}') = \emptyset$. Let $X \notin \mathsf{bnd}(\sigma\mathcal{B}\ \mathcal{F}')$. Then:

$$\begin{aligned}
[\![\sigma\mathcal{B}\ \mathcal{F}'\ \mathcal{G}]\!]\eta(X) &= [\![\mathcal{F}'\ \mathcal{G}]\!]\eta[X_I \mapsto \sigma X_I \,.\, \mathcal{B}([\![\mathcal{F}'\ \mathcal{G}]\!]\eta)](X) \\
&\overset{*}{=} [\![\mathcal{G}]\!]\eta[X_I \mapsto \sigma X_I \,.\, \mathcal{B}([\![\mathcal{F}'\ \mathcal{G}]\!]\eta)](X) \\
&\overset{\dagger}{=} [\![\mathcal{G}]\!]\eta(X)
\end{aligned}$$

where at $*$ we applied (IH) using the fact that $\mathsf{occ}(\mathcal{G}) \cap \mathsf{bnd}(\sigma\mathcal{B}\ \mathcal{F}') = \emptyset$ implies $\mathsf{occ}(\mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}') = \emptyset$, and at $\dagger$ we used $X_I \cap \mathsf{occ}(\mathcal{G}) = \emptyset$ and $X \notin X_I$. □

By the previous lemma, irrelevant equations at the start of an IBES can be removed. The following proposition generalizes this result to equations that occur anywhere in the IBES.

Proposition 6.6. *Let $\mathcal{E}, \mathcal{F}, \mathcal{G}$ be arbitrary IBESs. Then for all environments η,* $\mathsf{occ}(\mathcal{E}\ \mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}) = \emptyset \implies \forall X \notin \mathsf{bnd}(\mathcal{F}) \,.\, [\![\mathcal{E}\ \mathcal{F}\ \mathcal{G}]\!]\eta(X) = [\![\mathcal{E}\ \mathcal{G}]\!]\eta(X)$.

Proof. We use induction on the number of blocks in $\mathcal{E}$.

Base: $\mathcal{E}$ consists of zero blocks. Then $\mathsf{occ}(\mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}) = \emptyset$ and by lemma 6.5 we have $[\![\mathcal{F}\ \mathcal{G}]\!]\eta(X) = [\![\mathcal{G}]\!]\eta(X)$ for all $X \notin \mathsf{bnd}(\mathcal{F})$.

Step: Assume that for an IBES $\mathcal{E}'$ and all environments θ we have:

(IH) $\quad \mathsf{occ}(\mathcal{E}'\ \mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}) = \emptyset \implies \forall X \notin \mathsf{bnd}(\mathcal{F}) \,.\, [\![\mathcal{E}'\ \mathcal{F}\ \mathcal{G}]\!]\eta(X) = [\![\mathcal{E}'\ \mathcal{G}]\!]\eta(X)$.

Suppose that $\mathsf{occ}(\sigma\mathcal{B}\ \mathcal{E}'\ \mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}) = \emptyset$. Let $X \notin \mathsf{bnd}(\mathcal{F})$. Then:

$$\begin{aligned}
[\![\sigma\mathcal{B}\ \mathcal{E}'\ \mathcal{F}\ \mathcal{G}]\!]\eta(X) &= [\![\mathcal{E}'\ \mathcal{F}\ \mathcal{G}]\!]\eta[X_I \mapsto \sigma X_I \,.\, \mathcal{B}([\![\mathcal{E}'\ \mathcal{F}\ \mathcal{G}]\!]\eta)](X) \\
&\overset{(\mathrm{IH})}{=} [\![\mathcal{E}'\ \mathcal{G}]\!]\eta[X_I \mapsto \sigma X_I \,.\, \mathcal{B}([\![\mathcal{E}'\ \mathcal{F}\ \mathcal{G}]\!]\eta)](X) \\
&\overset{*}{=} [\![\mathcal{E}'\ \mathcal{G}]\!]\eta[X_I \mapsto \sigma X_I \,.\, \mathcal{B}([\![\mathcal{E}'\ \mathcal{G}]\!]\eta)](X) \\
&= [\![\sigma\mathcal{B}\ \mathcal{E}'\ \mathcal{G}]\!]\eta(X).
\end{aligned}$$

where at $*$ we used that $\mathsf{occ}(\sigma\mathcal{B}\ \mathcal{E}'\ \mathcal{G}) \cap \mathsf{bnd}(\mathcal{F}) = \emptyset$ implies $\mathcal{B}([\![\mathcal{E}'\ \mathcal{F}\ \mathcal{G}]\!]\eta) = \mathcal{B}([\![\mathcal{E}'\ \mathcal{G}]\!]\eta)$. □

Infinite BESs are used in section 6.4 when we instantiate PBESs containing countably infinite domains. We first look at instantiation on finite domains, for which normal BESs suffice.

Algorithm 6.1. The PBES instantiation algorithm $\textsc{Inst}_{\mathcal{P}}$.

For any $\mathcal{P} \subseteq \mathcal{X}$, with $\mathcal{P} \neq \emptyset$:

$$
\begin{aligned}
&\textsc{Inst}_{\emptyset}(\mathcal{E}) && := \mathcal{E} \\
&\textsc{Inst}_{\mathcal{P}}(\epsilon) && := \epsilon \\
&\textsc{Inst}_{\mathcal{P}}((\sigma X(d{:}D, e{:}E) = \varphi)\ \mathcal{E}) && := \\
&\qquad \begin{cases} \{(\sigma X_v(e{:}E) = \textsc{Sub}_{\mathcal{P}}(\varphi[v/d])) \mid v \in D\}\ \textsc{Inst}_{\mathcal{P}}(\mathcal{E}) & \text{if } X \in \mathcal{P} \\ (\sigma X(d{:}D, e{:}E) = \textsc{Sub}_{\mathcal{P}}(\varphi))\ \textsc{Inst}_{\mathcal{P}}(\mathcal{E}) & \text{otherwise,} \end{cases}
\end{aligned}
$$

where

$$
\begin{aligned}
&\textsc{Sub}_{\emptyset}(\varphi) && := \varphi \\
&\textsc{Sub}_{\mathcal{P}}(b) && := b \\
&\textsc{Sub}_{\mathcal{P}}(X(d,e)) && := \begin{cases} \bigvee_{v \in D}(v = d \wedge X_v(e)) & \text{if } X \in \mathcal{P} \\ X(d,e) & \text{otherwise} \end{cases} \\
&\textsc{Sub}_{\mathcal{P}}(\varphi_1 \oplus \varphi_2) && := \textsc{Sub}_{\mathcal{P}}(\varphi_1) \oplus \textsc{Sub}_{\mathcal{P}}(\varphi_2) \\
&\textsc{Sub}_{\mathcal{P}}(\mathsf{Q}d{:}D\,.\,\varphi) && := \mathsf{Q}d{:}D\,.\,\textsc{Sub}_{\mathcal{P}}(\varphi).
\end{aligned}
$$

6.3 Instantiation on finite domains

Instantiation is a transformation on PBESs that, for every variable X from a given set $\mathcal{P}$, replaces the equation $(\sigma X(d{:}D, e{:}E) = \varphi)$ by an entire PBES $(\sigma X_{d_1}(e{:}E) = \varphi_{d_1}) \cdots (\sigma X_{d_n}(e{:}E) = \varphi_{d_n})$. The transformation is given as algorithm 6.1 for general PBESs $\mathcal{E}$ and arbitrary sets $\mathcal{P}$. Although the basic idea of the transformation is elementary, the devil is in the detail: careful bookkeeping and a naming scheme have to be applied to make the transformation work. This is taken care of by the function $\textsc{Sub}_{\mathcal{P}}$ that is used in the main transformation $\textsc{Inst}_{\mathcal{P}}$. It correctly introduces new predicate variables in the right-hand sides of the equations of the PBES, as we prove below. In the definition of $\textsc{Sub}_{\mathcal{P}}$, the operand $\bigvee_{v \in D}$ abbreviates a finite disjunction over all basic elements v in D.

The soundness of the transformation is far from obvious due to the newly introduced predicate variables and the modifications to the right-hand sides. We prove that the transformation indeed preserves the solution of the original PBES, and claim a precise correspondence between the original PBES and the transformed PBES. Since the main proof is involved, we first prove correctness of algorithm 6.1 for $|\mathcal{P}| = 1$ – *i.e.* when a single variable is instantiated – in section 6.3.1. The correctness proof for the general case (arbitrary $\mathcal{P}$) is given in section 6.3.2 and relies on the results of section 6.3.1.

Without loss of generality, we assume that all predicate variables in this section are either of type $D \times E \to B$ or of type $E \to B$, for some finite sort D and some possibly infinite sort E. The finite sort D is used as the sort that is instantiated

for a given predicate variable. We use the sort F when we mean either domain D or $D \times E$. With each predicate variable $X : D \times E \to B$ we associate a finite set of predicate variables $\mathsf{all}(X) := \{X_d : E \to B \mid d \in D\}$. For any PBES $\mathcal{E}$, we say that the predicate variable X is *instantiation-fresh* for $\mathcal{E}$ iff $\mathsf{all}(X) \cap \mathsf{var}(\mathcal{E}) = \emptyset$.

6.3.1 Instantiation for a single predicate variable

In order to facilitate the proof of the main theorem of this section, we first address several lemmas concerning the functions $\textsc{Sub}_{\{X\}}$ and $\textsc{Inst}_{\{X\}}$ to which we refer as $\textsc{Sub}_X$ and $\textsc{Inst}_X$ for conciseness. The soundness of $\textsc{Sub}_X$ is established by the following lemma: for any φ the interpretations of φ and $\textsc{Sub}_X(\varphi)$ within the context of an environment η correspond, provided that $\eta(X)(\llbracket v \rrbracket) = \eta(X_v)$ for all $v \in D$.

Lemma 6.7. *Let φ be a predicate formula and $X{:}D \times E \to B$ be a predicate variable. Let η be an environment such that $\eta(X)(\llbracket v \rrbracket) = \eta(X_v)$ for all $v \in D$. Then for any environment ε, $\llbracket \varphi \rrbracket \eta\varepsilon = \llbracket \textsc{Sub}_X(\varphi) \rrbracket \eta\varepsilon$.*

Proof. Let ε be a data environment. We prove the statement by induction on the structure of φ.

1. $\varphi \equiv b$. This holds trivially.
2. $\varphi \equiv X(d, e)$. Then, using isomorphism between $\mathbb{D} \times \mathbb{E} \to \mathbb{B}$ and $\mathbb{D} \to \mathbb{E} \to \mathbb{B}$:
$$\begin{aligned}\llbracket X(d,e) \rrbracket \eta\varepsilon &= \eta(X)(\llbracket d \rrbracket \varepsilon)(\llbracket e \rrbracket \varepsilon) = \textstyle\bigvee_{v \in D}(\llbracket v \rrbracket = \llbracket d \rrbracket \varepsilon \wedge \eta(X)(\llbracket v \rrbracket)(\llbracket e \rrbracket \varepsilon)) \\ &= \textstyle\bigvee_{v \in D}(\llbracket v \rrbracket = \llbracket d \rrbracket \varepsilon \wedge \eta(X_v)(\llbracket e \rrbracket \varepsilon)) \\ &= \llbracket \textstyle\bigvee_{v \in D}(v = d \wedge X_v(e)) \rrbracket \eta\varepsilon = \llbracket \textsc{Sub}_X(X(d,e)) \rrbracket \eta\varepsilon.\end{aligned}$$
3. $\varphi \equiv Y(d, e)$ for $Y \neq X$. This holds trivially.
4. We assume for formulae φ_i, where $i \in \{1, 2\}$:

   (IH) $\qquad \llbracket \varphi_i \rrbracket \eta\varepsilon = \llbracket \textsc{Sub}_X(\varphi_i) \rrbracket \eta\varepsilon.$

   (a) $\varphi \equiv \varphi_1 \oplus \varphi_2$. Then:
$$\begin{aligned}\llbracket \varphi_1 \oplus \varphi_2 \rrbracket \eta\varepsilon &= \llbracket \varphi_1 \rrbracket \eta\varepsilon \oplus \llbracket \varphi_2 \rrbracket \eta\varepsilon \overset{\text{(IH)}}{=} \llbracket \textsc{Sub}_X(\varphi_1) \rrbracket \eta\varepsilon \oplus \llbracket \textsc{Sub}_X(\varphi_2) \rrbracket \eta\varepsilon \\ &= \llbracket \textsc{Sub}_X(\varphi_1 \oplus \varphi_2) \rrbracket \eta\varepsilon.\end{aligned}$$
   (b) $\varphi \equiv \mathsf{Q} f{:}F \,.\, \varphi_1$. Then:
$$\begin{aligned}\llbracket \mathsf{Q} f{:}F \,.\, \varphi_1 \rrbracket \eta\varepsilon &= \mathsf{Q} w \in \mathbb{F} \,.\, \llbracket \varphi_1 \rrbracket \eta(\varepsilon[f \mapsto w]) \\ &\overset{\text{(IH)}}{=} \mathsf{Q} w \in \mathbb{F} \,.\, \llbracket \textsc{Sub}_X(\varphi_1) \rrbracket \eta(\varepsilon[f \mapsto w]) \\ &= \llbracket \textsc{Sub}_X(\mathsf{Q} f{:}F \,.\, \varphi_1) \rrbracket \eta\varepsilon.\end{aligned}$$
   □

In the correctness proof below, we will encounter PBESs in which an unbound predicate variable is instantiated. As shown by the following lemma, instantiating an unbound variable X in a PBES $\mathcal{E}$ does not change the solution of $\mathcal{E}$ in the context of an environment η, provided that $\eta(X)(\llbracket v \rrbracket) = \eta(X_v)$ for all $v \in D$.

Lemma 6.8. *Let $\mathcal{E}$ be a PBES for which the predicate variable $X{:}D \times E \to B$ is instantiation-fresh and $X \notin \mathsf{bnd}(\mathcal{E})$. Let η be an environment such that $\eta(X)([\![v]\!]) = \eta(X_v)$ for all $v \in D$. Then for any environment ε, $[\![\mathcal{E}]\!]\eta\varepsilon = [\![\textsc{Inst}_X(\mathcal{E})]\!]\eta\varepsilon$.*

Proof. Let ε be a data environment. We prove the lemma by induction on the length of $\mathcal{E}$. If $\mathcal{E}$ is of length 0 then: $[\![\epsilon]\!]\eta\varepsilon = \eta = [\![\textsc{Inst}_X(\epsilon)]\!]\eta\varepsilon$. Suppose $\mathcal{E}$ is of length $m+1$ and for all $\mathcal{E}'$ of length m, we have, for any environment v:

$$\text{(IH)} \qquad [\![\mathcal{E}']\!]\eta v = [\![\textsc{Inst}_X(\mathcal{E}')]\!]\eta v.$$

Necessarily, $\mathcal{E}$ is of the form $(\sigma Z(f{:}F) = \varphi)\ \mathcal{F}$, where $\mathcal{F}$ is of length m. We derive:

$$\begin{aligned}
&[\![\mathcal{E}]\!]\eta\varepsilon = [\![(\sigma Z(f{:}F) = \varphi)\ \mathcal{F}]\!]\eta\varepsilon \\
&\quad = [\![\mathcal{F}]\!]\eta[Z \mapsto (\sigma g \in \mathbb{B}^{\mathbb{F}} \,.\, \lambda v \in \mathbb{F} \,.\, [\![\varphi]\!](([\![\mathcal{F}]\!]\eta[Z \mapsto g]\varepsilon)\varepsilon[f \mapsto v])]\varepsilon \\
&\quad \overset{*}{=} [\![\mathcal{F}]\!]\eta[Z \mapsto (\sigma g \in \mathbb{B}^{\mathbb{F}} \,.\, \lambda v \in \mathbb{F} \,.\, [\![\textsc{Sub}_X(\varphi)]\!](([\![\mathcal{F}]\!]\eta[Z \mapsto g]\varepsilon)\varepsilon[f \mapsto v])]\varepsilon \\
&\quad \overset{\text{(IH)}}{=} [\![\textsc{Inst}_X(\mathcal{F})]\!]\eta[Z \mapsto (\sigma g \in \mathbb{B}^{\mathbb{F}} \,.\, \lambda v \in \mathbb{F} \,. \\
&\qquad\qquad [\![\textsc{Sub}_X(\varphi)]\!](([\![\textsc{Inst}_X(\mathcal{F})]\!]\eta[Z \mapsto g]\varepsilon)\varepsilon[f \mapsto v])]\varepsilon \\
&\quad = [\![\textsc{Inst}_X((\sigma Z(f{:}F) = \varphi)\ \mathcal{F})]\!]\eta\varepsilon \\
&\quad = [\![\textsc{Inst}_X(\mathcal{E})]\!]\eta\varepsilon
\end{aligned}$$

where at $*$ we used the following equivalence:

$$(\sigma g \in \mathbb{B}^{\mathbb{F}}.[\![\varphi]\!](([\![\mathcal{F}]\!]\eta[Z \mapsto g]\varepsilon)\varepsilon) = (\sigma g \in \mathbb{B}^{\mathbb{F}}.[\![\textsc{Sub}_X(\varphi)]\!](([\![\mathcal{F}]\!]\eta[Z \mapsto g]\varepsilon)\varepsilon)$$

which follows readily from lemma 6.7. Observe that this lemma applies because $([\![\mathcal{F}]\!]\eta[Z \mapsto g]\varepsilon)(X)([\![v]\!]) = ([\![\mathcal{F}]\!]\eta[Z \mapsto g]\varepsilon)(X_v)$ for all $v \in D$ by assumption on η, instantiation-freshness of X for $\mathcal{E}$, $X \notin \mathsf{bnd}(\mathcal{F})$ and lemma 2.20. □

Suppose we have a PBES $\mathcal{E}$ in which the first equation is for variable X and X is instantiated in that PBES, yielding the PBES $\textsc{Inst}_X(\mathcal{E})$. The following lemma states that the solution to X in the original PBES and the solutions to its instantiated counterparts in the resulting PBES correspond.

Lemma 6.9. *Let $\mathcal{F}$ be a PBES of the form $(\sigma X(d{:}D, e{:}E) = \varphi)\ \mathcal{E}$ such that X is instantiation-fresh for $\mathcal{F}$. Then for any environments η, ε:*

$$\forall v \in D \,.\, ([\![\textsc{Inst}_X(\mathcal{F})]\!]\eta\varepsilon)(X_v) = (([\![\mathcal{F}]\!]\eta\varepsilon)(X))([\![v]\!]).$$

Proof. Assume that $D = \{v_1, \ldots, v_n\}$; then $|D| = |\mathbb{D}| = n$ and take $i \in \{1, \ldots, n\}$. Let η, ε be environments and $\mathcal{E}_\iota := \textsc{Inst}_X(\mathcal{E})$. First, we rewrite the left-hand side of the equality as follows:

$$
\begin{aligned}
&(\llbracket \textsc{Inst}_X(\mathcal{F})\rrbracket\eta\varepsilon)(X_{v_i})\\
&\overset{*}{=} \pi_i\big((\sigma(g_{v_1},\ldots,g_{v_n}) \in (\mathbb{B}^{\mathbb{E}})^n\,.\\
&\quad (\lambda w{\in}\mathbb{E}\,.\,\llbracket \textsc{Sub}_X(\varphi[v_1/d])\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[(X_{v_1},\ldots,X_{v_n}) \mapsto (g_{v_1},\ldots,g_{v_n})]\varepsilon)\varepsilon[e\mapsto w],\ldots,\\
&\quad\ \lambda w{\in}\mathbb{E}\,.\,\llbracket \textsc{Sub}_X(\varphi[v_n/d])\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[(X_{v_1},\ldots,X_{v_n}) \mapsto (g_{v_1},\ldots,g_{v_n})]\varepsilon)\varepsilon[e\mapsto w])\big)\big)\\
&\overset{\dagger}{=} \big(\sigma g \in \mathbb{B}^{\mathbb{D}\to\mathbb{E}}\,.\,(\lambda u\in\mathbb{D}\,.\,\lambda w\in\mathbb{E}\,.\\
&\qquad \llbracket \textsc{Sub}_X(\varphi)\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[X_{v_1}\mapsto g(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto g(\llbracket v_n\rrbracket)]\varepsilon)\varepsilon[d\mapsto u,e\mapsto w])\big)(\llbracket v_i\rrbracket).
\end{aligned}
$$

At $*$ we used Bekič's theorem [7] to replace n nested σ-fixpoints by a simultaneous σ-fixpoint over an n-tuple, and the fact that $X_{v_i} \in \mathsf{bnd}(\textsc{Inst}_X(\mathcal{F}))$. At $\dagger$ we used the assumption that the data theory is fully abstract, and the isomorphism between $(\mathbb{B}^{\mathbb{E}})^{|\mathbb{D}|}$ and $\mathbb{B}^{\mathbb{D}\to\mathbb{E}}$ to replace a tuple of functions $(g_{v_1},\ldots,g_{v_n}) : (\mathbb{E}\to\mathbb{B})^n$ by a single function $g : \mathbb{D}\to\mathbb{E}\to\mathbb{B}$ such that for any $u \in D$: $g(\llbracket u\rrbracket) = g_u$. For the right-hand side, we derive:

$$
\begin{aligned}
&(\llbracket\mathcal{F}\rrbracket\eta\varepsilon)(X(\llbracket v_i\rrbracket\varepsilon))\\
&= \big(\sigma f \in \mathbb{B}^{\mathbb{D}\times\mathbb{E}}\,.\,\lambda(u,w)\in(\mathbb{D}\times\mathbb{E})\,.\,\llbracket\varphi\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto f]\varepsilon)\varepsilon[(d,e)\mapsto(u,w)]\big)(\llbracket v_i\rrbracket)\\
&= \big(\sigma f \in \mathbb{B}^{\mathbb{D}\to\mathbb{E}}\,.\,\lambda u\in\mathbb{D}\,.\,\lambda w\in\mathbb{E}\,.\,\llbracket\varphi\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto f]\varepsilon)\varepsilon[d\mapsto u,e\mapsto w]\big)(\llbracket v_i\rrbracket).
\end{aligned}
$$

So it suffices to show the following equivalence:

$$
\begin{aligned}
&\big(\sigma f \in \mathbb{B}^{\mathbb{D}\to\mathbb{E}}\,.\,\lambda u\in\mathbb{D}\,.\,\lambda w\in\mathbb{E}\,.\,\llbracket\varphi\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto f]\varepsilon)\varepsilon[d\mapsto u,e\mapsto w]\big)\\
&= \big(\sigma g \in \mathbb{B}^{\mathbb{D}\to\mathbb{E}}\,.\,(\lambda u\in\mathbb{D}\,.\,\lambda w\in\mathbb{E}\,.\\
&\qquad \llbracket \textsc{Sub}_X(\varphi)\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[X_{v_1}\mapsto g(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto g(\llbracket v_n\rrbracket)]\varepsilon)\varepsilon[d\mapsto u,e\mapsto w])\big)
\end{aligned}
$$

which follows readily from:

$$
\begin{aligned}
&\llbracket\varphi\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto h]\varepsilon)\upsilon \;=\\
&\qquad \llbracket \textsc{Sub}_X(\varphi)\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[X_{v_1}\mapsto h(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto h(\llbracket v_n\rrbracket)]\varepsilon)\upsilon
\end{aligned}
\tag{6.1}
$$

for all environments υ and $h \in \mathbb{B}^{\mathbb{D}\to\mathbb{E}}$. Let υ be an environment and $h \in \mathbb{B}^{\mathbb{D}\to\mathbb{E}}$. We prove (6.1) using lemmas 6.7 and 6.8 as follows:

$$
\begin{aligned}
&\llbracket\varphi\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto h]\varepsilon)\upsilon\\
&\overset{*}{=} \llbracket\varphi\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto h][X_{v_1}\mapsto h(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto h(\llbracket v_n\rrbracket)]\varepsilon)\upsilon\\
&\overset{6.7}{=} \llbracket \textsc{Sub}_X(\varphi)\rrbracket(\llbracket\mathcal{E}\rrbracket\eta[X\mapsto h][X_{v_1}\mapsto h(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto h(\llbracket v_n\rrbracket)]\varepsilon)\upsilon\\
&\overset{6.8}{=} \llbracket \textsc{Sub}_X(\varphi)\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[X\mapsto h][X_{v_1}\mapsto h(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto h(\llbracket v_n\rrbracket)]\varepsilon)\upsilon\\
&\overset{\dagger}{=} \llbracket \textsc{Sub}_X(\varphi)\rrbracket(\llbracket\mathcal{E}_\iota\rrbracket\eta[X_{v_1}\mapsto h(\llbracket v_1\rrbracket),\ldots,X_{v_n}\mapsto h(\llbracket v_n\rrbracket)]\varepsilon)\upsilon
\end{aligned}
$$

where at $*$ we used that X is instantiation-fresh for $\mathcal{F}$ and at $\dagger$ we used that $X \notin \mathsf{occ}(\textsc{Sub}_X(\varphi))$. □

Note that the previous lemma does not state that instantiation does not have undesirable side-effects. This topic is addressed by the following lemma. It establishes that, when instantiating the variable of the first equation of a PBES, the solutions for non-instantiated variables are unaffected.

Lemma 6.10. *Let $\mathcal{F} := (\sigma X(d{:}D, e{:}E) = \varphi)\ \mathcal{E}$ be a PBES and let X be instantiation-fresh for $\mathcal{F}$. Then for all environments η, ε:*

$$\forall Y \in \mathcal{X} \,.\, Y \notin \mathsf{all}(X) \cup \{X\} \implies (\llbracket \textsc{Inst}_X(\mathcal{F}) \rrbracket \eta\varepsilon)(Y) = (\llbracket \mathcal{F} \rrbracket \eta\varepsilon)(Y).$$

Proof. Let η, ε be environments and $Y \in \mathcal{X}$ such that $Y \notin \mathsf{all}(X) \cup \{X\}$. Let $g : \mathbb{D} \times \mathbb{E} \to \mathbb{B}$ be such that:

$$\forall v \in D \,.\, g(\llbracket v \rrbracket) = (\llbracket \textsc{Inst}_X(\mathcal{F}) \rrbracket \eta\varepsilon)(X_v).$$

Then by lemma 6.9, we have $g = (\llbracket \mathcal{F} \rrbracket \eta\varepsilon)(X)$ and, using lemma 6.8:

$$\begin{aligned}
&(\llbracket \textsc{Inst}_X(\mathcal{F}) \rrbracket \eta\varepsilon)(Y) \\
&= (\llbracket \textsc{Inst}_X(\mathcal{E}) \rrbracket \eta[X_{v_1} \mapsto g(\llbracket v_1 \rrbracket), \ldots, X_{v_n} \mapsto g(\llbracket v_n \rrbracket)]\varepsilon)(Y) \\
&= (\llbracket \textsc{Inst}_X(\mathcal{E}) \rrbracket \eta[X \mapsto g][X_{v_1} \mapsto g(\llbracket v_1 \rrbracket), \ldots, X_{v_n} \mapsto g(\llbracket v_n \rrbracket)]\varepsilon)(Y) \\
&\overset{6.8}{=} (\llbracket \mathcal{E} \rrbracket \eta[X \mapsto g][X_{v_1} \mapsto g(\llbracket v_1 \rrbracket), \ldots, X_{v_n} \mapsto g(\llbracket v_n \rrbracket)]\varepsilon)(Y) \\
&= (\llbracket \mathcal{E} \rrbracket \eta[X \mapsto g]\varepsilon)(Y) \\
&= (\llbracket \mathcal{F} \rrbracket \eta\varepsilon)(Y).
\end{aligned}$$

□

We are now ready to prove the main theorem of this section, which generalizes lemmas 6.9 and 6.10: instantiation of a single, binding predicate variable X is sound for *arbitrary* PBESs, not just for PBESs in which X is bound in the first equation.

Theorem 6.11. *Let $\mathcal{E}$ be a PBES and $X \in \mathsf{bnd}(\mathcal{E})$ be instantiation-fresh for $\mathcal{E}$. Then for all environments η, ε:*

(a) $\forall v \in D \,.\, (\llbracket \textsc{Inst}_X(\mathcal{E}) \rrbracket \eta\varepsilon)(X_v) = (\llbracket \mathcal{E} \rrbracket \eta\varepsilon)(X(\llbracket v \rrbracket))$

(b) $\forall Y \in \mathcal{X} \,.\, Y \notin \mathsf{all}(X) \cup \{X\} \implies (\llbracket \textsc{Inst}_X(\mathcal{E}) \rrbracket \eta\varepsilon)(Y) = (\llbracket \mathcal{E} \rrbracket \eta\varepsilon)(Y).$

Proof. Observe that $\mathcal{E}$ is of the following form:

$$\mathcal{E} \ := \ \mathcal{E}_1\ \mathcal{F} \quad \text{with} \quad \mathcal{F} \ := \ (\sigma X(d{:}D, e{:}E) = \varphi)\ \mathcal{E}_2$$

for some predicate formula φ and PBESs $\mathcal{E}_1$ and $\mathcal{E}_2$. We prove the claim by induction on the structure of $\mathcal{E}_1$. For $\mathcal{E}_1 = \epsilon$, statements (a) and (b) follow immediately due to lemmas 6.9 and 6.10, respectively. Suppose $\mathcal{E}_1 = (\sigma' Z(f{:}F) = \psi)\ \mathcal{E}'$ for some PBES $\mathcal{E}'$. We assume as induction hypotheses, for all environments η', ε':

(IHa) $\forall v \in D \,.\, (\llbracket \textsc{Inst}_X(\mathcal{E}'\ \mathcal{F}) \rrbracket \eta'\varepsilon')(X_v) = (\llbracket \mathcal{E}'\ \mathcal{F} \rrbracket \eta'\varepsilon')(X(\llbracket v \rrbracket))$

(IHb) $\forall Y \in \mathcal{X} \,.\, Y \notin \mathsf{all}(X) \cup \{X\} \implies (\llbracket \textsc{Inst}_X(\mathcal{E}'\ \mathcal{F}) \rrbracket \eta'\varepsilon')(Y) = (\llbracket \mathcal{E}'\ \mathcal{F} \rrbracket \eta'\varepsilon')(Y).$

Let $v \in D$. Then for statement (a) we derive:

$$
\begin{aligned}
&(\llbracket \textsc{Inst}_X((\sigma' Z(f{:}F) = \psi)\ \mathcal{E}'\ \mathcal{F}) \rrbracket \eta\varepsilon)(X_v) \\
&= (\llbracket (\sigma' Z(f{:}F) = \textsc{Sub}_X(\psi))\ \textsc{Inst}_X(\mathcal{E}'\ \mathcal{F}) \rrbracket \eta\varepsilon)(X_v) \\
&= (\llbracket \textsc{Inst}_X(\mathcal{E}'\ \mathcal{F}) \rrbracket \eta[Z \mapsto (\sigma' h \in \mathbb{B}^{\mathbb{F}}\,. \\
&\qquad \lambda u \in \mathbb{F}\,. \llbracket \textsc{Sub}_X(\psi) \rrbracket (\llbracket \textsc{Inst}_X(\mathcal{E}'\ \mathcal{F}) \rrbracket \eta[Z \mapsto h]\varepsilon))]\varepsilon[f \mapsto u])(X_v) \\
&\overset{*}{=} (\llbracket \mathcal{E}'\ \mathcal{F} \rrbracket \eta[Z \mapsto \sigma' h \in \mathbb{B}^{\mathbb{F}}\,. \lambda u \in \mathbb{F}\,. \llbracket \psi \rrbracket (\llbracket \mathcal{E}'\ \mathcal{F} \rrbracket \eta[Z \mapsto h]\varepsilon)\varepsilon[f \mapsto u]]\varepsilon)(X(\llbracket v \rrbracket)) \\
&= (\llbracket (\sigma' Z(f{:}F) = \psi)\ \mathcal{E}'\ \mathcal{F} \rrbracket \eta\varepsilon)(X(\llbracket v \rrbracket)).
\end{aligned}
$$

At $*$ we used (IHa) and lemma 6.7 using both (IHa) and (IHb). The derivation for statement (b) follows the same line of reasoning and is therefore omitted. □

We demonstrate the use of $\textsc{Inst}_X$ by applying it to an example.

Example 6.12. Consider the following PBES $\mathcal{E}$:

$$
\begin{aligned}
\nu X(b{:}B) &= \exists n{:}N\,. Y(n) \wedge b \\
\mu Y(n{:}N) &= X(n \geq 10)\,.
\end{aligned}
$$

Instantiation of parameter b of X yields the PBES $\mathcal{E}'$ below, after minor rewriting:

$$
\begin{aligned}
\nu X_\top &= \exists n{:}N\,. Y(n) \\
\nu X_\bot &= \bot \\
\mu Y(n{:}N) &= (n \geq 10 \wedge X_\top) \vee (n < 10 \wedge X_\bot)\,.
\end{aligned}
$$

The PBES $\mathcal{E}'$ can be solved using migration, substitution and subsequent logic rewriting, which yields:

$$
\begin{aligned}
\nu X_\top &= \top \\
\mu Y(n{:}N) &= n \geq 10 \\
\nu X_\bot &= \bot\,.
\end{aligned}
$$

Hence, for arbitrary environments η, ε, we have the following correspondences:

- $(\llbracket \mathcal{E} \rrbracket \eta\varepsilon)(X)(\top) = (\llbracket \mathcal{E}' \rrbracket \eta\varepsilon)(X_\top) = \top$,
- $(\llbracket \mathcal{E} \rrbracket \eta\varepsilon)(X)(\bot) = (\llbracket \mathcal{E}' \rrbracket \eta\varepsilon)(X_\bot) = \bot$,
- $(\llbracket \mathcal{E} \rrbracket \eta\varepsilon)(Y) = (\llbracket \mathcal{E}' \rrbracket \eta\varepsilon)(Y) = (\lambda n{:}\mathbb{N}\,. n \geq 10)$.

Instantiation allows for solving a rather complex PBES $\mathcal{E}$ using standard PBES manipulation techniques and instantiation of a single predicate variable. □

By theorem 6.11 we have established the correctness of the instantiation algorithm $\textsc{Inst}_\mathcal{P}$ when $\mathcal{P} = \{X\}$ for some variable X. We now consider the more general case where an arbitrary set of variables $\mathcal{P}$ is instantiated simultaneously.

6.3.2 Simultaneous instantiation

Instantiation of a set of variables $\mathcal{P}$ in a PBES can be achieved by successively applying $\textsc{Inst}_X$ for every $X \in \mathcal{P}$. Sound as this strategy may be, it is undesirable as it is highly inefficient. The simultaneous instantiation performed by $\textsc{Inst}_{\mathcal{P}}$ is more efficient because it requires only a single pass over the entire PBES. We now prove soundness of $\textsc{Inst}_{\mathcal{P}}$ for any set $\mathcal{P}$ by showing that applying $\textsc{Inst}_{\mathcal{P}}$ yields a PBES that is syntactically equivalent to the one obtained by successively applying $\textsc{Inst}_X$ for every $X \in \mathcal{P}$. First, we prove several lemmas concerning $\textsc{Inst}_{\mathcal{P}}$ and $\textsc{Sub}_{\mathcal{P}}$. For a given formula φ, set of variables $\mathcal{P}$ and $X \in \mathcal{P}$, the following lemma states that applying $\textsc{Sub}_{\mathcal{P}}$ to φ yields the same result as applying $\textsc{Sub}_X$ after $\textsc{Sub}_{\mathcal{P}\setminus\{X\}}$ to φ.

Lemma 6.13. *Let φ be a predicate formula and $\mathcal{P}$ be a non-empty set of predicate variables such that for all $X, Y \in \mathcal{P}$, $X \notin \mathsf{all}(Y)$. Then for all $X \in \mathcal{P}$:*

$$\textsc{Sub}_{\mathcal{P}}(\varphi) = \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi)).$$

Proof. Let $X \in \mathcal{P}$. We prove the lemma by structural induction on φ.

1. $\varphi \equiv b$. Trivial.
2. $\varphi \equiv X(d, e)$. Then:
$$\begin{aligned}\textsc{Sub}_{\mathcal{P}}(X(d,e)) &= \textstyle\bigvee_{v\in D}(v = d \wedge X_v(e)) \;=\; \textsc{Sub}_X(X(d,e)) \\ &= \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(X(d,e)))\,.\end{aligned}$$
3. $\varphi \equiv Y(d, e)$ for some $Y \in \mathcal{X}$ with $Y \neq X$. If $Y \notin \mathcal{P}$ then this case is trivial. If $Y \in \mathcal{P}$ then:
$$\begin{aligned}\textsc{Sub}_{\mathcal{P}}(Y(d,e)) &= \textstyle\bigvee_{v\in D}(v = d \wedge Y_v(e)) \overset{*}{=} \textsc{Sub}_X(\textstyle\bigvee_{v\in D}(v = d \wedge Y_v(e))) \\ &\overset{\dagger}{=} \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(Y(d,e)))\end{aligned}$$
where at $*$ we used $X \notin \mathsf{all}(Y)$ and at $\dagger$ we used $Y \in \mathcal{P} \setminus \{X\}$.
4. Assume for predicate formulae φ_i, $i \in \{1, 2\}$:

   (IH) $\qquad \textsc{Sub}_{\mathcal{P}}(\varphi_i) = \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi_i))$.

   (a) $\varphi \equiv \varphi_1 \oplus \varphi_2$ for some $\oplus \in \{\vee, \wedge\}$. Then:
   $$\begin{aligned}\textsc{Sub}_{\mathcal{P}}(\varphi_1 \oplus \varphi_2) &= \textsc{Sub}_{\mathcal{P}}(\varphi_1) \oplus \textsc{Sub}_{\mathcal{P}}(\varphi_2) \\ &\overset{\text{(IH)}}{=} \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi_1)) \oplus \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi_2)) \\ &= \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi_1 \oplus \varphi_2))\,.\end{aligned}$$
   (b) $\varphi \equiv \mathsf{Q}d{:}D.\varphi_1$ for some $\mathsf{Q} \in \{\exists, \forall\}$. Then:
   $$\begin{aligned}\textsc{Sub}_{\mathcal{P}}(\mathsf{Q}d{:}D.\,\varphi_1) &= \mathsf{Q}d{:}D.\,\textsc{Sub}_{\mathcal{P}}(\varphi_1) \\ &\overset{\text{(IH)}}{=} \mathsf{Q}d{:}D.\,\textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi_1)) \\ &= \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\mathsf{Q}d{:}D\,.\,\varphi_1))\,.\end{aligned}$$ □

The next lemma establishes a similar result for $\textsc{Inst}_{\mathcal{P}}$ instead of $\textsc{Sub}_{\mathcal{P}}$.

Lemma 6.14. *Let $\mathcal{E}$ be a PBES and $\mathcal{P} \subseteq \mathsf{bnd}(\mathcal{E})$ be a non-empty set of predicate variables that is instantiation-fresh for $\mathcal{E}$. Then for all $X \in \mathcal{P}$:*

$$\textsc{Inst}_{\mathcal{P}}(\mathcal{E}) = \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E})).$$

Proof. In this proof we rely on lemma 6.13. This lemma applies because from $\mathcal{P} \subseteq \mathsf{bnd}(\mathcal{E})$ and the fact that $\mathcal{P}$ is instantiation-fresh for $\mathcal{E}$, it follows that for all $X, Y \in \mathcal{P}$, $X \notin \mathsf{all}(Y)$. We prove the lemma by induction on the length of $\mathcal{E}$. The case $\mathcal{E} = \epsilon$ is trivial. Suppose $\mathcal{E} = (\sigma Y(d{:}D, e{:}E) = \varphi)\ \mathcal{E}'$ for some PBES $\mathcal{E}'$. Assume that for all $X \in \mathcal{P}$:

$$\text{(IH)} \qquad \textsc{Inst}_{\mathcal{P}}(\mathcal{E}') = \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')).$$

Let $X \in \mathcal{P}$. We distinguish three cases and use lemma 6.13 and (IH) at every $*$:

1. $Y \notin \mathcal{P}$. Then:
$$\begin{aligned}
&\textsc{Inst}_{\mathcal{P}}((\sigma Y(d{:}D, e{:}E) = \varphi)\ \mathcal{E}') \\
&= (\sigma Y(d{:}D, e{:}E) = \textsc{Sub}_{\mathcal{P}}(\varphi))\ \textsc{Inst}_{\mathcal{P}}(\mathcal{E}') \\
&\overset{*}{=} (\sigma Y(d{:}D, e{:}E) = \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi)))\ \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')) \\
&= \textsc{Inst}_X((\sigma Y(d{:}D, e{:}E) = \textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi))\ \textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')) \\
&= \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}((\sigma Y(d{:}D, e{:}E) = \varphi)\ \mathcal{E}')).
\end{aligned}$$
2. $Y \in \mathcal{P} \wedge Y \neq X$. Observe that $X \notin \mathsf{all}(Y)$. Then:
$$\begin{aligned}
&\textsc{Inst}_{\mathcal{P}}((\sigma Y(d{:}D, e{:}E) = \varphi)\ \mathcal{E}') \\
&= \{(\sigma Y_v(e{:}E) = \textsc{Sub}_{\mathcal{P}}(\varphi[v/d])) \mid v \in D\}\ \textsc{Inst}_{\mathcal{P}}(\mathcal{E}') \\
&\overset{*}{=} \{(\sigma Y_v(e{:}E) = \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi[v/d]))) \mid v \in D\} \\
&\qquad \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')) \\
&= \textsc{Inst}_X(\{(\sigma Y_v(e{:}E) = \textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi[v/d])) \mid v \in D\}\ \textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')) \\
&= \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}((\sigma Y(d{:}D, e{:}E) = \varphi)\ \mathcal{E}')).
\end{aligned}$$
3. $Y = X$. Then:
$$\begin{aligned}
&\textsc{Inst}_{\mathcal{P}}((\sigma X(d{:}D, e{:}E) = \varphi)\ \mathcal{E}') \\
&= \{(\sigma X_v(e{:}E) = \textsc{Sub}_{\mathcal{P}}(\varphi[v/d])) \mid v \in D\}\ \textsc{Inst}_{\mathcal{P}}(\mathcal{E}') \\
&\overset{*}{=} \{(\sigma X_v(e{:}E) = \textsc{Sub}_X(\textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi[v/d]))) \mid v \in D\} \\
&\qquad \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')) \\
&= \textsc{Inst}_X((\sigma X(d{:}D, e{:}E) = \textsc{Sub}_{\mathcal{P}\setminus\{X\}}(\varphi))\ \textsc{Inst}_{\mathcal{P}\setminus\{X\}}(\mathcal{E}')) \\
&= \textsc{Inst}_X(\textsc{Inst}_{\mathcal{P}\setminus\{X\}}((\sigma X(d{:}D, e{:}E) = \varphi)\ \mathcal{E}')).
\end{aligned}$$

□

We introduce the following shorthand notation for functional composition of $\textsc{Inst}$ functions over a set of variables $\mathcal{P}$. Let $\leq$ be a total order on $\mathcal{X}$, being the set of

all predicate variables. Then:

$$\bigcirc_{X\in\mathcal{P}}\textsc{Inst}_X = \begin{cases} \mathcal{I} & \text{if } \mathcal{P}=\emptyset \\ \textsc{Inst}_Y \circ \bigcirc_{X\in\mathcal{P}\setminus\{Y\}}\textsc{Inst}_X & \text{for the } \leq\text{-minimal } Y\in\mathcal{P}, \\ & \text{otherwise} \end{cases}$$

where $\mathcal{I}$ denotes the identity function for PBESs, *i.e.* $\mathcal{I}(\mathcal{E})=\mathcal{E}$ for all $\mathcal{E}$. We now prove that instantiating a set of variables $\mathcal{P}$ yields the same equation system as successively instantiating for every variable in $\mathcal{P}$.

Lemma 6.15. *Let $\mathcal{E}$ be a PBES and $\mathcal{P}\subseteq\mathsf{bnd}(\mathcal{E})$ be a set of predicate variables that is instantiation-fresh for $\mathcal{E}$. Then:*

$$\textsc{Inst}_{\mathcal{P}}(\mathcal{E}) = (\bigcirc_{X\in\mathcal{P}}\textsc{Inst}_X)(\mathcal{E}).$$

Proof. The proof goes by induction on the size of $\mathcal{P}$. If $\mathcal{P}=\emptyset$ then trivially: $\textsc{Inst}_{\mathcal{P}}(\mathcal{E})=\mathcal{E}=\mathcal{I}(\mathcal{E})=(\bigcirc_{X\in\mathcal{P}}\textsc{Inst}_X)(\mathcal{E})$. Otherwise, let Y be the $\leq$-minimal element of $\mathcal{P}$ and assume:

$$\text{(IH)}\qquad \textsc{Inst}_{\mathcal{P}\setminus\{Y\}}(\mathcal{E}) = (\bigcirc_{X\in\mathcal{P}\setminus\{Y\}}\textsc{Inst}_X)(\mathcal{E}).$$

Then, using lemma 6.14:

$$\begin{aligned}\textsc{Inst}_{\mathcal{P}}(\mathcal{E}) &\overset{6.14}{=} \textsc{Inst}_Y(\textsc{Inst}_{\mathcal{P}\setminus\{Y\}}(\mathcal{E})) \overset{\text{(IH)}}{=} (\textsc{Inst}_Y \circ \bigcirc_{X\in\mathcal{P}\setminus\{Y\}}\textsc{Inst}_X)(\mathcal{E}) \\ &= (\bigcirc_{X\in\mathcal{P}}\textsc{Inst}_P)(\mathcal{E}).\end{aligned}$$

□

Together with the correctness of $\textsc{Inst}_X$ (theorem 6.11), the latter result allows us to prove the main theorem of this section, which states correctness of $\textsc{Inst}_{\mathcal{P}}$.

Theorem 6.16. *Let $\mathcal{E}$ be a PBES and $\mathcal{P}\subseteq\mathsf{bnd}(\mathcal{E})$ be a set of predicate variables that is instantiation-fresh for $\mathcal{E}$. Then for all environments η,ε:*
(a) $\forall X\in\mathcal{P}\,.\,\forall v\in D\,.\,(\llbracket\textsc{Inst}_{\mathcal{P}}(\mathcal{E})\rrbracket\eta\varepsilon)(X_v) = (\llbracket\mathcal{E}\rrbracket\eta\varepsilon)(X(\llbracket v\rrbracket))$
(b) $\forall Y\in\mathcal{X}\,.\,Y\notin\mathsf{all}(\mathcal{P})\cup\mathcal{P} \Longrightarrow (\llbracket\textsc{Inst}_{\mathcal{P}}(\mathcal{E})\rrbracket\eta\varepsilon)(Y) = (\llbracket\mathcal{E}\rrbracket\eta\varepsilon)(Y)$.

Proof. The proof goes by induction on the size of $\mathcal{P}$, relying on lemma 6.15 and theorem 6.11 for proving the correctness of instantiating a single variable. □

The above result allows for a full instantiation of a PBES to a BES. This is a sound strategy when (1) all data sorts that occur in the PBES are finite, (2) the PBES is closed and data-closed, and (3) it is possible to rewrite every data term that occurs in a right-hand side of the PBES to either $\top$ or $\bot$. We assume that the latter can be achieved by a data term evaluator eval, which can be implemented using *e.g.* rewriting technology; the data term evaluator can be lifted to PBESs in a straightforward manner. The following is then a corollary to lemma 6.15.

Corollary 6.17. *Let $\mathcal{E}$ be a PBES. If $\mathcal{E}$ is closed and data-closed, all data sorts occurring in $\mathcal{E}$ are finite, and a suitable term rewriter* eval *exists, then the equation system* $\mathsf{eval}(\textsc{Inst}_{\mathsf{bnd}(\mathcal{E})}(\mathcal{E}))$ *is a BES.*

We provide a small example to illustrate the transformation performed by $\textsc{Inst}_{\mathcal{P}}$.

Example 6.18. Consider the following PBES $\mathcal{E}$:

$$\begin{aligned} \nu X(b{:}B) &= \quad b \wedge Y(\neg b) \\ \mu Y(b{:}B) &= \neg b \vee X(b)\,. \end{aligned}$$

Instantiation of X and Y in $\mathcal{E}$ yields the BES $\mathcal{E}'$ below, after minor rewriting:

$$(\nu X_\top = Y_\bot)\ (\nu X_\bot = \bot)\ (\mu Y_\top = X_\top)\ (\mu Y_\bot = \top)\,.$$

The BES $\mathcal{E}'$ can be solved using substitution and approximation, by which we obtain the following correspondence between $\mathcal{E}$ and $\mathcal{E}'$, for any $b \in B$ and environments η and ε:

- $[\![\mathcal{E}]\!]\eta\varepsilon(X)(b) = [\![\mathcal{E}']\!]\eta\varepsilon(X_b) = [\![b]\!]$, and
- $[\![\mathcal{E}]\!]\eta\varepsilon(Y)(b) = [\![\mathcal{E}']\!]\eta\varepsilon(Y_b) = \top$. □

This concludes our treatment of instantiation on finite domains. In the next section we consider instantiation on countably infinite domains.

6.4 Instantiation on countable domains

In the previous section, we assumed that the domain D of the instantiated variable was finite. Instantiation then resulted in a PBES in which the predicate variables still carried parameters with a (possibly) infinite domain. In this section, we lift the restriction of finiteness and consider PBESs in which each predicate variable is either of type $D \to B$ or of type B, where D is a possibly infinite, yet countable sort. With each predicate variable $X : D \to B$, we associate a countable set of proposition variables $\mathsf{all}(X) := \{X_d{:}B \mid d \in D\}$.

The instantiation method[1] is listed in figure 6.1; it generates an IBES from a PBES. For every equation $\sigma X(d{:}D) = \varphi$ in the PBES, a block of countably many equations is generated, each of which is of the form $\sigma X_v = \omega_v$ for some $v \in D$ and infinite proposition formula $\omega_v = \textsc{Sub}_\infty(\varphi[v/d])$. To ensure that every ω_v is indeed a proper infinite proposition formula, we rely on the term evaluator eval to rewrite every data term in $\varphi[v/d]$ to either $\top$ or $\bot$. Hence, $\varphi[v/d]$ must be closed implying that φ may contain no free data variables other than d. This is ensured by allowing only data-closed PBESs for method $\textsc{Inst}_\infty$.

The main correspondence between the predicate variables of a PBES and the proposition variables of the IBES resulting from the instantiation, is given in theorem 6.20. Below, we first lift lemma 6.7 to countable domains.

Lemma 6.19. *Let φ be a closed predicate formula and η_1, η_2 be environments such that $\forall X \in \mathsf{occ}(\varphi)\,.\,\forall v \in D\,.\,\eta_1(X)([\![v]\!]) = \eta_2(X_v)$. Then for any environment ε:*

$$[\![\varphi]\!]\eta_1\varepsilon = [\![\textsc{Sub}_\infty(\varphi)]\!]\eta_2.$$

[1] We do not use the term "algorithm" as our method does not necessarily terminate.

$$
\begin{aligned}
&\textsc{Inst}_\infty(\epsilon) &&:= \epsilon \\
&\textsc{Inst}_\infty((\sigma X(d{:}D) = \varphi)\ \mathcal{E}) &&:= \{(\sigma X_v = \textsc{Sub}_\infty(\varphi[v/d])) \mid v \in D\}\ \textsc{Inst}_\infty(\mathcal{E})
\end{aligned}
$$

where

$$
\begin{aligned}
\textsc{Sub}_\infty(b) &:= \mathsf{eval}(b) \\
\textsc{Sub}_\infty(X(d)) &:= \textstyle\bigvee_{v \in D}\ (\mathsf{eval}(v = d) \wedge X_v) \\
\textsc{Sub}_\infty(\varphi_1 \oplus \varphi_2) &:= \textsc{Sub}_\infty(\varphi_1) \oplus \textsc{Sub}_\infty(\varphi_2) \\
\textsc{Sub}_\infty(\mathsf{Q}d{:}D\,.\,\varphi) &:= \textstyle\bigoplus_{v \in D} \textsc{Sub}_\infty(\varphi[v/d])
\end{aligned}
$$

where $\bigoplus = \bigwedge$ if $\mathsf{Q} = \forall$, and $\bigoplus = \bigvee$ if $\mathsf{Q} = \exists$.

Figure 6.1. The instantiation method for countable domains $\textsc{Inst}_\infty$.

Proof. The proof is similar to the proof of lemma 6.7 and is therefore omitted. □

Theorem 6.20. *Let $\mathcal{E}$ be a data-closed PBES such that every $X \in \mathsf{var}(\mathcal{E})$ is instantiation-fresh for $\mathcal{E}$, and let η be an environment satisfying:*

$$\forall Y \in \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})\,.\,\forall w \in D\,.\,\eta(Y_w) = \eta(Y)(\llbracket w \rrbracket). \tag{6.2}$$

Then, for any environment ε:

$$\forall X \in \mathsf{bnd}(\mathcal{E})\,.\,\forall v \in D\,.\,(\llbracket \textsc{Inst}_\infty(\mathcal{E}) \rrbracket \eta)(X_v) = (\llbracket \mathcal{E} \rrbracket \eta\varepsilon)(X)(\llbracket v \rrbracket).$$

Proof. Let ε be an environment. The proof goes by induction on the length of $\mathcal{E}$. If $\mathcal{E} = \epsilon$, the statement holds vacuously. For the inductive case we assume, for all PBESs $\mathcal{E}'$ of length m for which all variables are instantiation-fresh and environments η', ε' satisfying (6.2):

$$\text{(IH)} \qquad \forall X \in \mathsf{bnd}(\mathcal{E}')\,.\,\forall v \in D\,.\,(\llbracket \textsc{Inst}_\infty(\mathcal{E}') \rrbracket \eta')(X_v) = (\llbracket \mathcal{E}' \rrbracket \eta'\varepsilon')(X)(\llbracket v \rrbracket).$$

Suppose $\mathcal{E}$ is of length $m+1$, so $\mathcal{E} = (\sigma Y(d{:}D) = \varphi)\ \mathcal{E}'$ for some PBES $\mathcal{E}'$ of length m. We define the following shorthands:

$$
\begin{aligned}
\sigma\mathcal{B} &:= \{\sigma Y_w = \textsc{Sub}_\infty(\varphi[w/d]) \mid w \in D\} \\
f &:= \sigma g \in \mathbb{B}^D\,.\,\lambda w \in D\,.\,\llbracket \textsc{Sub}_\infty(\varphi[w/d]) \rrbracket(\llbracket \textsc{Inst}_\infty(\mathcal{E}') \rrbracket \eta[Y_D \mapsto g]) \\
h &:= \sigma k \in \mathbb{B}^{\mathbb{D}}\,.\,\lambda w \in \mathbb{D}\,.\,\llbracket \varphi \rrbracket(\llbracket \mathcal{E}' \rrbracket \eta[Y \mapsto k]\varepsilon)\varepsilon[d \mapsto w].
\end{aligned}
$$

Let $X \in \mathsf{bnd}(\mathcal{E})$ and $v \in D$. Then:

$$
\begin{aligned}
&[\![\textsc{Inst}_\infty((\sigma Y(d{:}D) = \varphi)\ \mathcal{E}')]\!]\eta(X_v)\\
&= [\![\sigma\mathcal{B}\ \textsc{Inst}_\infty(\mathcal{E}')]\!]\eta(X_v)\\
&= [\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\eta[Y_D \mapsto f](X_v)\\
&= [\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\eta[Y_D \mapsto f][Y \mapsto h](X_v)\\
&\overset{*}{=} ([\![\mathcal{E}']\!]\eta[Y_D \mapsto f][Y \mapsto h]\varepsilon)(X)([\![v]\!])\\
&= ([\![\mathcal{E}']\!]\eta[Y \mapsto h]\varepsilon)(X)([\![v]\!])\\
&= ([\![(\sigma Y(d{:}D) = \varphi)\ \mathcal{E}']\!]\eta\varepsilon)(X)([\![v]\!]).
\end{aligned}
$$

At $*$ we used (IH) for which we need to prove that:

$$\forall X \in \mathsf{var}(\mathcal{E}')\,.\,\mathsf{all}(X) \cap \mathsf{var}(\mathcal{E}') = \emptyset \tag{6.3}$$

$$\forall Z \in \mathsf{occ}(\mathcal{E}') \setminus \mathsf{bnd}(\mathcal{E}')\,.\,\forall x \in D\,. \quad \eta[Y_D \mapsto f][Y \mapsto h](Z_x) = \eta[Y_D \mapsto f][Y \mapsto h](Z)([\![x]\!]). \tag{6.4}$$

Property (6.3) follows from the facts that all variables in $\mathcal{E}$ are instantiation-fresh and $\mathsf{var}(\mathcal{E}') \subseteq \mathsf{var}(\mathcal{E})$. Regarding (6.4), let $Z \in \mathsf{occ}(\mathcal{E}') \setminus \mathsf{bnd}(\mathcal{E}')$ and $x \in D$. If $Z \neq Y$ then observe that $Z \in \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$. Then because $\mathcal{E}$ and η satisfy (6.2): $\eta[Y_D \mapsto f][Y \mapsto h](Z_x) = \eta(Z_x) = \eta(Z)([\![x]\!]) = \eta[Y_D \mapsto f][Y \mapsto h](Z)([\![x]\!])$.

If $Z = Y$ then $\eta[Y_D \mapsto f][Y \mapsto h](Y_x) = f(x)$ and $\eta[Y_D \mapsto f][Y \mapsto h](Y)([\![x]\!]) = h([\![x]\!])$, so we need to prove that $f(x) = h([\![x]\!])$. Let $g : D \to \mathbb{B}$ and for that g define $k : \mathbb{D} \to \mathbb{B}$ as follows: $k([\![d]\!]) = g(d)$ for all $d \in D$. Then $f(x) = h([\![x]\!])$ follows if:

$$
\begin{aligned}
&(\lambda w \in D\,.\,[\![\textsc{Sub}_\infty(\varphi[w/d])]\!]([\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\eta[Y_D \mapsto g]))(x)\\
&\quad = (\lambda w \in \mathbb{D}\,.\,[\![\varphi]\!]([\![\mathcal{E}']\!]\eta[Y \mapsto k]\varepsilon)\varepsilon[d \mapsto w])([\![x]\!]).
\end{aligned}
$$

We derive, starting at the right-hand side:

$$
\begin{aligned}
&(\lambda w \in \mathbb{D}\,.\,[\![\varphi]\!]([\![\mathcal{E}']\!]\eta[Y \mapsto k]\varepsilon)\varepsilon[d \mapsto w])([\![x]\!])\\
&= [\![\varphi]\!]([\![\mathcal{E}']\!]\eta[Y \mapsto k]\varepsilon)\varepsilon[d \mapsto [\![x]\!]]\\
&= [\![\varphi[x/d]]\!]([\![\mathcal{E}']\!]\eta[Y \mapsto k]\varepsilon)\varepsilon\\
&= [\![\varphi[x/d]]\!]([\![\mathcal{E}']\!]\eta[Y_D \mapsto g][Y \mapsto k]\varepsilon)\varepsilon\\
&\overset{*}{=} [\![\textsc{Sub}_\infty(\varphi[x/d])]\!]([\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\eta[Y_D \mapsto g][Y \mapsto k])\\
&= [\![\textsc{Sub}_\infty(\varphi[x/d])]\!]([\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\eta[Y_D \mapsto g])\\
&= (\lambda w \in D\,.\,[\![\textsc{Sub}_\infty(\varphi[w/d])]\!]([\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\eta[Y_D \mapsto g]))(x).
\end{aligned}
$$

For convenience we define $\theta := \eta[Y_D \mapsto g][Y \mapsto k]$. At $*$ we used lemma 6.19 which is allowed because $\varphi[x/d]$ is closed (by data-closedness of $\mathcal{E}$) and we have:

$$\forall W \in \mathsf{occ}(\varphi)\,.\,\forall w \in D\,.\,([\![\mathcal{E}']\!]\theta\varepsilon)(W)([\![w]\!]) = ([\![\textsc{Inst}_\infty(\mathcal{E}')]\!]\theta)(W_w)$$

which we prove now. Let $W \in \mathsf{occ}(\varphi)$ and $w \in D$. There are three cases:

1. $W = Y$. Then $(\llbracket \mathcal{E}' \rrbracket \theta\varepsilon)(Y)(\llbracket w \rrbracket) = k(\llbracket w \rrbracket) = g(w) = (\llbracket \textsc{Inst}_\infty(\mathcal{E}') \rrbracket \theta)(Y_w)$.
2. $W \neq Y$ and $W \notin \mathsf{bnd}(\mathcal{E})$. Then:

$$(\llbracket \mathcal{E}' \rrbracket \theta\varepsilon)(W)(\llbracket w \rrbracket) = \eta(W)(\llbracket w \rrbracket) \overset{\dagger}{=} \eta(W_w) \overset{\ddagger}{=} (\llbracket \textsc{Inst}_\infty(\mathcal{E}') \rrbracket \theta)(W_w).$$

   At † we used the fact that $\mathcal{E}$ and η satisfy (6.2) in combination with $W \in \mathsf{occ}(\mathcal{E})$ and $W \notin \mathsf{bnd}(\mathcal{E})$. At ‡ we used lemma 2.20 combined with $W_w \notin \mathsf{bnd}(\textsc{Inst}_\infty(\mathcal{E}'))$.
3. $W \neq Y$ and $W \in \mathsf{bnd}(\mathcal{E})$. Observe that $W \in \mathsf{bnd}(\mathcal{E}')$. Then the equivalence follows from (IH) if we prove that all variables in $\mathcal{E}'$ are instantiation-fresh and θ satisfies (6.2). The former follows from the fact that all variables in $\mathcal{E}$ are instantiation-fresh and $\mathsf{var}(\mathcal{E}') \subseteq \mathsf{var}(\mathcal{E})$. The latter follows using similar reasonings as in cases 1 and 2. □

From theorem 6.20 we obtain the following result.

Corollary 6.21. *Let $\mathcal{E}$ be a PBES. If $\mathcal{E}$ is closed and data-closed, and all data sorts occurring in $\mathcal{E}$ are countable and a suitable term rewriter* eval *exists, then* $\textsc{Inst}_\infty(\mathcal{E})$ *is an IBES.*

We remark that for typical verification problems, such as (local) model checking and equivalence checking, a partial solution to the PBES is often satisfactory. Using proposition 6.6, it is straightforward to turn the instantiation scheme for PBESs involving countable data sorts into a procedure for computing a BES. For example, this can be achieved by an on-the-fly, depth-first or breadth-first exploration of all the equations for the required (instantiated) binding variables of the theoretical IBES. A similar technique is discussed in [93] and we do not further explore this issue here.

Example 6.22. Consider the following PBES $\mathcal{E}$:

$$\nu X(n{:}N) = n \neq 1 \wedge X(n+1)\,.$$

Applying the transformation $\textsc{Inst}_\infty$, we obtain the following IBES:

$$\{(\nu X_0 = X_1)\ (\nu X_1 = \bot)\ (\nu X_n = X_{n+1}) \mid n \geq 2\}.$$

While solving, *e.g.*, $X(5)$ by means of a transformation to IBES would require an infinite computation, solving $X(0)$ or $X(1)$ would terminate using a local resolution: $X(0)$ depends on $X(1)$ which is immediately $\bot$. □

Note that the transformation from an IBES to a PBES is elementary, provided one has a sufficiently rich data language: the sorts that are used for the blocks in the IBES can be introduced as the data sorts of the PBES, and the infinite disjunction and conjunction that occur in the infinite proposition formulae can be converted to equality tests and universal and existential quantifications, respectively.

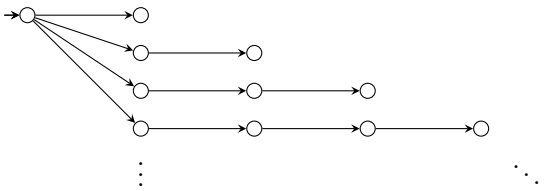


Figure 6.2. An infinite transition system in which every path from the initial state is of finite length.

6.5 Examples

In this section, we further demonstrate the instantiation techniques of the previous sections by applying them to several examples. Note that a prime example application of the manipulation is also given in section 5.4.1. First, we demonstrate that the partial instantiation of a PBES is a useful manipulation in itself. Next, we illustrate the feasibility of instantiating a given PBES to an (I)BES. We rely on several manipulation techniques for solving PBESs as they are described in section 2.6.3.

6.5.1 Model-checking infinite-state systems

Two smaller examples, derived from model-checking problems on infinite state systems, are given below. These problems first appeared in [15] and were revisited in [91] to demonstrate the efficacy of IBESs.

Checking for finite paths

Consider the following LPE:

$$P(b{:}B, n{:}N) = \sum_{i:N} b \to a \cdot X(\neg b, i) + \\ \neg b \wedge n > 0 \to a \cdot X(b, n-1)$$

with initial state $P(\top, 0)$. Its transition system is infinitely large and (partially) depicted in figure 6.2, where the a-labels have been omitted. The property that Bradfield [15] and Mader [91] verify is that every path that starts in the initial state has finite length only. This property is expressed by the following μ-calculus formula: $\mu X \,.\, [\top] X$. Note that the number of paths in the system is infinite.

The following PBES, consisting of a single equation, encodes the above model-checking problem, where satisfaction of the property by the initial state corresponds with $X(\top, 0)$:

$$\mu X(b{:}B, n{:}N) = (\forall i{:}N \,.\, \neg b \vee X(\neg b, i)) \wedge (b \vee n = 0 \vee X(b, n-1)) \,.$$

A straightforward approximation does not terminate and no patterns are applicable. Instantiation of X on the parameter b leads to the following PBES:

$$\mu X_\bot(n{:}N) = (n = 0) \vee X_\bot(n-1)$$
$$\mu X_\top(n{:}N) = \forall i{:}N \,.\, X_\bot(i).$$

Now, the equation for $X_\bot$ can easily be solved by means of a pattern (see section 2.6.3), which leads to the following equivalent equation system:

$$\mu X_\bot(n{:}N) = \exists i{:}N \,.\, n = i$$
$$\mu X_\top(n{:}N) = \forall i{:}N \,.\, X_\bot(i)\,.$$

Using standard logic, the above equation system can immediately be rewritten (even automatically [69]) to the following:

$$\mu X_\bot(n{:}N) = \top$$
$$\mu X_\top(n{:}N) = \top\,.$$

Hence the property holds for all reachable states, and the initial state in particular. The proof in [91] requires a manual construction of a set-based representation of an IBES, and requires showing the well-foundedness of mappings of this representation. The tableaux-based methods of [15] require the investigation of *extended paths*. Our proof strategy, using partial instantiation, is easier to understand and requires less effort.

Checking for a finite number of actions

Consider the following LPE:

$$\begin{aligned} P(b{:}B, n{:}N) = {} & b \rightarrow c \cdot P(\neg b, n) + \\ & b \rightarrow a \cdot P(b, n+1) + \\ & \neg b \wedge n > 0 \rightarrow a \cdot P(b, n-1) \end{aligned}$$

with initial state $P(\top, 0)$. This system originated from a Petri net given by Bradfield [15], and reappeared in [91] as a transition system. The LTS of P is depicted in figure 6.3. The property to be verified is that every behaviour in the transition system exhibits only a finite number of c actions: $\mu X \,.\, \nu Y \,.\, ([c]X \wedge [\neg c]Y)$. Note that this formula has alternation depth two.

The following PBES, consisting of two equations, encodes the above model-checking problem, where satisfaction of the formula by the initial state corresponds with $X(\top, 0)$:

$$\begin{aligned} &\mu X(b{:}B, n{:}N) = Y(b, n) \\ &\nu Y(b{:}B, n{:}N) = (\neg b \vee X(\neg b, n)) \wedge (\neg b \vee Y(b, n{+}1)) \wedge (b \vee n{=}0 \vee Y(b, n{-}1))\,. \end{aligned}$$

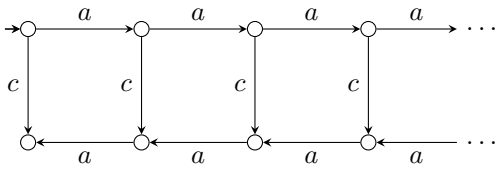


Figure 6.3. An infinite transition system that can perform only a finite number of c-steps.

The above equation system can be solved using a complex pattern which would require the introduction of an auxiliary selector function. Instantiation of Y on parameter b leads to the following PBES:

$$\begin{aligned}
\mu X(b{:}B, n{:}N) &= (b \wedge Y_\top(n)) \vee (\neg b \wedge Y_\bot(n)) \\
\nu Y_\top(n{:}N) &= X(\bot, n) \wedge Y_\top(n+1) \\
\nu Y_\bot(n{:}N) &= (n = 0) \vee Y_\bot(n-1)\,.
\end{aligned}$$

The equation for $Y_\top$ can now be solved easily by means of a pattern. The equation for $Y_\bot$ is solved instantly using symbolic approximation. This leads to the following equivalent equation system:

$$\begin{aligned}
\mu X(b{:}B, n{:}N) &= (b \wedge Y_\top(n)) \vee (\neg b \wedge Y_\bot(n)) \\
\nu Y_\top(n{:}N) &= \forall j{:}N\,.\, X(\bot, n+j) \\
\nu Y_\bot(n{:}N) &= \top\,.
\end{aligned}$$

The solutions to $Y_\bot$ and $Y_\top$ can then be substituted in the equation for X, yielding:

$$\mu X(b{:}B, n{:}N) = (b \wedge \forall j{:}N\,.\, X(\bot, n+j)) \vee \neg b\,.$$

A symbolic approximation yields the solution $\lambda b{:}B\,.\,\lambda n{:}N\,.\,\top$ for X as the third approximant. Hence the property holds for all reachable states, and the initial state in particular. Again, our proof using partial instantiation is straightforward and enables the use of simple pattern matching, while the earlier proofs by Mader and Bradfield require more effort.

6.5.2 Automatic verification

In the previous section, we used instantiation to manually solve PBESs that encoded model-checking problems on infinite-state systems. There, the use of instantiation simplified the derivation of the solution considerably, and allowed other techniques to be applied.

To assess the feasibility of automated PBES instantiation in practice, a prototype tool has been implemented that instantiates a given PBES.[2] Upon termination it has generated a BES that holds the answer to whether a particular equation in the original PBES is true for some data value. We apply the tool to verification problems on various models. Most of these models employ the data sort of natural numbers. A full instantiation of the PBES would therefore yield an IBES, but local resolution produces a finite BES in all cases.

All experiments were run on a 64-bits architecture computer having an Intel Core2 Quad 2.40 GHz CPU and 4 GB of RAM. It runs Fedora Core 8 Linux, kernel 2.6.26. We use revision 5839 of the mCRL2 toolset.

Checking for deadlocks

We used our tool to check for absence of deadlock on several publicly available benchmarks, consisting of industrial protocols and systems, and games. The deadlock property yields an alternation-free PBES. Of course, more involved properties – like fairness and liveness properties – can also be encoded, which may yield PBESs of higher alternation depths. As absence of deadlock requires all reachable states to be computed by the instantiation tool, it allows for a fair comparison with explicit state-space generation. For this, we use the mCRL2 tool lps2lts.

Table 6.1 contains the results of our experiments for each of the models. It lists the LTS sizes and the times needed to explore these LTSs by lps2lts. The rightmost column contains the times needed for instantiating the PBES to a BES *and* subsequently solving the BES using a combination of approximation and Gauß-elimination. For each model, the number of BES equations after instantiation is equal to the number of states in the corresponding LTS, as expected. The performance of the BES approach is in general comparable to that of the LTS approach; differences are attributed to minor differences in rewriting strategies.

Checking branching bisimilarity

The experiments above illustrate the efficacy of the full instantiation of alternation-free PBESs to (I)BESs. We now consider PBESs in which the branching bisimilarity problem on LPEs is encoded using the translation presented in section 5.3. The models on which we decide branching bisimilarity, are the *alternating-bit protocol* (ABP), the *concurrent alternating-bit protocol* (CABP) and the *one-place buffer* (OPB). Each protocol allows ten different messages to be communicated.

The translation of section 5.3 yields PBESs with an alternation depth of two. Every PBES is then instantiated to a BES by our tool. Finally, the BES is translated to a *parity game* which is then solved by a parity game solver[3] using the strategy from [113]. We compare this approach to an LTS-based approach, in which the LTS for each model is generated by lps2lts, after which a branching-bisimilarity

[2] The tool is called pbes2bool and is part of the mCRL2 toolset, see http://mcrl2.org.
[3] The tool PGSolver (version 2.0), available at http://www.tcs.ifi.lmu.de/pgsolver.

Table 6.1. Experimental results for checking for deadlocks in several models using both LTS exploration and PBES instantiation + BES solving.

	LTS Size		Time (sec.)	
Model	States	Transitions	LTS	PBES
BRP	10 548	12 168	5	4
Car lift	4 312	9 918	8	6
Chatbox	65 536	2 162 688	20	8
IEEE-1394	188 569	340 607	149	152
Clobber	600 161	2 221 553	92	103
Domineering	455 317	2 062 696	45	52
Othello	55 093	88 258	82	104

check on every pair of LTSs is performed by the mCRL2 tool ltscompare, that implements the algorithm from [67].

The results are listed in table 6.2. The times for solving the BES consist only of the times needed for solving the corresponding parity game. The PBES-based approach turns out to be reasonably fast for the cases involving OPB. It is also demonstrated to be feasible in the case of ABP and CABP, but there it is significantly slower than the LTS-based approach.

6.6 Conclusions

We have presented a new set of manipulations on PBESs, collectively called instantiation, by which data can be eliminated from a PBES. From a theoretical point of view, instantiation firmly relates PBESs to two other prominent types of equation systems, *viz.* BESs and IBESs, providing a different angle on the intricate fundamentals of PBESs. In practice, instantiation is a useful transformation on PBESs, allowing for a wider class of PBESs to be solved either automatically or by means of (syntactic) manipulations.

Given a PBES that contains finite parameter domains, partial instantiation can be used to selectively eliminate some of these parameters, yielding another, more simplified PBES. This can ease the process of finding a solution by allowing for other solution techniques, like patterns, to be applied. If all parameter domains are finite, a full instantiation can be applied to obtain a BES, provided that all data terms can be eliminated by a suitable term rewriter. The advantage of this approach is clear: solving BESs is decidable (see also chapter 7) and efficient tooling is readily available. We have extended full instantiation to PBESs that contain countably infinite domains. The result is an IBES that can generally not be constructed in a finite amount of time. However, in practice a local, on-the-fly

Table 6.2. Experimental results for deciding branching bisimilarity on the ABP, CABP and OPB models using both LTSs and PBESs.

LTS sizes and generation times.

Model	States	Transitions	Time (sec.)
ABP	362	460	< 0.1
CABP	3 536	13 791	0.3
OPB	11	20	< 0.1

BES sizes and times needed for PBES instantiation, BES solving and LTS comparison.

		Time (sec.)		
Equivalence	BES size	PBES inst.	BES solve	LTS
ABP $\leftrightarrows_{\mathsf{b}}$ OPB	2 884	< 0.1	< 0.1	< 0.1
OPB $\leftrightarrows_{\mathsf{b}}$ CABP	35 104	1.6	0.3	< 0.1
ABP $\leftrightarrows_{\mathsf{b}}$ CABP	268 064	9.2	13.8	< 0.1

approach can still yield a finite BES as one is usually interested in the solution for a particular variable only. We have established the necessary theoretical results to guarantee the soundness of such an approach.

We have applied our techniques to several examples. Instantiation simplifies the manual verification of two infinite-state systems considerably. Also, we have run a prototype tool on several typical verification problems. The tool implements the transformation from PBESs to (I)BESs that is required for a local resolution of the PBES, similar to the tool described in [95]. In view of this, our prototype tooling and the examples in section 6.5 demonstrate the feasibility of the approach outlined in [93].

When checking a μ-calculus formula on an LPE, the translation to a PBES may prune parts of the state space, depending on the formula. In this case, solving the PBES by a full instantiation does not require a full exploration of the state space. This is particularly beneficial when exploration of the entire (possibly infinite) state space is infeasible. The formula may prune large enough portions from the state space, such that the instantiation of the PBES becomes feasible and the resulting BES can be solved.

Chapter 7

Switching Graphs

7.1 Introduction

The previous chapter presented a technique for eliminating data from a PBES. Given a suitable data-term evaluator, this transformation yields a finite BES if all data domains are finite; if some domains are countably infinite, a finite BES may still be obtained by adopting an on-the-fly strategy. A transformation from PBESs to BESs is interesting, because, unlike PBESs, solving BESs is decidable. The problem is equivalent to the standard μ-calculus model-checking problem on finite transition systems (see *e.g.* [91]). This problem was shown to be in NP $\cap$ co-NP by Emerson, Jutla and Sistla, by showing equivalence to the non-emptiness problem on parity tree automata [40]. Stirling reduced the problem to that of deciding the winner in parity games [114]. In turn, this problem, and thereby solving BESs, was later shown to be in UP $\cap$ co-UP by Jurdziński [81]. This complexity class is a subset of NP $\cap$ co-NP and contains those problems that have a *unique* polynomial certificate (or, equivalently, can be decided in polynomial time by a nondeterministic Turing machine that has *at most one* accepting run for every problem instance).

In [60], continuing their earlier work in [59], Groote and Keinänen present a sub-quadratic algorithm for solving conjunctive and disjunctive BESs, which are BESs containing either conjunctions or disjunctions only. On the other hand, a variety of techniques for solving parity games has been proposed, among which there are intriguing hill-climbing algorithms that employ carefully chosen progress measures [82, 125]. In our search for a polynomial-time algorithm for solving general BESs (*i.e.* having both conjunctions and disjunctions), we took inspiration from these works and started to use what we call *switching graphs*. This enabled us to study the BES problem in a more intuitive setting, which allowed for easier reasoning. Also, it allowed us to define and study several variants of the problem in order to gain more insight into its nature.

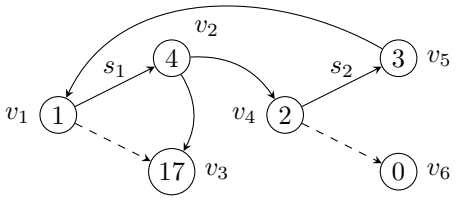


Figure 7.1. A directed labelled switching graph.

A switching graph is an ordinary graph that is extended with switches. A typical instance of a directed, (vertex-)labelled switching graph is given in figure 7.1. Note that the name of a vertex is depicted next to it, while its label is placed inside it. In figure 7.2(b) an undirected, unlabelled switching graph is depicted. A switch is a triple of vertices (v_1, v_2, v_3) in which either v_1 and v_2, or v_1 and v_3 are connected, depending on a *switch setting*, which is a function from switches to the Booleans. There are two switches in figure 7.1: $s_1 = (v_1, v_2, v_3)$ and $s_2 = (v_4, v_5, v_6)$. If the switch setting for s_1 is $\top$, the vertices v_1 and v_2 are connected, as indicated by the straight, solid arrow. If the switch setting for s_1 is $\bot$, only the dashed arrow is available. We call v_2 the $\top$-vertex and v_3 the $\bot$-vertex of s_1. Ordinary directed edges are represented by curved, solid arrows. We define switching graphs formally in section 7.2 along with other relevant concepts.

We study a number of problems on switching graphs in section 7.3. As it is equivalent to the problem of solving BESs (see below), we are particularly interested in the *v-parity loop problem*, that asks whether a switch setting exists such that the lowest label on every loop reachable from a vertex v is even. For example, in the graph of figure 7.1 there is a loop reachable from v_1 on which the lowest label is odd only if both s_1 and s_2 are set to $\top$. Any other switch setting is a witness for the fact that the answer to the v-parity loop problem is *yes*. We investigate the complexities of several variations of the v-parity loop problem, in order to gain more insight into its nature. If we relax the requirements on the labels, we obtain the polynomial *loop problem* and *1-2-loop problem*. If we relax the total ordering of labels, we obtain the NP-complete *cancellation loop problem*. If we relax the requirement of loops, we obtain the NP-complete *connection problem* and *disconnection problem*. The NP-completeness proofs of the cancellation loop problem and the disconnection problem rely on the property that switches can be *synchronized*. This means that the switch settings of certain groups of switches can be forced to always correspond. We prove that the parity loop problem does not have this property, which is an indication that it belongs to a lower complexity class.

In section 7.4 we give polynomial reductions between the problem of solving BESs and the v-parity loop problem, confirming their equivalence. It is not hard to see that the v-parity loop problem is essentially the same problem as that

of deciding the winner in parity games. As outlined above, this is known to be equivalent to the BES problem via the μ-calculus model-checking problem. Hence, the fact that the v-parity loop problem is equivalent to solving BESs is not surprising, neither is the immediate complexity result of UP $\cap$ co-UP. However, a direct encoding of BESs in switching graphs (and vice versa) is, to our opinion, more intuitive and provides a natural generalization of the encoding of conjunctive and disjunctive BESs in ordinary graphs, as proposed in [59, 60].

Switching graphs are a natural extension to ordinary graphs and form a natural abstract domain for representing concrete problems. As such, they make interesting objects of study by themselves, apart from their being related to model checking. Therefore, we expected switching graphs to be widely studied, but this appears not to be the case. In [98] a slightly different notion of switching graphs is used, essentially omitting switches by unifying nondeterminism with switches. Girard [53] proposes proof nets for linear logic which closely resemble directed switching graphs. In the same context, Danos and Regnier [33] construct proof structures which correspond with undirected switching graphs. Their criterion for a proof structure to be correct (and hence a proof net) is that for every switch setting the resulting graph must be connected and acyclic. Other notions in the literature (like switch graphs [75], bi-directed graphs [38] and skew-symmetric graphs [58]) also deal with choices excluding other options, but are of a different nature than switching graphs.

7.2 Preliminaries

7.2.1 Switching graphs

Definition 7.1. A *directed labelled switching graph* (*DLSG*) is a four-tuple $G = (V, \rightarrow, S, \ell)$ where $(V, \rightarrow, \ell)$ is a vertex-labelled graph with labelling function $\ell : V \rightarrow \mathbb{N}$, and $S \subseteq V \times V \times V$ is a set of *directed switches*.

For any switch $s = (v_1, v_2, v_3) \in S$, we call v_1, v_2 and v_3 the *root*, the $\top$*-vertex* and the $\bot$*-vertex* of s, respectively. A *switch setting* is a function $f : S \rightarrow \mathbb{B}$. For any switch setting f, an f*-path* from a vertex v to a vertex w is a sequence $v_1, v_2, \ldots, v_n$ with $v = v_1$, $w = v_n$ and for all $1 \leq i < n$ it either holds that $v_i \rightarrow v_{i+1}$, or there is a switch $s \in S$ with root v_i and $f(s)$-vertex v_{i+1}. An f*-loop* through v is an f-path from v to v. For any set of switches S and $b \in \mathbb{B}$, we denote by $f[S \mapsto b]$ the switch setting f in which every switch in S is set to b.

For any DLSG $G = (V, \rightarrow, S, \ell)$ and switch setting f on S, the f*-graph* of G is the directed, labelled graph $G_f = (V, \rightarrow', \ell)$, where:

$$\rightarrow' = \rightarrow \cup \{(v, w) \mid \exists u \in V \,.\, ((v, w, u) \in S \wedge f(v, w, u)) \vee ((v, u, w) \in S \wedge \neg f(v, u, w))\}$$

A simple example of a directed switching graph is depicted in figure 7.2(a). Compare this switching graph with an ordinary graph in which both branches of

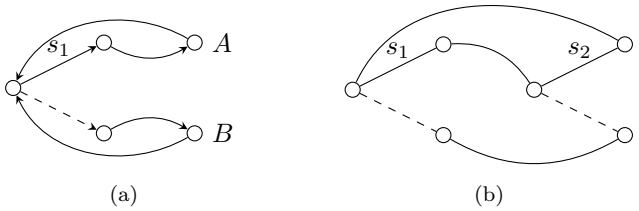


Figure 7.2. A directed (a) and an undirected (b) switching graph.

switch s_1 are ordinary edges. The question whether there is (a switch setting with) a loop through both A and B is answered with *yes* in the graph, and with *no* in the switching graph. If the switching graph is compared with an ordinary graph that contains only one of the branches, the question whether there is (a switch setting with) a loop through A and (a switch setting with) a loop through B is answered with *no* for the graph, and with *yes* for the switching graph.

A switching graph of which the edges are *unordered* pairs, is called *undirected*. An example of an undirected switching graph is given in figure 7.2(b). This graph is not connected only if both switches are set to $\top$. A typical application comes from investigating possible partitions of a set. The elements of the set are the vertices and two elements are equivalent (*i.e.* in the same block of the partition) if they are connected in the graph. Using switches, alternative relations between the individual elements can be represented. Typical questions can be whether a switch setting exists such that the equivalence partition has exactly k elements (or more than k, or exactly an even number, etc.).

7.2.2 The 3SAT problem

We use the well-known 3SAT problem for proving NP-completeness of several problems in section 7.3.

Definition 7.2. A *3CNF formula* is a logical formula of the form

$$\bigwedge_{i\in I}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$$

where I is a finite index set and $l_{i,j}$ are literals, *i.e.* formulae of the form p or $\neg p$ for a proposition letter p taken from a set P. The *3SAT problem* is the question whether a given 3CNF formula is satisfiable.

The famous Cook–Levin theorem states that the SAT problem, *i.e.* satisfiability of any Boolean formula, is NP-complete [29, 88]. By a reduction from SAT to 3SAT, Karp showed that the same holds for the 3SAT problem [84]. In this paper,

we assume some ordering $<$ on the proposition letters P and assume that the literals in each clause of a 3CNF formula are ordered according to $<$, such that the left-most proposition letter in that clause is the smallest, and the right-most is the largest.

7.2.3 Boolean equation systems

Boolean equation systems (BESs) are defined in definition 2.17 and the solution of a BES is already properly defined by definition 2.19. However, the definition can be simplified due to the absence of data and because we can use the fact that $\mu X \,.\, f(X) = f(\bot)$ and, dually, $\nu X \,.\, f(X) = f(\top)$ for any function f over the Booleans. Let $\mathcal{X}$ be a set of proposition variables and the interpretation $[\![\varphi]\!]$ of a proposition formula φ be as defined in definition 2.18.

Definition 7.3. Let $\eta : \mathcal{X} \to \mathbb{B}$ be a proposition environment. The *solution* $[\![\mathcal{E}]\!]\eta$ of a BES $\mathcal{E}$ in the context of η, is a proposition environment that is defined inductively as follows:

$$\begin{aligned} [\![\epsilon]\!]\eta &:= \eta \\ [\![(\sigma X = \varphi)\ \mathcal{E}]\!]\eta &:= [\![\mathcal{E}]\!]\eta[X \mapsto [\![\varphi]\!]([\![\mathcal{E}]\!]\eta[X \mapsto b_\sigma])] \end{aligned}$$

where $b_\mu = \bot$ and $b_\nu = \top$.

For any environment η, BES $\mathcal{E}$ and $X \in \mathsf{bnd}(\mathcal{E})$, the *BES problem* is the problem of determining $([\![\mathcal{E}]\!]\eta)(X)$. In this chapter, we only consider BESs of the following form:

$$\mathcal{E} = (\sigma_1 X_1 = \varphi_1) \ldots (\sigma_n X_n = \varphi_n)$$

for some $n \in \mathbb{N}$. Moreover, $\mathcal{E}$ is:

- in *standard form* if every φ_i is either of the form X_j or $X_j \vee X_k$ or $X_j \wedge X_k$;
- in *2-conjunctive normal form* (2CNF) if every φ_i is of the form $\bigwedge_{p \in I_i} c_{i,p}$ where I_i is an index set and $c_{i,p}$ is either X_j or $X_j \vee X_k$;
- *conjunctive* if no φ_i contains a $\vee$;
- *disjunctive* if no φ_i contains a $\wedge$.

It is known that every BES can be linearly transformed to standard form such that its solution is maintained, modulo renaming of variables [91]. For any conjunctive or disjunctive BES $\mathcal{E}$, the *dependency graph* of $\mathcal{E}$ is a directed, labelled graph $(V, \to, \ell)$ where:

- $V = \{1, \ldots, n\}$;
- $\to = \{(i, j) \mid X_j \text{ occurs in } \varphi_i\}$;
- $\ell(i) = 2 \cdot i + \mathit{sign}_{\sigma_i}$ with $\mathit{sign}_\mu = 1$ and $\mathit{sign}_\nu = 0$, for all $i \in V$.

The following result was obtained in [59] as lemma 3.[1]

Lemma 7.4. *Let η be a proposition environment, $\mathcal{E}$ be a conjunctive BES and G be its dependency graph. For any variable $X_i \in \mathsf{bnd}(\mathcal{E})$ we have that $([\![\mathcal{E}]\!]\eta)(X_i) = \bot$ iff G contains a loop that is reachable from i on which the lowest label is odd.*

For disjunctive BESs the dual result holds, *i.e.* replacing $\bot$ by $\top$ and *odd* by *even*. The loop problem on the dependency graph as described in lemma 7.4 can be solved in polynomial time, which is also shown in [59].

7.3 Switching-graph problems

In this section we investigate the complexity of several problems on a fixed DLSG $G = (V, \rightarrow, S, \ell)$. We define $N := |V|$ and $T := |\rightarrow| + |S|$. For some problems there are straightforward and efficient algorithms. In those cases we only state the complexity and sketch the algorithm for brevity.

7.3.1 Connection problems

Connection problems ask to connect particular vertices in a switching graph via a switch setting. We consider the following connection problems:

- The *v-w-connection problem* is the question whether there is a switch setting f such that there is an f-path in G from a given vertex v to a given vertex w.
- Given a set of pairs of vertices $\{(v_i, w_i) \mid i \in I\}$ for some index set I, the *connection problem* is the question whether there is a switch setting f such that there is an f-path in G from v_i to w_i, for every $i \in I$.

The latter is a generalization of the former.

Theorem 7.5. *The v-w-connection problem is solvable in $\mathcal{O}(T)$ time.*

Proof. A straightforward depth-first search suffices where both branches of each switch are taken as ordinary edges. When a path from v to w is found, no vertex, and hence no switch, occurs on this path more than once. The switch setting f is set accordingly, where those switches not on the path can be set arbitrarily. □

Theorem 7.6. *The connection problem is NP-complete.*

Proof. It is obvious that this problem is in NP and therefore we only show that it is NP-hard. We reduce the 3SAT problem to the connection problem in a straightforward way. Consider the following 3CNF formula:

$$\bigwedge_{i \in I} (l_{i,1} \vee l_{i,2} \vee l_{i,3})$$

[1] In [59] the labelling function is a function ℓ' that maps vertices to $\{\mu, \nu\}$. In the original lemma, G should contain a loop on which the vertex with the lowest *index* has label μ. It is obvious that $\ell'(i) = \mu$ iff $\ell(i)$ is odd and $\ell'(i) = \nu$ iff $\ell(i)$ is even. Also $i < j$ iff $\ell(i) < \ell(j)$. Hence, the result indeed carries over to our version here.

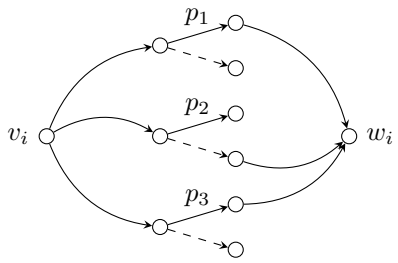


Figure 7.3. Translation of a clause i of the form $p_1 \vee \neg p_2 \vee p_3$ for the connection problem.

for an index set I. We introduce one switch for every proposition letter occurring in the formula. We translate each clause $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ in which proposition letters $p_{i,1}$, $p_{i,2}$ and $p_{i,3}$ occur, as follows. We introduce a start vertex v_i, and an end vertex w_i. The vertex v_i is connected to the roots of the switches $p_{i,1}$, $p_{i,2}$ and $p_{i,3}$. Moreover, if $l_{i,k}$ is a positive literal then the $\top$-vertex of switch $p_{i,k}$ is connected to w_i, otherwise its $\bot$-vertex is connected to w_i. See figure 7.3 for an example.

We can now directly observe that the 3CNF formula is satisfiable iff there is a switch setting f such that for all $i \in I$ there is an f-path from v_i to w_i. □

7.3.2 Disconnection problems

Disconnection problems ask to prevent particular vertices in a switching graph from being connected via a switch setting. We consider the following disconnection problems:

- The *v-w-disconnection problem* is the question whether there is a switch setting f such that there is no f-path in G from a given vertex v to a given vertex w.
- Given a set of pairs of vertices $\{(v_i, w_i) \mid i \in I\}$ for some index set I, the *disconnection problem* is the question whether there is a switch setting f such that there is no f-path in G from v_i to w_i, for every $i \in I$.

The latter is a generalization of the former.

Theorem 7.7. *The v-w-disconnection problem is solvable in $\mathcal{O}(T)$ time.*

Proof. Mark vertices backwards from w in a breadth-first fashion, *i.e.* we start from w and traverse the incoming edges and switch edges in the reverse direction in a breadth-first manner. A vertex u is marked if an outgoing edge leads to a marked vertex, or if both branches of a switch with root u lead to marked vertices. This marking algorithm will terminate in linear time. Answer *yes* if v was not marked and *no* otherwise. For obtaining the switch setting that is a witness in case of a positive answer, set every switch to an unmarked vertex; switches of

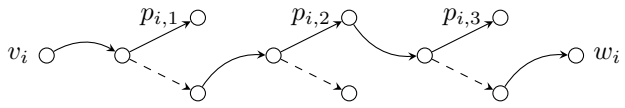


Figure 7.4. Translation of a clause $p_{i,1} \vee \neg p_{i,2} \vee p_{i,3}$ for the disconnection problem.

which the $\top$-vertex and the $\bot$-vertex are both marked or both unmarked, can be set arbitrarily. □

Theorem 7.8. *The disconnection problem is NP-complete.*

Proof. It is again easy to show that this problem is in NP. For showing NP-hardness, we again reduce the 3SAT problem to this problem but the translation is considerably more involved than for the connection problem. We start again with a 3CNF formula:

$$\bigwedge_{i \in I} (l_{i,1} \vee l_{i,2} \vee l_{i,3})$$

for an index set I. We introduce a switch for every *occurrence* of a proposition letter in a clause. For each clause $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ with proposition letters $p_{i,1}$, $p_{i,2}$ and $p_{i,3}$ a start vertex v_i, an end vertex w_i and three switches are generated. The vertex v_i is connected to the root of the first switch. If $l_{i,1}$ is a positive literal then the $\bot$-vertex is connected to the next switch, otherwise the $\top$-vertex is connected to the next switch. In the same way the second switch is connected to the third switch, which is in turn connected in the same way to the vertex w_i. If two or more of the proposition letters in a clause are the same, then either the clause is omitted in case the literals have opposite signs, or only one or two switches are necessary. See figure 7.4 for an example.

Now assume for each proposition letter p that all different switches for p are synchronized, *i.e.* they are all set to $\bot$, or they are all set to $\top$. Then it is straightforward to see that for all $i \in I$ there is *no* path from v_i to w_i iff the 3CNF formula is satisfiable.

We now show how all switches for a proposition letter p can be synchronized. This means that in any switch setting where the switches for p are not all set to the same position (either $\top$ or $\bot$), there will always be a path from a vertex v to a vertex w, where (v, w) is a pair of vertices between which there should be no path according to the disconnection problem. Hence, we ensure that in any switch setting that is a witness for the disconnection problem, all switches for every letter p are set to the same position. We add three pairs of start and end vertices for p, namely (A_p, A'_p), (B_p, B'_p) and (C_p, C'_p), and three switches $s_{p,1}$, $s_{p,2}$ and $s_{p,3}$. We connect the switches and vertices as indicated in figure 7.5. The column of switches under the letter p are all the switches introduced for every occurrence of p in some clause. Observe that the switches in figure 7.5 either

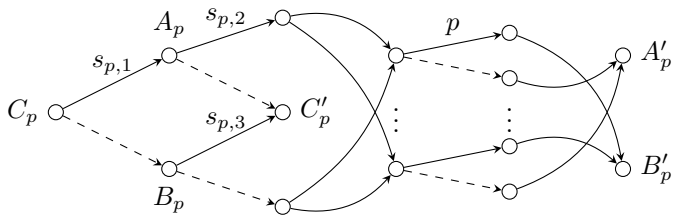


Figure 7.5. Construction for keeping all switches for one proposition letter p synchronized in the translation of the disconnection problem.

must all be set to $\bot$, or they must all be set to $\top$; otherwise there is a path from A_p to A'_p, from B_p to B'_p, or from C_p to C'_p. As the required case distinction is straightforward, we omit it here. □

7.3.3 Loop problems

Loop problems ask to find a switch setting such that loops with a particular property are present or absent. We consider the following loop problems:

- The *loop problem* is the question whether there is a switch setting f such that G contains an f-loop.
- The *v-parity loop problem* is the question whether there is a switch setting f such that every f-loop in G that is reachable from a given vertex v, has an even number as lowest label.
- The *parity loop problem* is the question whether there is a switch setting f such that every f-loop in G has an even number as lowest label.
- The *parity loop-through-v problem* is the question whether there is a switch setting f such that every f-loop through v in G has an even number as lowest label.
- Assuming that $\ell(v) \in \{1, 2\}$ for all $v \in V$, the *1-2-loop problem* is the question whether there is a switch setting f such that every f-loop in G has 2 as lowest label.
- The *cancellation loop problem* is the question whether there is a switch setting f such that every f-loop in G that contains an even label n does not contain the label $n - 1$.

We investigate each of these problems in turn below.

Theorem 7.9. *The loop problem is solvable in $\mathcal{O}(N \cdot T)$ time.*

Proof. This problem is equivalent to the v-v-connection problem for every vertex v. If for some vertex v a switch setting f is found such that there is an f-path from v to itself, then the answer is *yes*. Otherwise, the answer is *no*. □

Regarding all other problems, observe that we are allowed to modify the switching graph G in the following way, without affecting the answer to the problem: for every vertex $v \in V$, if v has no outgoing edges (*i.e.* there is no w such that $v \to w$, and there are no u, w such that $(v, u, w) \in S$) then we add an edge (v, v) to G and we set $\ell(v) := 2$. In this way we obtain a switching graph that is *total*, in the sense that every vertex has an outgoing edge or is the root of a switch. Hence, for the remaining problems of this section we can assume without loss of generality that G is total, which is convenient for the correspondence of the v-parity loop problem with solving parity games (as described below) and with solving BESs (as described in section 7.4).

A parity game is played on a game graph, which is a directed, vertex-labelled graph of which every vertex has an outgoing edge and is labelled by a natural number, called a *priority*. The parity game has two players, called *Even* and *Odd*, and the set of vertices of the graph is partitioned into two blocks: V_{Even} and V_{Odd}. The game proceeds as follows. A token is placed on vertex x and is repeatedly moved along an edge of the graph from its current vertex to an adjacent vertex by one of the players. If the token is currently on a vertex in V_{Even} then it is *Even*'s turn to move the token; if it is on a vertex in V_{Odd} then it is *Odd*'s turn. By repeatedly moving the token, the players construct an infinite path in the graph. Player *Even* wins the game if the lowest priority that occurs infinitely often on the path is even, and *Odd* wins if that priority is odd.

It is not hard to see that the v-parity loop problem on a total switching graph is equivalent to the problem of finding a winning strategy for player *Even* in a parity game, starting from a given vertex x of the game graph. Hence, the v-parity loop problem is in NP $\cap$ co-NP (and UP $\cap$ co-UP) and it is an open question whether a polynomial algorithm exists. We provide direct translations from this problem to that of solving BESs and vice versa in section 7.4.

The parity loop problem is very similar to the v-parity loop problem; in particular no polynomial algorithm is known. We show that it is not possible to keep two switches synchronized in the following sense. Consider a switching graph containing at least two switches. Suppose that whenever two switches are set to opposite positions, the answer to the parity loop problem is *yes*. Then there are no switch settings in which the same switches are set to equal positions and the answer to the parity loop problem on the resulting graph is *no*. For the disconnection problem and cancellation loop problem, the ability to synchronize switches allows us to prove NP-completeness (see theorems 7.8 and 7.12, respectively).

We define $L(f) := \top$ for a switch setting f if all f-loops in G have an even number as lowest label, and $L(f) := \bot$ otherwise. For a given f, $L(f)$ can be calculated in polynomial time by determining the strongly connected components of the f-graph [120].

Lemma 7.10. *Consider two switches s and s'. Suppose that for all switch settings f, $f(s) \neq f(s')$ implies $L(f) = \top$. Then there are no switch settings g and h such that $g(s) = g(s') = \bot$, $h(s) = h(s') = \top$ and $L(g) = L(h) = \bot$.*

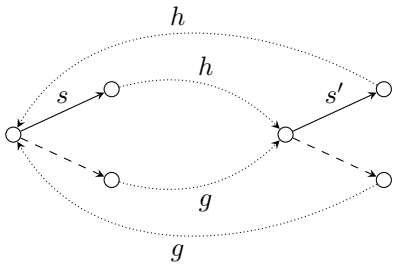


Figure 7.6. The situation in the proof of lemma 7.10. A dotted arrow marked by a switch setting f indicates an f-path in the switching graph.

Proof. Assume the contrary, *i.e.* there are switch settings g and h such that $g(s) = g(s') = \bot$, $h(s) = h(s') = \top$ and $L(g) = L(h) = \bot$. Hence, there is a g-loop on which the lowest label m is odd. Consider one of these loops with minimal m. If this loop does not pass through s, then the $g[s \mapsto \top]$-graph also has a loop with m as lowest number, hence $L(g[s \mapsto \top]) = \bot$ which contradicts the assumption of the lemma. Similarly, if the loop does not pass through s', then $L(g[s' \mapsto \top]) = \bot$ which is a contradiction. Hence, the loop passes through both s and s'. Using the same line of reasoning, there is an h-loop through both s and s' with an odd number n as lowest label, and we take one of these loops with minimal n. See figure 7.6.

Assume without loss of generality that $m \leq n$. There is a vertex labelled by m on the g-path from $s_\bot$ to the root of s' or on the g-path from $s'_\bot$ to the root of s. In both cases we can construct a switch setting f based on g and h such that $f(s) \neq f(s')$ and there is an f-loop that contains the label m. As $m \leq n$, there is no even label on this loop that is smaller than m. Hence, $L(f) = \bot$ which contradicts the assumption of the lemma. All cases have led to a contradiction, which proves the lemma. □

The parity loop-through-v problem is very similar to the parity loop problem, but it allows a single set of switches to be synchronized. In figure 7.7 the switches $s_1, \ldots, s_n$ are synchronized. All non-labelled nodes are assumed to have a label greater than 2. If all switches are set to $\top$, the lowest label of any loop through v is 2. If all switches are set to $\bot$, there is no loop through v. In all other cases, there is a loop through v on which the lowest label is 1.

The 1-2-loop problem is a simplification of the parity loop problem: the possible labels in the graph are restricted to 1 and 2. By this simplification, the problem becomes polynomially decidable as argued below.

Theorem 7.11. *The 1-2-loop problem is solvable in polynomial time.*

Proof. It is not hard to see that this problem is equivalent to the problem of deciding a winning strategy for a parity game, where d, being the number of

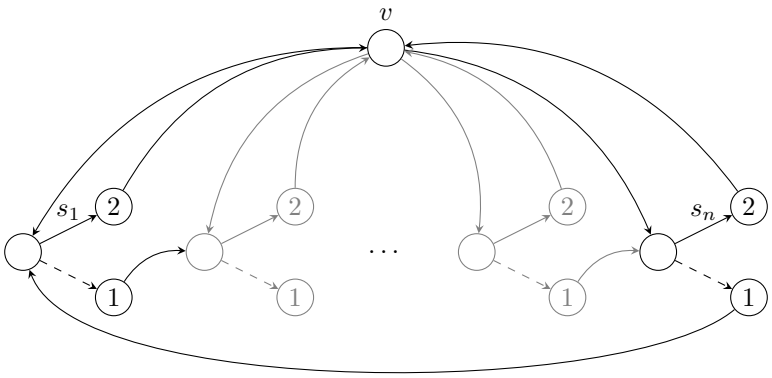


Figure 7.7. Synchronizing switches in the parity loop-through-v problem.

different priorities in the parity-game graph, is equal to 2. For fixed d, parity games are known to be solvable in polynomial time (see *e.g.* [82, 125]). □

The cancellation loop problem is another variation of the parity loop problem. It requires that if a loop contains an even label n then this value is not *cancelled* by the lower odd label $n-1$ on the same loop.

Theorem 7.12. *The cancellation loop problem is NP-complete.*

Proof. It is straightforward to see that this problem is in NP. If a switch setting is given, checking that every loop with even label n does not have a label $n-1$ can be done in polynomial time. In order to show that the problem is NP-hard, we reduce the 3SAT problem to the cancellation loop problem. We start with a 3CNF formula $\bigwedge_{i\in I}(l_{i,1}\vee l_{i,2}\vee l_{i,3})$ where I is an index set that does not contain 0. We generate a switch for every *occurrence* of a proposition letter in a clause. For each clause $l_{i,1}\vee l_{i,2}\vee l_{i,3}$ with proposition letters $p_{i,1}$, $p_{i,2}$ and $p_{i,3}$ three switches are generated. The root of the first switch is labelled with an even number $2i$. All other vertices have label 0. If $l_{i,1}$ is a positive literal then the $\bot$-vertex of the switch for $p_{i,1}$ is connected to the next switch and the $\top$-vertex is connected to the root of the switch for $p_{i,1}$. Otherwise, these connections are interchanged. In the same way the second switch is connected to the third switch, and the third switch to the first switch. If $l_{i,3}$ is positive, the $\bot$-vertex of the third switch is labelled with $2i-1$. Otherwise, its $\top$-vertex is labelled with $2i-1$. If two or more of the proposition letters in a clause are the same, then either the clause is omitted in case the literals have opposite signs, or only one or two switches are necessary. See figure 7.8 for an example.

Now assume for each proposition letter p that all different switches for p are synchronized, in the sense that they are all set to $\bot$, or they are all set to $\top$. Then it is straightforward to see that the formula is satisfiable if and only if there

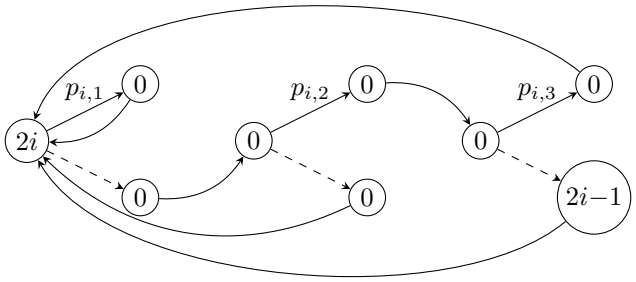


Figure 7.8. Translation of a clause $p_{i,1} \vee \neg p_{i,2} \vee p_{i,3}$ for the cancellation loop problem.

is a switch setting f such that every f-loop containing an even label n does not contain the label $n-1$.

We now show that synchronization of all the switches for a proposition letter p can be enforced. For the cancellation loop problem this means that for any switch setting f in which two switches for p are not set to the same value, the f-graph must contain a loop on which both the labels n and $n-1$ occur for some even number n. To achieve this, we introduce for every proposition letter p three unique even numbers greater than 0: a_p, b_p and c_p. Uniqueness means that these numbers are neither equal to any a_q, b_q or c_q for any other proposition letter q nor are they equal to $2i$ for any $i \in I$. For any p we add two switches $s_{p,1}$ and $s_{p,2}$ and we connect the switches as indicated in figure 7.9. The column of switches under the letter p are all the different switches introduced for every occurrence of p in some clause. We label certain vertices by a_p, b_p, c_p, a_p-1, b_p-1 and c_p-1, as indicated.

Note that the number of elements that is added to the switching graph in this way, is linear in the number of proposition letters that occur in the 3CNF formula. Now, the switches for p (including $s_{p,1}$ and $s_{p,2}$) either must all be set to $\bot$ or they must all be set to $\top$, otherwise there is a loop containing labels n and $n-1$ for some even number n. As the required case distinction is straightforward, we omit it here. □

This concludes our investigation of a number of loop problems on switching graphs. In particular, we focused on parity loop problems and studied several instances of this type of problem. Some of them turned out to be polynomially decidable while others were shown to be NP-complete. By its equivalence to solving parity games, the v-parity loop problem was shown to be in NP ∩ co-NP. We now prove its equivalence to the problem of solving BESs by direct reductions.

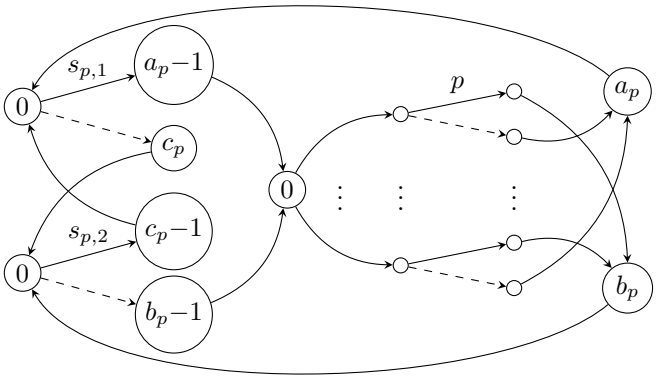


Figure 7.9. Construction for keeping all switches for a proposition letter p synchronized in the translation of the cancellation loop problem.

7.4 Equivalence to the BES problem

In this section we provide reductions from the BES problem to the v-parity loop problem and vice versa. For the proofs we need to be able to remove every disjunction from a BES in 2CNF by selecting one of the disjuncts and eliminating the other one, resulting in a conjunctive BES. The operation resembles that of obtaining the f-graph G_f from a DLSG G and a switch setting f.

Definition 7.13. Given the following BES in 2CNF:

$$\mathcal{E} = (\sigma_1 X_1 = \bigwedge_{j \in I_1} c_{1,j}) \ldots (\sigma_n X_n = \bigwedge_{j \in I_n} c_{n,j})$$

for clauses $c_{i,j}$ and index sets I_i. Let f be a function such that for every $1 \leq i \leq n$ and $j \in I_i$: $f(i, c_{i,j}) = X_k$ if $c_{i,j} = X_k$, and either $f(i, c_{i,j}) = X_k$ or $f(i, c_{i,j}) = X_l$ if $c_{i,j} = X_k \vee X_l$. We call f a *selection function*. Then the *f-BES* $\mathcal{E}_f$ is the following conjunctive BES:

$$\mathcal{E}_f = (\sigma_1 X_1 = \bigwedge_{j \in I_1} f(1, c_{1,j})) \ldots (\sigma_n X_n = \bigwedge_{j \in I_n} f(n, c_{n,j})).$$

By a lemma from [91], the solutions of an f-BES are stronger than those of the original BES.

Lemma 7.14. *Given a BES $\mathcal{E}$ and environment η. For all selection functions f and variables X of $\mathcal{E}$ we have $([\![\mathcal{E}_f]\!]\eta)(X)$ implies $([\![\mathcal{E}]\!]\eta)(X)$.*

Proof. By lemma 3.16 of [91]. □

The following result is obtained in [91] as proposition 3.36.

Proposition 7.15. *For every BES $\mathcal{E}$ and environment θ there is a selection function f such that $[\![\mathcal{E}_f]\!]\theta = [\![\mathcal{E}]\!]\theta$.* □

7.4.1 Reduction from BES to v-parity loop

Our translation is similar to the one for conjunctive and disjunctive BESs given in [59], which produces a directed labelled graph without switches. We fix a BES $\mathcal{E} = (\sigma_1 X_1 = \varphi_1) \dots (\sigma_n X_n = \varphi_n)$ for some $n \in \mathbb{N}$. Without loss of generality, we assume that $\mathcal{E}$ is in standard form.

Definition 7.16. The *DLSG corresponding to* $\mathcal{E}$ is a DLSG $(V, \to, S, \ell)$ where $V = \{1, \dots, n\}$ and:

- $\to = \{(i, j) \mid \varphi_i \text{ contains } X_j \text{ but not } \vee\}$;
- $S = \{(i, j, k) \mid \varphi_i = X_j \vee X_k\}$;
- $\ell(i) = 2i + sign_{\sigma_i}$ for all $i \in V$.

Observe that the size of this DLSG is polynomial in the size of $\mathcal{E}$.

Theorem 7.17. *Let G be the DLSG corresponding to $\mathcal{E}$, η be an arbitrary environment and $1 \leq i \leq n$. Then the BES-problem on $\mathcal{E}$, η and X_i is equivalent to the i-parity loop problem on G.*

Proof. We show that $([\![\mathcal{E}]\!]\eta)(X_i) = \top$ iff there is a switch setting of G such that on every loop reachable from vertex i the lowest label is even.

($\Rightarrow$) We prove this part by contraposition. Assume that for all switch settings of G there is a loop reachable from i on which the lowest label is odd. We have to show that $([\![\mathcal{E}]\!]\eta)(X_i) = \bot$. By proposition 7.15 there is a selection function f such that $[\![\mathcal{E}_f]\!]\eta = [\![\mathcal{E}]\!]\eta$. Take this f and construct a switch setting $\hat{f}$ of G as follows, for any $(i, j, k) \in S$: $\hat{f}(i, j, k) = \top$ if $f(i, X_j \vee X_k) = X_j$, and $\hat{f}(i, j, k) = \bot$ if $f(i, X_j \vee X_k) = X_k$. Note that $\varphi_i = X_j \vee X_k$ by construction of G. Observe that the $\hat{f}$-graph $G_{\hat{f}}$ is the dependency graph of the f-BES $\mathcal{E}_f$. Then by lemma 7.4 we have $([\![\mathcal{E}_f]\!]\eta)(X_i) = \bot$, hence $([\![\mathcal{E}]\!]\eta)(X_i) = \bot$.

($\Leftarrow$) Assume the existence of a switch setting $\hat{f}$ such that on every $\hat{f}$-loop reachable from i the lowest label is even. Construct a selection function f as follows. For any $(i, j) \in \to$ define $f(i, X_j) = X_j$ and for any $(i, j, k) \in S$ define:

$$f(i, X_j \vee X_k) = \begin{cases} X_j & \text{if } \hat{f}(i, j, k) = \top \\ X_k & \text{if } \hat{f}(i, j, k) = \bot. \end{cases}$$

Note that φ_i contains the required clauses – X_j or $X_j \vee X_k$ – by construction of G. Observe that the $\hat{f}$-graph $G_{\hat{f}}$ is the dependency graph of the f-BES $\mathcal{E}_f$ and that the lowest label on every loop in $G_{\hat{f}}$ reachable from i is even. Then by lemma 7.4 we have $([\![\mathcal{E}_f]\!]\eta)(X_i) = \top$. Hence, by lemma 7.14, $([\![\mathcal{E}]\!]\eta)(X_i) = \top$. □

7.4.2 Reduction from v-parity loop to BES

We fix a DLSG $G = (V, \rightarrow, S, \ell)$. Without loss of generality, we assume that $V = \{1, \ldots, n\}$ for some $n \geq 1$, $\ell(i) \leq \ell(i+1)$ for all $1 \leq i < n$, and G is total as described in section 7.3.3.

Definition 7.18. The *BES corresponding to* G is the following BES:

$$\mathcal{E} = (\sigma_1 X_1 = \varphi_1) \ldots (\sigma_n X_n = \varphi_n)$$

where, for all $1 \leq i \leq n$:

- $\sigma_i = \mu$ if $\ell(i)$ is odd and $\sigma_i = \nu$ otherwise;
- $\varphi_i = \bigwedge\{X_j \mid i \rightarrow j\} \ \wedge \ \bigwedge\{(X_j \vee X_k) \mid (i, j, k) \in S\}$.

Observe that this BES is in 2CNF and that its size is polynomial in the size of G.

Theorem 7.19. *Let $v \in V$, $\mathcal{E}$ be the BES corresponding to G and η be an arbitrary environment. Then the v-parity loop problem on G is equivalent to the BES problem on $\mathcal{E}$, η and X_v.*

Proof. We have to show that $([\![\mathcal{E}]\!]\eta)(X_v) = \top$ iff there is a switch setting of G such that on every loop reachable from vertex v the lowest label is even. The proof is very similar to that of theorem 7.17 and is therefore omitted. □

7.5 Conclusions

We have introduced switching graphs as a natural extension to ordinary graphs, and have determined the complexity of several problems on such graphs. The v-w-connection, v-w-disconnection, loop and 1-2-loop problems are solvable in polynomial time. The cancellation loop, connection and disconnection problems are NP-complete by reductions from the 3SAT problem. Finally, the v-parity loop problem is in NP ∩ co-NP, both by the already known result on parity games and by the direct translations to/from the BES problem presented here.

Of course, by its equivalence to the BES and μ-calculus model-checking problems, the v-parity loop problem is particularly interesting. Studying this problem and related problems in the context of switching graphs may help in answering the long-open question whether all of these problems are polynomially decidable.

Switching graphs are also interesting by themselves as a natural formalism for representing various kinds of combinatorial problems. Below we list a number of switching graph problems of which the complexity is not known. Note that problems having a polynomial solution on ordinary graphs, can become hard in the setting of switching graphs. Some open problems regarding undirected switching graphs are:

1. Is there a switch setting f such that the f-graph contains an Euler tour (*i.e.* a path through the f-graph that traverses each edge exactly once)?

2. Is there a switch setting f such that the f-graph is planar?
3. Find a switch setting f such that the f-graph has a minimal number of connected vertices (to some vertex v), or is k-connected (*i.e.* after removing k edges the f-graph is still connected).

Some open problems regarding directed switching graphs are:

1. Is there a switch setting f such that the f-graph is a strongly connected component?
2. Is there a switch setting f such that the f-graph can be topologically sorted? This is equivalent to finding a switch setting f such that the f-graph does not contain non-trivial connected components.
3. Is there a switch setting f such that the f-graph is a tree, has a network flow of sufficient capacity or has a path between two vertices shorter than a given k?
4. Given two vertices v and w, is there a switch setting f such that an f-loop through v and w exists?

Of course many more problems can be formulated. Resolving such problems may also help in finding the answer to the open question mentioned above.

Chapter 8

Conclusions

8.1 Discussion

We summarize and discuss our contributions to the field of verification. For more detailed conclusions we refer to the concluding sections of the relevant chapters.

As mentioned in the introduction, the greatest challenge when applying verification methods in practice is the complexity of the systems that are involved. This is reflected in the state explosion problem: model checking and equivalence checking become infeasible for very large state spaces. In practice, the maximal size that an automated checker can still handle properly, often depends on memory rather than time efficiency: when a program requires too much memory, modern computers start to employ slower secondary storage by which the computation effectively grinds to a halt. Apart from this, model checking and equivalence checking for infinite-state systems is undecidable and cannot be achieved using traditional techniques that are based on explicit enumeration of the state space.

The goal of the work in this thesis is to improve verification methods for large and infinite-state concurrent systems. In chapters 3 and 4 we focused on memory-efficient algorithms for equivalence checking on finite-state systems. For checking language and trace equivalence, we proposed five new determinization algorithms in chapter 3 that aimed to be more memory efficient on average than normal subset construction. In the experiments, two algorithms showed a significant gain in memory efficiency which allowed us to determinize an automaton that we could not determinize with the standard algorithm on the host compute, due to lack of memory. This stresses the importance of memory efficiency as we explained above. Three of the determinization algorithms relied heavily on the simulation preorder. For computing this preorder, we implemented the most time-efficient of the most space-efficient simulation algorithms known to this date. We found and repaired errors in this algorithm in chapter 4. We proved that our corrections did not affect the space and time complexities of the algorithm.

In chapters 3 and 4 we limited ourselves to concrete and finite-state systems. We lifted both of these restrictions in chapter 5, where we considered the more general problem of deciding several branching-time equivalences on infinite-state systems with silent steps. We translated this problem to that of solving a parameterized Boolean equation system (PBES), thereby obtaining a finite and symbolic representation of the equivalence-checking problem on infinite-state systems. As the model-checking problem for the first-order μ-calculus could already be encoded in PBESs, this showed that PBESs form a generic framework for studying various verification problems on infinite-state systems. This makes PBESs a valuable formalism in its own right and justifies efforts to gain more insight into their underlying dynamics and come up with novel solution or simplification techniques.

The fact that solving a PBES is generally undecidable, makes this a challenging task. Our instantiation technique of chapter 6 confirms the intuition that this undecidability is mainly due to the involved data sorts. It provides a way to simplify PBESs so that other techniques can be more readily applied. Moreover, if all involved data sorts are finite, solving a PBES reduces to solving a BES. In the presence of countably infinite data sorts, only finite subsets of these may have to be considered in order to determine the solution for a particular equation. In that case a BES can be obtained by adopting an on-the-fly instantiation strategy.

Solving a BES is decidable; it is in NP and co-NP. In an attempt to find a polynomial-time algorithm for this problem, we introduced the novel notion of switching graphs in chapter 7. We gave direct, polynomial reductions from the BES problem to the v-parity loop problem on switching graphs, and vice versa. We also studied several switching-graph problems related to the v-parity loop problem; some were shown to be in P while others were shown to be NP-complete. We believe switching graphs to be an intuitive and interesting formalism by itself that can be used for the representation of many combinatorial problems, not limited to model checking and equivalence checking.

8.2 Future work

For future directions of research, we envisage the following possibilities:

- *Extension of the simulation algorithm to weak and branching variants.* The idea of [51] to represent the simulation problem as a generalized coarsest partition problem could be further exploited in order to obtain space-efficient algorithms for the weak and branching simulation preorders. Here, it may be possible to borrow techniques from the partitioning algorithm for deciding branching bisimilarity [67].
- *Translation of other equivalences into PBESs.* Based on our translation for the (bi)simulation equivalences, equivalence checking for other branching-time equivalences may be translated into PBESs in a similar manner. Also, translation for linear-time equivalences may be obtained by exploiting the fact that these coincide with bisimilarity for deterministic processes. For instance, the

PBES for strong trace equivalence would then be similar to the one for strong bisimilarity, except for the fact that the PBES parameters now represent *sets* of LPE states, rather than single LPE states.

- *Expansion of the library of PBES manipulations.* As PBESs have proven to be versatile verification vehicles, the development of techniques for simplifying and solving them should be continued and intensified. Examples of open questions concern the exploitation of confluence, and the abstraction from large data domains in general and dense domains (real time) in particular. For this, the definition of a more intuitive semantics or of useful equivalences on PBESs may turn out to be essential. On a practical level, the usability of PBESs can be greatly enhanced by the generation of counterexamples for model checking and equivalence checking.
- *Extension of PBESs to quantitative domains.* In order to evaluate the performance of probabilistic and stochastic systems, the PBES framework could be extended to allow for quantitative model checking, *viz.* over the interval of reals $[0, 1]$ rather than the Booleans. For this, a quantitative μ-calculus is already available, while a proper symbolic representation of probabilistic processes and Markov chains in the spirit of LPEs should be investigated.
- *Expanding our knowledge of switching graphs.* More problems on switching graphs should be investigated in order to gain more insight into these structures. There the aim could be to find a polynomial algorithm for the parity loop problem, but switching-graph problems can equally be studied as interesting combinatorial problems in their own right. The translation of other problems into switching graphs could be a fruitful path for obtaining new insights into either domain.

Bibliography

[1] M. Abadi and L. Lamport (1995): *Conjoining Specifications.* ACM Transactions on Programming Languages and Systems 17(3), pp. 507–534. (3)

[2] H.R. Andersen (1994): *Model checking and boolean graphs.* Theoretical Computer Science 126(1), pp. 3–30. (81)

[3] J.C.M. Baeten, J.A. Bergstra and J.W. Klop (1987): *Decidability of bisimulation equivalence for processes generating context-free languages.* In A.J. Nijman, J.W. de Bakker and P.C. Treleaven, editors: Proc. Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages, LNCS 259, pp. 94–111. Springer. (3)

[4] J.C.M. Baeten and W.P. Weijland (1990): *Process Algebra.* Cambridge University Press. (15)

[5] C. Baier and J.P. Katoen (2008): *Principles of Model Checking.* The MIT Press. (7)

[6] T. Basten (1996): *Branching bisimilarity is an equivalence indeed!* Information Processing Letters 58(3), pp. 141–147. (83)

[7] H. Bekič (1984): *Definable operations in general algebras, and the theory of automata and flowcharts.* In C.B. Jones, editor: Programming Languages and Their Definition, LNCS 177, pp. 30–55. Springer. (108)

[8] J.A. Bergstra and J.W. Klop (1984): *Process algebra for synchronous communication.* Information and Control 60(1-3), pp. 109–137. (2, 15)

[9] J.A. Bergstra and J.W. Klop (1985): *Algebra of Communicating Processes with Abstraction.* Theoretical Computer Science 37(1), pp. 77–121. (2, 15)

[10] B. Bloom, S. Istrail and A.R. Meyer (1995): *Bisimulation Can't Be Traced.* Journal of the ACM 42(1), pp. 232–268. (55)

[11] B. Bloom and R. Paige (1995): *Transformational design and implementation of a new efficient solution to the ready simulation problem.* Science of Computer Programming 24(3), pp. 189–220. (3, 37, 55)

[12] B. Boigelot and P. Godefroid (1999): *Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs.* Formal Methods in System Design 14(3), pp. 237–255. (3)

[13] J. Bradfield and C. Stirling (1992): *Local model checking for infinite state spaces.* Theoretical Computer Science 96(1), pp. 157–174. (3)

[14] J. Bradfield and C. Stirling (2001): *Modal logics and mu-calculi.* In J.A. Bergstra, A. Ponse and S.A. Smolka, editors: Handbook of Process Algebra, pp. 293–332. Elsevier. (23)

[15] J.C. Bradfield (1992): *Verifying Temporal Properties of Systems.* Birkhäuser. (118, 119)

[16] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe (1984): *A Theory of Communicating Sequential Processes.* Journal of the ACM 31(3), pp. 560–599. (3)

[17] J.A. Brzozowski (1963): *Canonical Regular Expressions and Minimal State Graphs for Definite Events.* In J. Fox, editor: Proc. of the Symposium on Mathematical Theory of Automata, MRI Symposia Series 12, pp. 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn. (30)

[18] J.R. Burch, E.M. Clarke, D.L. Dill, J. Hwang and K.L. McMillan (1992): *Symbolic model checking:* 10^{20} *states and beyond.* Information and Computation 98(2), pp. 142–171. (3)

[19] D. Bustan and O. Grumberg (2003): *Simulation-based minimization.* ACM Transactions on Computational Logic 4(2), pp. 181–206. (55, 56)

[20] S. Christensen, Y. Hirshfeld and F. Moller (1993): *Decomposability, Decidability and Axiomatisability for Bisimulation Equivalence on Basic Parallel Processes.* In: Proc. 8th Annual IEEE Symposium on Logic in Computer Science (LICS 1993), pp. 386–396. IEEE Computer Society Press. (3)

[21] S. Christensen, H. Hüttel and C. Stirling (1992): *Bisimulation Equivalence is Decidable for all Context-Free Processes.* In R. Cleaveland, editor: Proc. 3rd International Conference on Concurrency Theory (CONCUR 1992), LNCS 630, pp. 138–147. Springer. (3)

[22] E.M. Clarke (2008): *The Birth of Model Checking.* In O. Grumberg and H. Veith, editors: 25 Years of Model Checking, LNCS 5000, pp. 1–26. Springer. (7)

[23] E.M. Clarke and E.A. Emerson (1981): *Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic.* In D. Kozen, editor: Proc. Workshop on Logics of Programs, LNCS 131, pp. 52–71. Springer. (2, 7, 21)

[24] E.M. Clarke, E.A. Emerson and A.P. Sistla (1986): *Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems 8(2), pp. 244–263. (2)

[25] E.M. Clarke, R. Enders, T. Filkorn and S. Jha (1996): *Exploiting Symmetry in Temporal Logic Model Checking.* Formal Methods in System Design 9(1–2), pp. 77–104. (3)

[26] E.M. Clarke, O. Grumberg and D.A. Peled (1999): *Model Checking.* The MIT Press. (7, 15)

[27] R. Cleaveland, J. Parrow and B. Steffen (1989): *The Concurrency Workbench.* In J. Sifakis, editor: Proc. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407, pp. 24–37. Springer. (2, 3)

[28] R. Cleaveland and B. Steffen (1991): *Computing behavioural relations, logically.* In J. Leach Albert, B. Monien and M. Rodríguez Artalejo, editors: Proc. 18th International Colloquium on Automata, Languages and Programming (ICALP 1991), LNCS 510, pp. 127–138. Springer. (81)

[29] S.A. Cook (1971): *The Complexity of Theorem Proving Procedures.* In: Proc. 3rd ACM Symposium on Theory of Computing, pp. 151–158. ACM. (128)

[30] C. Courcoubetis, M.Y. Vardi, P. Wolper and M. Yannakakis (1990): *Memory Efficient Algorithms for the Verification of Temporal Properties.* In E.M. Clarke and R.P. Kurshan, editors: Proc. 2nd Conference on Computer Aided Verification (CAV 1990), LNCS 531, pp. 233–242. Springer. (55)

[31] D. Dams, R. Gerth and O. Grumberg (1993): *Generation of Reduced Models for Checking Fragments of CTL.* In C. Courcoubetis, editor: Proc. 5th Conference on Computer Aided Verification (CAV 1993), LNCS 697, pp. 479–490. Springer. (55)

[32] D. Dams, R. Gerth and O. Grumberg (1997): *Abstract Interpretation of Reactive Systems.* ACM Transactions on Programming Languages and Systems 19(2), pp. 253–291. (3)

[33] V. Danos and L. Regnier (1989): *The Structure of Multiplicatives.* Archive for Mathematical Logic 28(3), pp. 181–203. (127)

[34] R. De Nicola and F.W. Vaandrager (1990): *Action versus State based Logics for Transition Systems.* In I. Guessarian, editor: Semantics of Systems of Concurrent Processes, LNCS 469, pp. 407–419. Springer. (15, 21)

[35] E.W. Dijkstra (1976): *A Discipline of Programming.* Prentice Hall. (1)

[36] D.L. Dill, A.J. Hu and H. Wong-Toi (1992): *Checking for Language Inclusion Using Simulation Preorders.* In K.G. Larsen and A. Skou, editors: Proc. 3rd Conference on Computer Aided Verification (CAV 1991), LNCS 575, pp. 255–265. Springer. (37)

[37] J. Dingel and T. Filkorn (1995): *Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving.* In P. Wolper, editor: Proc. 7th International Conference on Computer Aided Verification (CAV 1995), LNCS 939, pp. 54–69. Springer. (3)

[38] J. Edmonds and E.L. Johnson (2001): *Matching: A Well-Solved Class of Integer Linear Programs.* In M. Jünger, G. Reinelt and G. Rinaldi, editors: Combinatorial Optimization, LNCS 2570, pp. 27–30. Springer. (127)

[39] E.A. Emerson (2008): *The Beginning of Model Checking: A Personal Perspective.* In O. Grumberg and H. Veith, editors: 25 Years of Model Checking, LNCS 5000, pp. 27–45. Springer. (7)

[40] E.A. Emerson, C.S. Jutla and A.P. Sistla (1993): *On Model-Checking for Fragments of Mu-Calculus.* In C. Courcoubetis, editor: Proc. 5th Conference on Computer Aided Verification (CAV 1993), LNCS 697, pp. 385–396. Springer. (125)

[41] E.A. Emerson and C.L. Lei (1986): *Efficient Model Checking in Fragments of the Propositional μ-Calculus.* In: Proc. 1st Annual Symposium on Logic in Computer Science (LICS 1986), pp. 267–278. IEEE Computer Society Press. (23)

[42] E.A. Emerson and K.S. Namjoshi (1998): *On Model Checking for Nondeterministic Infinite State Systems.* In: Proc. 13th IEEE Symposium on Logic in Computer Science (LICS 1998), pp. 70–80. IEEE Press. (3)

[43] E.A. Emerson and A.P. Sistla (1984): *Deciding Full Branching Time Logic.* Information and Control 61(3), pp. 175–201. (21)

[44] E.A. Emerson and A.P. Sistla (1996): *Symmetry and model checking.* Formal Methods in System Design 9(1–2), pp. 105–131. (3)

[45] J. Engelfriet (1985): *Determinacy $\Rightarrow$ (observation equivalence = trace equivalence).* Theoretical Computer Science 36(1), pp. 21–25. (20)

[46] S. Evangelista and J.F. Pradat-Peyre (2005): *Memory Efficient State Space Storage in Explicit Software Model Checking.* In P. Godefroid, editor: Proc. 12th International SPIN Workshop on Model Checking Software, LNCS 3639, pp. 43–57. Springer. (55)

[47] J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighireanu (1996): *CADP - A Protocol Validation and Verification Toolbox.* In R. Alur and T.A. Henzinger, editors: Proc. 8th International Conference on Computer Aided Verification, (CAV 1996), LNCS 1102, pp. 437–440. Springer. (2, 3)

[48] R.W. Floyd and R. Beigel (1994): *The Language of Machines.* Freeman. (37)

[49] W. Fokkink, J. Pang and J. van de Pol (2006): *Cones and foci: A mechanical framework for protocol verification.* Formal Methods in System Design 29(1), pp. 1–31. (82)

[50] H. Garavel, R. Mateescu, F. Lang and W. Serwe (2007): *CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes.* In W. Damm and H. Hermanns, editors: Proc. 19th International Conference on Computer Aided Verification (CAV 2007), LNCS 4590, pp. 158–163. Springer. (2, 3)

[51] R. Gentilini, C. Piazza and A. Policriti (2003): *From Bisimulation to Simulation: Coarsest Partition Problems.* Journal of Automated Reasoning 31(1), pp. 73–103. (44, 55, 56, 57, 59, 60, 61, 62, 63, 65, 69, 80, 144)

[52] R. Gentilini, C. Piazza and A. Policriti (2003): *From Bisimulation to Simulation: Coarsest Partition Problems.* RR 12-2003, Dep. of Computer Science, University of Udine, Italy. (69)

[53] J.Y. Girard (1987): *Linear Logic.* Theoretical Computer Science 50, pp. 1–102. (127)

[54] R.J. van Glabbeek (1993): *The Linear Time — Branching Time Spectrum II.* In E. Best, editor: Proc. 4th International Conference on Concurrency Theory (CONCUR 1993), LNCS 715, pp. 66–81. Springer. (20, 100)

[55] R.J. van Glabbeek (2001): *The Linear Time — Branching Time Spectrum I. The semantics of concrete, sequential processes.* In J.A. Bergstra, A. Ponse and S.A. Smolka, editors: Handbook of Process Algebra, pp. 3–99. Elsevier. (3, 20, 51)

[56] R.J. van Glabbeek and W.P. Weijland (1996): *Branching time and abstraction in bisimulation semantics.* Journal of the ACM 43(3), pp. 555–600. (19, 81)

[57] P. Godefroid (1991): *Using Partial Orders to Improve Automatic Verification Methods.* In E.M. Clarke and R.P. Kurshan, editors: Proc. 2nd International Conference on Computer Aided Verification (CAV 1990), LNCS 531, pp. 176–185. Springer. (3)

[58] A.V. Goldberg and A.V. Karzanov (1996): *Path Problems in Skew-Symmetric Graphs.* Combinatorica 16(3), pp. 353–382. (127)

[59] J.F. Groote and M. Keinänen (2004): *Solving Disjunctive/Conjunctive Boolean Equation Systems with Alternating Fixed Points.* In K. Jensen and A. Podelski, editors: Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), LNCS 2988, pp. 436–450. Springer. (125, 127, 130, 139)

[60] J.F. Groote and M. Keinänen (2005): *A Sub-quadratic Algorithm for Conjunctive and Disjunctive Boolean Equation Systems.* In D.V. Hung and M. Wirsing, editors: Proc. 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC 2005), LNCS 3722, pp. 532–545. Springer. (125, 127)

[61] J.F. Groote and R. Mateescu (1999): *Verification of temporal properties of processes in a setting with data.* In A.M. Haeberer, editor: Proc. 7th International Conference on Algebraic Methodology and Software Technology (AMAST 1998), LNCS 1548, pp. 74–90. Springer. (21, 23, 24)

[62] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko and M. van Weerdenburg (2007): *The Formal Specification Language mCRL2.* In E. Brinksma, D. Harel, A. Mader, P. Stevens and R. Wieringa, editors: Methods for Modelling Software Systems (MMOSS), Dagstuhl Seminar Proceedings 06351. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). (15)

[63] J.F. Groote, A. Mathijssen, M.A. Reniers, Y.S. Usenko and M. van Weerdenburg (2008): *Analysis of Distributed Systems with mCRL2.* In M. Alexander and W. Gardner, editors: Process Algebra for Parallel and Distributed Processing, pp. 99–128. CRC Press. (15)

[64] J.F. Groote and J. van de Pol (1996): *A bounded retransmission protocol for large data packets.* In M. Wirsing and M. Nivat, editors: Proc. 5th International Conference on Algebraic Methodology and Software Technology (AMAST 1996), LNCS 1101, pp. 536–550. Springer. (82)

[65] J.F. Groote and A. Ponse (1995): *The Syntax and Semantics of μCRL.* In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, editors: Proc. 1st Workshop on the Algebra of Communicating Processes (ACP 1994), Workshops in Computing. Springer. (15)

[66] J.F. Groote and M.A. Reniers (2001): *Algebraic process verification.* In J.A. Bergstra, A. Ponse and S.A. Smolka, editors: Handbook of Process Algebra, pp. 1151–1208. Elsevier. (15)

[67] J.F. Groote and F.W. Vaandrager (1990): *An efficient algorithm for branching bisimulation and stuttering equivalence.* In M.S. Paterson, editor: Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP 1990), LNCS 443, pp. 626–638. Springer. (81, 122, 144)

[68] J.F. Groote and F.W. Vaandrager (1992): *Structured Operational Semantics and Bisimulation as a Congruence.* Information and Computation 100(2), pp. 202–260. (55)

[69] J.F. Groote and T.A.C. Willemse (2005): *Model-checking processes with data.* Science of Computer Programming 56(3), pp. 251–273. (23, 24, 28, 119)

[70] J.F. Groote and T.A.C. Willemse (2005): *Parameterised boolean equation systems.* Theoretical Computer Science 343(3), pp. 332–369. (24, 27, 28)

[71] M. Hennessy and R. Milner (1980): *On Observing Nondeterminism and Concurrency.* In J.W. de Bakker and J. van Leeuwen, editors: Proc. 7th International Colloquium on Automata, Languages and Programming (ICALP 1980), LNCS 85, pp. 299–309. Springer. (21)

[72] M.R. Henzinger, T.A. Henzinger and P.W. Kopke (1995): *Computing Simulations on Finite and Infinite Graphs.* In: Proc. 36th Annual Symposium on Foundations of Computer Science (FOCS 1995), pp. 453–462. IEEE Computer Society Press. (3, 37, 55)

[73] C.A.R. Hoare (1980): *A Model for Communicating Sequential Processes.* In R.M. McKeag and A.M. Macnaghten, editors: On the Construction of Programs, pp. 229–254. Cambridge University Press. (3)

[74] C.A.R. Hoare (1985): *Communicating Sequential Processes.* Prentice Hall. (2)

[75] J.M. Hochstein and K. Weihe (2004): *Edge-Disjoint Routing in Plane Switch Graphs in Linear Time.* Journal of the ACM 51(4), pp. 636–670. (127)

[76] G.J. Holzmann (1988): *An Improved Protocol Reachability Analysis Technique.* Software Practice and Experience 18(2), pp. 137–161. (55)

[77] G.J. Holzmann (1990): *Design and Validation of Computer Protocols.* Prentice-Hall. (2)

[78] G.J. Holzmann (2003): *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley. (2)

[79] J.E. Hopcroft (1971): *An $n \log n$ Algorithm for Minimizing States in a Finite Automaton*. In Z. Kohavi, editor: Theory of Machines and Computations, pp. 189–196. Academic Press. (30)

[80] J.E. Hopcroft and J.D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley. (30)

[81] M. Jurdziński (1998): *Deciding the Winner in Parity Games Is in UP ∩ co-UP*. Information Processing Letters 68(3), pp. 119–124. (125)

[82] M. Jurdziński (2000): *Small Progress Measures for Solving Parity Games*. In H. Reichel and S. Tison, editors: Proc. 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2000), LNCS 1770, pp. 290–301. Springer. (125, 136)

[83] P.C. Kanellakis and S.A. Smolka (1990): *CCS Expressions, Finite State Processes, and Three Problems of Equivalence*. Information and Computation 86(1), pp. 43–68. (3)

[84] R.M. Karp (1972): *Reducibility among combinatorial problems*. In R.E. Miller and J.W. Thatcher, editors: Complexity of Computer Computations, pp. 85–103. Plenum Press. (128)

[85] D. Kozen (1983): *Results on the propositional μ-calculus*. Theoretical Computer Science 27(1), pp. 333–354. (21)

[86] H. Kwak, J. Choi, I. Lee and A. Philippou (1998): *Symbolic weak bisimulation for value-passing calculi*. Technical Report MS-CIS-98-22, University of Pennsylvania. (82)

[87] M. Leuschel and T. Massart (1999): *Infinite State Model Checking by Abstract Interpretation and Program Specialisation*. In A. Bossi, editor: Selected Papers 9th International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 1999), LNCS 1817, pp. 62–81. Springer. (3)

[88] L. Levin (1973): *Universal search problems*. Problemy Peredachi Informatsii 9(3), pp. 265–266. In Russian. (128)

[89] H. Lin (1996): *Symbolic transition graph with assignment*. In U. Montanari and V. Sassone, editors: Proc. 7th International Conference on Concurrency Theory (CONCUR 1996), LNCS 1119, pp. 50–65. Springer. (82, 97)

[90] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani and S. Bensalem (1995): *Property Preserving Abstractions for the Verification of Concurrent Systems*. Formal Methods in System Design 6(1), pp. 11–44. (55)

[91] A. Mader (1997): *Verification of Modal Properties using Boolean Equation Systems.* Ph.D. thesis, Technische Universität München. (24, 27, 81, 102, 118, 119, 125, 129, 138, 139)

[92] A. Mader (1997): *Verification of Modal Properties Using Infinite Boolean Equation Systems.* Technical Report CSI-R9727, University of Nijmegen. (101)

[93] R. Mateescu (1998): *Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus.* In: Proc. 2nd International Workshop on Verification, Model Checking and Abstract Interpretation (VMCAI 1998). (102, 117, 123)

[94] R. Mateescu (2003): *A generic on-the-fly solver for alternation-free boolean equation systems.* In H. Garavel and J. Hatcliff, editors: Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), LNCS 2619, pp. 81–96. Springer. (81, 83)

[95] R. Mateescu and D. Thivolle (2008): *A Model Checking Language for Concurrent Value-Passing Systems.* In J. Cuéllar, T.S.E. Maibaum and K. Sere, editors: Proc. 15th International Symposium on Formal Methods (FM 2008), LNCS 5014, pp. 148–164. Springer. (102, 123)

[96] K.L. McMillan (1993): *Symbolic Model Checking.* Kluwer Academic Publishers. (2, 3)

[97] K.L. McMillan (1999): *Verification of Infinite State Systems by Compositional Model Checking.* In L. Pierre and T. Kropf, editors: Proc. 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 1999), LNCS 1703, pp. 219–234. Springer. (3)

[98] C. Meinel (1989): *Switching graphs and their complexity.* In A. Kreczmar and G. Mirkowska, editors: Proc. 14th International Symposium on Mathematical Foundations of Computer Science (MFCS 1989), LNCS 379, pp. 350–359. Springer. (127)

[99] R. Milner (1980): *A Calculus of Communicating Systems.* Springer. (2, 3, 18)

[100] R. Milner, J. Parrow and D. Walker (1992): *A calculus of mobile processes (Part I/II).* Information and Computation 100(1), pp. 1–77. (2, 100)

[101] J. Misra and K.M. Chandy (1981): *Proofs of Networks of Processes.* IEEE Transactions on Software Engineering 7(4), pp. 417–426. (3)

[102] E.F. Moore (1956): *Gedanken-experiments on sequential machines.* In C.E. Shannon and J. McCarthy, editors: Automata Studies, pp. 129–153. Princeton University Press. (3)

[103] S. Orzan and T.A.C. Willemse (2008): *Invariants for Parameterised Boolean Equation Systems.* In F. van Breugel and M. Chechik, editors: Proc. 19th International Conference on Concurrency Theory (CONCUR 2008), LNCS 5201, pp. 187–202. Springer. (24, 28, 95)

[104] R. Paige and R.E. Tarjan (1987): *Three Partition Refinement Algorithms.* SIAM Journal on Computing 16(6), pp. 973–989. (3, 81)

[105] D.M.R. Park (1981): *Concurrency and Automata on Infinite Sequences.* In P. Deussen, editor: Proc. 5th GI-Conference on Theoretical Computer Science, LNCS 104, pp. 167–183. Springer. (3, 18, 20)

[106] D. Peled (1996): *Combining Partial Order Reductions with On-the-Fly Model-Checking.* Formal Methods in System Design 8(1), pp. 39–64. (3)

[107] C.A. Petri (1962): *Kommunikation mit Automaten.* Ph.D. thesis, University of Bonn. (2)

[108] C.A. Petri and W. Reisig (2008): *Petri net.* Scholarpedia 3(4), 6477. (2)

[109] A. Pnueli (1977): *The temporal logic of programs.* In: Proc. 18th IEEE Symposium on the Foundations of Computer Science (FOCS 1977), pp. 46–57. IEEE Computer Society Press. (2, 21)

[110] J.P. Queille and J. Sifakis (1982): *Specification and Verification of Concurrent Systems in CESAR.* In M. Dezani-Ciancaglini and U. Montanari, editors: Proc. 5th International Symposium in Programming, LNCS 137, pp. 337–351. Springer. (2, 7)

[111] F. Ranzato and F. Tapparo (2007): *A new efficient simulation equivalence algorithm.* In: Proc. 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007), pp. 171–180. IEEE Computer Society Press. (55, 56)

[112] A.W. Roscoe (1994): *Model-checking CSP.* In: A classical mind: essays in honour of C. A. R. Hoare, pp. 353–378. Prentice Hall. (3)

[113] S. Schewe (2008): *An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games.* In M. Kaminski and S. Martini, editors: Proc. 22nd International Workshop on Computer Science Logic (CSL 2008), LNCS 5213, pp. 369–384. Springer. (121)

[114] C. Stirling (1995): *Local Model Checking Games.* In I. Lee and S.A. Smolka, editors: Proc. 6th International Conference on Concurrency Theory (CONCUR 1995), LNCS 962, pp. 1–11. Springer. (125)

[115] L.J. Stockmeyer and A.R. Meyer (1973): *Word problems requiring exponential time.* In: Proc. 5th Annual ACM Symposium on Theory of Computing (STOC 1973), pp. 1–9. ACM Press, New York. (37, 55)

[116] B. Stroustrup (1986): *The C++ Programming Language.* Addison-Wesley. (44)

[117] K. Sutner (2003): *The size of power automata.* Theoretical Computer Science 295(1–3), pp. 371–386. (46)

[118] D. Tabakov and M.Y. Vardi (2005): *Experimental Evaluation of Classical Automata Constructions.* In G. Sutcliffe and A. Voronkov, editors: Proc. 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005), LNCS 3835, pp. 396–411. Springer. (30, 47, 49)

[119] L. Tan and R. Cleaveland (2001): *Simulation Revisited.* In T. Margaria and W. Yi, editors: Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), LNCS 2031, pp. 480–495. Springer. (55)

[120] R.E. Tarjan (1972): *Depth-first search and linear graph algorithms.* SIAM Journal on Computing 1(2), pp. 146–160. (134)

[121] A. Tarski (1955): *A lattice-theoretical fixpoint theorem and its applications.* Pacific Journal of Mathematics 5(2), pp. 285–309. (10)

[122] A.M. Turing (1936): *On Computable Numbers, with an Application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society 2(42), pp. 230–265. (1)

[123] Y.S. Usenko (2002): *Linearization in μCRL.* Ph.D. thesis, Technische Universiteit Eindhoven. (15)

[124] A. Valmari (1992): *A Stubborn Attack on State Explosion.* Formal Methods in System Design 1(4), pp. 297–322. (3)

[125] J. Vöge and M. Jurdziński (2000): *A Discrete Strategy Improvement Algorithm for Solving Parity Games.* In E.A. Emerson and A.P. Sistla, editors: Proc. 12th Conference on Computer Aided Verification (CAV 2000), LNCS 1855, pp. 202–215. Springer. (125, 136)

[126] B.W. Watson (1995): *Taxonomies and Toolkits of Regular Language Algorithms.* Ph.D. thesis, Technische Universiteit Eindhoven. (30)

[127] S. Wolfram (1984): *Computation theory of cellular automata.* Communications in Mathematical Physics 96(1), pp. 15–57. (46)

[128] S. Wolfram (2002): *A New Kind of Science.* Wolfram Media, Inc. (44)

[129] D. Zhang and R. Cleaveland (2005): *Fast generic model-checking for data-based systems.* In F. Wang, editor: Proc. 25th Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005), LNCS 3731, pp. 83–97. Springer. (24)

Summary

A concurrent system is a system in which a number of processes operate in parallel and interact with each other to perform certain tasks or computations. In this context, the purpose of verification methods is to prove that a concurrent system behaves correctly. A well established verification technique is model checking, where a model of a system is constructed and it is checked whether this model satisfies or violates certain properties. Another useful verification technique is called equivalence checking, where two models at different levels of abstraction (usually a specification and an implementation) are constructed and it is checked whether these models are behaviourally equivalent.

An infamous problem when applying verification in practice is the state space explosion problem: models can become extremely large, whereby automated verification quickly breaks down. A number of techniques have been proposed in the literature to address this problem. As we found existing techniques to be inadequate for the verification of certain large and infinite-state concurrent systems, we propose improved verification methods to deal with those types of systems in this dissertation. In particular, we address the following topics that are related to model checking and equivalence checking.

- For checking the language and trace preorders and equivalences on finite-state models, we present five new determinization algorithms based on one of the standard algorithms from the literature. The aim is to reduce the memory consumption, which is often the bottleneck when applying automatic verification techniques in practice. By experimental evaluation we show that implementations of two of our algorithms consume significantly less memory on examples describing patterns in a cellular automaton, which makes the determinization of larger NFAs feasible.

- A space-efficient algorithm for deciding simulation preorder and equivalence on finite-state models is corrected. The algorithm was found in the literature and is the most space-efficient algorithm for this problem known to this date. Several flaws in the supporting fixpoint theory are identified and the algorithm is repaired, without degrading its time and space complexities.

- The problems of deciding strong, weak and branching (bi)simulation equiv-

alences on infinite-state models are translated to the problem of solving sequences of fixpoint equations. Such sequences, called parameterized Boolean equation systems (PBES), could already be used to encode the problem of checking a first-order μ-calculus formula on a possibly infinite-state model. By the proposed translations, it is shown that PBESs are more versatile: they are generic vehicles for finitely representing various kinds of verification problems on infinite-state models. One of the translations is proven to be correct and illustrated by examples.

- A transformation on PBESs, called instantiation, is proposed that aims to eliminate data domains from a PBES. This can allow for other solution techniques to be applied more readily. In particular, if all data domains are finite, the resulting equation system is a BES for which computing the solution is decidable. Instantiation can also be applied in an on-the-fly manner which allows for local model checking. The transformation is proven to be correct and has been implemented. It is applied to several examples, both manually and automatically.

- The concept of switching graphs is introduced as an intuitive formalism on which various interesting problems can be defined. The complexity classes of several switching-graph problems are investigated. In particular, it is shown that solving BESs is equivalent to the parity-loop problem on switching graphs. Some variations of this problem are shown to be polynomial-time decidable, while other variations are shown to be NP-complete. Arguably, switching graphs allow for more intuitive reasoning about abstract problems like μ-calculus model checking and solving BESs. Thereby, they may aid in finding an answer to the long open question of whether these problems are decidable in polynomial time.

Curriculum Vitae

Sebastiaan Cornelis Willem (Bas) Ploeger was born on 9 March 1982 in Gouda and raised in Oostvoorne, The Netherlands. After completing his secondary education in 1999, Bas moved to Eindhoven to study Computer Science at the Eindhoven University of Technology. He received his Master of Science degree with honours in August 2005. His Master's thesis was titled "Analysis of Concurrent State Machines in Embedded Copier Software".

Immediately after his graduation Bas started as a PhD candidate at the same university. His project was supervised by Professor Jan Friso Groote and supported by a grant from the Netherlands Organisation for Scientific Research (NWO). Within this project, Bas initially focused on the visualization of transition systems that model system behaviour, but he soon became interested in formal verification techniques. His scientific contributions to these areas were presented at various international conferences and published in their proceedings. As one of the core developers, he also contributed to the publicly available mCRL2 tool set for the analysis of system behaviour by writing code, documentation and web pages.

In October 2009, Bas starts working as a Software Engineer in Test at Google in Zürich, Switzerland.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a*

Toolkit. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electri-

cal Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20