Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Sascha Klüppelholz
geboren am 21.10.1976 in Haan

Gutachter:

Prof. Dr. rer. nat. Christel Baier
Technische Universität Dresden

Prof. Dr. Farhad Arbab
Universität Leiden, Niederlande

Tag der Verteidigung: 19. März 2012

Abstract

Information and communication systems enter an increasing number of areas of daily lives. Our reliance and dependence on the functioning of such systems is rapidly growing together with the costs and the impact of system failures. At the same time the complexity of hardware and software systems extends to new limits as modern hardware architectures become more and more parallel, dynamic and heterogenous. These trends demand for a closer integration of formal methods and system engineering to show the correctness of complex systems within the design phase of large projects.

The goal of this thesis is to introduce a formal holistic approach for modeling, analysis and synthesis of parallel systems that potentially addresses complex system behavior at any layer of the hardware/software stack. Due to the complexity of modern hardware and software systems, we aim to have a hierarchical modeling framework that allows to specify the behavior of a parallel system at various levels of abstraction and that facilitates designing complex systems in an iterative refinement procedure, in which more detailed behavior is added successively to the system description. In this context, the major challenge is to provide modeling formalisms that are expressive enough to address all of the above issues and are at the same time amenable to the application of formal methods for proving that the system behavior conforms to its specification. Our focus in this thesis will be on model checking which yields an algorithmic approach to show the correctness of a system with respect to its specification. For the modeling, we are interested in specification formalisms that allow to apply formal verification techniques such that the underlying model checking problems are still decidable within reasonable time and space bounds.

The presented work relies on an exogenous modeling approach that allows a clear separation of coordination and computation and provides an operational semantic model where formal methods such as model checking are well suited and applicable. The channel-based exogenous coordination language Reo is used as modeling formalism as it supports hierarchical modeling in an iterative top-down refinement procedure. It facilitates reusability, exchangeability, and heterogeneity of components and forms the basis to apply formal verification methods. At the same time Reo has a clear formal semantics based on automata, which serve as foundation to apply formal methods such as model checking.

In this thesis new modeling languages are presented that allow specifying complex systems in terms of Reo and automata models which yield the basis for a holistic approach on modeling, verification and synthesis of parallel systems. The second main contribution of this thesis consists of tailored branching-time and alternating time temporal logics as well as corresponding model checking algorithms. The thesis includes results on the theoretical complexity of the underlying model checking problems as well as practical results. For the latter the presented approach has been implemented in the symbolic verification tool set Vereofy. The implementation within Vereofy and evaluation of the branching-time and alternating-time model checker comprise the third main contribution of this thesis.

Acknowledgements

First and foremost I want to thank my advisor *Christel Baier* for giving me the opportunity to be a member of her group for many years. I am very thankful for all her encouragement and contributions of ideas, time and funding which made my PhD experience a wonderful, productive and stimulating time. It is only due to her geniality, ingenuity and guidance that I was able to successfully complete my PhD. I want to thank her from the bottom of my heart.

I also want to thank my parents and their new marriage partners, *Monika* and *Peter Pietzsch*, *Klaus* and *Urszula Klüppelholz*, my grandparents *Katharina* and *Paul Wahler* and the whole family for proving me any imaginable kind of support. I owe special gratitude to my caring mother *Monika* who worked hard and provided continuous and unconditional support.

I wish to thank my girlfriend *Claudia Engler* for her love, devotion and unstinting support after all those busy weekends and evenings. I also want to thank the Engler family for warmly welcoming me. I would also like to express my warm thanks to my extremely-long-standing friends together with their family members who have always been there when I needed them the most, as there are *Bettina* and *Christian Wunner*, *Christoph Münch*, *David Calaminus*, *Frank Birkenbeil*, *Nora Timmerbeil* and *Ricarda Essrich* as well as long and outstanding friends from Bonn like *Andreas Niers* and his family, *Arthur Glogowski*, *Jan Tietjen* and family, *Kristina* and *Sebastian Lützeler-Gerhards*, *Kristina Judith*, and many many more. I also want to thank all the wonderful people from Villigst and especially *André Koch* who also finished his PhD recently. Thank you for all your love and support and thank you *Ricarda* and *Bettina* for all the proofreading of my thesis.

It was a pleasure to share doctoral studies and life in Dresden and Bonn with wonderful people like *Tobias Blechmann*, *Frank Ciesinski*, *Marcus Größer* and *Joachim Klein*, who shared this exiting adventure starting from the early days in Bonn. In the final phase of this thesis I very much appreciated the help of *Joachim Klein* who constantly gave very helpful comments, reviewed many things with me at the whiteboard, implemented some nifty helper scripts for efficiently running the case studies and kept poking me to work thoroughly on the remaining parts.

I was also very lucky to meet and get to know some delightful and knowledgeable colleagues in Dresden like *Manuela Berg*, *Clemens Dubslaff*, *Markus Daum*, *Steffen Märcker*, *Michael Ummels*, and all the others who joined (and left) our group in the recent years and of course all the other nice people I met at the institute for theoretical computer science and the faculty of computer science. A very special thanks I want to express to *Kerstin Achtruth* for just everything that she did for us. I would also like to thank the professors forming the committee of the doctoral examination procedure. *Prof. Uwe Assmann* and *Prof. Hermann Härtig* for forming and leading the committee, *Prof. Christel Baier* for supervising and reviewing the thesis, *Prof. Farhad Arbab* for all his effort he spent on reviewing the thesis as well as for the traveling, and *Prof. Heiko Vogler* for giving very helpful comments in preparation of the defense.

In the context of a collaboration within the former EU project CREDO and the bilateral DFG/NWO project SYANCO I had the opportunity to travel a lot, see different places and meet wonderful people. Working with them has been a great pleasure and beyond the project scope I still feel grateful to people like *Bernhard Aichernig*, *Farhad Arbab*, *Frank de Boer*, *David Clarke*, *David Costa*, *Immo Grabe*, *Andreas Griesmayer*, *Mahdi Jaghouri*, *Einar Broch Johnsen*, *Stephanie Kemper*, *Jetty Kleijn*, *Christian Krause*, *Marcel Kyas*, *Wolfgang Leister*, *Xuedong Liang*, *Bjarte Østvold*, *Olaf Owe*, *Jose Proenca*, *Willem-Paul de Roever*, *Jan Rutten*, *Alfons Salden*, *Rudolph Schlatte*, *Andries Stam*, *Martin Steffen*, *Simon Tschirner*, and *Wang Yi*.

Unfortunately, some of my closest people cannot witness the end of this experience. I feel very certain that my late father *Klaus Klüppelholz* and his wife *Urszula Klüppelholz*, who died just six days after him, would be very proud seeing this. We all miss *Tobias Blechmann* our dear colleague and friend who passed away much too early. Some of the work presented in this thesis has been created in collaboration with him within endless working sessions on the whiteboard. Just to mention two of his contributions: some important language features and the name of our modeling language CARML were originally proposed by him and the bisimulation engine he created is still part of our model checking tool set Vereofy. We should have pushed this much further – together. Lastly, and most importantly, I want to thank a very special person. My grandfather *Paul Wahler* was and is one of my major role models from whom I learned a lot for life. I dedicate this thesis to him.

Contents

CHAPTER 1 Introduction **1**

CHAPTER 2 Constraint automata **9**

2.1. Introduction 9

2.2. Symbolic representation of constraint automata 16

2.3. Data types and polarity 18

CHAPTER 3 Modeling formalism and languages **23**

3.1. The coordination language Reo 23

3.2. The modeling languages RSL and CARML 34

3.2.1. Reo scripting language (RSL) 35

3.2.2. Constraint automata reactive module language (CARML) 46

CHAPTER 4 Alternating-time stream logic **53**

4.1. Constraint automata as multi-player games 53

4.2. Syntax and semantics 58

4.3. Branching-time and alternating-time model checking 66

4.3.1. BTSL model checking 67

4.3.2. ASL model checking 71

4.4. Complexity of the ASL model-checking problem 88

4.5. ASL with fairness 95

CHAPTER 5 Implementation **99**

5.1. The model-checking tool Vereofy 99

5.2. BDD-based model checking 102

5.3. Tool evaluation 109

5.3.1. Tic-Tac-Toe 110

5.3.2. Other game examples 130

5.3.3. A dining philosophers example 130

5.3.4. Other protocol examples 138
5.3.5. The ASK communication system 138

CHAPTER 6 Conclusion **147**

Bibliography 151

List of Figures

2.1. Constraint automaton for a component or connector 11

2.2. Product automaton of two FIFO channels with capacity one and $\mathsf{Data} = \{0\}$ 14

2.3. Hiding variants applied to the constraint automaton from Example 2.1.9 . . 15

2.4. Constraint automaton for a component or connector (IOC-syntax) 17

2.5. Constraint automaton with memory for a component or connector 18

3.1. Basic set of Reo channels 24

3.2. Constraint automata for Reo channels 27

3.3. Reo nodes: a) sink node, b) source node, and c) mixed Reo node 28

3.4. Replacement of a Reo node N by a component C_N 29

3.5. Constraint automata for a standard Reo node N 29

3.6. Constraint automata for a Reo route node N 30

3.7. Reo join for two FIFO channels 30

3.8. Automata for two FIFO1 channels, the standard Reo node and their product 31

3.9. Reo component connector for data-dependent routing 32

3.10. A writer and two readers connected by a data-dependent router 32

3.11. Constraint automaton (after hiding the internals) for a data-dependent router 33

3.12. Schema of an RSL circuit 36

3.13. Two RSL scripts composing the same connector 40

3.14. An RSL main program to compose the Reo network of Example 3.1.5 . . . 41

3.15. Schema for RSL scripts with dynamic reconfiguration 42

3.16. A dynamic router used in a simple Reo network 43

3.17. Schema of a CARML module 47

3.18. CARML module for a FIFO1 channel with initial value *init_value* 50

3.19. Induced constraint automaton for the CARML module of Figure 3.18 . . . 51

3.20. CARML module for a stack with capacity cap 51

4.1. Constraint automaton and a finite-memory N-strategy 57

4.2. Two components connected via a FIFO channel 62

4.3. Network (left) and constraint automaton (right) 63
4.4. Network with three components and its constraint automaton 64
4.5. Constraint automaton $\mathcal{A}$ and NFA $\mathcal{Z}$ with $\mathcal{L}(\mathcal{Z}) = (c(A) = 1)^*$ 70
4.6. Product automaton $\mathcal{A} \otimes \mathcal{Z}$ 71
4.7. Schema for the treatment of $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ 77
4.8. Schema for the treatment of $\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi$ 85

5.1. Main elements of the Vereofy tool set 100
5.2. Component model for a two-player game 109
5.3. RSL declarations for the Tic-Tac-Toe game 111
5.4. CARML module for the rules of the Tic-Tac-Toe game 111
5.5. CARML module for the game arena of the Tic-Tac-Toe game 112
5.6. RSL circuit composing the Tic-Tac-Toe game 113
5.7. Executing Vereofy for the Tic-Tac-Toe game and formula Φ_4 116
5.8. Tic-Tac-Toe draw configuration as reached in the witness path in Figure 5.7 117
5.9. BDD nodes during fixpoint computation for Tic-Tac-Toe 3×3 (gcd=default) 124
5.10. BDD nodes during fixpoint computation for Tic-Tac-Toe 4×4 (gcd=default) 125
5.11. BDD nodes during fixpoint computation for Tic-Tac-Toe 4×4 (gcd=inf) . . 126
5.12. BDD nodes during fixpoint computation for Tic-Tac-Toe 4×4 (gcd=0) . . . 127
5.13. BDD nodes during fixpoint computation for Tic-Tac-Toe 3×3 (gcd=0) . . . 128
5.14. Time needed for fixpoint computation in Tic-Tac-Toe (gcd=inf) 129
5.15. Component model for the dining philosophers 131
5.16. Constraint automata for the dining philosophers components 131
5.17. Alternative component model for the dining philosophers without deadlock 132
5.18. Composition time in seconds for the dining philosophers protocol 134
5.19. BDD nodes: precomputing reachable states of the dining philosophers . . . 135
5.20. ASK core overview 139
5.21. Abstraction levels in the ASK model and components contained in the level 140
5.22. Reo network for the ASK core 141
5.23. Reo network for the scheduler process 142
5.24. Time and memory usage for the scheduler process 144
5.25. Time and memory usage for the reception process 146

1 Introduction

In our everyday life we observe that information and communication systems enter an increasing number of areas. Our reliance and dependence on the functioning of such systems are rapidly growing, and so are the costs and the impact of system failures. At the same, time the complexity of hardware and software systems extends to new limits. In the last decades the performance of hardware systems was mainly improved by integrating more transistors on an integrated circuit and increasing the processor frequency.

This process has reached its physical limitations for current CMOS-based technology. Continuing Moore's Law in the future demands for new designs and technologies [BC11]. Current hardware multi- and many-core architectures are providing more and more resources available for parallel and asynchronous computing. Instead of synchronizing on a single global clock whose frequency is increased to gain more performance, modern architectures tend to be more and more asynchronous. Given the fact that local hardware components still work in a synchronized way it is rather natural to design systems following GALS paradigm [Cha84] (*globally asynchronous and (only) locally synchronous*). Better performance can now potentially be achieved by increasing the number of cores. Together with the growing number of cores, future hardware becomes more flexible with respect to dynamic reconfiguration and runtime adaptations. Notable improvements can be achieved when changing the communication from classical bus- or crossbar-based interconnects to network-on-chip (NoC) communication which is more eligible for flexible adaptations at runtime. Hence, reconfigurable hardware has the chance of achieving a better performance at a lower energy consumption. The latter has become a very important issue not only for mobile devices and large server farms. Another important aspect results from the development of promising alternative material such as silicon nanowires. Current hardware architectures are heterogenous only in the sense that they may for example use graphics processing units (GPUs) for computations instead of the CPU. In the future we expect more heterogenous hardware systems that may contain special-purpose hardware based on alternative materials. This makes future hardware platforms even more flexible and provides another opportunity for improving the overall computing performance. Despite the fact that the performance of future hardware increases by providing more flexible parallel and heterogenous hardware resources, the overall performance of a system will not benefit from these developments automatically. We see already today that our static sequential programs do not profit in full from CPUs with four or even more cores. An integrated approach is needed that covers the hardware, the operating system, programming languages and compilers, libraries and middle-ware as well as aspects of the application layer.

These developments show that future hardware and hence the software will be even more complex and error-prone. It will become more difficult to build systems that behave correctly with respect to their specification. This demands for a closer integration of systems engineering and formal methods, i.e., applied mathematics for modeling and analyzing information and communication systems, to show the correctness of complex systems within the early design phase of large projects.

The goal of this thesis is to present a holistic approach for the modeling, analysis and synthesis of complex current and future hardware and software systems. The proposed approach should be applicable at any layer of the hardware/software stack (and even across layers) and support heterogeneity. Moreover, the proposed approach should be compositional and support hierarchical modeling of parallel systems by means of a stepwise refinement procedure that facilitates reusability, interchangeability, and system maintenance. To achieve this goal, we plan to inspect well-known concepts, methods, and techniques from the literature and combine them into a holistic framework. To obtain efficiency in this framework, this surely involves some nontrivial adaptions and modification. The main challenge in developing such a holistic approach is to find a reasonable balance between different aspects. We demand for modeling languages that are comprehensive and expressive and at the same time simple, elegant, intuitive and easy to learn. The underlying semantics of the modeling formalisms needs to capture all the relevant aspects and details of the system. Also the formal methods that are applied to show the correctness of a parallel system model should be powerful enough to address all relevant aspects, e.g., functionality, costs, time, utility and various types of system properties from the safety-liveness spectrum. On the other hand, the proposed combination of languages, their operational semantics, and formal methods should still be eligible for systematic verification and synthesis supported by tools, and the algorithmic problems should still be efficiently decidable.

In this respect, component-based modeling is a promising approach facilitating the vision of "correctness by construction" [Hal02, HC02, Hal05, GS05]. In the component-based modeling approach the basic principle is (as in software engineering) to divide a complex system into logical entities. These entities are called *components*. In principle, components are abstract entities that have a well-defined interface consisting of (input and output) ports which they use for communication and interaction. At each layer of the hardware/software stack components can stand for different parts of the system. E.g., at the hardware level, the components of a component-based system model could stand for different parts of hardware such as the CPU or the memory. At the programming language layer, components could model the behavior of objects or method calls. Hence, the component-based view can in principle be applied at any layer or even across layers of the hardware/software stack. Component-based modeling also allows and facilitates a systematic hierarchical system design, as modeling complex systems can be done in an iterative top-down refinement procedure. Starting from a rather abstract, purely structural component-model, one can successively provide details of the concrete implementation for each of the components. On the other hand, starting with fully specified (and verified) components, one can compositionally build complex systems in a bottom-up manner.

A multitude of component-based modeling formalisms have been proposed in the literature. They are based on different formalisms to describe the components themselves and a variety of paradigms for interaction and communication of the components. One of the most simple types of synchronization relies on the principle of *synchronized parallelism* that has been suggested by Milner [Mil83], and is used, e.g., to model the communication between transition systems [Arn94]. Famous examples of communication paradigms are based on *shared variables* which were originally proposed by Dijkstra [Dij65] in 1965 or *handshake communication* as it is the case in process algebras such as CSP (*Communicating Sequential Processes*) [Hoa78, Hoa85], CCS (*Calculus of Communicating Systems*) [Mil89, Mil99], ACP (*Algebra of Communicating Processes*) [BK85], π-Calculus [MPW92], etc., where parallel processes communicate over synchronous and asynchronous channels. While process algebras are very expressive, their view is action-based and focuses on algebraic reasoning about equivalence, simulation, bisimulation, or trace-based refinement relations. Coordination languages have a different focus which is on the *glue code* that coordinates and orchestrates the components. Coordination languages aim for a clear separation between coordination and computation. Moreover, coordination languages potentially allow for the application of iterative and hierarchical refinement procedures as needed when modeling complex parallel software and hardware systems and they align perfectly with refinement procedures known from the software engineering and hence do facilitate reusability and exchangeability of components (and the orchestrating structures as well).

A variety of coordination models and languages have been introduced. In general they can be categorized by the following two aspects. Coordination languages are either *control-oriented* or *data-oriented*. Whereas control-oriented languages have a data-abstract perspective and tend to center around the control-flow between components (only), the data-oriented languages tend to manage the dataflow of communicating components. The second aspect is *exogenous* vs. *endogenous* coordination. Endogenous coordination languages require to incorporate coordination primitives within the code that specifies the behavior of the components. A typical example is the data-oriented and endogenous coordination language *Linda* [GCCC85] where components are described in a computational language extended by operators to store or retrieve data objects from a global tuple space. A cleaner separation of computation and coordination is provided by exogenous coordination models where the components do not need to be aware of each other. Instead, they are controlled from "outside" via their interfaces.

Several approaches for exogenous coordination have been suggested, e.g., an aspect-oriented approach [KLM$^+$97, CSZ04], a variant of the π-calculus with anonymous peer-to-peer communication [GSAdBB05, GS07], glue code provided by scripting languages [SLN01, ALSN01], and formalism that rely on the construction of component connectors, such as interaction systems [GS03, MCM08] or the declarative channel-based language Reo [Arb04]. In this thesis we will use the exogenous coordination language Reo for specifying the communication, synchronization and interaction of components. Reo is based on the ideal worker ideal manager (IWIM) model [Arb96], extending classical models such as Kahn networks and dataflow languages. Reo allows any kind of peer-to-peer communication, hierarchical modeling and provides support for dynamic reconfiguration. Reo is more expressive than, e.g., dataflow models, workflow models, and Petri nets [Pet77, Pet81]. In particular, standard CCS-like handshaking via synchronous channels can be modeled in Reo.

The interface behavior of the components as well as the behavior of the basic Reo channels are described in terms of automata models. Constraint automata [BSAR06] serve for both, as an operational semantics for Reo which supports composition of channels and at the same time constraint automata are used to describe the interface behavior of components. This approach is fully compositional and yields the basis for analyzing the components in isolation (which carry out the computation), the orchestrating Reo network in isolation (which is responsible for communication and interaction between components), or the entire system model. The underlying operational semantics for components and their interactions is now based on an automata model that yields the perfect basis for applying formal methods to show the correctness of a system with respect to its specification.

In general there are different ways to validate the correctness of a designed system. With *testing* and *simulation* one can find many types of failures and errors within existing systems and system models respectively. However, testing and simulation come with the drawback of incompleteness as neither of the two methods formally proves the absence of system errors. In contrast *theorem proving* and *model checking* are complementary verification methods that are complete in the sense that properties which have been proven to hold, do hold for all possible system behavior. Whereas theorem proving is a semi-automatic deductive technique which requires expert knowledge to operate, model checking is a fully automatic and algorithmic approach. Theorem proving, model checking, testing and simulation form a useful toolbox of complementary methods that can be used to validate the correctness of future hardware and software systems. In this thesis we will consider model checking techniques which target towards a closer integration in the system design phase of complex parallel (and distributed) systems.

The term model checking was introduced [CE81] in 1981 and about the same time a similar technique was suggested in [QS82]. In model checking the operational behavior of a system is given in terms of an automata-based model and we check the specification, which is usually given in terms of a temporal logic formula. In the early days, model checking was mostly applied for verifying hardware circuits and protocols where exhaustive testing is not feasible due to the huge number of test cases. Later, model checking has also been successfully applied to software systems, e.g. [Hol93]. An introduction to model checking can be found, e.g., in [CK96, CS00, CW96, Mer01, Wol95]. Extensive knowledge about model checking can be found in the early books [Hol90, McM93, Kur94], the 1999 book *Model Checking* [CGP99], which serves as a standard reference, and other books [HR99, Sch04a, BBF$^+$01] as well as the book [BK08] from 2008.

The major challenge in the classical model checking approach arises from the fact that the corresponding algorithms work on the representation of the entire systems, whose size becomes huge and reaches (or even extends) the memory limitations of modern computer systems very quickly. This challenge is known as the *state-explosion problem* and demands for additional techniques to handle gigantically large state spaces, see e.g. [AK86]. Many techniques have been proposed to overcome the state-space-explosion problem such as partial order reduction [Val92, God96, Pel93], symmetry reduction [CEFJ96], slicing for concurrent programs [Wei84, Kri03], (learning-based) assume-guarantee reasoning [Pnu85, HQR98, CCF$^+$10], bounded and unbounded SAT-based model checking as in [BCCZ99] and [WBCG00], iterative (and counterexample-guided) abstraction-refinement algorithms

[GS97, CGJ+00], as well as symbolic model checking based on *binary decision diagrams* (BDDs) [BCM+92].

Many instances of model checking have been studied in the literature. They vary in the way the behavior of the system is specified, the underlying operational model and the type of properties that can be checked. One of the most prominent candidates to formally describe the behavior of systems are variants of transition systems [Kel76], which do serve as a semantic model for formalisms such as process algebras [Hoa85, Mil89, Mil99, BPe01], Petri nets [Pet77, Pet81], UML [RJB99], statecharts [Har87], finite-state Mealy and Moore automata [Kro99], program graphs [MP92], etc. Transition systems form the basis for most model checkers.

Over the years, a large number of model checkers have been developed in academia and industry[1]. We now look at some of the most prominent formalisms and model checkers in more detail. SPIN [Hol97] is a tool for protocol design and validation that targets on efficient software verification, not hardware verification. SPIN supports LTL (*linear temporal logic*) [Pnu77] model checking [VW86, Var96] by an on-the-fly exploration of the state space. The input language of SPIN is Promela – a nondeterministic guarded command language. The communication between components is limited to simple synchronous and asynchronous channels. Hence, Promela and SPIN hardly support hierarchical top-down or bottom-up modeling. SMV (*symbolic model verifier*) [McM93] was the first BDD-based CTL (*computation tree logic*) model checker. SMV is designed for hardware verification and its capability of handling complex data structures is limited. A recent variant of SMV is the model checker NuSMV [CCGR00] which now also supports LTL. Modeling complex software systems with NuSMV leads to rather artificial models which are not close to the real system. MOCHA [AHM+98] supports alternating-time symbolic model checking, e.g., ATL [AHK02] model checking of reactive modules [AH99].

The above model checkers, SPIN, NuSMV, and MOCHA, are very limited in communication and coordination between components directly and none of them treats exogenous coordination as a first class citizen. Hence, the compositional and hierarchical modeling approach is hardly supported by any of the above model checkers. In contrast, there are other formalisms like classical process algebras and corresponding tools that in principle would provide this support. E.g, CADP [GLMS11] is a powerful set of tools to design communication protocols and distributed systems. It is connected to various complementary tools and widely used in (industrial) projects. CADP uses the process algebra LOTOS (*Language of Temporal Ordering Specification*) [BB87] as the modeling language, which then is translated into a simplified process algebra called SUBLOTOS. From that an intermediate Petri Net model [Pet77, Pet81] is generated and in the end the labeled transition system (LTS) is produced by performing a reachability analysis on the Petri net. The LTS are stored in an explicit manner and CADP uses XTL (eXecutable Temporal Language), a functional-like programming language designed for the implementation of user defined temporal logics. FDR2 (Failures-Divergence Refinement Tool) is a refinement checker, whose modeling formalism is based on Hoare's process algebra CSP. It is used primarily for checking security properties.

[1] An incomplete overview is given e.g. at `http://anna.fi.muni.cz/yahoda/`

There are many more tools from the process algebra community like the *concurrency work bench* (CWB) [Ste97, CPS89] based on Milner's CCS for bisimulation checking and μ-calculus model checking, and μCRL [DG95, Wou01, vdP01, KKdV12] for μ-calculus model checking of process algebras, but none of them treats coordination as first class object in the first place. Although it is possible to specify coordination within by means of special operators, the specification becomes rather complex and artificial. In contrast, the declarative nature of Reo promotes specifying the coordination glue code in a very intuitive and flexible manner. The goal of this thesis is to introduce a model checker that directly supports the constructs of Reo and hence allows to link back counter examples and other properties to the original model effectively.

Goal of the thesis. The major goal of this thesis is to introduce a holistic approach for the modeling, analysis, and synthesis of component-based system models with exogenous coordination, which includes input languages that are expressive and simple, allow a clear separation of computation and coordination, support compositional and hierarchical modeling, allow specifying systems at any layer of the hardware/software stack, have clear formal semantics eligible to formal methods as well as temporal logics and model checking algorithms that treat exogenous coordination in component-based systems as a first class citizen, allow reasoning about aspects of coordination and dataflow, allow reasoning about alternating-time aspects such as controllability, and are decidable within reasonable time and space bounds. It is intended to evaluate the introduced techniques and methods for modeling, analysis, and synthesis within a newly implemented tool set.

Contribution. In this thesis we will introduce the syntax and formal semantics of two modeling languages CARML and RSL. CARML (Constraint Automata Reactive Module Language) is a guarded command language [Dij76] which is inspired by reactive modules [AH99] and serves for the specification of the interface behavior of components and Reo channels. RSL (Reo scripting language) is a scripting language that allows to create new instances of Reo channels and components specified in CARML and compositionally build larger networks and systems. Combining the two languages yields a compositional and hierarchical modeling framework. The combination of guarded command languages together with synchronous and asynchronous channel-based communication is also used in Promela [Hol93], the input language of the model checker SPIN [Hol97, Hol03], but Reo facilities a clear distinction between coordination and computation.

In this thesis we will present a branching-time and an alternating-time temporal logic tailored to our modeling framework based on Reo and constraint automata [BSAR06]. The branching-time logic BTSL combines standard features known from CTL [CE81, CGP99] with modalities from *propositional dynamic logics* like PDL [FL79] and the *timed data stream logic* (TDSL) [ABdBR04, CCA04] for reasoning about the observable dataflow at interface ports of Reo channels and components. The presented alternating-time logic ASL is inspired by classical alternating-time logics such as ATL [AHK02]. In this thesis a two-player game-based interpretation of constraint automata is proposed that allows reasoning about strategies in the Reo and constraint automata framework. Furthermore, we present the algorithms for the corresponding model checking problems and analyze them with respect to their theoretical complexity. For a practical evaluation of the model checking algorithms we implemented them within the model checking tool set Vereofy.

Vereofy is a tool set developed in our group at the TU Dresden that uses CARML and RSL as input languages and supports the introduced branching-time and alternating-time model checking as well as linear-time model checking and equivalence checking using bisimulation. To overcome the state-space explosion problem, Vereofy uses BDDs as data structures to represent constraint automata compactly. Also, the implementation of the model checking algorithms is based on BDDs. In this thesis we present the relevant implementation details of Vereofy as well as practical results achieved in a number of case studies.

Outline. Chapter 2 will give an overview on constraint automata and basic notations used throughout this thesis. In Chapter 3 we introduce our modeling framework based on Reo and constraint automata as well as syntax and semantics of our modeling languages CARML and RSL. The main contribution of this thesis can be found in Chapter 4 where syntax and semantics of our branching-time logic BTSL and the alternating-time logic ASL are presented. In this chapter we provide the model checking algorithms together with results on the complexity of the model checking problems. In Chapter 5 the details on the symbolic implementation within our verification tool set Vereofy are given. In this chapter we provide examples and case studies that have been carried out with Vereofy and present practical results. Chapter 6 concludes this thesis and presents ideas for future work.

2 | Constraint automata

Constraint automata [BSAR06] provide a generic operational model to formalize the behavioral interfaces of the components, the network (i.e., the glue code or connector) that coordinates a set of components, and a composite system consisting of components and the glue code. Constraint automata are variants of labeled transition systems (LTS) where the labels of the transitions represent the (possibly data-dependent) I/O-operations of the components and the network. Constraint automata are adequate to represent any kind of synchronous and asynchronous peer-to-peer communication. The states of a constraint automaton represent the local states of components and/or configurations of a connector. The transition labels characterize the possible I/O-activities at the dataflow locations where I/O-operations can take place, such as the interface ports of components, nodes in the connector network, etc. Constraint automata will also serve as a formal semantics for our input languages RSL and CARML [BBKK09a, BKK12] as introduced in Section 3.2.

Organization. Section 2.1 provides the basic notation of constraint automata, executions, paths and operators used for composition. In Section 2.2 we present an alternative syntax for constraint automata. This symbolic characterization is closer to what is used in the implementation and allows a more compact (graphical) representation. In Section 2.3 the concept of constraint automata will be enriched with polarity and type information.

2.1. Introduction

To formalize the I/O-activity, constraint automata use a finite set $\mathcal{N}$ of names for *dataflow locations*. Dataflow locations are points in the network where dataflow is observable, e.g., the I/O-ports of components or nodes of the network that serve as a router. We assume here that the data domain Data is a fixed and finite set of data items that can be observed at the dataflow locations.

For the syntax of constraint automata we slightly depart from the original one presented in [BSAR06]. Instead of having guard conditions on the possible dataflow as transition labels, the observable dataflow in a constraint automaton is in the first place formalized by means of concurrent I/O-operations (CIOs). CIOs can be understood as the possible actions of a component, a connector, or as interactions between several components and the connector in a composite system. The meaning of a transition instance $q \xrightarrow{c} p$ is that in configuration q, the concurrent I/O-operation c is enabled and state p is a possible successor configuration of q executing c. A concurrent I/O-operation specifies the dataflow locations, where at some specific time instance dataflow can be observed simultaneously.

Definition 2.1.1 (Concurrent I/O-operations)**.** Let $\mathcal{N}$ be a finite, nonempty set of names for dataflow locations. We define a concurrent I/O-operation as a function

$$c : \mathcal{N} \to \mathsf{Data} \cup \{\bot\},$$

where the symbol $\bot$ means that there is no dataflow at the corresponding dataflow location. We write $\mathsf{active}(c)$ for the set of names of dataflow locations $A \in \mathcal{N}$ such that $c(A) \in \mathsf{Data}$, where Data is the data domain. For technical reasons, we also allow the *empty* concurrent I/O-operation $c_\emptyset$ with $\mathsf{active}(c_\emptyset) = \emptyset$. It represents either an internal step of some component, or an unobservable step of a connector, where dataflow appears at some hidden (invisible) dataflow locations only. We refer to CIO as the set of all concurrent I/O-operations (including $c_\emptyset$). As we suppose $\mathcal{N}$ and Data to be finite, the set CIO of concurrent I/O-operations is finite as well. When reasoning about finite behavior we will also need a special symbol $\surd$ that indicates that dataflow has stopped. The set $\mathsf{CIO}_\surd$ stands for $\mathsf{CIO} \cup \{\surd\}$.

■

Definition 2.1.2 (Constraint automata [BSAR06])**.** A constraint automaton is a tuple

$$\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle,$$

where Q is a finite and nonempty set of states, $\mathcal{N}$ a finite set of port names, and $\longrightarrow$ is a subset of $Q \times \mathsf{CIO} \times Q$ called the transition relation of $\mathcal{A}$. We will write $q \xrightarrow{c} p$ instead of $\langle q, c, p\rangle \in \longrightarrow$. The set $Q_0 \subseteq Q$ is a nonempty set of initial states, AP a finite set of atomic propositions, $L : Q \to 2^{AP}$ a labeling function, and $\mathcal{Q} \subseteq Q$ a set of so called *quiescent states* such that $\{q \in Q \mid \text{there is no transition } q \xrightarrow{c} q'\} \subseteq \mathcal{Q}$.

■

The set $\mathcal{Q}$ of quiescent states characterizes those states $q \in Q$ where dataflow can stop completely (although there might be outgoing transitions).

Example 2.1.3 (Constraint automata)**.** Figure 2.1 shows an example of a constraint automaton providing the interface behavior of a component or a connector with two dataflow locations. The constraint automaton consists of the three states $Q = \{q_0, q_1, q_2\}$, the set $\mathcal{N} = \{A, B\}$ of names for the dataflow locations, and the depicted transition relation. The set of starting states $Q_0 = \{q_0\}$ is a singleton and contains the state q_0 only. As all states $q \in Q$ have outgoing transitions we can assume here, that the set of quiescent states is empty, i.e., $\mathcal{Q} \overset{\text{def}}{=} \emptyset$.

Assuming that A is an input port and B is an output port, the given behavior agrees with the behavior of a FIFO channel with capacity one which can store the two values of the data domain $\mathsf{Data} = \{0, 1\}$.

For the atomic propositions we define the set

$$AP = \{\text{empty}, \text{full}, \text{contains}_0, \text{contains}_1\}$$

and the labeling function as follows:

$$\begin{aligned} L(q_0) &= \{\text{empty}\} \\ L(q_1) &= \{\text{full}, \text{contains}_0\} \\ L(q_2) &= \{\text{full}, \text{contains}_1\}. \end{aligned}$$

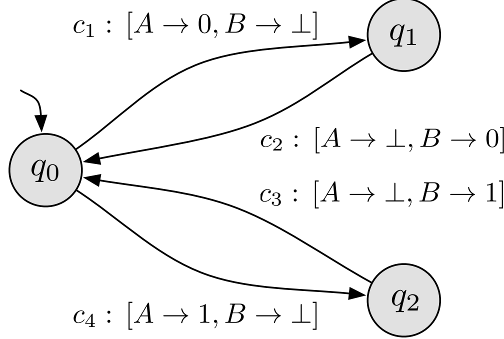


Figure 2.1: Constraint automaton for a component or connector

■

Definition 2.1.4 (Enabledness)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton as in Definition 2.1.2, $q \in Q$ a state, and $c \in \mathsf{CIO}$ a concurrent I/O-operation. We call c to be *enabled* in q iff c can take place in q and define the set of all I/O-operations enabled in q as

$$\mathsf{CIO}(q) \stackrel{\text{def}}{=} \left\{ c \in \mathsf{CIO} \,:\, q \xrightarrow{c} p \text{ for some } p \in Q \right\}.$$

State q is called *terminal* iff $\mathsf{CIO}(q) = \emptyset$.

■

Definition 2.1.5 (Execution, completeness of executions)**.** Let $\mathcal{A}$ be a constraint automaton. An *execution* in $\mathcal{A}$ is a finite or infinite sequence built of consecutive transitions:

$$\eta \;=\; q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$$

where $q_0, q_1, \ldots \in Q$, $c_1, c_2, \ldots \in \mathsf{CIO}$, and $q_i \xrightarrow{c_{i+1}} q_{i+1}$ for all $i \geq 0$. An execution is said to be *complete* if it is either infinite or it is finite and ends in a quiescent state $q_n \in \mathcal{Q}$.

■

The notion of complete executions serves to reason about "maximal" behaviors of constraint automata, called paths. A *path* of $\mathcal{A}$ is either an infinite execution or arises from a finite complete execution by adding a special transition symbol $\surd$ to denote termination.

Definition 2.1.6 (Path)**.** Each infinite execution of $\mathcal{A}$ is called an infinite *path*. Finite *paths* in $\mathcal{A}$ have the form

$$\pi \;=\; q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n \xrightarrow{\surd} q_n$$

where q_n is a quiescent state.

■

In the sequel, we shall use the symbol η for executions and the symbol π to range over paths. We write $\mathsf{Paths}(q)$ to denote the set of all paths starting in q and $\mathsf{Exec}_{\mathsf{fin}}(q)$ for the set of all finite executions starting in q. The *length* $|\pi|$ of a path π is the total number of transitions taken in π (including the pseudo-transition with label $\surd$). Thus, the length of an infinite path is ∞, while the length of a finite path π as in Definition 2.1.6 is $n{+}1$.

Let $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ be a path and $0 \leq n < |\pi|$. Then $\pi \downarrow n$ denotes the prefix of path π with length n. Thus,

$$\pi \downarrow n \overset{\text{def}}{=} q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n$$

is an execution. For finite paths of length n, the n-th prefix is defined to be π. In particular, if $n = |\pi|$ then $\pi \downarrow n = \pi$ is a path.

The *I/O-stream* $\mathsf{ios}(\eta)$ of a finite execution η is the finite word over CIO that is obtained by taking the projection to the labels of the transitions. Similarly, for finite paths the I/O-stream is a finite word over $\mathsf{CIO}_{\surd}$.

Definition 2.1.7 (I/O-stream (IOS))**.** If $\eta = q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n$ is a finite execution then $\mathsf{ios}(\eta) \overset{\text{def}}{=} c_1 \ldots c_n \in \mathsf{CIO}^*$. The associated I/O-stream for a finite path

$$\pi = q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n \xrightarrow{\surd} q_n$$

is defined by $\mathsf{ios}(\pi) \overset{\text{def}}{=} c_1 \ldots c_n\surd \in \mathsf{CIO}^*\surd$. The set of all I/O-streams is denoted by $\mathsf{IOS} \overset{\text{def}}{=} \mathsf{CIO}^* \cup \mathsf{CIO}^*\surd$. The associated I/O-stream for an infinite execution or path $\pi' = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ is defined by $\mathsf{ios}(\pi') \overset{\text{def}}{=} c_1 c_2 \ldots \in \mathsf{CIO}^{\omega}$.

■

Composition. Constraint automata provide a compositional semantics which is based on a parallel operator that allows synchronous and asynchronous I/O-activity. We present here a basic version of the product for arbitrary constraint automata. In the later chapters we will use a refined version of product that allows the product operator to be applied for *composable* constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ only. The formal definition of composability will be given at the end of this section.

Definition 2.1.8 (Constraint automata product [BSAR06])**.** The product automaton of two constraint automata $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \longrightarrow_i, Q_{0,i}, \mathcal{Q}_i, AP_i, L_i \rangle$ for $i \in \{1, 2\}$ is

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \longrightarrow, Q_{0,1} \times Q_{0,2}, \mathcal{Q}, AP_1 \cup AP_2, L \rangle,$$

where $\longrightarrow$ is defined by the following two rules for the asynchronous behavior

$$\frac{q \xrightarrow{c}_1 q' \ \wedge\ \mathsf{active}(c) \cap \mathcal{N}_2 = \emptyset}{\langle q, p \rangle \xrightarrow{c} \langle q', p \rangle} \qquad \frac{p \xrightarrow{c}_2 p' \ \wedge\ \mathsf{active}(c) \cap \mathcal{N}_1 = \emptyset}{\langle q, p \rangle \xrightarrow{c} \langle q, p' \rangle} \tag{2.1}$$

and one synchronization rule

$$\frac{q \xrightarrow{c_1}_1 q' \ \wedge \ p \xrightarrow{c_2}_2 p' \ \text{ with } \ c_1|_{\mathcal{N}_1 \cap \mathcal{N}_2} = c_2|_{\mathcal{N}_1 \cap \mathcal{N}_2}}{\langle q, p \rangle \xrightarrow{c_1 \oplus c_2} \langle q', p' \rangle} \tag{2.2}$$

where

$$c|_N(A) \stackrel{\text{def}}{=} \begin{cases} c(A) & \text{for all } A \in N \\ \bot & \text{else} \end{cases}$$

and $c_1 \oplus c_2$ is the unique concurrent I/O-operation with

$$\mathsf{active}(c_1 \oplus c_2) = \mathsf{active}(c_1) \cup \mathsf{active}(c_2) \text{ and } (c_1 \oplus c_2)|_{\mathcal{N}_i} = c_i.$$

The set of quiescent states in the product is defined as follows

$$\mathcal{Q} \stackrel{\text{def}}{=} \mathcal{Q}_1 \times \mathcal{Q}_2 \ \cup \ \{\langle q, p \rangle \in Q_1 \times Q_2 \mid \mathsf{CIO}(\langle q, p \rangle) = \emptyset\}$$

and the labeling function L of the product automaton is given by

$$L(\langle p, q \rangle) \stackrel{\text{def}}{=} L_1(p) \cup L_2(q).$$

■

The constraint automaton product operator is commutative and associative.

Example 2.1.9 (Product automaton)**.** On the left of Figure 2.2 the two constraint automata (without labels) of two FIFO channels with capacity one are depicted. Both automata are similar to the automaton shown in Example 2.1.3, but now for the singleton data domain $\mathtt{Data} = \{0\}$. Here, we assume that the two automata share and synchronize on dataflow location B. The product of these two automata shown on the right of Figure 2.2 can be interpreted as a FIFO channel with capacity two.

The concurrent I/O-operations of the product automaton are defined as follows:

$$\begin{array}{rcllll} c_{\text{in}} & = & [& A \to 0, & B \to \bot, & C \to \bot &] \\ c_{\text{trans}} & = & [& A \to \bot, & B \to 0, & C \to \bot &] \\ c_{\text{out}} & = & [& A \to \bot, & B \to \bot, & C \to 0 &] \\ c_{\text{both}} & = & [& A \to 0, & B \to \bot, & C \to 0 &] \end{array}$$

where c_{in} stands for the concurrent I/O-operation when the data item will be written to the first FIFO (via port A), c_{trans} describes the transfer of the data item from the first to the second FIFO via their shared dataflow location B and c_{out} stands for a read operation from the second FIFO (via port C). The remaining $c_{\text{both}} \in \mathsf{CIO}$ stands for the possibility that the read from the second FIFO and the write to the first FIFO can happen in the same time step.

■

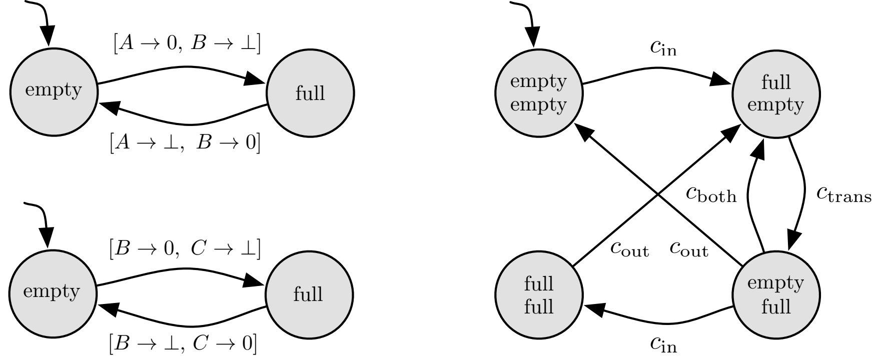


Figure 2.2: Product automaton of two FIFO channels with capacity one and $\mathsf{Data} = \{0\}$

Hiding. The hide operator for constraint automata provides a means to declare certain dataflow locations as local and non-observable from outside. This can, e.g., be used to abstract from internal behavior of a component connector or coordinating network or remove information about the channel-ends.

For the hiding there are two important variants. The first one is the *structure-preserving hide* which simply removes all names for dataflow locations $N \subseteq \mathcal{N}$ which should be hidden from the transition labels. The hiding in constraint automata is achieved by applying a projection $c|_{\mathcal{N} \setminus N}$ on all transitions.

Definition 2.1.10 (Structure-preserving hide)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L \rangle$ be a constraint automaton. The result of hiding some dataflow locations $N \subseteq \mathcal{N}$ from $\mathcal{A}$ is the constraint automaton

$$\mathcal{A}_{\setminus N} = \langle Q, \mathcal{N} \setminus N, \longrightarrow_{\setminus}, Q_0, \mathcal{Q}_{\setminus}, AP, L \rangle,$$

where the transition relation $\longrightarrow_{\setminus}$ is given by $q \xrightarrow{c} q'$ iff $q \xrightarrow{c|_{\mathcal{N}\setminus N}}_{\setminus} q'$. The set of quiescent states $\mathcal{Q}$ is defined as

$$\mathcal{Q}_{\setminus} \stackrel{\text{def}}{=} \mathcal{Q} \setminus \{q \in \mathcal{Q} \mid \text{there exists some } q \xrightarrow{c} q' \text{ with } \emptyset \neq \mathsf{active}(c) \subseteq N\}.$$

Hence, dataflow must not stop in states with newly introduced $c_\emptyset$-transitions.

■

Note that, for transitions in $\mathcal{A}$ of the form $q \xrightarrow{c} q'$ with $\mathsf{active}(c) \subseteq N$, the structure-preserving hide will produce transitions of the form $q \xrightarrow{c_\emptyset}_{\setminus} q'$, but the structure of the constraint automaton remains unchanged. Although there are other variants of hiding, the structure-preserving hide will be the most relevant in the context of this thesis as it preserves the structure and hence the branching behavior of the underlying constraint automaton.

Another important variant of hiding, called *aggregating hide* also uses the projection $c|_{\mathcal{N}\setminus N}$, but it removes newly introduced $c_\emptyset$-transitions. Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton. The result of hiding some dataflow locations $N \subseteq \mathcal{N}$ from $\mathcal{A}$ is the constraint automaton

$$\mathcal{A}_{\times N} = \langle Q, \mathcal{N} \setminus N, \longrightarrow_{\times}, Q_0, \mathcal{Q}_{\times}, AP, L\rangle,$$

where the transition relation $\longrightarrow_\times$ is given by:

$$\longrightarrow_\times \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} R^{(i)},$$

where the $R^{(i)}$ are defined as follows

$$\begin{aligned} R^{(0)} &\stackrel{\text{def}}{=} \big\{\langle q, c|_{\mathcal{N}\setminus N}, p\rangle \mid \text{there exists a transition } q \xrightarrow{c} q'\big\} \text{ and} \\ R^{(i+1)} &\stackrel{\text{def}}{=} \big\{\langle q, c_2, q'\rangle \mid \text{there exists } q \xrightarrow{c_1 \in \mathsf{CIO}_N} p \text{ and } \langle p, c_2, q'\rangle \in R^{(i)}\big\} \cup \\ &\qquad \big\{\langle q, c_1, q'\rangle \mid \text{there exists } p \xrightarrow{c_2 \in \mathsf{CIO}_N} q' \text{ and } \langle q, c_1, p\rangle \in R^{(i)}\big\}. \end{aligned}$$

Here, CIO_N denotes the set of all concurrent I/O-operations $c \in \mathsf{CIO}$ with $\mathsf{active}(c) \subseteq N$. The set of quiescent states $\mathcal{Q}$ is defined as for the structure-preserving hide:

$$\mathcal{Q}_\setminus \stackrel{\text{def}}{=} \mathcal{Q} \setminus \{q \in \mathcal{Q} \mid \text{there exists some } q \xrightarrow{c} q' \text{ with } \emptyset \neq \mathsf{active}(c) \subseteq N\}.$$

Note that applying the aggregating hide operator may change the branching behavior of the underlying constraint automaton.

Example 2.1.11 (Hiding variants)**.** Figure 2.3 a) shows the constraint automaton $\mathcal{A}$ for a FIFO channel with capacity two as considered in Example 2.1.9.

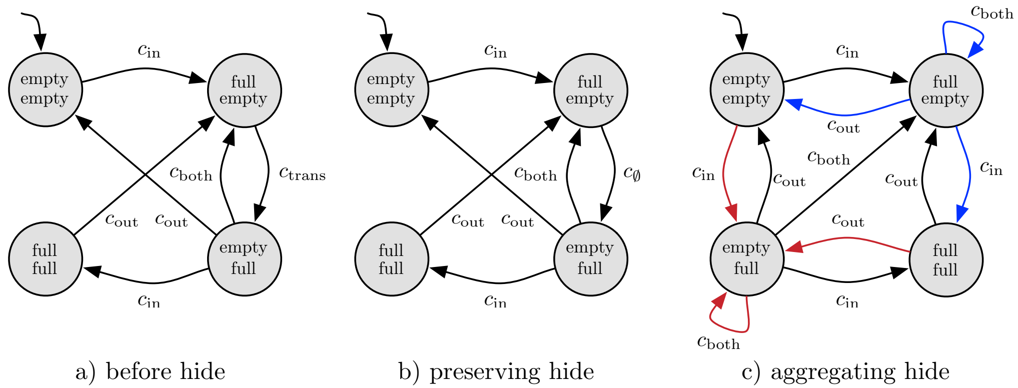


Figure 2.3: Hiding variants applied to the constraint automaton from Example 2.1.9

The transition labels are as before:

$$\begin{array}{rcl} c_{\text{in}} & = & [\ A \to 0, \quad B \to \bot, \quad C \to \bot\ ] \\ c_{\text{trans}} & = & [\ A \to \bot, \quad B \to 0, \quad C \to \bot\ ] \\ c_{\text{out}} & = & [\ A \to \bot, \quad B \to \bot, \quad C \to 0\ ] \\ c_{\text{both}} & = & [\ A \to 0, \quad B \to \bot, \quad C \to 0\ ] \end{array}$$

Figure 2.3 b) shows the constraint automaton $\mathcal{A}_{\setminus N}$ for the port-set $N = \{B\}$. Hence, port B will be removed from all transitions $q \xrightarrow{c} q'$ with $c(B) \neq \bot$. For the given constraint automaton $\mathcal{A}$ this affects the c_{trans} transition only.

Figure 2.3 c) depicts the constraint automaton $\mathcal{A}_{\times N}$ for the same port-set $N = \{B\}$. The newly added transitions $q \xrightarrow{c}_{\times} q'$ result from combining "regular transitions" in $\mathcal{A}$ (where $c \neq c_{\emptyset}$) with $c_{\emptyset}$-transitions, either of the form $q \xrightarrow{c} p \xrightarrow{c_{\emptyset}} q'$ or $q \xrightarrow{c_{\emptyset}} p \xrightarrow{c} q'$ with $c \neq c_{\emptyset}$.

■

2.2. Symbolic representation of constraint automata

Sometimes it is advantageous to use a symbolic representation of the transition relation by combining transitions with the same starting and target state (e.g., in our implementation of the model checker). For this purpose, we deal with I/O-constraints, i.e., propositional formulas in positive normal form that stand for *sets* of concurrent I/O-operations. The I/O-constraints may impose conditions on the dataflow locations that may or may not be involved and on the data items written on or read from them.

Definition 2.2.1 (I/O-constraints (IOC))**.** The abstract syntax of I/O-constraints is given by the grammar:

$$\mathsf{ioc} ::= tt \mid \mathit{ff} \mid A \mid \neg A \mid (d_{A_1}, \ldots, d_{A_k}) \in D \mid \mathsf{ioc}_1 \wedge \mathsf{ioc}_2 \mid \mathsf{ioc}_1 \vee \mathsf{ioc}_2$$

where $A \in \mathcal{N}$, $A_1, \ldots, A_k$ are pairwise distinct dataflow locations in $\mathcal{N}$ and $D \subseteq \mathsf{Data}^k$.

■

The meaning of an I/O-constraint ioc is a subset $\lceil \mathsf{ioc} \rceil$ of CIO defined in the expected way. We define $\lceil tt \rceil \stackrel{\text{def}}{=} \mathsf{CIO}$, $\lceil \mathit{ff} \rceil \stackrel{\text{def}}{=} \emptyset$, and for the literals $A \in \mathcal{N}$ and their negations $\neg A$:

$$\begin{array}{rcl} \lceil A \rceil & \stackrel{\text{def}}{=} & \{\, c \in \mathsf{CIO} \,:\, A \in \mathsf{active}(c) \,\} \\ \lceil \neg A \rceil & \stackrel{\text{def}}{=} & \{\, c \in \mathsf{CIO} \,:\, A \notin \mathsf{active}(c) \,\} \end{array}$$

The I/O-constraints $(d_{A_1}, \ldots, d_{A_k}) \in D$ impose conditions for the written and read data items. That is, $\lceil (d_{A_1}, \ldots, d_{A_k}) \in D \rceil$ agrees with the set

$$\{\, c \in \mathsf{CIO} \,:\, \{A_1, \ldots, A_k\} \subseteq \mathsf{active}(c) \text{ and } (c(A_1), \ldots, c(A_k)) \in D \,\}.$$

Conjunction and disjunction have their standard meaning, i.e.,

$$\begin{array}{rcl} \lceil \mathsf{ioc}_1 \wedge \mathsf{ioc}_2 \rceil & \stackrel{\text{def}}{=} & \lceil \mathsf{ioc}_1 \rceil \cap \lceil \mathsf{ioc}_2 \rceil \\ \lceil \mathsf{ioc}_1 \vee \mathsf{ioc}_2 \rceil & \stackrel{\text{def}}{=} & \lceil \mathsf{ioc}_1 \rceil \cup \lceil \mathsf{ioc}_2 \rceil \end{array}$$

We often use simplified notations for the IOC of the form $(d_{A_1}, \ldots, d_{A_k}) \in D$. E.g., the notation $d_A = d_B$ is a shorthand for $(d_A, d_B) \in \{(d_1, d_2) \in \mathsf{Data}^2 : d_1 = d_2\}$, while $A \wedge (d_B \in D)$ stands for the set $\{c \in \mathsf{CIO} : \{A, B\} \subseteq \mathsf{active}(c) \wedge c(B) \in D\}$. The notation $\{A, B\}$ is used as a shorthand for the set $\{c \in \mathsf{CIO} : \mathsf{active}(c) = \{A, B\}\}$. Let IOC denote the set of IO-constraints.

Definition 2.2.2 (Constraint automata (IOC-syntax))**.** A constraint automaton in IOC-syntax is a tuple $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle$, where Q is a finite and nonempty set of states, $\mathcal{N}$ a finite set of port names, $\longrightarrow$ is a subset of $Q \times \mathsf{IOC} \times Q$ called the transition relation of $\mathcal{A}$, $Q_0 \subseteq Q$ a nonempty set of initial states, AP a finite set of atomic propositions, and $L : Q \to 2^{AP}$ a labeling function. We write $q \xrightarrow{g} p$ instead of $\langle q, g, p\rangle \in \longrightarrow$.

■

Example 2.2.3 (Constraint automata (IOC-syntax))**.** As an example we revisit the constraint automaton from Example 2.1.3 and present it using the IOC-syntax (cf. Figure 2.4).

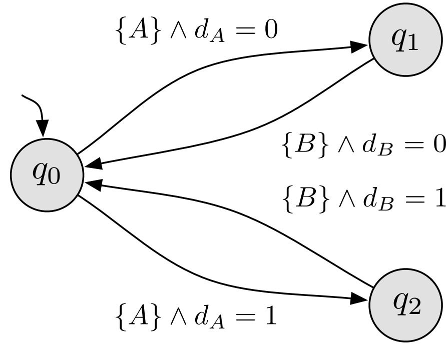


Figure 2.4: Constraint automaton for a component or connector (IOC-syntax)

■

The IOC-syntax equates to the original syntax of constraint automata as introduced in [BSAR06]. The use of this alternative syntax allows a (graphical) representation of constraint automata which is more compact and independent from a specific data domain. The same idea can be applied to states rather than on transition labels. By adding variables to states we can avoid the explicit enumeration of states for different data values. Figure 2.5 depicts a parameterized version of the constraint automata from Example 2.2.3 in IOC-syntax.

Here, the state $q_1\langle d\rangle$ is equipped with memory. The variable $d \in \mathsf{Data}$ is holding the value of the buffer. Incoming and outgoing transitions of state $q_1\langle d\rangle$ can refer to the actual value of d whereas the value is irrelevant for other states. This idea allows prototype definitions of components and connectors for arbitrary data domains. Our specification language CARML, which will be introduced in Subsection 3.2.2, makes use of this feature. E.g., FIFO1 channels can be defined to store arbitrary values without knowing the data domain in advance.

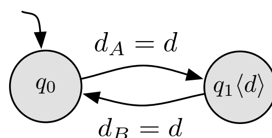


Figure 2.5: Constraint automaton with memory for a component or connector

The IOC-syntax uses I/O-constraints rather than concurrent I/O-operations for the transition labels, i.e., the labels are now Boolean conditions on active ports and the dataflow, which serve as guards. The I/O-constraints can immediately be translated into a symbolic representation as it is used in our implementation of the model checker (cf. Section 5.2). For this thesis we will similarly use both notions when adequate as they are equally expressive.

The basic definition of constraint automata as presented by now lacks some important information which will be added to the concept of constraint automata in the next section. We will add the concepts of *polarity*, as the direction in which data flows plays a crucial role for the composition in Reo. The second information added to constraint automata are *data types* which play an important role for defining the state space of a constraint automaton as well for specifying the domain for the observable data at dataflow locations.

2.3. Data types and polarity

Data types will play an important role for the syntax and semantics of our modeling languages RSL and CARML as they support variables, messages and ports that may have different types. We will now provide the basic terminology for data types and related notions that will be used throughout this thesis. First we will look at the set DT of data types with a fixed semantics. When using our modeling languages RSL and CARML the set DT consists of Booleans, finite integer ranges, enumerations, and structured data types such as arrays and unions. For the concepts introduced throughout this section the precise definition of the data types in DT is not required.

Data types with fixed semantics. Let DT denote a set of data types covering standard data types, such as Booleans or finite integer ranges, or user-defined data types with a fixed semantics, such as arrays and unions over elements of a predefined data type or enumeration types. Let Op denote a set of operators on the data types in DT such as conjunction, disjunction, negation for Booleans and arithmetic operations like addition and multiplication for integers. Furthermore, Pred denotes a set of predicates, such as the standard binary comparison predicates $=$, $<$, $\leq$, and so on.

Formally, the elements of DT are fixed finite non-empty sets and each operator $op \in \mathsf{Op}$ is a function

$$op : T_1 \times \ldots \times T_k \to T$$

where $(T_1, \ldots, T_k, T) \in \mathsf{DT}^{k+1}$ and $k \in \mathbb{N}$. The tuple $\langle T_1, \ldots, T_k, T\rangle$ is called the type of

op, denoted $\mathfrak{type}(op)$. Each predicate $pr \in \mathsf{Pred}$ is a subset of $T_1 \times \ldots \times T_k$ where $k \geq 1$ and $T_1, \ldots, T_k \in \mathsf{DT}$. We write $\mathfrak{type}(pr)$ for the tuple $\langle T_1, \ldots, T_k \rangle$.

Definition 2.3.1 (Signature)**.** A signature is a tuple $\mathfrak{Sig} = \langle \mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var} \rangle$ where $\mathsf{DT} \subseteq \mathfrak{DT}$, $\mathsf{Op} \subseteq \mathfrak{Op}$, $\mathsf{Pred} \subseteq \mathfrak{Pred}$ and $\mathfrak{Var}$ is a set of typed variables, i.e., for each variable $V \in \mathfrak{Var}$ its type, which is denoted by $\mathfrak{type}(V)$, is an element of $\mathfrak{DT}$.

■

Terms and atomic propositions. Terms over $\mathfrak{Sig}$ are built by variables, constants, and operator symbols in a type-consistent manner. Formally, terms over $\mathfrak{Sig}$ are defined recursively according to the following statements:

(1) Each variable $V \in \mathfrak{Var}$ is a term of type $\mathfrak{type}(V)$ and each constant $op \in \mathfrak{Op}$ (i.e., 0-ary operator in $\mathfrak{Op}$) is a term of type $\mathfrak{type}(op)$.

(2) If $t_1, \ldots, t_k$ are terms such that the type of t_i is T_i and $op \in \mathfrak{Op}$ with $\mathfrak{type}(op) = (T_1, \ldots, T_k, T)$ then $op(t_1, \ldots, t_k)$ is a term of type T.

Atomic propositions over $\mathfrak{Sig}$ are type-consistent expressions stating that a certain tuple of terms is an element of a predicate in $\mathfrak{Pred}$.

Formally, if $pr \in \mathfrak{Pred}$ with $\mathfrak{type}(pr) = (T_1, \ldots, T_k)$ and $t_1, \ldots, t_k$ are terms over $\mathfrak{Sig}$ such that t_i is of type T_i then $pr(t_1, \ldots, t_k)$ is called an atomic proposition over $\mathfrak{Sig}$.

Interpretations of signatures. For the semantics of terms and atomic propositions over a signature, we have to consider an interpretation $\mathcal{I}$ that provides type-consistent meanings for the uninterpreted data types $T \in \mathfrak{DT} \setminus \mathsf{DT}$, operator symbols $op \in \mathfrak{Op} \setminus \mathsf{Op}$, predicate symbols $pr \in \mathfrak{Pred} \setminus \mathsf{Pred}$, and the variables $V \in \mathfrak{Var}$. That is, $\mathcal{I}$ assigns a finite set $T^{\mathcal{I}} \neq \emptyset$ to each data type symbol T, an element of $V^{\mathcal{I}} \in T^{\mathcal{I}}$ to each variable of type T, a function $op^{\mathcal{I}} : T_1^{\mathcal{I}} \times \ldots \times T_k^{\mathcal{I}} \to T^{\mathcal{I}}$ to each operator symbol op of type $(T_1, \ldots, T_k, T)$ and a predicate $pr^{\mathcal{I}} \subseteq T_1^{\mathcal{I}} \times \ldots \times T_k^{\mathcal{I}}$ to each predicate symbol pr of type $(T_1, \ldots, T_k)$.

We treat $\mathcal{I}$ as an interpretation for all symbols of $\mathfrak{Sig}$ by putting $S^{\mathcal{I}} = S$ for all predefined symbols $S \in \mathsf{DT} \cup \mathsf{Op} \cup \mathsf{Pred}$. The semantics $t^{\mathcal{I}} \in T^{\mathcal{I}}$ of a term of type T and the truth value $pr(t_1, \ldots, t_k)^{\mathcal{I}} \in \{\mathsf{true}, \mathsf{false}\}$ for an interpretation $\mathcal{I}$ and an atomic proposition $pr(t_1, \ldots, t_k)$ are defined in the obvious way.

Apart from the type information, we require to add information about the polarity of the dataflow locations, i.e., whether the dataflow locations serve for input or output. For this, we will add the sets $\mathcal{N}^{\mathrm{src}}$ and $\mathcal{N}^{\mathrm{snk}}$ which are disjoint subsets of $\mathcal{N}$. Intuitively, $\mathcal{N}^{\mathrm{src}}$ stands for the set of input ports (so called *sources*) and $\mathcal{N}^{\mathrm{snk}}$ for the set of output ports (called *sinks*). With the help of *dataflow vocabulary* we may add type and polarity information to the concept of constraint automata.

Definition 2.3.2 (Dataflow vocabulary)**.** A dataflow vocabulary over a signature $\mathfrak{Sig}$ is a tuple $\mathfrak{Voc} = \langle \mathcal{N}, \mathcal{N}^{\text{src}}, \mathcal{N}^{\text{snk}}, \lambda \rangle$ where $\mathcal{N}^{\text{src}} \cup \mathcal{N}^{\text{snk}} \subseteq \mathcal{N}$ is a set of dataflow locations containing all source and sink locations such that $\mathcal{N}^{\text{src}} \cap \mathcal{N}^{\text{snk}} = \emptyset$ and $\lambda : \mathcal{N} \to \mathfrak{DT}$ is a function that assigns to each dataflow location A its message-type $\lambda(A)$, i.e., the type of data items that can be passed via dataflow location A.

■

Definition 2.3.3 (Extended constraint automaton)**.** An *extended constraint automaton* is a tuple $\langle \mathcal{A}, \mathfrak{Voc} \rangle$ where $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L \rangle$ is a constraint automaton and $\mathfrak{Voc} = \langle \mathcal{N}, \mathcal{N}^{\text{src}}, \mathcal{N}^{\text{snk}}, \lambda \rangle$ its associated dataflow vocabulary. Then, adding type information to ports of $\mathcal{A}$ imposes some obvious side conditions on the possible transitions. That is, any concurrent I/O-operation c that appears in a transition $q \xrightarrow{c} q'$ maps some $A \in \mathcal{N}$ to an element of its type $\lambda(A)$, i.e.,

$$\text{if } c(A) = d \neq \bot \text{ then } d \in \lambda(A) \text{ for all } A \in \mathcal{N}.$$

A concurrent I/O-operation $c \in \mathsf{CIO}$ is called *type-safe* with respect to $\mathfrak{Voc}$ if it satisfies the above condition. For an extended constraint automaton $\langle \mathcal{A}, \mathfrak{Voc} \rangle$ where $\mathcal{A}$ is specified in IOC-syntax this means that for a given transition $q \xrightarrow{g} q'$ in $\mathcal{A}$ we allow only type-safe concurrent IO-operations to fire when reasoning about runs in $\mathcal{A}$, i.e., $q \xrightarrow{c} q'$ if $c \in \lceil g \rceil$ and c is type-safe.

■

Definition 2.3.4 (Composability)**.** Let $\langle \mathcal{A}_i, \mathfrak{Voc}_i \rangle$, $i \in \{1, 2\}$ be two extended constraint automata with constraint automata $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \longrightarrow_i, Q_{0,i}, \mathcal{Q}_i, AP_i, L_i \rangle$ and associated dataflow vocabularies $\mathfrak{Voc}_i = \langle \mathcal{N}_i, \mathcal{N}_i^{\text{src}}, \mathcal{N}_i^{\text{snk}}, \lambda_i \rangle$. The two extended constraint automata $\langle \mathcal{A}_1, \mathfrak{Voc}_1 \rangle$ and $\langle \mathcal{A}_2, \mathfrak{Voc}_2 \rangle$ are called *composable* if the following conditions hold for all dataflow locations $A \in \mathcal{N}_1 \cap \mathcal{N}_2$:

i) $AP_1 \cap AP_2 = \emptyset$

ii) $A \in \mathcal{N}_1^{\text{src}} \cap \mathcal{N}_2^{\text{snk}}$ or $A \in \mathcal{N}_2^{\text{src}} \cap \mathcal{N}_1^{\text{snk}}$

iii) $\lambda_1(A) = \lambda_2(A)$

Please note, that here we require strict type consistency in the third condition to simplify the presentation throughout the next chapters. In the implementation that is presented in Chapter 5 we support some implicit casts for simple data types.

■

Definition 2.3.5 (Extended product)**.** Let $\langle \mathcal{A}_i, \mathfrak{Voc}_i \rangle$, $i \in \{1,2\}$ be two composable extended constraint automata as in Definition 2.3.4. The *extended product automaton* is defined as $\langle \mathcal{A}_1 \bowtie \mathcal{A}_2, \mathfrak{Voc} \rangle$, where $\mathfrak{Voc} = \langle \mathcal{N}, \mathcal{N}^{\text{src}}, \mathcal{N}^{\text{snk}}, \lambda \rangle$ is the associated dataflow vocabulary with

$$\begin{aligned}
\mathcal{N} &\stackrel{\text{def}}{=} \mathcal{N}_1 \cup \mathcal{N}_2, \\
\mathcal{N}^{\text{src}} &\stackrel{\text{def}}{=} (\mathcal{N}_1^{\text{src}} \cup \mathcal{N}_2^{\text{src}}) \setminus (\mathcal{N}_1 \cap \mathcal{N}_2), \\
\mathcal{N}^{\text{snk}} &\stackrel{\text{def}}{=} (\mathcal{N}_1^{\text{snk}} \cup \mathcal{N}_2^{\text{snk}}) \setminus (\mathcal{N}_1 \cap \mathcal{N}_2), \text{ and} \\
\lambda(A) &\stackrel{\text{def}}{=} \begin{cases} \lambda_1(A) & \text{if } A \in \mathcal{N} \setminus \mathcal{N}_2 \\ \lambda_2(A) & \text{else} \end{cases}
\end{aligned}$$

■

In the next chapter we will look at the modeling formalism and languages used throughout this thesis. To keep notations simple we will identify extended constraint automata $\langle \mathcal{A}, \mathfrak{Voc} \rangle$ with $\mathcal{A}$ whenever possible. Furthermore, we assume that $\langle \mathcal{A}_1, \mathfrak{Voc}_1 \rangle$ and $\langle \mathcal{A}_2, \mathfrak{Voc}_2 \rangle$ are composable (with respect to Definition 2.3.4) whenever applying the constraint automata product to $\mathcal{A}_1$ and $\mathcal{A}_2$.

3 Modeling formalism and languages

Reo [Arb04] is a declarative modeling language for specifying the glue code for coordinating components from outside. In this *exogenous* approach, the components do not need to be aware of the presence of each other. The components communicate by sending and receiving their data via a network of channels.

The *Reo scripting language* (RSL) [BBKK09a, BKK12] is inspired by Reo and yields an elegant declarative framework for the compositional construction of connectors by creating instances of components and channels and then glueing their channels-ends, the I/O-ports of components, or sub-connectors together. This is done via join operations, resulting in a network called *Reo circuit*. RSL mainly serves as a specification language for Reo component connectors and networks. The formal semantics of RSL is (and Reo itself) based on constraint automata [BBKK09a], which can be constructed in a compositional way by providing constraint automata models for each channel and component and mimicking Reo's operators by corresponding operators for constraint automata [BSAR06].

The interface behavior of components, channels, and connectors instantiated and composed by RSL scripts can be specified within our second modeling language CARML (constraint automata reactive module language). CARML uses features of reactive modules [AH99] and together with RSL the two languages form a hybrid modeling approach. As the two languages are based on the same operational semantic model, this hybrid modeling approach allows to either specify the interface behavior of any entity of a component-based system by providing a CARML module or by providing an RSL script composing the entity from smaller building blocks.

Organization. Section 3.1 summarizes the syntax, semantics and main concepts of the coordination language Reo as introduced in [Arb04]. Section 3.2 we will introduce syntax and semantics of our modeling languages RSL and CARML.

The work presented in the latter section has partially been published in [BBKK09a, BKK12].

3.1. The coordination language Reo

In this section we give a brief introduction the channel-based coordination language Reo as introduced in [Arb04]. The primitives in Reo are channels. A Reo channel consists of exactly two *channel-ends* which are either declared to be *source-end* if they permit writing of data or *sink-end* if they allow the reading of data.

Primitives. Reo suggests a set of basic channels. This set can be extended with user-defined channels by providing a behavioral description for each channel. The graphical representation for the set of basic channels that is used throughout this thesis is shown in Figure 3.1.

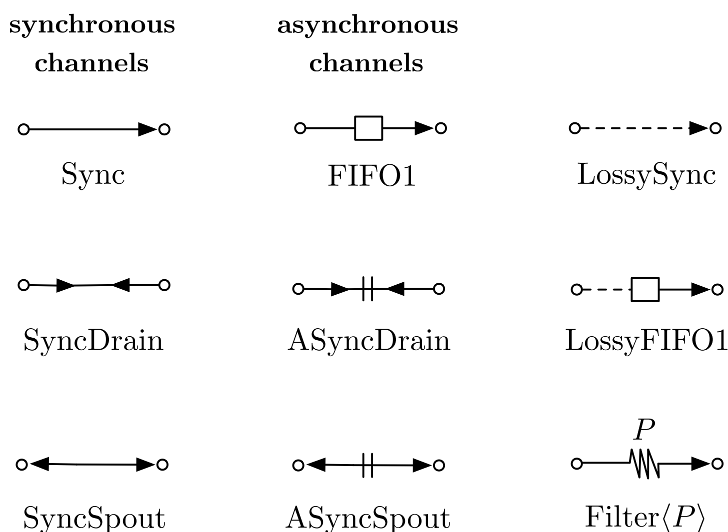


Figure 3.1: Basic set of Reo channels

Channels working in a completely synchronous manner, i.e., both channel-ends are always active at the same time, are called *synchronous channels*. Channels working in a complete asynchronous fashion are called *asynchronous channels*. The behavior of channels may depend on any combination of their current configuration, the available data to be transmitted, on a (purely) nondeterministic choice and the context in which they are deployed.

The intuitive behavior of the primitive channels depicted in Figure 3.1 is as follows. The *synchronous channel* (Sync) accepts data at its source-end and simultaneously writes it to the sink-end. Another very useful channel for the design of complex coordination principles in Reo is the *synchronous drain* channel (SyncDrain). It has two source-ends (but no sink-end). A data item has to be written on both ends simultaneously and both get destroyed. A *synchronous spout* (SyncSpout) produces unspecified data and writes it simultaneously on both of its sink-ends. The SyncDrain and the SyncSpout can be used to synchronize read and write operations respectively. The *asynchronous drain* (ASyncDrain) and *asynchronous spout* (ASyncSpout) act as their synchronous counterparts, except that their channel-ends fire mutually exclusive rather than simultaneously. The *FIFO* channel (FIFO1) is also an asynchronous channel, but in contrast to the previous channels it has memory. The FIFO1 has a buffer cell where the buffer is initially empty. Writing a data item at the source-end is enabled as long as the buffer is empty. The effect of writing a data item $d \in \mathsf{Data}$ is that d will be stored in the buffer. Reading data at the sink-end is enabled if the buffer is filled, in which case the data item is taken off the channel. The third column of Figure 3.1 depicts

some channels that are neither synchronous nor asynchronous. The first one is a nondeterministic *lossy synchronous channel* (LossySync) which acts as the Sync, but it may also loose the data. In the original work of [Arb04] a lossy synchronous channel was proposed that loses the data if and only if there is no reader at the sink-end ready to take the data. Throughout this thesis we will use a nondeterministic variant of this channel as it is sufficient in the context where it will be applied. The *lossy FIFO* channel (LossyFIFO1) is a lossy variant of the FIFO1. The losing of data depends on the configuration of the buffer. A write on the source-end will succeed in any case and the data will be lost if and only if the buffer is full. The *P-filter* channel (Filter$\langle P \rangle$) with filter condition $P \subseteq \mathsf{Data}$ is a variant of the LossySync. In this case losing depends on the data value written on the source-end of a filter. The data values $d \in P$ can pass whereas a $d' \in \mathsf{Data} \setminus P$ will be lost.

Remark 3.1.1. As Reo channels are intended to work with any type of data, we assume here that Data is a data type that serves as a placeholder for any data that may flow through a Reo network. Before building a concrete Reo network concrete decisions for the message-types of all ports have to be made.

■

Reo semantics. Several alternative semantics have been proposed for Reo. Some important candidates are enumerated below.

1. The basic semantic model is that of *timed-data streams* (TDS) [AR02]. This coalgebraic model of a stream calculus formally describes the timed dataflow through channels and connectors as sequences of discrete events. The TDS semantics captures best the linear-time behavior and is too abstract for branching-time model checking.
2. The *connector coloring* (CC) [CCA07, Cos10] describes the behavior of a Reo network in terms of the detailed dataflow behavior of all its constituent channels and nodes. The semantics of a Reo network is the set of all of its dataflow alternatives in each static configuration. Each such alternative is a consistent composition of the dataflow alternatives of each of its constituent channels and nodes, expressed in terms of two colors that represent the basic flow and no-flow alternatives. A more sophisticated model using three colors is necessary to capture the context sensitive behavior of primitives such as the *LossySync* channel. The CC semantics captures the possible interactions for individual configurations rather then the operational behavior [JA11].
3. In [CPLA11] the view changed from a graphical representation of Reo channels, whose overall behavior is sometimes difficult to grasp, to constraint based perspective. This constraint-based semantics (CBS) of Reo is expressed in terms of behavioral constraints imposed by the primitive channels which are propagated by the composition. As the CC semantics does, the CBS semantics captures the possible interactions for each individual configurations rather then the operational behavior.

4. An alternative semantics is based on *constraint automata* (CA) [BSAR06], which is consistent with the TDS semantics. Constraint automata are variants of labeled transition systems where the labels of the transitions represent the (possibly data-dependent) I/O-operations of the components and the network. The intuitive meaning of constraint automaton as an operational model for Reo connectors is similar to the interpretation of labeled transition systems as formal models for reactive systems. A Constraint automaton contains a set of port names, which holds the names of the dataflow locations. The states represent the configurations of the connector, the transitions the possible one-step behavior. For details on the relationship of constraint automata and the connector coloring semantics as well as their expressiveness we refer to [JA11].

Throughout this thesis we will apply the constraint automata semantics, as a semantics based on automata serve well for temporal logic model checking of branching-time and alternating-time properties. Constraint automata capture the operational behavior of Reo. At the same time we will use constraint automata as semantics for the connected components to describe their stepwise I/O-behavior. Thus, constraint automata serve as a common formal semantics for the components, Reo channels, and consequently for component connectors and the composite system. Moreover, constraint automata preserve the compositional nature of Reo, as constraint automata provide an product operator equivalent to the Reo join, which is used for composition.

Remark 3.1.2 (Finiteness)**.** For the concepts illustrated in this section our assumption of a finite state space for constraint automata is not relevant. Indeed, we could provide constraint automata with infinite state space, e.g., to model the behavior of a FIFO channel with infinite capacity. Only in later chapters on model checking the assumption that the state space of a constraint automaton is finite will be crucial.

■

We will now provide the operational behavior for each of these channels in terms of extended constraint automata specifications $\langle \mathcal{A}, \mathfrak{Voc} \rangle$. Figure 3.2 shows the extended constraint automata for the Reo primitive channels.

Remark 3.1.3. We will still ignore the message-type information for the I/O-ports of the constraint automata in the first place and assume that $\lambda(A) = \mathsf{Data}$ for all $A \in \mathcal{N}$. Hence, for their definition we will use the data type Data as a placeholder, and the effect is that we can provide definitions for Reo channels that are able to deal with any kind of data. This way the constraint automata for the individual parts of a Reo network are composable (cf. Definition 2.3.4). The message-type of I/O-ports has to be declared when building a constraint automaton for a concrete channel, as the message-type may affect the state space Q (e.g., for a FIFO1) and also the transition relation $\longrightarrow$ (e.g., in case of a Filter channel). In what follows we assume that the message-type of the ports is resolved in a way such that the constraint automata for the individual parts of a Reo network are still composable.

■

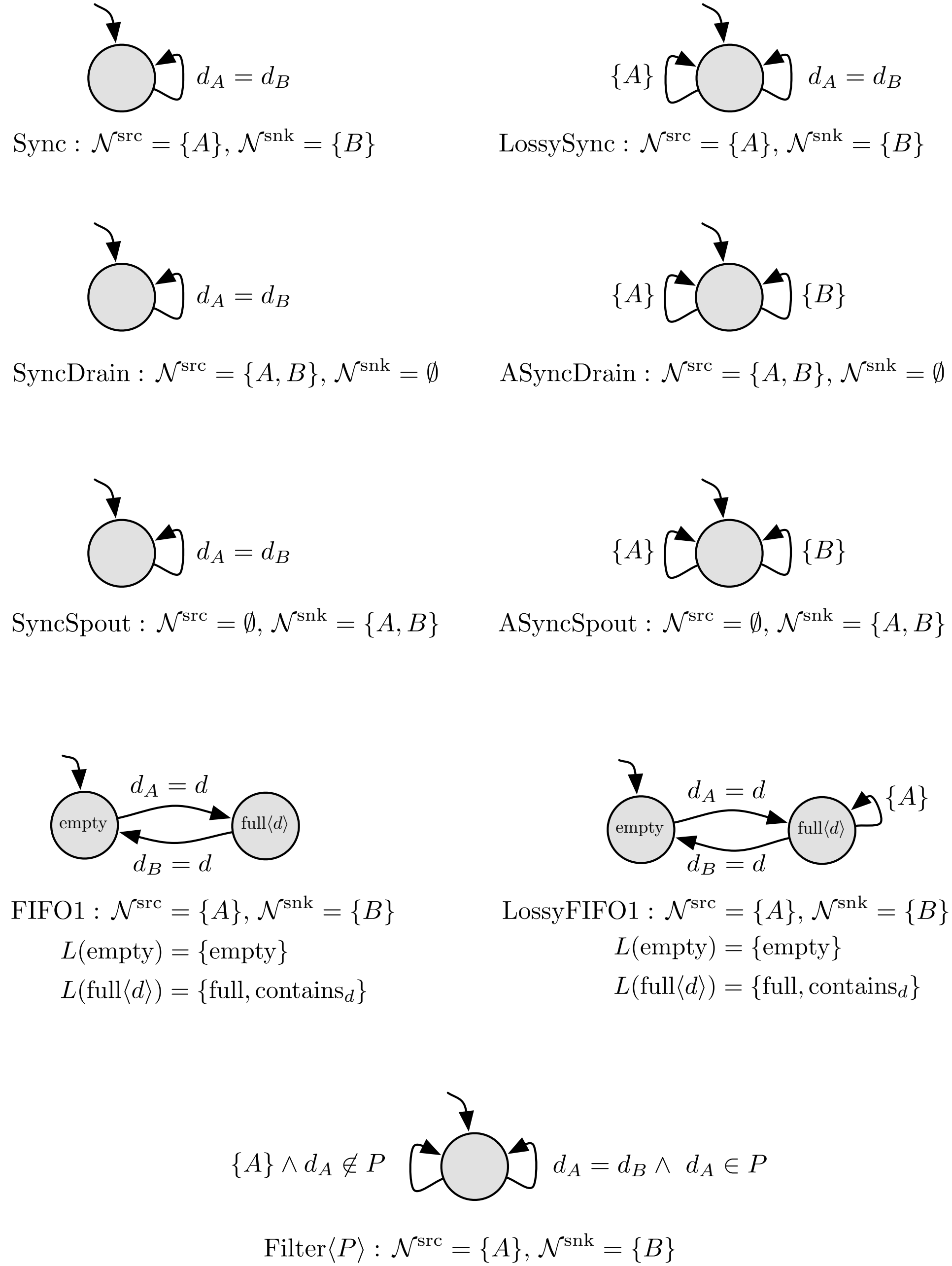


Figure 3.2: Constraint automata for Reo channels

One can see that some of the channels are based on the same structure, i.e., they have the same constraint automaton and differ on the associated dataflow vocabulary $\mathfrak{Voc} = \langle \mathcal{N}, \mathcal{N}^{\mathrm{src}}, \mathcal{N}^{\mathrm{snk}}, \lambda \rangle$ only. Here, the dataflow vocabulary (cf. Definition 2.3.2) provides the polarity attribute (either sink or source) for each channel-end. Hence, a Sync channel can be distinguished from a SyncDrain and a SyncSpout by the associated dataflow vocabularies.

In Figure 3.2 the two variants of the FIFO1 their "full" states are equipped with a variable $d \in \mathsf{Data}$ holding the value of the buffer. Incoming and outgoing transitions of state "full" can refer to the actual value of d whereas the value is irrelevant for other states. For all channels except for the two FIFO1 channels the set of atomic propositions AP is irrelevant as the state spaces of their underlying constraint automata are singleton sets. For the FIFO1 channels we provide the set $AP \stackrel{\text{def}}{=} \{\text{empty,full}\} \cup \{\text{contains}_d \mid d \in \mathsf{Data}\}$ with the illustrated labeling function.

For the quiescence of the constraint automata, we assume that all states are quiescent states (i.e., $\mathcal{Q} = Q$) in all constraint automata for the selected Reo channels. Hence, dataflow can potentially stop at any time in execution.

From the set of individual channels a network can be composed. The composition of Reo channels makes use of *Reo nodes* and the compositional semantics is based on the product operator for constraint automata. Please note, that only sink ports will be joint with source ports (of the same message-type), and thus the constraint automata for channels, Reo nodes, and components are composable in the sense of Definition 2.3.4. We will now look at the composition in more detail.

Composition. A *Reo node* is a nonempty set of dataflow locations, i.e. channel-ends and I/O-ports of components which is either source, sink or mixed, which are classified into *source*, *sink* and *mixed nodes*. Depending on whether all channel-ends that coincide on a node N are source-ends (then N is a source node), sink-ends (then N is a sink node), or a mixture of sink and source-ends (then N is a mixed node). The mixed nodes serve as routers where data items are transmitted through the network. The join operator of Reo takes a set of channel-ends and creates a Reo node. Figure 3.3 illustrated the different types of Reo nodes to be generated.

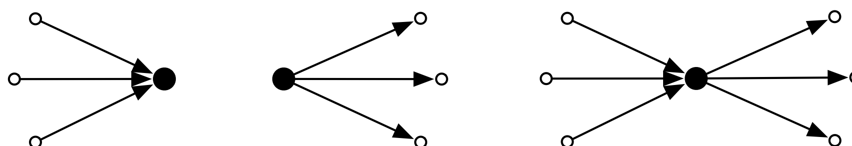


Figure 3.3: Reo nodes: a) sink node, b) source node, and c) mixed Reo node

The operational semantics of Reo nodes is also based on constraint automata. In general we may support any behavior inside a Reo node as long as we do provide a constraint automaton describing its behavior. Hence, any Reo node N could be replaced by a component C_N with an appropriate number of input and output ports (cf. Figure 3.4) having the same routing behavior as the Reo node. The component C_N plays now the role of the Reo node.

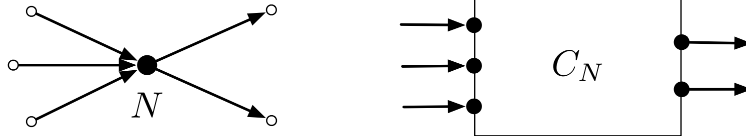


Figure 3.4: Replacement of a Reo node N by a component C_N

Throughout this thesis, we use two variants of nodes, the *standard Reo nodes* and *Reo route nodes*. In what follows we will look at these two variants in more detail and assume that the set of source-ends associated with the Reo node N is given by $\mathcal{N}^{\text{src}} = \{A_1, \ldots, A_k\}$ and the sink-ends by $\mathcal{N}^{\text{snk}} = \{B_1, \ldots, B_\ell\}$. The port-set $\mathcal{N}$ of the constraint automaton for a Reo node N is then given by

$$\mathcal{N} \stackrel{\text{def}}{=} \mathcal{N}^{\text{src}} \cup \mathcal{N}^{\text{snk}} \cup \{N\}$$

as we add a fresh name N to the port-set.

The *standard Reo node* (depicted as ●) has a *merge-and-replicate*-semantics, where simultaneously a datum is taken from one of the sink-ends (chosen nondeterministically) and a copy of that datum is sent to all source-ends. Figure 3.5 shows the behavior of a standard Reo node N in terms of constraint automata.

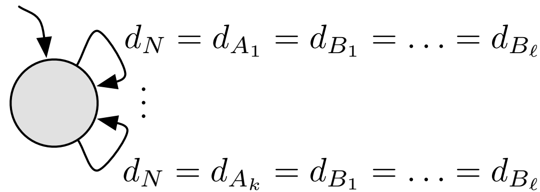


Figure 3.5: Constraint automata for a standard Reo node N

The *Reo route node* (depicted as ⊗) has a *merge-and-route*-semantics, as a datum is simultaneously taken from one of the sink-ends and forwarded to exactly one of the of the source-ends. The choice of the two ends is completely nondeterministic. Figure 3.6 depicts the constraint automaton of a Reo route node N.

It is worth mentioning that for these two types for Reo nodes the polarity of the channel-ends, i.e., whether the channel-ends are source or sink-ends, is important, as source and sink-ends are being treated differently.

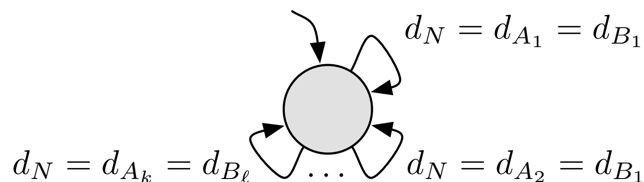


Figure 3.6: Constraint automata for a Reo route node N

The compositional semantics of Reo is based on the product operator ($\bowtie$) of constraint automata. When composing a network of channels the compositional behavior arises from applying the constraint automaton product operator to the constraint automata for the channels, and the Reo nodes, and the connected components.

Example 3.1.4 (Composition of two FIFO1 channels)**.** We start with two FIFO1 channels as depicted in Figure 3.7. The result of joining B_1 with B_2 will be a new standard Reo node B. The overall result of the composition will be a FIFO channel with capacity two.

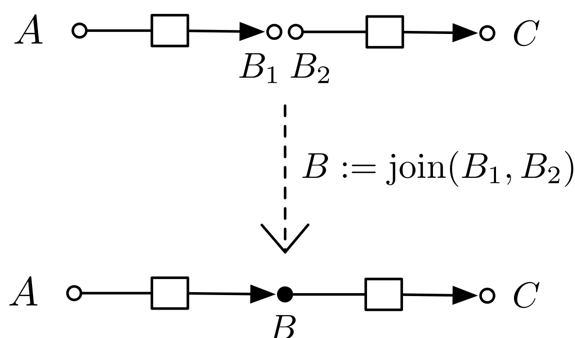


Figure 3.7: Reo join for two FIFO channels

The behavior of the FIFO channel with capacity two arises from building the product automaton of the automata $\mathcal{A}_1$ (with port-set $\mathcal{N}_1 = \{A, B_1\}$, where $A \in \mathcal{N}_1^{\text{src}}$ and $B_1 \in \mathcal{N}_1^{\text{snk}}$) and $\mathcal{A}_2$ (with port-set $\mathcal{N}_2 = \{B_2, C\}$ where $B_2 \in \mathcal{N}_1^{\text{src}}$ and $C \in \mathcal{N}_1^{\text{snk}}$) for the channels and the automaton $\mathcal{A}_B$ for newly created Reo node B. The latter has port-set $\mathcal{N}_B = \{B_1, B_2, B\}$ where $B_1 \in \mathcal{N}^{\text{src}}$ and $B_2 \in \mathcal{N}^{\text{snk}}$. For the composition we will have to fix the message-type of all I/O-ports (instead of using Data as a placeholder for all message-types). To simplify the presentation we choose $\lambda(A') = T = \{0\}$ for all $A' \in \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_B$. On the left of Figure 3.8 the three automata are shown. The product automaton $\mathcal{A} = \mathcal{A}_1 \bowtie \mathcal{A}_2 \bowtie \mathcal{A}_B$ with port-set $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_B$ is depicted on the right of Figure 3.8.

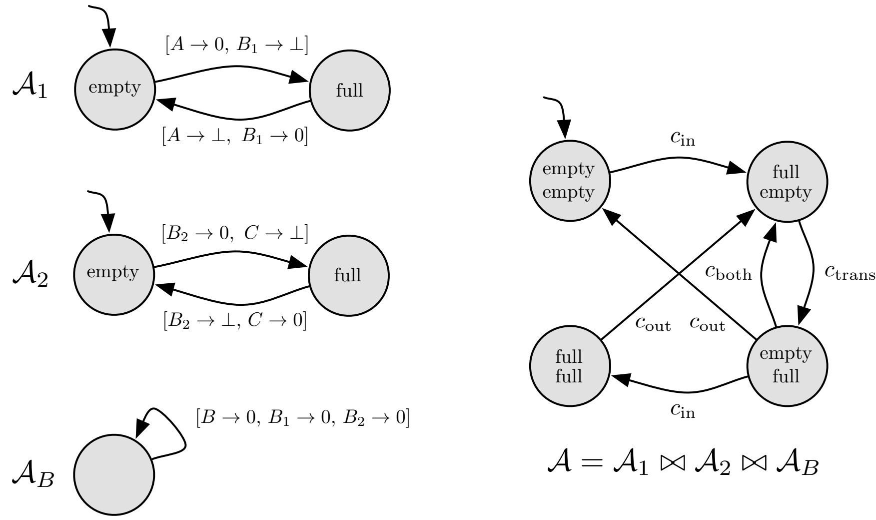


Figure 3.8: Automata for two FIFO1 channels, the standard Reo node and their product

The concurrent I/O-operations of the product automaton are as follows:

$$\begin{array}{rcllllll}
c_{\text{in}} & = & [& A \to 0, & B \to \bot, & B_1 \to \bot, & B_2 \to \bot, & C \to \bot \quad] \\
c_{\text{trans}} & = & [& A \to \bot, & B \to 0, & B_1 \to 0, & B_2 \to 0, & C \to \bot \quad] \\
c_{\text{out}} & = & [& A \to \bot, & B \to \bot, & B_1 \to \bot, & B_2 \to \bot, & C \to 0 \quad] \\
c_{\text{both}} & = & [& A \to 0, & B \to \bot, & B_1 \to \bot, & B_2 \to \bot, & C \to 0 \quad]
\end{array}$$

The product automaton $\mathcal{A}$ agrees with the automaton presented in Example 2.1.9 except that we identified B_1 and B_2 with B.

■

A *Reo component connector* consists of a nonempty set of Reo channels and a set of Reo nodes. The I/O-ports form the interface of the connector, where other components may connect to. When components are connected, channel-ends and I/O-ports of components are joined in a Reo node. The *degree* of a connector is the number of I/O-ports. Reo's primitive channels are the most simple form of Reo connector. Connectors can have memory which is formed by buffer cells, e.g., FIFO1 channels. The buffer cells can either be empty or full. In connectors without memory data items may either be produced, pass through or get lost within the connector.

Example 3.1.5 (Reo component connector)**.** Figure 3.9 depicts a simple Reo component connector, which accepts data items at its interface port A_1 and forwards the data to either port B_1 or B_2.

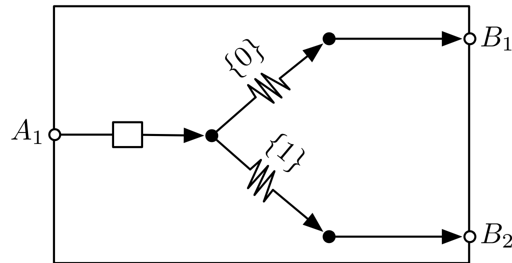


Figure 3.9: Reo component connector for data-dependent routing

The data items are stored and then routed according to their value with the help of the two filter channels. Hence, all data with value 0 will be routed to port B_1 and data with value 1 will be routed to B_2. All other data items will be lost. Here we make the assumption that $\lambda(A) = T_A \subseteq \{0, 1\}$ for all dataflow locations A. From now on we will call this connector a data-dependent router.

■

Example 3.1.6 (Reo component connector)**.** Components can be connected with the help of connectors to form a *Reo network*. A Reo network consists of a set of components, Reo connectors, and Reo nodes. We assume that at most one component will be connected per I/O-port of a connector.

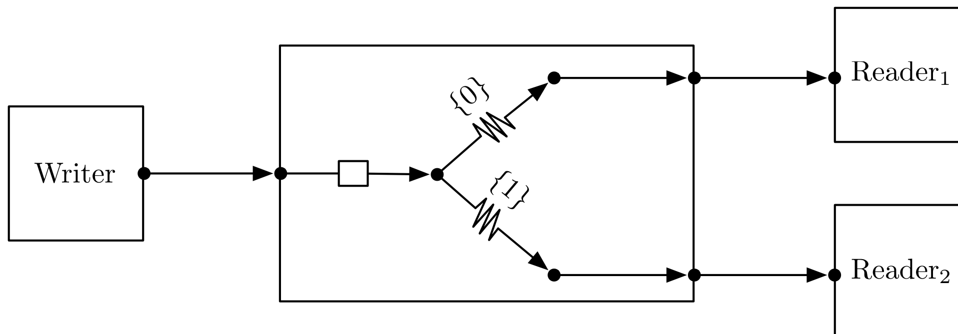


Figure 3.10: A writer and two readers connected by a data-dependent router

In the Reo network shown in Figure 3.10 the Reader_1 component will receive all data with value 0 sent by the Writer component whereas the Reader_2 component receives the ones with value 1. The specification for the readers and the writer can be provided in terms of constraint automata as well.

■

What can be seen in this example is, that the connector depicted in Figure 3.9 with its interface ports A_1, B_1, and B_2 forms a simple coordination pattern (or connector) which can be applied in various situations, e.g. to distribute data (or job messages) among readers (or workers, respectively).

Hiding. Another important Reo operator realizes the concept of *hiding*. The intuitive idea of hide is to hide the internal details at the end of the composition phase. Hidden Reo nodes or channel-ends must not be accessed for further composition. In general the mixed nodes of a Reo network will be hidden such that only the behavior at the interface ports of a connector remains visible at the end of the composition. The semantics of hide is based on the structure-preserving hide operator for constraint automata (cf. Definition 2.1.10).

Example 3.1.7 (Reo hide operator)**.** As an example we revise the data-dependent router presented in Example 3.1.5 (cf. Figure 3.9). Let $\mathcal{A}$ be the constraint automaton representing the behavior of the Reo component connector. As $\mathcal{A}$ arises from building the product of the constraint automata for the FIFO1, the two Filter, the two Sync, and the Reo nodes, the constraint automaton $\mathcal{A}$ contains all internal details of the Reo component connector. Its port-set $\mathcal{N}$ contains names for all dataflow locations, i.e., all channel-ends and Reo nodes. As we are interested in the I/O-behavior of the component connector at its interface ports A_1, B_1, and B_2 we will apply the structure-preserving hide operator for constraint automata with the set $N = \mathcal{N} \setminus \{A_1, B_1, B_2\}$. The constraint automaton $\mathcal{A}_{\setminus N}$ for $\mathsf{Data} = \{0, 1\}$ is depicted in Figure 3.11.

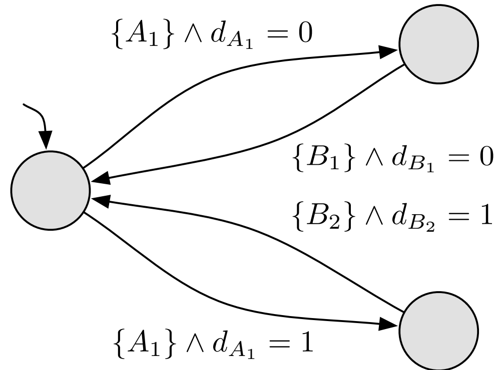


Figure 3.11: Constraint automaton (after hiding the internals) for a data-dependent router

■

Other Reo operators. There are more operations offered by Reo like the split which may abolish the effect of previous join operations. For this thesis we will not consider these operators as any static network structure which can be constructed by the complete list of Reo operations can also be obtained by a sequence of instantiations, join, and hide operations.

For specifying dynamic changes in the network topology at runtime Reo supports other operators. As in general those operators lead to infinite state automaton representations we will restrict ourselves to dynamic changes of network structures which can we represented by enumerating a finite set of network structures having the same interface. Thus we omit explanations of additional Reo operations.

3.2. The modeling languages RSL and CARML

In this sections we introduce the two new modeling languages CARML and RSL that are part of the contribution of this thesis. CARML and RSL yield the basis for our hybrid modeling approach based on Reo and constraint automata.

The scripting language (RSL) is an input language of our verification tool set Vereofy (cf. Section 5.1). RSL provides a convenient way for specifying (dynamically changing) Reo networks. It mainly serves for the specification of the system structure by creating and composing instances of components and connectors. The components and connectors are treated in the exact same way. In what follows, the notion *module* will be used as an umbrella term for components, component connectors (and channels). Beyond instantiation of modules and support of Reo's join operator RSL includes the standard features of imperative languages such as variables, arrays, conditional branching, and loops.

The behavior of modules can be defined in terms of RSL sub-scripts refining a module description into smaller concepts or by providing a constraint automaton definition. For the latter, our verification tool set Vereofy provides a guarded command language CARML which mainly serves to specify the I/O-ports of components (or connectors) and their stepwise "observable" behavior. For this section it is important to recognize that the behavioral description for any CARML module $\mathcal{M}$ together with an interpretation $\mathcal{I}$ for uninterpreted symbols (e.g., for data types, operators, etc.) will be provided in terms of a constraint automaton $\mathcal{A}_{\mathcal{M},\mathcal{I}}$. A detailed description of CARML will be given in Subsection 3.2.2.

As both, RSL scripts and CARML code, can be used to specify the interface behavior of a module, they do serve as a (parameterized) prototype descriptor for modules. E.g., a FIFO channel with capacity k can be specified in terms of a parameterized RSL script that composes k instances of a FIFO1 with capacity one, or by providing a parameterized CARML module implementing the FIFO behavior with the help of an array data structure of size k. Here, k serves a parameter for the capacity of the FIFO. This hybrid modeling approach with CARML and RSL allows nesting of the two specification languages, supports compositional and hierarchical design, modular verification and reusability of components and component connectors.

The behavioral descriptions for Reo's primitive channels are part of the predefined built-in library of Vereofy which also contains some other very helpful components and connectors for typical coordination tasks. User-defined modules can be collected in libraries as well.

3.2.1. Reo scripting language (RSL)

We will now look at the syntax and formal semantics of RSL in detail.

RSL program. On the top-level an RSL program is used to specify the structure and the behavior of a system. An RSL program consists of the following four parts:

(1) a declaration part of an RSL program contains the declarations of global variables and the definition of user-defined data types (like enumerations, arrays and unions over predefined or previously user-defined data types) as well as constants, operators of arity ≥ 1 and predicates,

(2) a list of include-instructions to access modules from a library,

(3) a list of the CARML modules and RSL scripts that might be instantiated in the main system, and optionally

(4) an instantiation of an RSL script (or CARML module) that serves as the main system.

When omitting (4), an non-parameterized module called "main" is required to be contained in (3).

Vereofy supports various data types for messages sent through the network (i.e., as data domain), including finite integer ranges, enumerations and structured data types such as structs and arrays. The ability to use these data types for the local variables of a CARML module yields a convenient way of modeling complex behavior subject to guard conditions on local states and messages.

Uninterpreted data types and signatures. Beside data types with fixed semantics, our languages also allow for uninterpreted symbols for data types, operators and predicates. These can be used as parameters for RSL and CARML specifications which then serve as templates for components or connectors. In other words, the declarations in (3) of CARML modules and RSL scripts may contain template parameters including the user-defined data types, operators (or constants), predicates as declared in (1), and values of some variables. We assume that all the relevant definitions are given in (1) and the value for the parameters will be provided on instantiation. Hence, all dependencies can be resolved before instantiation.

Let $\mathfrak{Sig} = \langle \mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var} \rangle$ be a signature that consists of predefined and uninterpreted symbols. $\mathfrak{Sig}$ yields the basis for terms and atomic propositions that can be used in the instructions of CARML or RSL code. The uninterpreted symbols for data type operators and predicates are then given by the elements of

$$\Theta = \mathfrak{DT} \setminus \mathsf{DT},\ \Omega = \mathfrak{Op} \setminus \mathsf{Op} \text{ and } \Pi = \mathfrak{Pred} \setminus \mathsf{Pred},$$

respectively. The data type symbols $T \in \Theta$ can be seen as placeholders for sets (data types). The operator symbols $op \in \Omega$ and predicate symbols $pr \in \Pi$ are associated with a type.

Recall that the type of an operator symbol $op \in \Omega$ is a tuple $\mathfrak{type}(op) = (T_1, \ldots, T_k, T) \in \mathfrak{DT}^{k+1}$ for some $k \in \mathbb{N}$. It declares that op takes as arguments k elements $v_1, \ldots, v_k$ where v_i is of type T_i and returns an element of type T. That is, op stands for a function $op : T_1 \times \ldots \times T_k \to T$. The number k is called the arity of op.

Similarly, the type of a predicate symbol $pr \in \Pi$ is a tuple $\mathfrak{type}(pr) = (T_1, \ldots, T_k) \in \mathfrak{DT}^k$ for some $k \geq 1$, denoting that pr has to be interpreted by a predicate consisting of tuples $(v_1, \ldots, v_k)$ where v_i is an element of data type T_i.

RSL scripts for networks with static topology. As mentioned in the previous section we will provide limited support for dynamically changing network structures. First we will explain the static fragment of RSL before giving a brief overview on how to specify dynamic connectors in RSL. The schema for the RSL script of a Reo circuit without dynamic reconfiguration is shown in Figure 3.12. The shorthand notation

$$\text{“}\mathsf{type} : \Theta, \mathsf{op} : \Omega, \mathsf{pred} : \Pi\text{”}$$

refers to a list where all uninterpreted symbols $S \in \Theta \cup \Omega \cup \Pi$ are encountered together with the corresponding keyword type, op or pred and their types in case of the operator and predicate symbols. Similarly, the notation “var : Υ” stands short for an enumeration of all variables in Υ together with the keyword var and their types. All variables in Υ are passed according to the concept “call-by-value”.

CIRCUIT $\mathcal{C}\langle \mathsf{type} : \Theta, \mathsf{op} : \Omega, \mathsf{pred} : \Pi, \mathsf{var} : \Upsilon \rangle \{$
    stmt; // stepwise construction of a Reo circuit
    interface_decl // declaration of the interface ports of the Reo circuit
$\}$

Figure 3.12: Schema of an RSL circuit

The body of an RSL circuit consists of a *statement* that describes the stepwise construction of a Reo circuit and an *interface declaration* where the exported I/O-ports are specified.

Statements. The statement in the body of an RSL circuit is build by basic operations and control flow instructions (sequential composition, conditional branching and for-loops). The abstract syntax for statements is given by the grammar

$$\begin{array}{rcl} \textit{stmt} & ::= & \textit{instantiation} \;\big|\; \textit{Reo_operation} \;\big|\; \textit{assignment} \;\big|\; \textit{stmt}\,;\, \textit{stmt} \;\big| \\ & & \mathsf{if}\ (\textit{bexpr})\ \{\textit{stmt}\}\ \mathsf{else}\ \{\textit{stmt}\} \;\big|\; \mathsf{for}\ (i = j, \ldots, k)\ \{\textit{stmt}\} \end{array}$$

Instantiation. The instantiation of a module is performed via instructions of the form

$$\mathsf{new}\ \texttt{module_template}\langle\Theta', \Omega', \Pi', \Upsilon'\rangle(A_1, \ldots, A_r; B_1, \ldots, B_s)$$

where $\Theta', \Omega', \Pi', \Upsilon'$ are lists of data types, operators, predicates and variables or constants that provide meanings for the parameters in the CARML or RSL code for the module template. The elements of $\Theta', \Omega', \Pi', \Upsilon'$ have to be contained in the signature of $\mathcal{C}$ (cf. Definition 2.3.1) which is given by $\mathfrak{Sig}_{\mathcal{C}} = \langle\mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var}_{\mathcal{C}}\rangle$ where

$$\mathfrak{DT} = \mathsf{DT} \cup \Theta,\ \mathfrak{Op} = \mathsf{Op} \cup \Omega,\ \mathfrak{Pred} = \mathsf{Pred} \cup \Pi$$

and $\mathfrak{Var}_{\mathcal{C}} = \Upsilon$ and the following obvious side-constraints.

1. If $\mathsf{type} : \Theta$ stands for $\mathsf{type} : T_1, \ldots,\ \mathsf{type} : T_n$ then Θ' must be a list $U_1, \ldots, U_n$ consisting of n elements of elements in $\mathfrak{DT}$.
2. If $\mathsf{op} : \Omega$ encounters m operator symbols then Ω' must be a list of m elements in $\mathfrak{Op}$, and if the i-th element in $\mathsf{op} : \Omega$ is $\mathsf{op} : (T_{i_1}, \ldots, T_{i_k}, T_j)\ f$ then the i-th element of Ω' has to be an element $f' \in \mathfrak{Op}$ of the type $(U_{i_1}, \ldots, U_{i_k}, U_j)$.

Analogous conditions are required for Π' and Υ'.

Optionally, the list of meanings for the uninterpreted symbols in the template for some module $\mathcal{M}$ is followed by a list $(A_1, \ldots, A_r; B_1, \ldots, B_s)$ of names for the source and sink ports of $\mathcal{M}$. Thus, A_i will serve as the name for the i-th source port of the generated instance of the module, and B_j as the name for the j-th sink port of that instance. The names A_i and B_j have to be pairwise distinct. They can be fresh names or can represent an existing location, in which case consistency of the message-types is required. If A_i is an already existing location then the message-types of A_i and the i-th source port of $\mathcal{M}$ must agree, and from now on location A_i is joined with the i-th source port of the created instance of $\mathcal{M}$. Hence, this implicit join creates a new standard Reo node with merge-and-replicate-semantics. Analogous conditions are required for $B_1, \ldots, B_s$ and the sink ports of the created instance of $\mathcal{M}$.

The second type of instantiation is the explicit creation of a *Reo node*. The effect of an instantiation of a standard Reo node via the instruction

$$\mathsf{NODE}\langle\mathsf{type} : T\rangle\ N$$

is the creation of a fresh Reo node N without any channel-end or I/O-port associated. The message-type of N is $T \in \mathfrak{DT}$ which indicates that only data items of type T can flow through N. Analogously a Reo route node of type $T \in \mathfrak{DT}$ can be created by the statement

$$\mathsf{ROUTE_NODE}\langle\mathsf{type} : T\rangle\ N.$$

Reo operations. RSL supports Reo's main operations for the composition of complex circuits. The join operation $\mathsf{join}(N_1, \ldots, N_n)$ in RSL takes a list $N_1, \ldots, N_n$ of at least two I/O-ports or Reo nodes of the same message-type $T \in \mathfrak{DT}$ as arguments. Additionally we

require that all Reo nodes $N_i \in \{N_1, \ldots, N_n\}$ have the same routing behavior, i.e., either all Reo nodes are Reo route nodes or Reo standard nodes.

The join creates a new Reo node N of message-type T where all I/O-ports and channel-ends of $N_1, \ldots, N_n$ are combined. If all $N_i \in \{N_1, \ldots, N_n\}$ are channel-ends rather than Reo nodes the created Reo node will have merge-and-replicate behavior. Otherwise N will inherit the behavior of the Reo nodes which served as parameter of the join statement.

Script variables and assignments. For the sequential composition carried out by an RSL script, the I/O-ports, Reo nodes created by a join operation as well as the result of an instantiation (either a Reo node or module) can be stored into local script variables. Script variables can also be used to hold values of predefined data types (typically an integer value). The script variables are "dynamically typed" and do not have to be declared in advance. An assignment for a script variable sv has the syntax $sv := \mathcal{V}$ where

$$\mathcal{V} ::= \textit{instantiation} \mid \mathsf{join}(N_1, \ldots, N_n) \mid \textit{expression} \mid \textit{script_variable}$$

and $expression$ is a term over the signature induced by the predefined data types and the variables $V \in \Upsilon$ (typically arithmetic expressions). Script variables are dynamically sized arrays, with sv being shorthand for $sv[0]$. A script variable sv referring to an instantiated module $\mathcal{M}$ provides access to the interface I/O-ports via $sv.\mathsf{in}[i]$ and $sv.\mathsf{out}[j]$. An RSL circuit can refer to its own source and sink ports via $\mathsf{in}[i]$ and $\mathsf{out}[j]$ (see the explanations below). To hide dataflow locations they can be made anonymous by assigning the value NULL to all script variables holding a reference to the dataflow location, i.e., $sv := \mathsf{NULL}$. The hiding of dataflow locations will be done automatically at the end of the composition phase. The presented version of RSL does not contain an explicit hiding operator. Instead hiding is inherent in the semantics of RSL circuits.

Control flow instructions. The control flow features (sequential composition, conditional and repetitive commands) have the standard meaning. These and the script variables serve for the stepwise construction of a Reo circuit, and should not be confused with the operational behavior of the network given by the constraint automaton for the Reo circuit that results from executing the RSL script. The control flow statements make use of Boolean expressions (*bexpr*) that impose conditions on the values of script variables and variables in Υ. In the sloppy notation provided for the syntax of for-loops, we assume that i is an integer script variable and j and k are either integer script values or constants.

Interface declaration. In the schema sketched in Figure 3.12, the body of an RSL circuit ends with a definition of the nodes that are exported to the higher level as source and sink ports. The syntax to specify that the i-th source port is A_i and the j-th sink port is B_j is the following.

$$\mathsf{in}\text{: } A_1; \ldots; \mathsf{in}\text{: } A_r; \mathsf{out}\text{: } B_1; \ldots; \mathsf{out}\text{: } B_s$$

It is required that the A_i's are sources and the B_j's are sinks (I/O-ports or Reo nodes) that have been defined in *stmt* via an assignment or module instantiation. Furthermore, the A_i's and B_j's are required to be pairwise distinct. Alternatively, one may depart from the schema in Figure 3.12 and define the interface ports within *stmt* via the references $\mathsf{in}[i]$ and $\mathsf{out}[j]$, either assignments:

$$\mathsf{in}\,[i] := \ldots \quad \text{and} \quad \mathsf{out}\,[j] := \ldots$$

or by instantiations:

$$\mathsf{new}\;\texttt{module_template}\;\langle\ldots\rangle\;(\,\ldots,\mathsf{in[i]},\ldots;\ldots,\mathsf{out[j]}\,,\ldots).$$

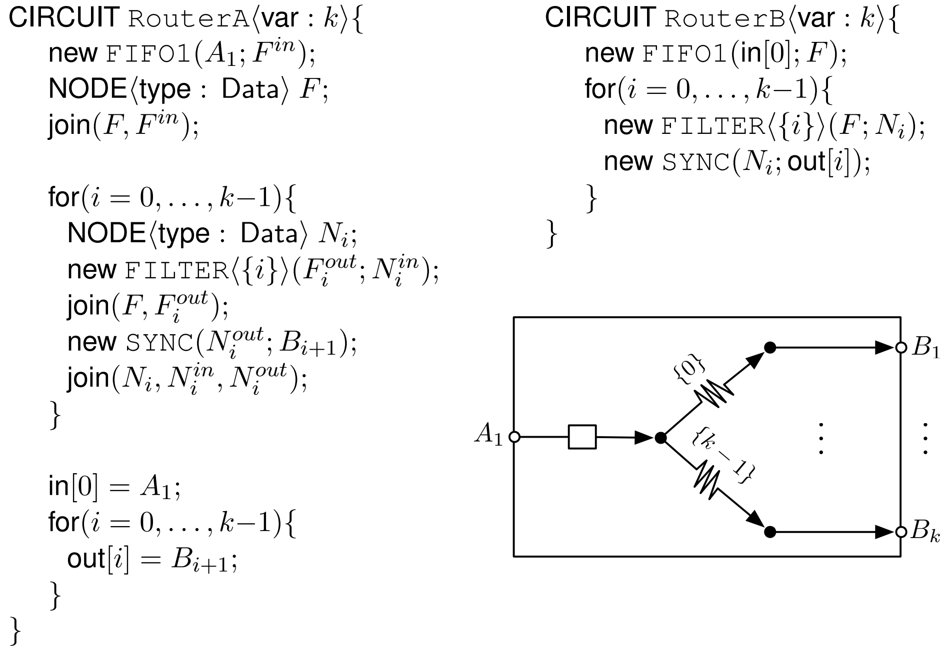
The indices i and j for the exported source and sink ports have to be consecutive starting with 0. If there are two or more assignments for, e.g., $\mathsf{in}\,[i]$ then the last one declares the i-th source port.

Remark 3.2.1. Hiding will be applied automatically and recursively when executing an RSL script. All internal Reo nodes, ports and channel-ends inside of modules that are instantiated by an RSL main program will be hidden. For the dataflow locations on the top most level the hiding depends on whether or not there exists a script variable holding a reference to the dataflow location at the end of the RSL script. Reo nodes, ports and channel-ends which no script variable holds a reference to are called *anonymous*. All anonymous dataflow locations will automatically be hidden when the end of an RSL script is reached.

■

Example 3.2.2 (RSL script for a simple connector)**.** The RSL scripts shown in Figure 3.13 both define a prototype for the component connector presented in Example 3.1.5. The definitions are parameterized in the number k of output ports, Sync and filter channels being created. For this the parameter k has to be an integer type $T \in \mathfrak{DT}$. For the data domain we assume that $\{0, \ldots, k\} \subseteq \mathsf{Data}$.

Basically, the two scripts depicted in Figure 3.13 are two variants to compose the same component connector. The scrips consecutively create all channels and nodes when being executed for a concrete value of $k \in \mathbb{N}$. At the end of the scripts follows the interface declaration. The parameter passed to the filter channels is denoted as a singleton set containing the data value which is allowed to pass the filter channel. Formally, filter conditions can be seen as user-defined unary predicates $pr \in \Pi$ which evaluate to true for all the data values that belong to the current set.



```
CIRCUIT RouterA⟨var : k⟩{
  new FIFO1(A_1; F^in);
  NODE⟨type : Data⟩ F;
  join(F, F^in);

  for(i = 0, ..., k−1){
    NODE⟨type : Data⟩ N_i;
    new FILTER⟨{i}⟩(F_i^out; N_i^in);
    join(F, F_i^out);
    new SYNC(N_i^out; B_{i+1});
    join(N_i, N_i^in, N_i^out);
  }

  in[0] = A_1;
  for(i = 0, ..., k−1){
    out[i] = B_{i+1};
  }
}
```

```
CIRCUIT RouterB⟨var : k⟩{
  new FIFO1(in[0]; F);
  for(i = 0, ..., k−1){
    new FILTER⟨{i}⟩(F; N_i);
    new SYNC(N_i; out[i]);
  }
}
```


Figure 3.13: Two RSL scripts composing the same connector

The script on the right of Figure 3.13 makes use of the fact that channel-ends or ports which do appear more than once are joined implicitly and do automatically form a Reo node. Thus, neither the explicit definition of the Reo nodes nor the join statements are required. Moreover, the definition of the interface ports has moved from the *interface_decl* part into the *stmt* part (compared with the schema depicted in Figure 3.12).

Although the two circuits created by the scripts depicted in Figure 3.13 will have the same interface behavior, they are not equivalent when instantiated at the top most level of an RSL program, because the left script keeps references to all the channel-ends. Hence, these will not be hidden.

■

Based on the definition of the data-dependent router above, we may now form an RSL main program using the router as a connector for some reader and writer components.

Example 3.2.3 (RSL script for a simple connector)**.** The RSL script shown in Figure 3.14 composes the Reo network consisting of a writer and two readers from the built-in library of Vereofy and an instance of the router with size two.

■

```
include "builtin"
include "routers.rsl"

REPLACE("SomeRouter","RouterB");

CONST size = 2;
TYPE mytype_t = int(0, ..., 2 · size);

TYPE Data = mytype_t;

CIRCUIT main{
 TheWriter     =  new Writer;
 TheRouter     =  new someRouter⟨size⟩;
                  new SYNC(TheWriter.out[0]; TheRouter.in[0]);

 for(i = 0, ..., size−1){
   TheReader[i]  =  new Reader;
                    new SYNC(TheRouter.out[i]; TheReader[i].in[0]);
 }
}
```

Figure 3.14: An RSL main program to compose the Reo network of Example 3.1.5

The above RSL main program illustrates some important language features of RSL:

(1) the include statements in the beginning of the main program loads the definitions from built-in library as well as the user defined library containing the RSL circuit definitions for the routers. (2) The replace statement can be used to exchange one prototype definition by another at each instantiation. Here, the effect will be that all instances of "someRouter" will be instances of "RouterB". Replace statements can be used for abstraction and refinement as specifications can be exchanged with others. The replacement requires the two prototypes to have the same interface, i.e., they have the same number of input and output ports and the type of the interface ports agrees as well. (3) The CONST statement defines the value of the constant named "size" which will serve as a parameter for the size of the router structure. (4) User-defined types can be specified by using the TYPE statement and (5) the data domain Data can have any of the supported data types of Vereofy. In the above program the domain will be of type "mytype_t" which corresponds to the finite set $\{0, \ldots, 2 \cdot \text{size}\}$. (6) Beyond simple arithmetic operators for unsigned integers, as it is used here to compute the size of the data domain, Vereofy provides support for some built-in functions and additional operators for Booleans and unsigned integers [BKK12]. (7) The names of I/O-ports are optional when creating new instances of components. (8) Instead, RSL script variables can be used to hold references to component or channel instances. (9) These references then yield a way to access the interface ports of components via their in and out arrays. (10) As all ports are connected, the circuit does not contain any interface definition and thus the created network will be a closed system, to which neither other components nor the environment can connect.

Dynamic reconfiguration. RSL provides support for specifying component connectors with multiple network topologies. The interface of a dynamic connector $\mathcal{C}$ contains a special input port $\mathcal{C}$.reconf, called *reconfiguration port*.

```
CIRCUIT C ⟨type : Θ, op : Ω, pred : Π, var : Υ⟩ {
   stmt;                    // construction of a common sub-circuit
   interface_decl           // declaration of the exported source/sink ports

   TOPO(id1) = {stmt1}      // additional sub-circuit for topology id1
      ⋮
   TOPO(idt) = {stmtt}      // additional sub-circuit for topology idt
}
```

Figure 3.15: Schema for RSL scripts with dynamic reconfiguration

The schema of RSL scripts with dynamic reconfigurations is shown in Figure 3.15. The parameter list is the same as before. The body of a dynamic RSL circuit $\mathcal{C}$ consists of statement *stmt* that specifies a common (static) sub-circuit $\mathcal{C}_{stmt}$ of all network topologies, followed by the interface declaration and instructions of the form

$$\mathsf{TOPO}(id_i)\{stmt_i\}$$

for $i = 1, \ldots, t$. Here, id_i denotes an identifier for the i-th network topology and $stmt_i$ is a statement. The network topology with an identifier id_i is generated by the composite statement $stmt\,;\, stmt_i$, resulting in the (static) Reo circuit $\mathcal{C}'_i$. The interface declaration specifies the input and output ports of $\mathcal{C}$, except for the reconfiguration port *circ*.reconf of the instances of the circuit $\mathcal{C}$ in Figure 3.15. No special declaration is required for the reconfiguration port. The assignments in the interface declaration can only refer to the nodes and ports that appear in *stmt*, but not to entities created in $stmt_1, \ldots, stmt_t$. Any other RSL circuit *circ′* creating an instance *circ* of the circuit $\mathcal{C}$ (cf. Figure 3.15) can access the reconfiguration port *circ*.reconf as any port in the interface of *circ*. Hence, any module that is connected in *circ′* to the reconfiguration port *circ*.reconf can serve as a driver and trigger switches in the topology of *circ* by sending the identifier of the new topology.

Example 3.2.4 (Simple dynamic Reo network)**.** Figure 3.16 depicts a Reo network consisting of a writer and two readers which communicate via a dynamic router circuit as it could be composed by the following RSL script.

```
CIRCUIT DynamicRouter⟨var : k⟩{
 new FIFO1(in[0]; F);

 for(i = 1, . . . , k){
   out[i−1] := NODE⟨type : Data⟩ B_i;
   TOPO(id_i) = {new SYNC(F; N_i); new SYNC(N_i; B_i); }
 }
}
```

Additionally, there can be a driver component connected at the reconfiguration port of the connector triggering changes in the network topology. In case the reconfiguration port is not connected with a driver, the composed system is assumed to be an open system where nondeterministic writes on open ports can occur at any time.

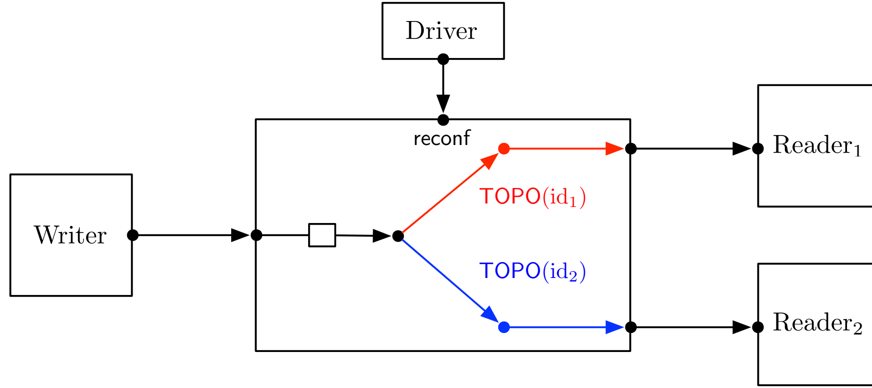


Figure 3.16: A dynamic router used in a simple Reo network

The driver sends either id_1 or id_2 on the reconfiguration port of the dynamic router. The router initially starts in the topology with identifier id_1 and switches according to the received reconfiguration signals. The result is that in the topology with identifier id_i the Reader_i will receive all data stored in the buffer.

■

The above Reo network constitutes a perfect example for exogenous coordination, as the communication medium has been exchanged in respect of Example 3.1.5 without changing the specification of the reader and the writer.

We will now look at the semantics of RSL, which also relies on constraint automata.

Semantics of static RSL circuits. Let $\mathcal{C}$ be an RSL circuit with the parameters Θ, Ω, Π and Υ as in Figure 3.12. To provide an operational semantics for circuit $\mathcal{C}$, we will

1. fix an interpretation $\mathcal{I}$ for the symbols in the parameter list of $\mathcal{C}$ (which yields an interpretation for the induced signature $\mathfrak{Sig}_{\mathcal{C}}$) and then construct a Reo circuit $\mathcal{R}_{\mathcal{C},\mathcal{I}}$ for $\mathcal{C}$ by means of the instructions given in *stmt*.

2. construct the extended constraint automata for all individual parts of $\mathcal{R}_{\mathcal{C},\mathcal{I}}$ either directly for Reo nodes, recursively in case the behavioral description of the entity is given in terms of another RSL script, or by applying the machinery described in the next section if the behavior is given as a CARML module.

   The Reo circuit $\mathcal{R}_{\mathcal{C},\mathcal{I}}$ is obtained by executing the RSL script given by the instructions in the body of $\mathcal{C}$. When instantiating a module the meanings of the uninterpreted data types, operator or predicate symbols are taken according to $\mathcal{I}$.

The instantiation of a CARML module $\mathcal{M}(\Theta_0, \Omega_0, \Pi_0, \Upsilon_0)$ with n sources and m sinks via the instruction

$$comp := \mathsf{new}\, \mathcal{M}\langle \Theta', \Omega', \Pi', \Upsilon' \rangle (D_1, \dots, D_n, E_1, \dots, E_m)$$

means binding an instance *comp* of $\mathcal{M}$ where the i-th source of *comp* is identified with the possibly already existing node D_i and the j-th sink of *comp* with E_j. In the Reo circuit $\mathcal{R}_{\mathcal{C},\mathcal{I}}$, *comp* is viewed as a black-box component. However, by applying the algorithm of [BSAR06] (extended to handle global variables) to construct a constraint automaton from $\mathcal{R}_{\mathcal{C},\mathcal{I}}$ we use a constraint automaton $\mathcal{A}_{comp,\mathcal{I}}$ as specification for the behavioral interface of that instance *comp* of $\mathcal{M}$. Automaton $\mathcal{A}_{comp,\mathcal{I}}$ can be obtained as follows:

Let $\mathcal{I}_0$ be the interpretation that arises from $\mathcal{I}$ by the substituting Θ_0 with Θ' (i.e., if the i-th data type symbol in Θ_0 is T and the i-th element of Θ' is U then $T^{\mathcal{I}_0} = U^{\mathcal{I}}$) and substituting Ω_0 with Ω', Π_0 with Π', and Υ_0 with Υ'.

We now regard the constraint automaton $\mathcal{A}_{\mathcal{M},\mathcal{I}_0}$ and replace the i-th source port of $\mathcal{M}$ with D_i and the j-th sink port of $\mathcal{M}$ with E_j in the dataflow vocabulary of $\mathcal{A}_{\mathcal{M},\mathcal{I}_0}$ and the concurrent I/O-operations that appear as labels for the transitions of $\mathcal{A}_{\mathcal{M},\mathcal{I}_0}$. The resulting automaton is $\mathcal{A}_{comp,\mathcal{I}}$.

The meaning of an instantiation of another RSL script $\mathcal{C}'$ via the instruction

$$comp := \mathsf{new}\, \mathcal{C}'\langle \Theta', \Omega', \Pi', \Upsilon' \rangle (D_1, \dots, D_n, E_1, \dots, E_m)$$

is analogous. It has the effect of including the Reo circuit associated with the generated instance of $\mathcal{C}'$.

3. Finally, we apply the product operator for constraint automata to output one large constraint automaton $\mathcal{A}_{\mathcal{C},\mathcal{I}}$ for $\mathcal{R}_{\mathcal{C},\mathcal{I}}$ with associated dataflow vocabulary $\mathfrak{Voc}_{\mathcal{C}}$ which is defined according to the interface declaration, i.e.,

$$\mathfrak{Voc}_{\mathcal{C}} = (\mathcal{N}_{\mathcal{C}}, \mathcal{N}_{\mathcal{C}}^{\text{src}}, \mathcal{N}_{\mathcal{C}}^{\text{snk}}, \lambda_{\mathcal{C}})$$

where $\mathcal{N}_{\mathcal{C}}^{\text{src}} = \{A_1, \dots, A_r\}$ and $\mathcal{N}_{\mathcal{C}}^{\text{snk}} = \{B_1, \dots, B_s\}$ if the interface declaration specifies $A_1, \dots, A_s$ as source ports and $B_1, \dots, B_s$ as sink ports. The function $\lambda_{\mathcal{C}}$ is the obvious one and assigns the message-type of A_i to the i-th source port ($\mathsf{in}[i-1]$) and the message-type of B_j to the j-th sink port ($\mathsf{out}[j-1]$). The set of all observable locations of $\mathcal{C}$ is

$$\mathcal{N}_{\mathcal{C}} = \mathcal{N}_{\mathcal{C}}^{\text{src}} \cup \mathcal{N}_{\mathcal{C}}^{\text{snk}} \cup \mathcal{N}_{\mathcal{C}}^{\text{int}}$$

where the set $\mathcal{N}_{\mathcal{C}}^{\text{int}}$ holds the names for all internal dataflow locations that have not been hidden during the composition. The set $\mathcal{N}_{\mathcal{C}}^{\text{int}}$ will be empty if $\mathcal{C}$ is not on the top-level of the RSL program as for those circuits all internals are hidden automatically.

The procedure above results in an extended constraint automaton $\langle \overline{\mathcal{A}_{\mathcal{C},\mathcal{I}}}, \mathfrak{Voc}_{\mathcal{C}} \rangle$, where $\overline{\mathcal{A}_{\mathcal{C},\mathcal{I}}}$ arises from $\mathcal{A}_{\mathcal{C},\mathcal{I}}$ by applying the structure-preserving hide operator. Please notice that applying the hide operator removes internal ports from the port-set and it may also effect the set of quiescent states, as the hiding may introduce new $c_{\emptyset}$-transitions to some states which were quiescent in $\mathcal{A}_{\mathcal{C},\mathcal{I}}$ and will hence not be quiescent in $\overline{\mathcal{A}_{\mathcal{C},\mathcal{I}}}$.

Please note that we provide the operational semantics for RSL circuits (static as well as dynamic) in terms of constraint automata $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q} \rangle$ without labeling, as the a precise definition of the labeling of states is irrelevant for the semantic of RSL. Instead we will assume that the set of atomic propositions of the constraint automaton $\mathcal{A}_{\mathcal{C},\mathcal{I}}$ contains all quantifier-free formulas over signature $\mathfrak{Sig}_{\mathcal{C}}$. Additional atomic propositions, which may serve as shortcut notation, can be defined in CARML and will be lifted to the circuit level via the constraint automaton product construction.

Semantics of dynamic RSL circuits. For dynamic RSL circuit the dataflow vocabulary $\mathfrak{Voc}_{\mathcal{C}}$ is defined according to the interface declaration and internal dataflow locations which have not been hidden, completed with the reconfiguration port $\mathcal{C}.\mathsf{reconf}$ which is a source port of message-type $\{id_1, \ldots, id_t\}$. That is, the dataflow vocabulary

$$\mathfrak{Voc}_{\mathcal{C}} = (\mathcal{N}_{\mathcal{C}}, \mathcal{N}_{\mathcal{C}}^{\text{src}}, \mathcal{N}_{\mathcal{C}}^{\text{snk}}, \lambda_{\mathcal{C}})$$

with $\mathcal{N}_{\mathcal{C}} = \mathcal{N}_{\mathcal{C}}^{\text{src}} \cup \mathcal{N}_{\mathcal{C}}^{\text{snk}} \cup \mathcal{N}_{\mathcal{C}}^{\text{int}}$ is as for static circuits, but here

$$\mathcal{N}_{\mathcal{C}}^{\text{src}} = \{\mathcal{C}.\mathsf{reconf}, A_1, \ldots, A_r\} \text{ with } \mathfrak{type}(\mathcal{C}.\mathsf{reconf}) = \{id_1, \ldots, id_t\}.$$

We use the aforementioned construction for each constraint automaton (without labeling)

$$\mathcal{A}_i' = \mathcal{A}_{stmt;stmt_i,\mathcal{I}} = \left\langle Q^{(i)}, \mathcal{N}, \longrightarrow_i, Q_0^{(i)}, \mathcal{Q}^{(i)} \right\rangle$$

for the circuit $\mathcal{C}_i'$ induced by the statement *stmt* ; $stmt_i$. The port-set $\mathcal{N} = \mathcal{N}_{\mathcal{C}} \setminus \{\mathcal{C}.\mathsf{reconf}\}$ consists of all interface ports of the circuit $\mathcal{C}$ except the reconfiguration port. The states in $Q^{(i)}$ can be written in the form $\langle q, q^{(i)} \rangle$ where q stands for a state in the constraint automaton $\mathcal{A}_{stmt,\mathcal{I}}$ for the (static) common sub-circuit $\mathcal{C}_{stmt}$ of all circuits $\mathcal{C}_1', \ldots, \mathcal{C}_t'$ and $q^{(i)}$ a state of constraint automaton $\mathcal{A}_{stmt_i,\mathcal{I}}$ for the sub-circuit induced by $stmt_i$. W.l.o.g. we can assume that $Q^{(i)} \cap Q^{(j)} = \emptyset$ for $1 \leq i < j \leq t$.

The constraint automaton (without labeling) $\mathcal{A}_{\mathcal{C},\mathcal{I}} = \langle Q, \mathcal{N}_{\mathcal{C}}, \longrightarrow, Q_0, \mathcal{Q} \rangle$ for the dynamic connector $\mathcal{C}$ is then obtained by combining $\mathcal{A}_1', \ldots, \mathcal{A}_t'$ as follows:

- The state space Q of $\mathcal{A}_{\mathcal{C},\mathcal{I}}$ is the disjoint union of the state spaces of $\mathcal{A}_1', \ldots, \mathcal{A}_t'$, that is, $Q = Q^{(1)} \cup \ldots \cup Q^{(t)}$.
- The set Q_0 of initial states is the set of all states $\langle q, q^{(i)} \rangle$ where q is an initial state in the constraint automaton $\mathcal{A}_{stmt,\mathcal{I}}$ for *stmt* and $q^{(i)}$ an initial state in the constraint automaton $\mathcal{A}_{stmt_i,\mathcal{I}}$ for $stmt_i$.
- The transitions $\longrightarrow$ are given by the following two rules, where the first stands for the receipt of the signal to switch to the j-th network topology and the second rule stands for the execution of a concurrent I/O-operation in the i-th topology:

$$\frac{q^{(j)} \text{ initial state of } \mathcal{A}_{stmt_j,\mathcal{I}} \text{ for } stmt_j}{\langle q, q^{(i)} \rangle \xrightarrow{\mathcal{C}.\mathsf{reconf}?id_j} \langle q, q^{(j)} \rangle} \qquad \frac{\langle q, q^{(i)} \rangle \xrightarrow{c}_i \langle q, p^{(i)} \rangle}{\langle q, q^{(i)} \rangle \xrightarrow{c} \langle q, p^{(i)} \rangle} \tag{3.1}$$

where $\mathcal{C}.\mathsf{reconf}?id_j$ denotes the unique concurrent I/O-operation c with

$$\mathsf{active}(c) = \{\mathcal{C}.\mathsf{reconf}\} \text{ and } c(\mathcal{C}.\mathsf{reconf}) = id_j.$$

- The set of quiescent states is defined as $\mathcal{Q} \stackrel{\text{def}}{=} \mathcal{Q}^{(1)} \cup \ldots \cup \mathcal{Q}^{(t)}$

3.2.2. Constraint automata reactive module language (CARML)

The input languages of Vereofy for the specification of components (and connectors) in terms of constraint automata is a guarded command language, called CARML (constraint automata reactive module language), that describes the transitions of constraint automata in a symbolic way, i.e., by means of Boolean conditions on the states and the enabled concurrent I/O-operations. CARML provides a convenient way to specify the component interfaces and to provide a high-level description of the operational behavior of components. CARML supports channel-based message passing and communication over shared variables. The latter is irrelevant for exogenous coordination, but can be useful to incorporate the coordination primitives of an endogenous approach e.g. for existing systems where the coordination protocol is given in an imperative language. In this case, modeling the protocol by means of the coordination language can be much harder than providing a CARML specification. CARML is even expressive enough to specify complex component connectors. To ease the automatic translation of CARML specifications into a compact internal BDD-based representation, we adapted some concepts of reactive modules [AH99] for the syntax of CARML modules.

Standard data types such as Boolean, finite integer ranges, arrays, unions and enumerations together with the usual operators and predicates on them can be used as data types for variables and message-types for I/O-ports. The set of data types, operators and predicates with fixed semantics is denoted as for RSL by DT, Op and Pred. CARML modules can also use uninterpreted symbols for data types, operators and predicates. That is, a CARML module $\mathcal{M}$ can be parameterized by a set Θ of data types and a set Ω of operator symbols, a set Π of predicate symbols, and a set Υ of variables with types in $\mathfrak{DT} = \mathsf{DT} \cup \Theta$.

The general schema of a CARML module is shown in Figure 3.17.

MODULE $\mathcal{M}\langle \mathsf{type} : \Theta, \mathsf{op} : \Omega, \mathsf{pred} : \Pi, \mathsf{var} : \Upsilon \rangle$ {

// interface declaration: source ports

in : $T_1^{\mathsf{in}}\ A_1$;

$\vdots$ // A_i with message-types $T_i^{\mathsf{in}} \in \mathfrak{DT}$

in : $T_k^{\mathsf{in}}\ A_r$;

// interface declaration: sink ports

out : $T_1^{\mathsf{out}}\ B_1$;

$\vdots$ // B_i with message-types $T_i^{\mathsf{out}} \in \mathfrak{DT}$

out : $T_\ell^{\mathsf{out}}\ B_s$;

// definition of local variables X_i with data types $T_i^{\mathsf{var}} \in \mathfrak{DT}$

// with initial value $\mathit{init_value}_i \in T_i^{\mathsf{var}}$ (optional)

var : $T_1^{\mathsf{var}}\ X_1$ init := $\mathit{init_value}_1$;

$\vdots$

var : $T_\ell^{\mathsf{var}}\ X_\ell$ init := $\mathit{init_value}_\ell$;

// definition of atomic propositions a_i

ap : $a_1 \leftrightarrow \mathit{state_guard}_{a_1}$;

$\vdots$

ap : $a_m \leftrightarrow \mathit{state_guard}_{a_m}$;

// transition definitions

$\mathit{state_guard}_1 \ {-}[\ \mathit{I/O_guard}_1\ ]{\rightarrow}\ \mathit{state_assignment}_1$;

$\vdots \qquad \vdots \qquad \vdots$

$\mathit{state_guard}_n \ {-}[\ \mathit{I/O_guard}_n\ ]{\rightarrow}\ \mathit{state_assignment}_n$;

}

Figure 3.17: Schema of a CARML module

Interface declaration. It consists of a (possibly empty) parameter list, the interface declaration where the source and sink ports of a component and its local variables are defined followed by the transition definitions specifying the behavioral interface. The shorthand notation

$$\text{“}\mathsf{type} : \Theta, \mathsf{op} : \Omega, \mathsf{pred} : \Pi\text{”}$$

in Figure 3.17 refers to a list where all uninterpreted symbols $S \in \Theta \cup \Omega \cup \Pi$ are encountered together with the corresponding keyword type, op or pred and their types in case of the operator and predicate symbols. Similarly, the notation $\mathsf{var} : \Upsilon$ stands short for an enumeration of all variables in Υ together with the keyword var and their types. All variables in Υ are passed according to the concept “call-by-value”.

Let $\mathcal{M}$ be the name of the CARML module in Figure 3.17. The data types with fixed semantics together with the parameters Θ, Ω, Π, Υ and the set $\mathfrak{Var}_{\mathcal{M}}$ of variables that can be used in $\mathcal{M}$ (see below) constitute the signature of $\mathcal{M}$ which is given by

$$\mathfrak{Sig}_{\mathcal{M}} = (\mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var}_{\mathcal{M}})$$

where $\mathfrak{DT} = \mathsf{DT} \cup \Theta$. $\mathfrak{Op} = \mathsf{Op} \cup \Omega$ and $\mathfrak{Pred} = \mathsf{Pred} \cup \Pi$.

Local variables. The variables that can be used in a CARML module $\mathcal{M}$ are *local variables* and the variables in Υ. The local variables together with their types have to be listed in the declaration part of $\mathcal{M}$. Thus, a CARML module conforming to the schema shown in Figure 3.17 has the local variables $X_1, \ldots, X_\ell$. The type of local variable X_i is T_i^{var} which has to be an element of $\mathfrak{DT}$. The specification of an initial value for the local variables is optional.

Global variables. With our hybrid modeling approach, where CARML specifications can be embedded in the scripting language RSL (cf. Subsection 3.2.1), CARML modules can also use *global* or *shared* variables which have to be declared in the RSL (main) program. The treatment of shared variables requires some additional rules in the constraint automaton product to ensure mutually exclusive writes. As global variables are an additional concept outside the scope of our exogenous coordination setting we will not go into the technical details for the treatment of global variables.

The interface declaration of a CARML module $\mathcal{M}$ following the schema shown in Figure 3.17 will induce the port-set $\mathcal{N} = \{A_1, \ldots, A_r, B_1, \ldots, B_s\}$ of a constraint automaton

$$\mathcal{A}_{\mathcal{M},\mathcal{I}} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L \rangle,$$

where $\mathcal{I}$ is an interpretation for the uninterpreted symbols $S \in \Theta \cup \Omega \cup \Pi$. The type of source port A_i is T_i^{in}, while sink port B_j is of type T_j^{out}. Again, the types T_i^{in} and T_j^{out} are elements of $\mathfrak{DT}$. The evaluations for the local variables of a module will form the state space Q of the resulting constraint automaton $\mathcal{A}_{\mathcal{M},\mathcal{I}}$, while the set of initial states Q_0 is defined according to the provided initial values for each local variable V_i for $1 \leq i \leq \ell$. The set of quiescent states in $\mathcal{A}_{\mathcal{M},\mathcal{I}}$ is defined as $\mathcal{Q} \overset{\text{def}}{=} \{q \in Q \mid c_\emptyset \notin \mathsf{CIO}(q)\}$.

Atomic propositions. The formal notion of atomic propositions over $\mathfrak{Sig}_{\mathcal{M}}$ are predicates of the form $pr(t_1, \ldots, t_k)$ where $\mathfrak{type}(pr) = (T_1, \ldots, T_k)$ and $t_1, \ldots, t_k$ are terms over $\mathfrak{Sig}_{\mathcal{M}}$ such that t_i is of type T_i. For the standard set AP of atomic propositions for a CARML module $\mathcal{M}$, we assume that it contains all boolean expressions of predicates $pr(t_1, \ldots, t_k)$ over $\mathfrak{Sig}_{\mathcal{M}}$. Additionally, we allow to enrich the standard set of atomic propositions (cf. Figure 3.17). This allows labeling of states according to the evaluation of their local variables. These atomic propositions serve as shortcut notations useful for transition definitions and within formulas.

For a CARML module as the one depicted in Figure 3.17 the set $\{a_1, \ldots, a_m\}$ will additionally be included in the set of atomic propositions AP. For $a_1, \ldots, a_m$ new predicate symbols are being introduced in the signature $\mathfrak{Sig}_{\mathcal{M}}$. The labeling function is defined according to the definition provided in the CARML module:

$$a_i \in L(q) \text{ iff } \textit{state_guard}_{a_i} \text{ holds in state } q \in Q.$$

Formally, a state guard is a (possibly empty) conjunction of atomic propositions, i.e., predicates $pr(t_1, \ldots, t_k)$ over $\mathfrak{Sig}_{\mathcal{M}}$. The transition definitions will yield the transition relation $\longrightarrow$ as follows.

Transition definitions. Each *transition definition* consists of local conditions on the current state (a state guard), conditions on the concurrent I/O-operations to be fired (an I/O-guard) and the effect of firing such an I/O-operation on the states (formalized by the state assignments). An *I/O-guard* is a condition on the observable dataflow, formalized by a Boolean combination of atomic propositions over an extended signature $\mathfrak{Sig}_{\mathcal{M}}^{\mathcal{N}}$ that allows to reason about the data items that are observable at the locations in $\mathcal{N}$. The motivation for having symbols for state variables as well as for dataflow locations in the extended signature $\mathfrak{Sig}_{\mathcal{M}}^{\mathcal{N}}$ is that we want define predicates which serve for comparing the observable dataflow at some location with the evaluation of local variable(s), i.e., an I/O-guard is a Boolean combination of atoms of the form $\mathsf{data}_A \asymp rhs$, where $\asymp$ is a Boolean operator from the set $\{<, \leq, =, \neq, \geq, >\}$ and the abstract syntax for the right hand side (rhs) is given by:

$$rhs \ ::= \ k \;\big|\; \mathsf{data}_B \;\big|\; X \;\big|\; \gamma(rhs_1, \ldots, rhs_n)$$

Here, data_A and data_B refer to the observable dataflow at some ports A and B. The observable dataflow can be compared to either a constant value k, the data value data_B observable at port some port B, a value of a local variable X, or the value of an n-ary function ρ. Formally, $\mathfrak{Sig}_{\mathcal{M}}^{\mathcal{N}}$ denotes the signature that results from $\mathfrak{Sig}_{\mathcal{M}}$ by adding

- a special type $T_{\text{I/O}}$ that serves for a characterization of the I/O-ports,
- a new monadic predicate symbol active with $\mathfrak{type}(\mathsf{active}) = T_{\text{I/O}}$,
- constant symbols data_A with $\mathfrak{type}(\mathsf{data}_A) = \lambda_{\mathcal{M}}(A)$ for all $A \in \mathcal{N}$.

The special type symbol $T_{\text{I/O}}$ is needed for technical reasons only. Note that the location-symbol A is of type $T_{\text{I/O}}$, while its message-type is $\lambda_{\mathcal{M}}(A) = \mathfrak{type}(data_A)$. For the interpretations $\mathcal{I}$ of $\mathfrak{Sig}_{\mathcal{M}}^{\mathcal{N}}$ we require that ${T_{\text{I/O}}}^{\mathcal{I}} = \mathcal{N}_{\mathcal{M}}$. The intuitive meaning of the atomic proposition $active(A)$ is a port activity flag which indicates that dataflow at location A is observed.

A *state assignment* is a (possibly empty) conjunction of assignments for local variables, i.e., state assignments have the form

$$V_1 := t_1 \wedge V_2 := t_2 \wedge \ldots \wedge V_p := t_p$$

where $V_1, \ldots, V_p$ are pairwise distinct variables in $\mathfrak{Var}_{\mathcal{M}}$ and t_j are terms over the extended signature $\mathfrak{Sig}_{\mathcal{M}}^{\mathcal{N}}$. Intuitively, when firing a transition via a concurrent I/O-operation c with a state assignment as above as then in the next state the value of the variables V_i agrees with the value of the term t_i under the interpretation given by the current state and the c. Variables $V \in \mathfrak{Var}_{\mathcal{M}} \setminus \{V_1, \ldots, V_p\}$ keep their value after the transition has been taken.

Example 3.2.5 (CARML module of FIFO1 channel with initial value)**.** Figure 3.18 shows a CARML module for a FIFO channel with capacity one whose initial value is set according to the parameter variable *init_value*.

```
MODULE FIFO1⟨var : init_value⟩{
    in : Data A;
    out : Data B;

    var : enum{empty,full} state init := full;
    var : Data value init := init_value;

    state = empty −[ {A} ]→ state := full ∧ value := data_A;
    state = full −[ {B} ∧ data_B = value ]→ state := empty;
}
```

Figure 3.18: CARML module for a FIFO1 channel with initial value *init_value*

The module description of the FIFO1 with initial value contains two local variables "state" and "value" representing the current configuration and of the FIFO1 (empty or full) and the current value $d \in$ Data stored in the buffer. The initial configuration is that the FIFO1 channel is filled with data item *init_value* $\in$ Data provided in the parameter list of the module. The behavior agrees with the standard FIFO1 behavior, i.e., data can be stored if the FIFO1 is in the configuration "empty" and taken off in the "full" configuration.

Figure 3.19 shows the induced constraint automaton for Data $= \{0, 1\}$ and *init_value* $= 1$.

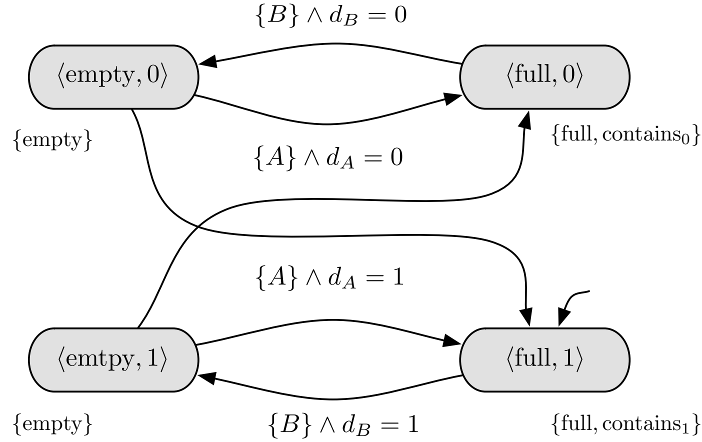


Figure 3.19: Induced constraint automaton for the CARML module of Figure 3.18

■

Example 3.2.6 (CARML module of a stack data structure with capacity *cap*)**.** In Figure 3.20 shows a CARML module for a stack which stores values of type T. The data type T is the first parameter of the module definition whereas the second parameter *cap* determines the capacity of a stack.

```
MODULE Stack⟨type: T, var : cap⟩{
    in : T A;
    out : T B;

    var : (T ∪ {⊥})[cap] values init := ⊥;
    var : int(0, cap) pos init := 0;

    pos < cap −[ {A} ]→ values[pos] := data_A ∧ pos := pos+1;
    pos > 0 −[ {B} ∧ data_B = values[pos] ]→ values[pos] := ⊥ ∧ pos := pos−1;
}
```

Figure 3.20: CARML module for a stack with capacity *cap*

The module for the stack contains two local variables. The variable values is an array of type $T \cup \{\perp\}$ and size *cap* and holds the actual values pushed on the stack via input port A. The second variable pos is of integer type and holds the position of the top-most element of the stack. The value of pos will be increased on any push operation carried out by a write on the input port A and decreased on every pop operation happening when the top-most element is written on output port B.

■

4 | Alternating-time stream logic

Temporal logics such as LTL (linear temporal logic) [Pnu77], and CTL (*computation tree logic*) [CE81] and related temporal logics are well investigated formalisms for the modeling, verification, and synthesis of reactive systems. Within this family, the alternating-time temporal logic (ATL) [AHK02] introduced a useful generalization that helps modeling and analyzing distributed systems. Whereas, classical temporal logics are interpreted over transition-system-like structures, alternating-time logics are interpreted over game structures. Game logics such as alternating-time temporal logic (ATL) [AHK02] allow reasoning about strategies for coalitions of components. A typical scenario for this game-based view of a distributed system would consist of a coalition of controllable components that is playing against its environment and/or a coalition of components that is assumed to behave in an uncontrollable way. ATL now provides modalities for reasoning about the existence of a strategy for the controllable components coalition that ensures a certain temporal logic property to hold no matter how the opponents behave.

In this chapter we introduce the alternating-time stream logic (ASL) for reasoning about strategies in the Reo and constraint automata framework. For this, constraint automata are being interpreted as multi-player game structures. We are interested in the existence of a strategy for the controllable parts of the system to cooperate in such a way that they achieve a common goal. Here, a strategy means that the controllable components can restrict or even refuse performing certain actions or participate to synchronize actions with its environment and the uncontrollable parts of the system.

Organization. In Section 4.1 we provide an interpretation of constraint automata as multi-player games. The syntax and semantics of ASL are presented in Section 4.2. The ASL model checking procedures are introduced in Section 4.3, whereas Section 4.4 contains a discussion about the complexity of ASL model checking. In Section 4.5 ASL we discuss the concept of fairness for ASL.

The work presented in this section has partially been published in [KB09, KB10].

4.1. Constraint automata as multi-player games

In this section we introduce a game-based interpretation of constraint automata. The data type and polarity information for the ports of constraint automata played an important role for the modeling, but throughout this chapter the dataflow vocabulary can be ignored as it is not essential for the introduced temporal logic and the model checking.

For this game-based view of constraint automata, it is assumed that the constraint automaton under consideration models a system where several components are glued via a network consisting of several (a)synchronous channels. The players of the game are the individual components. Each of the players has control over his write and read operations at its interface ports. A player might refuse some or even any synchronization operation with other players. Players might build arbitrary coalitions to achieve a certain common goal, e.g., to enforce that a certain temporal property holds. In our approach, a coalition of players is given by a set of names for the controllable ports $N \subseteq \mathcal{N}$, the union of all controllable coalition ports, for which the players might try to develop a common strategy to achieve their objectives.

Let $N \subseteq \mathcal{N}$ be the set of controllable ports. Then a transition $q \xrightarrow{c} q'$ in a constraint automaton is called *controllable* from the perspective of the N-agents if $\emptyset \neq \mathsf{active}(c) \subseteq N$. The opponents cannot prevent a controllable transition as they are not involved in the concurrent I/O-operation and the transition may fire even if they refuse an interaction with their environment. The transition is *preventable* from the perspective of the N-agents if $\mathsf{active}(c) \cap N \neq \emptyset$ and *unpreventable* otherwise. Hence, for the N-agent there is a symmetric argument, saying that they can only prevent transitions if they are involved in the concurrent I/O-operation. Otherwise, the transition is unpreventable from the perspective of the N-agents. Hence, transitions with the empty concurrent I/O-operation $c_\emptyset$ are unpreventable for the N-agents and their opponents.

Intuitively, a strategy for the set N of controllable ports, briefly called an N-strategy takes the history of the system, formalized by a finite execution, as input (i.e., we suppose here perfect recall) and declares the conditions under which the N-agents (members of the coalition) are willing to cooperate with each other and their opponents. For instance, an N-strategy might offer to write data value 0 at a dataflow location $A \in N$, but refuse to write data value 1. Furthermore, an N-strategy might suggest that the N-agents completely refuse any participation in concurrent I/O-operations. The special symbol $\surd$ will be used for this purpose.

Definition 4.1.1 (Strategy)**.** Let $\mathcal{A}$ be a constraint automaton as in Definition 2.1.2, and let $N \subseteq \mathcal{N}$ be a set of port names. An N-strategy is a function

$$\mathfrak{S} : \mathsf{Exec}_{\mathsf{fin}}(\mathcal{A}) \to 2^{\mathsf{CIO}_\surd},$$

assigning to any finite execution η a set $\mathfrak{S}(\eta)$ consisting of I/O-operations $c \in \mathsf{CIO}$ or the special symbol $\surd$ such that if $c \in \mathsf{CIO}$ and $\mathsf{active}(c) \cap N = \emptyset$ then $c \in \mathfrak{S}(\eta)$. The latter condition ensures that unpreventable transitions cannot be refused by the N-agents.

■

Unlike strategies in standard multi-player games, an N-strategy does not necessarily determine unique activities for the controlled components. Instead it yields a set of potential interactions. This is reasonable for the game semantics of constraint automata since the availability of a concurrent I/O-operation c with $\mathsf{active}(c) = M$ depends on the agreement of all components that have an I/O-port in M to perform synchronized write and read operations according to c. Hence, only if M is a nonempty subset of N the concurrent

I/O-operation c is controllable by N in the sense that the N-ports can offer c, although they cannot enforce that c will indeed be taken.

The rationale behind the condition requiring that $c \in \mathfrak{S}(\eta)$ whenever c is a concurrent I/O-operation with $\mathsf{active}(c) \cap N = \emptyset$ is that the N-ports are not in the position to refuse an I/O-operation c where none of the N-ports is involved. In particular, invisible I/O-operations (i.e., the concurrent I/O-operation $c_\emptyset$ with $\mathsf{active}(c) = \emptyset$) cannot be ruled out by an N-strategy. Thus, $c_\emptyset \in \mathfrak{S}(\eta)$ for each execution η and N-strategy $\mathfrak{S}$.

Given an N-strategy $\mathfrak{S}$, the $\mathfrak{S}$-paths are those paths in $\mathcal{A}$ that can be obtained when the I/O-operations performed at the ports in N are consistent with $\mathfrak{S}$. For finite paths, consistency with $\mathfrak{S}$ requires that $\mathfrak{S}$ returns the special symbol $\surd$ (which indicates that the N-agents refuse any further interactions at their ports) or that the opponents have been in the position to stop dataflow by refusing any I/O-request. This is formalized by the notion of quiescence. From now on we will consider a state $q \in Q$ to be *quiescent* if all enabled concurrent I/O-operations require some activity of at least one I/O-port of either the N-agents or the opponents.

Definition 4.1.2 (Quiescent states)**.** Let $q \in Q$ be a state in the constraint automaton $\mathcal{A}$. State q is said to be *quiescent* if for all concurrent I/O-operations c that are enabled in state q (i.e., all $c \in \mathsf{CIO}(q)$), the port-set $\mathsf{active}(c)$ is nonempty. Hence, the set of all quiescent states $\mathcal{Q}$ is defined as $\mathcal{Q} = \{q \in Q \mid \mathsf{active}(c) \neq \emptyset \text{ for all } c \in \mathsf{CIO}(q)\}$.

■

Stated differently, the set $\mathcal{Q}$ consists of all states q is in which $c_\emptyset \notin \mathsf{CIO}(q)$ holds. Note that dataflow does not need to stop in quiescent states. Instead dataflow continues if there is an enabled concurrent I/O-operation c where the involved components agree on interacting with each other by means of performing the write and read operation specified by c. For each non-quiescent state q, at least one invisible transition is enabled, i.e., we have $c_\emptyset \in \mathsf{CIO}(q)$. This I/O-operation does not require any interaction with the components and will fire, unless another transition is taken.

Definition 4.1.3 ($\mathfrak{S}$-executions, $\mathfrak{S}$-completeness)**.** Let $\mathfrak{S}$ be an N-strategy. A $\mathfrak{S}$-*execution* is a finite or infinite execution

$$\eta = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \dots$$

such that for any position $i \in \mathbb{N}$ with $i < |\eta|$ we have $c_{i+1} \in \mathfrak{S}(\eta \downarrow i)$.

Each infinite $\mathfrak{S}$-execution is said to be $\mathfrak{S}$-complete. A finite $\mathfrak{S}$-execution η is called $\mathfrak{S}$-*complete* if its last state q is quiescent (cf. Definition 4.1.2) and at least one of the following two conditions (i) or (ii) holds:

(i) $\surd \in \mathfrak{S}(\eta)$

(ii) there is no $c \in \mathsf{CIO}(q) \cap \mathfrak{S}(\eta)$ such that $\mathsf{active}(c) \subseteq N$.

■

Condition (i) indicates that refusing any dataflow on the N-ports is a potential behavior under strategy $\mathfrak{S}$, while (ii) stands for the possibility of the opponents to do the same on their part (i.e. refusing any synchronization on the $\mathcal{N} \setminus N$-ports).

Definition 4.1.4 ($\mathfrak{S}$-paths)**.** Let $\mathfrak{S}$ be an N-strategy. A $\mathfrak{S}$-*path* denotes any infinite $\mathfrak{S}$-execution or any finite path

$$\pi \;=\; q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n \xrightarrow{\surd} q_n$$

where $\pi \downarrow n$ is a $\mathfrak{S}$-complete $\mathfrak{S}$-execution.

■

We write $\mathsf{Paths}(q, \mathfrak{S})$ to denote all $\mathfrak{S}$-paths starting in q. Similarly, $\mathsf{Exec}_{\mathrm{fin}}(q, \mathfrak{S})$ denotes the set of all finite $\mathfrak{S}$-executions from q.

The general definition of strategies does not impose any restrictions on their realizability. E.g., strategies may even be not computable. As we will see later, for our purposes strategies with finite memory are sufficient. These are strategies that make their decisions on the basis of a finite automaton rather than the full history.

Definition 4.1.5 (Finite-memory and memoryless strategies)**.** A finite-memory N-strategy is a tuple $\mathfrak{M} = \langle \mathsf{Modes}, \Delta, \mu, m_0 \rangle$, where

- Modes is a finite set (of so-called modes),
- $m_0 \in \mathsf{Modes}$ the starting mode,
- $\mu : Q \times \mathsf{Modes} \to 2^{\mathsf{CIO}_\surd}$ the decision function, and
- $\Delta : \mathsf{Modes} \times (Q \times \mathsf{CIO} \times Q) \to \mathsf{Modes}$ the next-mode function.

For the decision function μ we require that $\mu(q, m) \supseteq \{c \in \mathsf{CIO} : \mathsf{active}(c) \cap N = \emptyset\}$ for all states $q \in Q$ and modes $m \in \mathsf{Modes}$. If Modes is a singleton then we refer to $\mathfrak{M}$ as a *memoryless* strategy. Memoryless strategies are typically specified as functions $\mathfrak{S} : Q \to 2^{\mathsf{CIO}_\surd}$.

■

Given a finite-memory N-strategy $\mathfrak{M}$ then the associated N-strategy $\mathfrak{S}_{\mathfrak{M}}$ (in the sense of Definition 4.1.1) is given by:

$$\mathfrak{S}_{\mathfrak{M}}\big(q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_i} q_i\big) \;=\; \mu\big(q_i, \Delta^*(m_0, q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_i} q_i)\big)$$

where $\Delta^*(m, \eta)$ is defined by induction on the length of η. For executions of length 0 we put $\Delta^*(m, q_0) \stackrel{\text{def}}{=} m$. For executions of length 1, Δ^* agrees with the next-mode function Δ, i.e.,

$$\Delta^*(m, q_0 \xrightarrow{c_1} q_1) \;\stackrel{\text{def}}{=}\; \Delta(m, q_0 \xrightarrow{c_1} q_1),$$

while for executions of length $i \geq 2$ we define:

$$\Delta^*\big(m, q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots \xrightarrow{c_i} q_i\big) \;\stackrel{\text{def}}{=}\; \Delta^*\big(\,\Delta(m, q_0 \xrightarrow{c_1} q_1),\, q_1 \xrightarrow{c_2} \ldots \xrightarrow{c_i} q_i\big).$$

Example 4.1.6 (Finite-memory strategies)**.** Let $\mathcal{A}$ be a constraint automaton as depicted on the left of Figure 4.1, with

$$\mathsf{active}(c_1) \cap N \neq \emptyset,\ \mathsf{active}(c_2) \subseteq N,\ \text{and } \mathsf{active}(c_3) \cap N = \emptyset$$

for a set of port names $N \subseteq \mathcal{N}$, i.e., c_1 is a preventable, c_2 is a controllable, and c_3 is an unpreventable concurrent I/O-operation from the perspective of the N-agents. Let the goal for this example now be to specify a finite-memory N-strategy $\mathfrak{S}$ which ensures that the number of visits of state q_2' is at most one and that state q_1 is not visited at all along all paths $\pi \in \mathsf{Paths}(q_0, \mathfrak{S})$.

Let $\mathfrak{M} = \langle \mathsf{Modes}, \Delta, \mu, m_0 \rangle$ be a finite-memory N-strategy with $\mathsf{Modes} = \{m_0, m_1\}$ and $m_0 \in \mathsf{Modes}$ the starting mode. The next-mode function Δ is a partial function for which the relevant parts are shown on the right of Figure 4.1 and the decision function

$$\mu(q_i, m_0) = \{c_\emptyset, c_2, c_3\} \text{ and } \mu(q_i, m_1) = \{c_\emptyset, c_3\}$$

for all $i \in \mathbb{N}$. The ratio behind $\mathfrak{M}$ is 1) that the preventable transition $q_0 \xrightarrow{c_1} q_1$ should

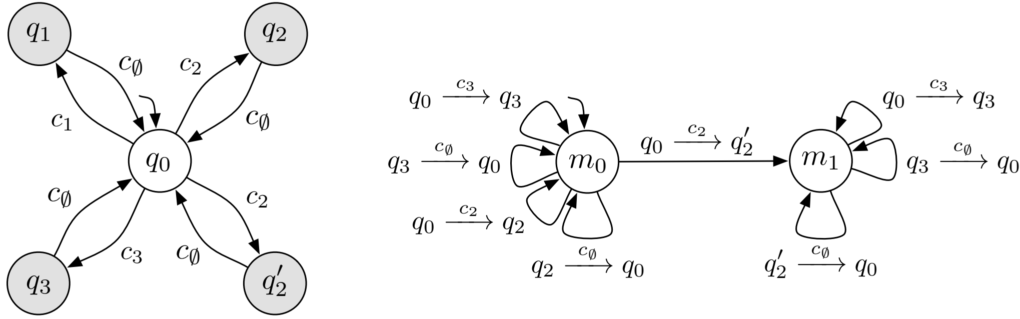


Figure 4.1: Constraint automaton and a finite-memory N-strategy

never be taken and 2) that the controllable transitions $q_0 \xrightarrow{c_2} q_2$ and $q_0 \xrightarrow{c_2} q_2'$ with the concurrent I/O-operation c_2 are supported in the initial mode m_0. Once the transition $q_0 \xrightarrow{c_2} q_2'$ has fired the finite-memory N-strategy $\mathfrak{M}$ changes its mode to m_1 and from then on refuses any synchronization with either c_1 or c_2. Thus, in $\mathcal{A}$ the state q_2' can be visited at most once.

The associated N-strategy $\mathfrak{S}_\mathfrak{M}$ is

$$\mathfrak{S}_\mathfrak{M}(q_0 \xrightarrow{c_1} \dots \xrightarrow{c_i} q_i) = \begin{cases} \mu(q_i, m_0) = \{c_\emptyset, c_2, c_3\} & \text{if } q_j \neq q_2' \text{ for all } 0 \leq j \leq i \\ \mu(q_i, m_1) = \{c_\emptyset, c_3\} & \text{else} \end{cases}$$

■

4.2. Syntax and semantics

Syntax of ASL state and path formulas. State formulas (denoted by capital Greek letters Φ, Ψ) and path formulas (denoted by lower case Greek letters φ, ψ) of ASL are built by the following grammar:

$$\Phi \;::=\; \mathsf{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \mathbb{E}_N\varphi$$
$$\varphi \;::=\; \langle\mathcal{Z}\rangle\Phi \;\Big|\; [\![\mathcal{Z}]\!]\Phi \;\Big|\; \Phi_1 \,\mathrm{U}\, \Phi_2 \;\Big|\; \Phi_1 \,\mathrm{R}\, \Phi_2$$

where $N \subseteq \mathcal{N}$, $a \in AP$ and $\mathcal{Z} = \langle Z, \Sigma, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ is a nondeterministic finite automaton (NFA), where Z is a finite set of states, $\Sigma = \mathsf{CIO}_{\surd}$ a finite alphabet, $\longrightarrow_{\mathcal{Z}} \subseteq Z \times \Sigma \times Z$ the transition relation, $Z_0 \subseteq Z$ a set of starting states, and Z_F a set of final states.

The operators $\langle\mathcal{Z}\rangle$ and $[\![\mathcal{Z}]\!]$ are used as existential and universal quantifiers on prefixes characterized by $\mathcal{Z}$. The intended meaning of $\langle\mathcal{Z}\rangle\Phi$ is that it holds for a path π iff π has a finite prefix generating an I/O-stream which is accepted by $\mathcal{Z}$ and Φ holds for the state reached afterwards. $[\![\mathcal{Z}]\!]\Phi$ is the dual operator of $\langle\mathcal{Z}\rangle\Phi$ and holds for a path π iff for all finite prefixes of π generating an I/O-stream accepted by $\mathcal{Z}$, the formula Φ holds for the last state of the prefix.

Beside the special $\surd$-transitions, $\mathcal{Z}$ can be viewed as a constraint automaton $\mathcal{Z}$ with an additional set Z_F of final (accept) states. The atomic propositions and labeling function are irrelevant for $\mathcal{Z}$.

We call a run $\theta = z_0 \xrightarrow{c_1}_{\mathcal{Z}} \dots \xrightarrow{c_n}_{\mathcal{Z}} z_n$ in $\mathcal{Z}$ an *accepting run* if $z_0 \in Z_0$ and $z_n \in Z_F$. The language accepted by $\mathcal{Z}$ is defined as

$$\mathcal{L}(\mathcal{Z}) \stackrel{\text{def}}{=} \{c_1 \dots c_n \in \mathsf{CIO}^*_{\surd} \mid z_0 \xrightarrow{c_1}_{\mathcal{Z}} \dots \xrightarrow{c_n}_{\mathcal{Z}} z_n \text{ is an accepting run in } \mathcal{Z}\}.$$

For $\mathcal{L}(\mathcal{Z})$ we require that $\mathcal{L}(\mathcal{Z}) \subseteq \mathsf{CIO}^* \cup \mathsf{CIO}^*\surd$. By the special role of the end symbol $\surd$, we may assume that $\mathcal{Z}$'s state space contains a subset $Z_{\surd} \subseteq Z$ such that

1. $z \xrightarrow{\surd}_{\mathcal{Z}} z'$ implies $z' \in Z_{\surd}$,
2. $z \xrightarrow{c}_{\mathcal{Z}} z'$ with $z' \in Z_{\surd}$ implies $c = \surd$ and
3. $z \in Z_{\surd}$ implies that z is a terminal state.

As a consequence we have that $\mathcal{L}(\mathcal{Z}) \subseteq \mathsf{CIO}^* \cup \mathsf{CIO}^*\surd$.

Derived operators. The operator $\mathbb{E}_N$ corresponds to an existential quantification over all N-strategies. The dual operator $\mathbb{A}_N\varphi$, stating that no strategy for the ports in N can avoid φ to hold, is defined by:

$$\mathbb{A}_N\langle\mathcal{Z}\rangle\Phi \stackrel{\text{def}}{=} \neg\mathbb{E}_N[\![\mathcal{Z}]\!]\neg\Phi \qquad \mathbb{A}_N(\Phi_1 \,\mathrm{U}\, \Phi_2) \stackrel{\text{def}}{=} \neg\mathbb{E}_N(\neg\Phi_1 \,\mathrm{R}\, \neg\Phi_2)$$
$$\mathbb{A}_N[\![\mathcal{Z}]\!]\Phi \stackrel{\text{def}}{=} \neg\mathbb{E}_N\langle\mathcal{Z}\rangle\neg\Phi \qquad \mathbb{A}_N(\Phi_1 \,\mathrm{R}\, \Phi_2) \stackrel{\text{def}}{=} \neg\mathbb{E}_N(\neg\Phi_1 \,\mathrm{U}\, \neg\Phi_2)$$

Other Boolean connectives, like disjunction or implication, are obtained in the standard way. In the following, we shortly write

$$\mathbb{E}_A\varphi \text{ for } \mathbb{E}_{\{A\}}\varphi \text{ and } \mathbb{A}_A\varphi \text{ for } \mathbb{A}_{\{A\}}\varphi.$$

ASL path formulas are interpreted over paths in a constraint automaton. The modalities $\mathtt{U}$ and $\mathtt{R}$ denote the standard until-operator and release-operator, respectively. The eventually and always operator are obtained in the usual way by

$$\Diamond\Phi \overset{\text{def}}{=} (\mathsf{true}\,\mathtt{U}\,\Phi) \text{ and } \Box\Phi \overset{\text{def}}{=} (\mathsf{false}\,\mathtt{R}\,\Phi).$$

The standard *next* operator is derived in ASL by

$$\mathtt{X}\,\Phi \overset{\text{def}}{=} \langle\mathcal{Z}_{tt}\rangle\Phi$$

where $\mathcal{Z}_{tt}$ is an NFA accepting words over CIO of length one. Thus, $\mathtt{X}\,\Phi$ holds for all paths where the underlying execution has at least one transition and Φ holds afterwards.

Semantics of ASL. Let $\mathcal{A}$ be a constraint automaton and π a path in $\mathcal{A}$. The satisfaction relation $\models$ for ASL state formulas is defined by structural induction as shown below:

$$\begin{array}{lll}
q \models \mathsf{true} & & \\
q \models a & \text{iff} & a \in L(q) \\
q \models \Phi_1 \wedge \Phi_2 & \text{iff} & q \models \Phi_1 \text{ and } q \models \Phi_2 \\
q \models \neg\Phi & \text{iff} & q \not\models \Phi \\
q \models \mathbb{E}_N\varphi & \text{iff} & \text{there is an } N\text{-strategy } \mathfrak{S} \text{ such that} \\
 & & \text{for all } \pi \in \mathsf{Paths}(q,\mathfrak{S}) \text{ we have: } \pi \models \varphi.
\end{array}$$

For the dual alternating-time modality $\mathbb{A}_N$ we obtain the expected semantics:

$$\begin{array}{lll}
q \models \mathbb{A}_N\varphi & \text{iff} & \text{for all } N\text{-strategies } \mathfrak{S} \\
 & & \text{there exists } \pi \in \mathsf{Paths}(q,\mathfrak{S}) \text{ such that } \pi \models \varphi.
\end{array}$$

The satisfaction relation $\models$ for ASL path formulas and the path π in $\mathcal{A}$ is defined as follows:

$$\begin{array}{lll}
\pi \models \langle\mathcal{Z}\rangle\Phi & \text{iff} & \text{there exists } n \in \mathbb{N} \text{ such that } 0 \leq n \leq |\pi| \text{ and} \\
 & & \mathsf{ios}(\pi\downarrow n) \in \mathcal{L}(\mathcal{Z}) \text{ and } q_n \models \Phi \\
\pi \models [\![\mathcal{Z}]\!]\Phi & \text{iff} & \text{for all } n \in \mathbb{N} \text{ such that } 0 \leq n \leq |\pi| \text{ we have:} \\
 & & \mathsf{ios}(\pi\downarrow n) \in \mathcal{L}(\mathcal{Z}) \text{ implies } q_n \models \Phi \\
\pi \models \Phi_1\,\mathtt{U}\,\Phi_2 & \text{iff} & \text{there exists } n \in \mathbb{N} \text{ such that } 0 \leq n < |\pi| \text{ where} \\
 & & q_n \models \Phi_2 \text{ and } q_i \models \Phi_1 \text{ for } 0 \leq i < n \\
\pi \models \Phi_1\,\mathtt{R}\,\Phi_2 & \text{iff} & \text{at least one of the following conditions (i) or (ii) holds:} \\
 & & \text{(i) for all } n \in \mathbb{N} \text{ with } 0 \leq n < |\pi| \text{ we have: } q_n \models \Phi_2 \\
 & & \text{(ii) there exists some } n \in \mathbb{N} \text{ with } 0 \leq n \leq |\pi| \text{ such that:} \\
 & & \quad q_n \models \Phi_1 \wedge \Phi_2 \text{ and } q_i \models \Phi_2 \text{ for } 0 \leq i \leq n.
\end{array}$$

Definition 4.2.1 (Winning strategy)**.** Given a state q and an ASL path formula φ, an N-strategy $\mathfrak{S}$ is called *winning* for the tuple $\langle q, \varphi \rangle$ if φ holds for all $\mathfrak{S}$-paths starting in q. Thus, $q \models \mathbb{E}_N \varphi$ iff there exists a winning N-strategy for $\langle q, \varphi \rangle$. We say that $\mathfrak{S}$ is winning for φ if $\mathfrak{S}$ is winning for all pairs $\langle q, \varphi \rangle$ where $q \models \mathbb{E}_N \varphi$.

■

Remark 4.2.2 (Nesting of strategy quantifiers)**.** For formulas of the form $\mathbb{E}_N \varphi$ where φ itself contains a strategy quantifier $\mathbb{E}_{N'} \varphi'$ it is worth noticing that a winning strategy for the inner formula is in general not related to the winning strategy for the outer formula even if N' and N agree. In general the two strategies cannot be combined into a single winning strategy. E.g., let $\Phi_0 = \mathbb{E}_N \Box(\Phi \wedge \mathbb{E}_N \Diamond \neg \Phi)$ be a nested ASL formula. Any winning N-strategy $\mathfrak{S}$ for $\varphi = \Box(\Phi \wedge \mathbb{E}_N \Diamond \neg \Phi)$ ensures Φ to hold globally and at the same time it ensures staying in states from which there exists a different N-strategy $\mathfrak{S}'$ which is winning for $\varphi' = \Diamond \neg \Phi$. Obviously the two strategies $\mathfrak{S}$ and $\mathfrak{S}'$ cannot be combined into a single N-strategy which is winning for φ and φ' at the same time. Similar phenomena can be observed when nesting $\mathbb{E}_N \varphi$ with $\mathbb{A}_{N'} \varphi'$ in any possible combination. These phenomena are not specific to the ASL, but do exist already in branching-time logics such as CTL, which quantify over paths (or prefixes of paths) rather than over strategies. E.g., for the CTL state formula $\forall \Box(\Phi \wedge \forall \Diamond \neg \Phi)$ we observe a similar situation.

■

Regular I/O-stream expressions. Instead of an NFA we will often use a *regular I/O-stream expression* α which uses I/O-constraints as atoms. The abstract syntax of regular I/O-stream expressions, briefly called *stream expressions*, is given by the following grammar:

$$\alpha \ ::= \ g \ \Big| \ \surd \ \Big| \ \alpha^* \ \Big| \ \alpha_1; \alpha_2 \ \Big| \ \alpha_1 \cup \alpha_2$$

where $g \in \mathsf{IOC}$ ranges over all I/O-constraints. Any stream expression represents a regular set of I/O-streams. In the sequel we denote the set of all I/O-streams by IOS.

The formal definition of the regular languages $\mathsf{IOS}(\alpha) \subseteq \mathsf{CIO}^* \cup \mathsf{CIO}^* \surd$ is given by structural induction. $\mathsf{IOS}(g)$ is the set consisting of the I/O-streams of length 1 given by g, i.e., $\mathsf{IOS}(g) \stackrel{\text{def}}{=} \lceil g \rceil$. (Recall that the semantics $\lceil g \rceil$ of an I/O-constraint $g \in \mathsf{IOC}$ has been defined in Definition 2.2.1). Similarly, $\mathsf{IOS}(\surd)$ is the singleton set consisting of the I/O-stream $\surd$. Union ($\cup$), Kleene star ($*$) and concatenation (;) have their standard meaning.

$$\begin{aligned} \mathsf{IOS}(\alpha_1 \cup \alpha_2) &\stackrel{\text{def}}{=} \mathsf{IOS}(\alpha_1) \cup \mathsf{IOS}(\alpha_2) \\ \mathsf{IOS}(\alpha_1; \alpha_2) &\stackrel{\text{def}}{=} \mathsf{IOS}(\alpha_1); \mathsf{IOS}(\alpha_2). \end{aligned}$$

The Kleene star and concatenation rely on a special treatment of the termination symbol $\surd$. For $\mathcal{I}_1, \mathcal{I}_2 \subseteq \mathsf{IOS}$ we define

$$\begin{aligned} \mathcal{I}_1; \mathcal{I}_2 &\stackrel{\text{def}}{=} \left\{ \mathfrak{i}_1 \mathfrak{i}_2 \, : \, \mathfrak{i}_1 \in \mathcal{I}_1 \cap \mathsf{CIO}^* \wedge \mathfrak{i}_2 \in \mathcal{I}_2 \right\} \\ \mathsf{IOS}(\alpha^*) &\stackrel{\text{def}}{=} \bigcup_{n \geq 0} \mathsf{IOS}(\alpha)^{(n)} \end{aligned}$$

where $\mathcal{I}_1^{(0)} \stackrel{\text{def}}{=} \{\varepsilon\}$, $\mathcal{I}_1^{(1)} = \mathcal{I}_1$, and $\mathcal{I}_1^{(n+1)} \stackrel{\text{def}}{=} \mathcal{I}_1; \mathcal{I}_1^{(n)}$.

As usual, we write α^+ for $\alpha; \alpha^*$. Using standard methods for regular languages (see e.g. [HMU06]), we first generate a nondeterministic finite automaton (NFA) $\mathcal{Z}$ over the alphabet $\Sigma = \mathsf{IOC}_{\surd}$ for a stream expression α and then transfer it into an NFA $\mathcal{Z}_\alpha$ with alphabet $\Sigma = \mathsf{CIO}_{\surd}$ such that

$$\mathcal{L}(\mathcal{Z}) = \mathcal{L}(\mathcal{Z}_\alpha) = \mathsf{IOS}(\alpha).$$

Let from now on $\mathcal{Z}_\alpha$ denote an NFA with alphabet $\Sigma = \mathsf{CIO}_{\surd}$ with the above property.

The quiescent states are characterized by the state formula $\exists\langle\surd\rangle\mathsf{true}$, while $\forall\langle\surd\rangle\mathsf{true}$ is satisfied in exactly those states where no concurrent I/O-operation is enabled. The path formula $[\![tt^*; \surd]\!]\mathsf{false}$ is characteristic for the infinite paths, while $\langle tt^*; \surd\rangle\mathsf{true}$ holds exactly for the finite paths.

Clearly, the use of stream expressions rather than NFA in ASL path formulas does not affect the expressiveness of ASL. Using well-known techniques for regular expressions and NFA, each ASL path formula can be transformed to an equivalent formula with stream expressions in the modalities $\langle\alpha\rangle$ and $[\![\alpha]\!]$ rather than $\langle\mathcal{Z}\rangle$ and $[\![\mathcal{Z}]\!]$. The transformation, however, can cause an exponential blow-up.

Sub-logics (ASL $\supset$ BTSL $\supset$ CTL). The sub-logic BTSL as presented in [KB09] is a branching-time logic which can be derived from ASL. The standard existential and universal path quantifiers that range over all paths can be derived using the $\mathbb{E}_N$ and $\mathbb{A}_N$-quantifiers with the port-set $N = \emptyset$.

$$\forall\varphi \stackrel{\text{def}}{=} \mathbb{E}_\emptyset\varphi \text{ and } \exists\varphi \stackrel{\text{def}}{=} \mathbb{A}_\emptyset\varphi$$

This is due to the fact that N-strategies for the empty port-set $N = \emptyset$ cannot rule out any concurrent I/O-operation, i.e., $\mathsf{CIO} \subseteq \mathfrak{S}(\eta)$ for each $\emptyset$-strategy $\mathfrak{S}$ and execution η. Hence, all paths are $\mathfrak{S}$-paths.

The abstract syntax of BTSL can be described by the following grammar.

$$\begin{array}{lcl} \Phi & ::= & \mathsf{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \\ \varphi & ::= & \langle\mathcal{Z}\rangle\Phi \mid [\![\mathcal{Z}]\!]\Phi \mid \Phi_1 \,\mathrm{U}\, \Phi_2 \mid \Phi_1 \,\mathrm{R}\, \Phi_2 \end{array}$$

The sub-logic CTL (*computation tree logic*) first proposed in [CE81] corresponds to BTSL, but without the dataflow quantifiers $\langle\mathcal{Z}\rangle$ and $[\![\mathcal{Z}]\!]$. To preserve the CTL next step operator we explicitly include it into the syntax.

$$\begin{array}{lcl} \Phi & ::= & \mathsf{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \\ \varphi & ::= & \mathrm{X}\,\Phi \mid \Phi_1 \,\mathrm{U}\, \Phi_2 \mid \Phi_1 \,\mathrm{R}\, \Phi_2 \end{array}$$

Example 4.2.3. For a simple example, consider a system consisting of two components C_A and C_B that are connected via a FIFO1 channel with a single buffer cell as illustrated in the left part of Figure 4.2. The corresponding constraint automaton for the data domain $\mathsf{Data} = \{0, 1\}$ is depicted on the right where we treat C_A and C_B as black box components. Suppose that empty is an atomic proposition stating that the buffer is empty. Then, the ASL

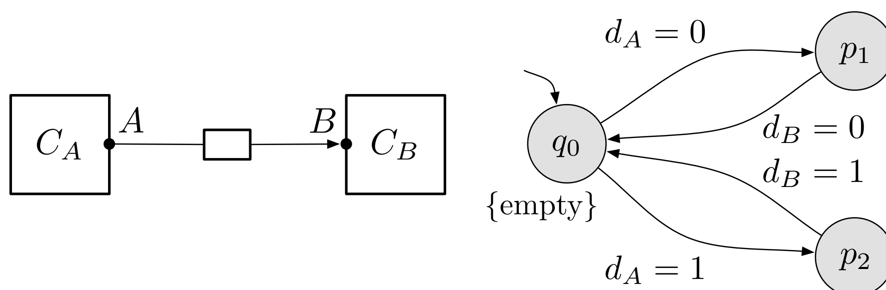


Figure 4.2: Two components connected via a FIFO channel

state formula $\mathbb{E}_A \Box\, \mathsf{empty}$ asserts that component C_A has the power to ensure that the buffer remains always empty. In fact, we have

$$q_0 \models \mathbb{E}_A \Box\, \mathsf{empty}$$

as the memoryless A-strategy that assigns $\{\surd\} \cup \{c \in \mathsf{CIO} : B \in \mathsf{active}(c)\}$ to all states is winning for state q_0 and the objective $\Box\, \mathsf{empty}$. Similarly,

$$q_0 \models \mathbb{E}_A \Box\, (\mathsf{buffer} \neq 0)$$

holds where $(\text{buffer} \neq 0)$ is an atomic proposition stating that either the buffer is empty or contains a data value different from 0. However, A has no strategy that ensures that A can write two times to the buffer. That is,

$$q_0 \not\models \mathbb{E}_A \langle tt^*; A; tt; A \rangle \mathsf{true}.$$

The reason for this is that once C_A has written some value into the buffer then C_A is not in the position to force C_B to read the written value eventually. But if C_B never takes the element from the buffer then C_A's writing request at port A remains disabled forever.

■

Standard turn-based games are determined [BL69, Mar75]. In our setting, determinacy means that given a state q, a coalition $\mathcal{C}$ and a winning objective φ, then either $\mathcal{C}$ has a strategy to ensure that φ holds or the opponents, i.e., all agents not in $\mathcal{C}$, have a strategy to ensure that $\neg\varphi$ holds. This does not hold for the ASL games. We illustrate this phenomenon by means of two toy examples.

Example 4.2.4 (ASL games are not determined)**.** A system with two components C_A and C_B with one output port each (called A and B, respectively) is shown on the left of Figure 4.3 where we used the Reo syntax to depict the network. The glue code consists of two circular FIFO channels with a single buffer cell. The picture on the right of Figure 4.3 shows a corresponding constraint automaton where we treat C_A and C_B as black box components and assume that the internals of the Reo connector are hidden. Initially (state q_0), the upper buffer contains the data item 0, while the lower buffer is empty. For the data domain we assume that $\mathsf{Data} = \{0\}$. As long as neither C_A nor C_B writes into the buffer then the data item 0 can alternate between the upper and lower buffers via a non-observable step (the empty concurrent I/O-constraint $c_\emptyset$). In state q_0 the write operation at port A of component C_A is blocked (as the upper buffer is full), while component C_B can write into the lower buffer by performing a write operation at port B. The situation where the lower buffer is full and the upper buffer is empty (state q_1) is symmetric. As soon as one of the components writes on its output port, both buffers become full (state q_2) which blocks any further dataflow in the network. We now consider the ASL state formula $\Phi = \mathbb{E}_A\Diamond\neg\exists\,\mathrm{X}\,\mathsf{true}$. We

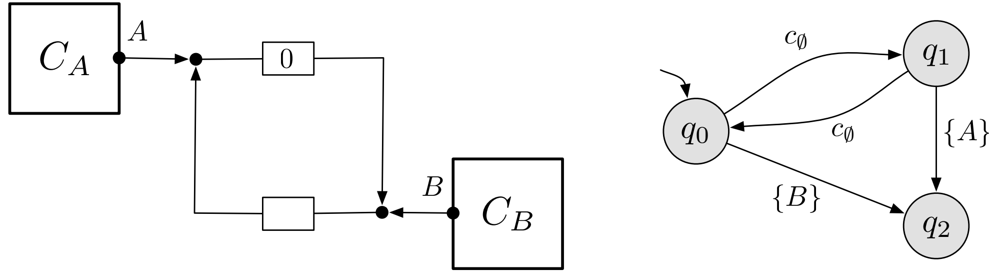


Figure 4.3: Network (left) and constraint automaton (right)

first observe that $q \models \neg\exists\,\mathrm{X}\,\mathsf{true}$ iff $q = q_2$. Thus, Φ asserts that port A (i.e., component C_A) has a strategy to ensure that eventually state q_2 will be reached. Intuitively, one might expect Φ to hold for states q_0 and q_1, since a write operation at port A is enabled in state q_1 and leads to state q_2. However, this is not the case since C_A cannot avoid the path

$$\pi \;=\; q_0 \xrightarrow{c_\emptyset} q_1 \xrightarrow{c_\emptyset} q_0 \xrightarrow{c_\emptyset} \ldots$$

where no I/O-operation at A or B will be performed. Note that π is a $\mathfrak{S}$-path for any A-strategy $\mathfrak{S}$ that offers a write operation at port A whenever the system is in state q_1, e.g., the memoryless strategy $\mathfrak{S}(q_i) = \mathsf{CIO}$. Thus, A has no winning strategy for $\langle q_0, \Diamond\neg\exists\,\mathrm{X}\,\mathsf{true}\rangle$. Therefore

$$q_0 \not\models \mathbb{E}_A\Diamond\neg\exists\,\mathrm{X}\,\mathsf{true}$$

which can be rephrased as $q_0 \models \mathbb{A}_A\Box\exists\,\mathrm{X}\,\mathsf{true}$ stating that A cannot avoid that the system is always in one of the states q_0 or q_1. On the other hand, the opponent C_B does not have a winning strategy for the objective $\neg\Diamond\neg\exists\,\mathrm{X}\,\mathsf{true} \;\equiv\; \Box\exists\,\mathrm{X}\,\mathsf{true}$ either, as C_B cannot avoid that in state q_1 the A-transition to state q_2 will be taken eventually.

■

In the above example, the property to reach state q_2 cannot be enforced by A since the writing request by component C_A might be ignored forever. This pathological case can be ruled out by imposing appropriate fairness assumptions where all controllable concurrent I/O-operations that are infinitely often offered by a strategy will be taken infinitely often (see Section 4.5). However, even under such fairness assumptions the multi-player game associated with a constraint automaton is not determined, because of the internal nondeterminism that is inherent in the choice between transitions with the same source state and the same I/O-operation.

Example 4.2.5 (ASL games are not determined (internal nondeterminism))**.** Consider a network with three components C_A, C_1, C_2 as shown on the left in Figure 4.4. The components C_1 and C_2 have one input and one output port, whereas C_A has one output port A^{out} and two input ports A_1^{in} and A_2^{in}. We assume that the interface specifications of C_A, C_1, and C_2 declare that their read and write actions alternate. That is, component C_A serves as an initial producer (i.e., it starts with a write) and C_1 and C_2 as initial consumers (i.e., each starts with a read). Again we use Reo syntax to specify the glue code and hide the internals of the connector when looking at the constraint automaton. The network consists of three synchronous channels, a component connector that realizes an exclusive router and two FIFO1 channels. The (data-abstract) operational semantics of this network is modeled by the constraint automaton with the port-set

$$\mathcal{N} = \{A_1^{in}, A_2^{in}, A^{out}, B_1^{in}, B_1^{out}, B_2^{in}, B_2^{out}\}$$

shown on the right in Figure 4.4. In the initial state q_0 the two buffers are empty and the

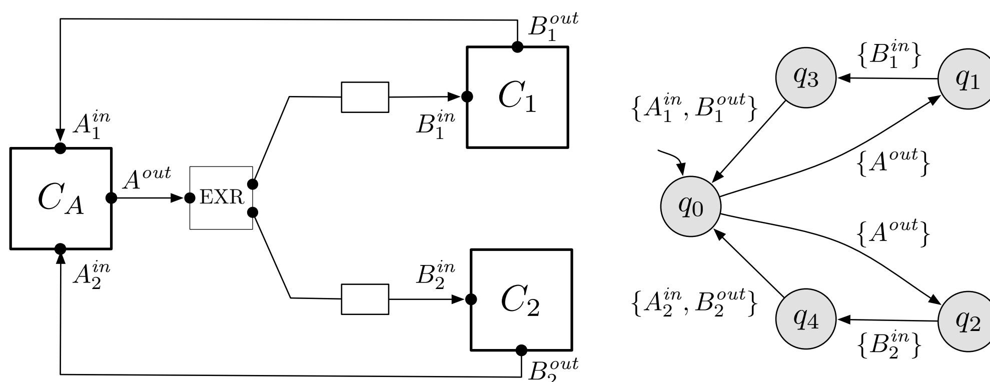


Figure 4.4: Network with three components and its constraint automaton

only possible action is a write operation performed by C_A at its output port A^{out}. Then, the exclusive router delivers the data item written by C_A to one of the two buffers where the decision of which buffer is chosen is resolved internally in a nondeterministic way. Depending on which buffer is full, we are in state q_i where $i = 1$ or $i = 2$, and then component C_i can perform a read operation on its input port B_i^{in} to take the data item written by C_A from the buffer. The resulting state is q_3 or q_4, and C_A can synchronize with C_i via the synchronous channel $A_i^{in}\, B_i^{out}$ which leads back to the initial state q_0.

Because of the nondeterminism in the initial state q_0, neither C_A has the power to enforce that state q_1 will be reached eventually, nor does the coalition $\{C_1, C_2\}$ have a strategy to guarantee that state q_1 will never be reached. Thus, we have that

$$q_0 \not\models \mathbb{E}_N \Diamond \exists \langle B_1^{in} \rangle \mathsf{true} \quad \text{and} \quad q_0 \not\models \mathbb{E}_{\mathcal{N} \setminus N} \Box \neg \exists \langle B_1^{in} \rangle \mathsf{true}$$

where $N = \{A_1^{in}, A_2^{in}, A^{out}\}$. Note that $q \models \exists \langle B_1^{in} \rangle \mathsf{true}$ iff $q = q_1$.

■

As we have seen in the above two examples the reason for ASL not to be determined is based on the fact that there are situations where neither the N-agents nor their opponents have a winning strategy for φ and $\neg\varphi$, respectively. We will now consider deterministic turn-based constraint automata, as for this class of system models ASL will become determined.

Definition 4.2.6 (Deterministic constraint automata)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L \rangle$ be a constraint automaton. $\mathcal{A}$ is *deterministic* if the transition relation can be written as a partial function

$$\delta : Q \times \mathsf{CIO} \times Q$$

such that $\delta(q, c_\emptyset) \neq \bot$ implies $\delta(q, c) = \bot$ for all $c \in \mathsf{CIO} \setminus \{c_\emptyset\}$.

■

Definition 4.2.7 ((Non)deterministic two-player turn-based game structures)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L \rangle$ be a constraint automaton and $N \subseteq \mathcal{N}$ a set of ports. $\mathcal{A}$ can be interpreted as a *two-player turn-based game structure* for player N and player $\mathcal{N} \setminus N$ if the following three conditions hold:

i) The set of states Q can be partitioned into sets Q_1 and Q_2

ii) The transition relation $\longrightarrow$ can be written as $\longrightarrow \,=\, \longrightarrow_1 \cup \longrightarrow_2$, where:

$$\begin{aligned} \longrightarrow_1 \subseteq Q_1 \times \mathsf{CIO} \times Q_2 \text{ with } \langle q, c, q' \rangle \in \longrightarrow_1 \quad &\text{implies} \quad \emptyset \neq \mathsf{active}(c) \subseteq N \\ \longrightarrow_2 \subseteq Q_2 \times \mathsf{CIO} \times Q_1 \text{ with } \langle q, c, q' \rangle \in \longrightarrow_2 \quad &\text{implies} \quad \emptyset \neq \mathsf{active}(c) \subseteq \mathcal{N} \setminus N \end{aligned}$$

iii) The set of initial states Q_0 is either a subset of Q_1 or of Q_2.

A two-player turn-based game structure is deterministic if $\longrightarrow_1$ and $\longrightarrow_2$ can be written as partial functions (cf. Definition 4.2.6).

■

Lemma 4.2.8 (ASL for turn-based games)**.** Let $\mathcal{A}$ be a constraint automaton which is structured as in Definition 4.2.7 with player N and player $\mathcal{N} \setminus N$ for some $N \subseteq \mathcal{N}$. Furthermore, let $q \in Q$ be a state in $\mathcal{A}$ and φ an ASL path formula. Then,

$$q \models \mathbb{E}_N \varphi \iff q \not\models \mathbb{E}_{\mathcal{N} \setminus N} \neg\varphi \; \big(\iff q \models \mathbb{A}_{\mathcal{N} \setminus N} \varphi \big)$$

where the right equivalence holds by definition.

■

Proof. We will prove this by contradiction and start with the assumption that

$$q \models \mathbb{E}_N \varphi \;\text{ and }\; q \models \mathbb{E}_{\mathcal{N} \setminus N} \neg\varphi.$$

Hence, there exists an N-strategy $\mathfrak{S}$ and a $\mathcal{N} \setminus N$-strategy $\mathfrak{S}'$ such that

i) $\pi \models \varphi$ for all $\mathfrak{S}$-paths π starting in q and

ii) $\pi \not\models \varphi$ for all $\mathfrak{S}'$-paths π starting in q

Thus, there is no path $\pi \in \mathsf{Paths}(q, \mathfrak{S}) \cap \mathsf{Paths}(q, \mathfrak{S}')$. We will now construct such a path which is in the intersection of both strategies. Let

$$\eta = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} q_n$$

be an execution of length n, which is a $\mathfrak{S}$-execution and at the same time a $\mathfrak{S}'$-execution.

For the case $q_n \in Q_1$ (according to Definition 4.2.7) we have that, if η is a complete $\mathfrak{S}$-execution (according to Definition 4.1.3), then η is also a complete $\mathfrak{S}'$-execution. This is due to the fact, that if $\mathfrak{S}$ suggests to stop in $q_n \in Q_1$ (either by $\surd \in \mathfrak{S}(\eta)$ or by $\mathfrak{S}(\eta) = \emptyset$), the opponents strategy $\mathfrak{S}'$ can not force the game to continue. A symmetric argument holds for $q_n \in Q_2$. Here we have that, if η is a complete $\mathfrak{S}'$-execution then η is also a complete $\mathfrak{S}$-execution. Hence we can conclude that

$$\eta \xrightarrow{\surd} q_n \;\text{ is a path } \pi \in \mathsf{Paths}(q, \mathfrak{S}) \cap \mathsf{Paths}(q, \mathfrak{S}').$$

Otherwise, if the execution η is not a complete execution in the above sense, we assume w.l.o.g. that $q_n \in Q_1$ and show that η can be extended to a $\mathfrak{S}$-execution $\eta' = \eta \xrightarrow{c} q_{n+1}$ of length $n{+}1$ for some $c \in \mathfrak{S}(\eta)$. As $\mathsf{active}(c) \subseteq N$ we observe that $c \in \mathfrak{S}'(\eta)$ and hence η' is also a $\mathfrak{S}'$-execution of length $n{+}1$. The execution η' is either $\mathfrak{S}'$-complete or can be further extended by some $c' \in \mathfrak{S}'(\eta')$ to an $\mathfrak{S}'$-execution of length $n{+}2$ which is also a $\mathfrak{S}$-execution now due to the fact the last state q_{n+2} of η' is in Q_2 and hence $\mathsf{active}(c') \subseteq \mathcal{N} \setminus N$.

Together with the observation that the execution $\eta = q_0 = q$ of length 0 is an $\mathfrak{S}$-execution and a $\mathfrak{S}'$-execution at the same time and by repeating the above procedure, we can construct a path $\pi \in \mathsf{Paths}(q, \mathfrak{S}) \cap \mathsf{Paths}(q, \mathfrak{S}')$.

□

4.3. Branching-time and alternating-time model checking

This subsection explains the model-checking procedures for ASL state formulas. First we will illustrate the model checking for the sublogic BTSL. We then present the methods for the ASL operators $\mathbb{E}_N(\Phi_1 \,\mathrm{U}\, \Phi_2)$, $\mathbb{E}_N(\Phi_1 \,\mathrm{R}\, \Phi_2)$, $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ and $\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi$ as the remaining operators can be handled with the help of standard methods for CTL model checking [CES86].

The model-checking problem for ASL asks whether, for a given constraint automaton $\mathcal{A}$ and ASL state formula Φ_0, all initial states q_0 of $\mathcal{A}$ satisfy Φ_0. The main procedure for ASL model checking follows the standard approach for CTL-like branching-time [CES86] and alternating-time logics [AHK02]. It recursively calculates the satisfaction sets

$$\mathsf{Sat}(\Phi) \stackrel{\text{def}}{=} \{q \in Q : q \models \Phi\}$$

for all subformulas Φ of Φ_0. Thus, the CTL fragment of ASL can be checked by means of a standard CTL model checker.

4.3.1. BTSL model checking

We explain here an algorithm for $\exists\langle\!\langle \mathcal{Z} \rangle\!\rangle\Phi$ and $\exists[\![\mathcal{Z}]\!]\Phi$. The treatment of formulas $\forall\langle\!\langle \mathcal{Z} \rangle\!\rangle\Phi$ and $\forall[\![\mathcal{Z}]\!]\Phi$ is obtained by the duality laws

$$\forall\langle\!\langle \mathcal{Z} \rangle\!\rangle\Phi \equiv \neg\exists[\![\mathcal{Z}]\!]\neg\Phi \ \text{ and } \ \forall[\![\mathcal{Z}]\!]\Phi \equiv \neg\exists\langle\!\langle \mathcal{Z} \rangle\!\rangle\neg\Phi.$$

For formulas of the form $\exists\langle\!\langle \mathcal{Z} \rangle\!\rangle\Phi$ or $\exists[\![\mathcal{Z}]\!]\Phi$, we build the product $\mathcal{A} \otimes \mathcal{Z}$ where the states are pairs $\langle q, z\rangle$ consisting of a state q in $\mathcal{A}$ and a state z in $\mathcal{Z}$.

Definition 4.3.1 (BTSL automata product)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L\rangle$ and $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ be as before. We define the automaton $\mathcal{A} \otimes \mathcal{Z}$ as follows:

$$\mathcal{A} \otimes \mathcal{Z} \stackrel{\text{def}}{=} \langle Q \times Z, \mathcal{N}, \longrightarrow_{\mathcal{A}\otimes\mathcal{Z}}, Q_0 \times Z_0, \mathcal{Q}_{\otimes}, AP', L'\rangle.$$

The transitions in $\mathcal{A} \otimes \mathcal{Z}$ are obtained by the following rules:

$$\frac{q \xrightarrow{c}_{\mathcal{A}} q' \ \wedge \ z \xrightarrow{c}_{\mathcal{Z}} z'}{\langle q, z\rangle \xrightarrow{c}_{\mathcal{A}\otimes\mathcal{Z}} \langle q', z'\rangle} \qquad \frac{q \text{ is quiescent in } \mathcal{A} \ \wedge \ z \xrightarrow{\surd}_{\mathcal{Z}} z'}{\langle q, z\rangle \xrightarrow{\surd}_{\mathcal{A}\otimes\mathcal{Z}} \langle q, z'\rangle} \tag{4.1}$$

where we use the subscripts $\mathcal{A}$, $\mathcal{Z}$ or $\mathcal{A} \otimes \mathcal{Z}$ for the transition relations in $\mathcal{A}$, $\mathcal{Z}$ and $\mathcal{A} \otimes \mathcal{Z}$, respectively. The atomic propositions and labeling function in $\mathcal{A} \otimes \mathcal{Z}$ are given by the set $AP' = AP \cup \{a_\Phi, \mathit{final}\}$ and the following conditions:

$$\begin{aligned} a_\Phi \in L'(\langle q, z\rangle) \quad &\text{iff} \quad q \models \Phi \\ \mathit{final} \in L'(\langle q, z\rangle) \quad &\text{iff} \quad z \in Z_F \\ a \in L'(\langle q, z\rangle) \quad &\text{iff} \quad a \in L(q) \end{aligned}$$

■

Please note, that $\mathcal{Z}$ and $\mathcal{A}$ make use of the same port-set $\mathcal{N}$ and thus (besides the special treatment of $\surd$) their product $\mathcal{A} \otimes \mathcal{Z}$ corresponds to the synchronous part of the constraint automaton product ($\bowtie$) as introduced by rule (2.2) in Definition 2.1.8.

The following proposition now provides a reduction to CTL.

Proposition 4.3.2 (Reduction to CTL)**.**

(a) $q \models_{\mathcal{A}} \exists\langle\mathcal{Z}\rangle\Phi$ iff there exists $z_0 \in Z_0$ with

$$\langle q, z_0\rangle \models_{\mathcal{A}\otimes\mathcal{Z}} \exists\Diamond(a_\Phi \wedge \mathit{final}).$$

(b) If $\mathcal{Z}$ is deterministic then

$$q \models_{\mathcal{A}} \exists[\![\mathcal{Z}]\!]\Phi \text{ iff } \langle q, z_0\rangle \models_{\mathcal{A}\otimes\mathcal{Z}} \exists\Box(\mathit{final} \rightarrow a_\Phi)$$

where z_0 is the initial state of $\mathcal{Z}$.

■

Proof.

(a) If $q \models_{\mathcal{A}} \exists\langle\mathcal{Z}\rangle\Phi$ then there exists a finite execution

$$\eta = \underset{\overset{\uparrow}{q}}{q_0} \xrightarrow{c_1} \ldots \xrightarrow{c_k} q_k \in \mathsf{Exec}_{\mathrm{fin}}(\mathcal{A})$$

such that $q = q_0$, $c_1 \ldots c_k \in \mathcal{L}(\mathcal{Z})$, and $q_k \models_{\mathcal{A}} \Phi$.

Let $\theta = z_0 \xrightarrow{c_1} \ldots \xrightarrow{c_k} z_k$ be an accepting run in $\mathcal{Z}$ for $c_1 \ldots c_k$, i.e., $z_k \in Z_F$ and $z_0 \in \mathcal{Z}_0$. Then,

$$\tilde{\eta} = \langle q_0, z_0\rangle \xrightarrow{c_1} \ldots \xrightarrow{c_k} \langle q_k, z_k\rangle \in \mathsf{Exec}_{\mathrm{fin}}(\mathcal{A}\otimes\mathcal{Z})$$

and hence, $\langle q, z_0\rangle \models_{\mathcal{A}\otimes\mathcal{Z}} \exists\Diamond(a_\Phi \wedge \mathit{final})$.

Let us now assume that $\langle q, z_0\rangle \models_{\mathcal{A}\otimes\mathcal{Z}} \exists\Diamond(a_\Phi \wedge \mathit{final})$ where $z_0 \in Z_0$. Then, there exists a finite execution

$$\tilde{\eta} = \langle q_0, z_0\rangle \xrightarrow{c_1} \ldots \xrightarrow{c_k} \langle q_k, z_k\rangle \in \mathsf{Exec}_{\mathrm{fin}}(\mathcal{A}\otimes\mathcal{Z})$$

with $\langle q_0, z_0\rangle \in Q_0 \times Z_0$ and $\langle q_k, z_k\rangle \models_{\mathcal{A}\otimes\mathcal{Z}} (a_\Phi \wedge \mathit{final})$, i.e., $q_k \models_{\mathcal{A}} \Phi$ and $z_k \in Z_F$. Thus,

$$\theta = z_0 \xrightarrow{c_1} \ldots \xrightarrow{c_k} z_k$$

is an accepting run for $c_1 \ldots c_k$ in $\mathcal{Z}$. This yields $c_1 \ldots c_k \in \mathcal{L}(\mathcal{Z})$. Since $q_k \models_{\mathcal{A}} \Phi$,

$$\eta = \overset{\overset{q}{\downarrow}}{q_0} \xrightarrow{c_1} \ldots \xrightarrow{c_k} q_k$$

is an execution in $\mathcal{A}$, which can be extended to a complete execution, i.e., a path π, such that $\pi \models \langle\mathcal{Z}\rangle\Phi$.

(b) Let $\mathcal{Z}$ be deterministic and z_0 the initial state of $\mathcal{Z}$. If $q \models_{\mathcal{A}} \exists[\![\mathcal{Z}]\!]\Phi$ then there exists a path

$$\pi = \underset{\overset{\uparrow}{q}}{q_0} \xrightarrow{c_1} q_1 \xrightarrow{c_2} q_2 \xrightarrow{c_3} \ldots \in \mathsf{Paths}(\mathcal{A})$$

such that $\pi \models [\![\mathcal{Z}]\!]\Phi$. Let us consider a prefix $\eta = \pi \downarrow k = q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_k} q_k$ of path π. If there is no corresponding run for $c_1 \ldots c_k$ in $\mathcal{Z}$ and hence $\mathcal{Z}$ does not accept $c_1 \ldots c_k$, π can be lifted to a path in $\mathcal{A} \otimes \mathcal{Z}$ where $\Box(\mathit{final} \to a_\Phi)$ holds. Otherwise let

$$\theta = z_0 \xrightarrow{c_1} \ldots \xrightarrow{c_k} z_k$$

be the unique run for $c_1 \ldots c_k$ in $\mathcal{Z}$. Then, $z_k \in Z_F$ implies $q_k \models_{\mathcal{A}} \Phi$. Thus, path π can be lifted to a path in $\mathcal{A} \otimes \mathcal{Z}$ where $\Box(\mathit{final} \to a_\Phi)$ holds. This yields $\langle q, z_0 \rangle \models_{\mathcal{A} \otimes \mathcal{Z}} \exists\Box(\mathit{final} \to a_\Phi)$.

Let us now assume that $\langle q, z_0 \rangle \models_{\mathcal{A} \otimes \mathcal{Z}} \exists\Box(\mathit{final} \to a_\Phi)$ and let

$$\tilde{\pi} = \langle q_0, z_0 \rangle \xrightarrow{c_1} \langle q_1, z_1 \rangle \xrightarrow{c_2} \ldots \in \mathsf{Paths}(\mathcal{A} \otimes \mathcal{Z})$$

be a path in $\mathcal{A} \otimes \mathcal{Z}$ where $q = q_0$ and $\tilde{\pi} \models_{\mathcal{A} \otimes \mathcal{Z}} \Box(\mathit{final} \to a_\Phi)$.

The projection of $\tilde{\pi}$ to the $\mathcal{A}$-components yields a path π in $\mathcal{A}$ starting in state q. We now show that $\pi \models_{\mathcal{A}} [\![\mathcal{Z}]\!]\Phi$. Let

$$\eta = \pi \downarrow k = \underset{\overset{\uparrow}{q}}{q_0} \xrightarrow{c_1} \ldots \xrightarrow{c_k} q_k$$

be a prefix of π. Then, $z_0 \xrightarrow{c_1} \ldots \xrightarrow{c_k} z_k$ is the (unique) run in $\mathcal{Z}$ for the word $c_1 \ldots c_k$. Since $\eta \models \Box(\mathit{final} \to a_\Phi)$, we have: $z_k \in Z_F$ implies $q_k \models_{\mathcal{A}} \Phi$. This yields the claim.

$\Box$

Part (a) of Proposition 4.3.2 allows to compute $\mathsf{Sat}(\exists\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi)$ by means of a backward reachability analysis in $\mathcal{A} \otimes \mathcal{Z}$ as shown in Algorithm 1, where $\mathsf{Pre}_{\mathcal{A} \otimes \mathcal{Z}}(P)$ returns the set of all predecessor states of $P \subseteq Q \times Z$ in $\mathcal{A} \otimes \mathcal{Z}$, i.e.,

$$\begin{aligned} \mathsf{Pre}_{\mathcal{A} \otimes \mathcal{Z}}(P) &\overset{\text{def}}{=} \{\langle q, z \rangle \in Q \times Z \mid \langle q, z \rangle \xrightarrow{c} \langle q', z' \rangle \in P\} \\ &\cup \quad \{\langle q, z \rangle \in Q \times Z \mid \langle q, z \rangle \xrightarrow{\checkmark} \langle q, z' \rangle \in P\} \end{aligned}$$

Algorithm 1 Computation of $\mathsf{Sat}\big(\exists\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi\big)$

$P := \{\langle q, z\rangle \in Q \times Z \mid q \in \mathsf{Sat}(\Phi) \wedge z \in Z_F\}$;
repeat
  $\mathcal{V} := P$;
  $P := P \cup \mathsf{Pre}_{\mathcal{A}\otimes\mathcal{Z}}(P)$;
until $(\mathcal{V} = P)$;
return $\{q \in Q \mid \exists z_0 \in Z_0 \text{ s.t. } \langle q, z_0\rangle \in P\}$;

Note that part (b) of Proposition 4.3.2 becomes wrong if $\mathcal{Z}$ is nondeterministic. For an NFA $\mathcal{Z}$, there may exist two runs

$$\theta = z_0 \xrightarrow{c_1}_{\mathcal{Z}} z_1 \xrightarrow{c_2}_{\mathcal{Z}} \ldots \quad \text{and} \quad \theta' = z_0' \xrightarrow{c_1}_{\mathcal{Z}} z_1' \xrightarrow{c_2}_{\mathcal{Z}} \ldots$$

in $\mathcal{Z}$ for the same sequence of concurrent I/O-operations, where only one of them is accepting. This fact may induce two different paths in the product automaton $\mathcal{A}\otimes\mathcal{Z}$ (corresponding to a single path in $\mathcal{A}$), one of them satisfying $\Box(\mathit{final} \rightarrow a_\Phi)$, while this property does not hold for the other path. The problem is that the non-accepting run yields a witness for

$$\exists\Box(\mathit{final} \rightarrow a_\Phi)$$

in the product. The following example illustrates this phenomenon.

Example 4.3.3 (NFA vs. DFA)**.** Figure 4.5 shows a constraint automaton $\mathcal{A}$ and an NFA $\mathcal{Z}$ where the boxed state $z_0 \in Z_F$ is accepting. Let $\mathcal{N} = \{A\}$ be the common port-set of $\mathcal{A}$

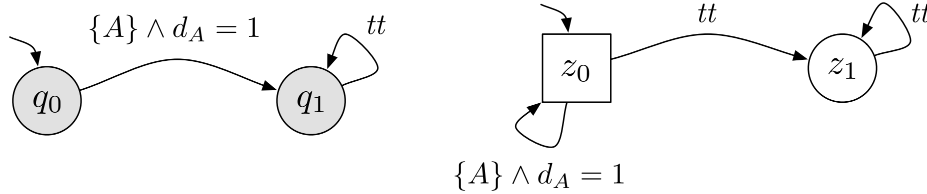


Figure 4.5: Constraint automaton $\mathcal{A}$ and NFA $\mathcal{Z}$ with $\mathcal{L}(\mathcal{Z}) = (c(A) = 1)^*$

and $\mathcal{Z}$. The recognized language of $\mathcal{Z}$ is

$$\mathcal{L}(\mathcal{Z}) = (c(A) = 1)^*$$

Assuming that $q_0 \models \Phi$ and $q_1 \not\models \Phi$ it becomes obvious that $\mathcal{A} \not\models \exists[\![\mathcal{Z}]\!]\Phi$.

When building the product as shown in Figure 4.6 it turns out that

$$\mathcal{A}\otimes\mathcal{Z} \models \exists\Box(\mathit{final} \rightarrow a_\Phi)$$

holds, since

$$\tilde{\pi} = \langle q_0, z_0\rangle \xrightarrow{c} \langle q_1, z_1\rangle \xrightarrow{c} \langle q_1, z_1\rangle \ldots \in \mathsf{Paths}(\mathcal{A}\otimes\mathcal{Z})$$

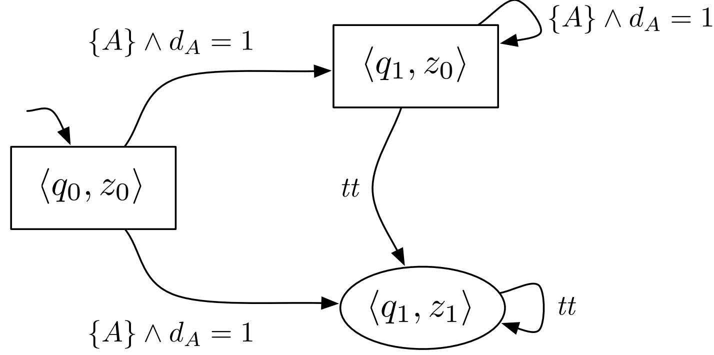


Figure 4.6: Product automaton $\mathcal{A} \otimes \mathcal{Z}$

with $c(A) = 1$ is a possible path within the product automaton.

■

For $\mathsf{Sat}(\exists[\![\mathcal{Z}]\!]\Phi)$, part (b) of Proposition 4.3.2, therefore, suggests to switch from $\mathcal{Z}$ to an equivalent deterministic finite automaton (DFA) and a fixpoint computation on the product of $\mathcal{A}$ and the DFA $\mathcal{Z}$. Algorithm 2 computes $\mathsf{Sat}(\exists[\![\mathcal{Z}]\!]\Phi)$.

Algorithm 2 Computation of $\mathsf{Sat}\big(\exists[\![\mathcal{Z}]\!]\Phi\big)$ for NFA $\mathcal{Z}$

construct a DFA $\mathcal{Z}' = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, z_0, Z_F\rangle$ from NFA $\mathcal{Z}$;
$P := \{\langle q, z\rangle \in Q \times Z \mid q \in \mathsf{Sat}(\Phi) \vee z \notin Z_F\}$;
repeat
  $\mathcal{V} := P$;
  $P := P \cap \mathsf{Pre}_{\mathcal{A}\otimes\mathcal{Z}'}(P)$;
until $(\mathcal{V} = P)$;
return $\{q \in Q \mid \langle q, z_0\rangle \in P\}$;

4.3.2. ASL model checking

The interesting part of the ASL model checking is the calculation of $\mathsf{Sat}(\mathbb{E}_N\varphi)$ for an ASL path formula φ and port-set $N \subseteq \mathcal{N}$. The essential ingredient for this is a modified predecessor operator $\mathsf{Pre}(P, N)$ for some $P \subseteq Q$, which in the case of ASL depends on the set N and is defined as the set of all states $q \in Q$ such that the agents controlling the N-ports have a strategy which guarantees to move within one step to a state in $P \subseteq Q$.

Definition 4.3.4 (Post, Pre-operator)**.** If c is a concurrent I/O-operation and q a state then

$$\mathsf{Post}[c](q) \stackrel{\text{def}}{=} \{p \in Q : q \stackrel{c}{\longrightarrow} p\}.$$

Let $P \subseteq Q$ and let $N \subseteq \mathcal{N}$ be a port-set. Then, $\mathsf{Pre}(P, N)$ denotes the set of all states $q \in Q$ such that the following two conditions hold:

(1) for all $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \cap N = \emptyset$ we have $\mathsf{Post}[c](q) \subseteq P$

(2) there exists $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \subseteq N$ and $\mathsf{Post}[c](q) \subseteq P$.

Recall that $\mathsf{CIO}(q)$ denotes the set of all concurrent I/O-operations that are enabled in state q.

■

Condition (1) is needed to ensure that no uncontrollable transition (from the view of the N-agents) leads to a state outside of P, while condition (2) asserts the existence of at least one concurrent I/O-operation that is controllable by the N-agents and certainly leads to a state in P.

Lemma 4.3.5 (Correctness of the Pre-operator)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L \rangle$ be a constraint automaton as before, $P \subseteq Q$ a set of states in $\mathcal{A}$, and let $N \subseteq \mathcal{N}$ be a port-set. Then,

$$\mathsf{Pre}(P, N) = \left\{ q \in Q \,:\, q \models \mathbb{E}_N \, \mathrm{X}\, P \right\}.$$

■

Proof. "$\subseteq$": Suppose that $q \in \mathsf{Pre}(P, N)$. Let $\mathfrak{S}$ be a memoryless N-strategy such that

$$\begin{aligned} \mathfrak{S}(q) \;=\; & \{c \in \mathsf{CIO} : \mathsf{active}(c) \cap N = \emptyset\} \,\cup \\ & \{c \in \mathsf{CIO}(q) : \mathsf{active}(c) \subseteq N \wedge \mathsf{Post}[c](q) \subseteq P\}. \end{aligned}$$

We then have:

- $\{c \in \mathfrak{S}(q) : \mathsf{active}(c) \subseteq N\} \neq \emptyset$ by condition (2) in Definition 4.3.4,
- $\mathsf{Post}[c](q) \subseteq P$ for all $c \in \mathfrak{S}(q) \cap \mathsf{CIO}(q)$ by condition (1) in Definition 4.3.4.

Hence, each $\mathfrak{S}$-complete $\mathfrak{S}$-execution η from q starts with a transition $q \xrightarrow{c} p$ where $c \in \mathfrak{S}(q) \cap \mathsf{CIO}(q)$. But then $\pi \models \mathrm{X}\, P$ for all $\pi \in \mathsf{Paths}(q, \mathfrak{S})$. Thus, $\mathfrak{S}$ yields a witness for $q \models \mathbb{E}_N \, \mathrm{X}\, P$.

"$\supseteq$": Suppose now that $q \models \mathbb{E}_N \, \mathrm{X}\, P$. We have to check conditions (1) and (2) in Definition 4.3.4. Let $\mathfrak{S}$ be a memoryless N-strategy winning for $\langle q, \, \mathrm{X}\, P \rangle$.

(1) Let $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \cap N = \emptyset$ and let $p \in \mathsf{Post}[c](q)$. By the requirements for N-strategies, we have $c \in \mathfrak{S}(q)$. Let π be an arbitrary $\mathfrak{S}$-path that starts with the transition $q \xrightarrow{c} p$. Since $\pi \models \mathrm{X}\, P$ we get $p \in P$.

(2) If q is quiescent then the execution q of length 0 cannot be $\mathfrak{S}$-complete, since

$$q \xrightarrow{\surd} q \not\models \mathtt{X}\,\mathsf{true}$$

and $\mathtt{X}\,P$ holds for all $\mathfrak{S}$-paths from q. Hence, there exists a concurrent I/O-operation $c \in \mathsf{CIO}(q) \cap \mathfrak{S}(q)$ such that $\mathsf{active}(c) \subseteq N$. We now show that $\mathsf{Post}[c](q) \subseteq P$. For each state $p \in \mathsf{Post}[c](q)$ there exists a $\mathfrak{S}$-path π that starts with the transition $q \xrightarrow{c} p$. As $\pi \models \mathtt{X}\,P$ we get $p \in P$.

□

As for standard CTL (and ATL), the semantics of the until and release operator have a fixed point characterization. The set $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2))$ is the least fixpoint, while the set $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2))$ is the greatest fixpoint of the following operators $2^Q \to 2^Q$:

$$\begin{array}{lll} P & \mapsto & \mathsf{Sat}(\Phi_2) \cup (\mathsf{Pre}(P, N) \cap \mathsf{Sat}(\Phi_1)) \qquad \text{(until)} \\ P & \mapsto & \mathsf{Sat}(\Phi_2) \cap (\mathsf{Pre}(P, N) \cup \mathsf{Sat}(\Phi_1)) \qquad \text{(release)} \end{array}$$

The treatment of the modalities until and release in ASL relies (as for standard CTL and ATL) on *expansion laws*. For ASL we have the following expansion laws:

$$\begin{array}{lll} \mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2) & \equiv & \Phi_2 \vee (\Phi_1 \wedge \mathbb{E}_N \,\mathtt{X}\, \mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2)) \\ \mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2) & \equiv & \Phi_2 \wedge (\Phi_1 \vee \mathbb{E}_N \,\mathtt{X}\, \mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2)) \end{array}$$

where $\equiv$ denotes equivalence of ASL state formulas. On the basis of the expansion laws, we obtain that for winning objectives formalized by ASL path formulas φ of the form $(\Phi_1 \,\mathtt{U}\, \Phi_2)$ or $(\Phi_1 \,\mathtt{R}\, \Phi_2)$, memoryless strategies are sufficient. Furthermore, the satisfaction set $\mathsf{Sat}(\mathbb{E}_N\varphi)$ can be computed by means of the standard procedures to compute least and greatest fixed points of monotonic operators.

The main steps are summarized in Algorithms 3 and 4 which, at the same time, compute a memoryless N-strategy $\mathfrak{S}$ that is winning for all states q where $\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2)$ (respectively $\mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2)$) holds. The correctness of these algorithms is stated in the following lemma.

Lemma 4.3.6 (Correctness of Algorithms 3 and 4)**.** Let $\mathcal{A}$ be a constraint automaton as before, $N \subseteq \mathcal{N}$ a port-set and let Φ_1 and Φ_2 be ASL state formulas. Then:

(a) Algorithm 3 correctly returns the set $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2))$ and the computed memoryless N-strategy $\mathfrak{S}$ is winning for all states $q \in \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2))$ and the ASL path formula $(\Phi_1 \,\mathtt{U}\, \Phi_2)$.

(b) Algorithm 4 correctly returns the set $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2))$ and the computed memoryless N-strategy $\mathfrak{S}$ is winning for all states $q \in \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2))$ and the ASL path formula $(\Phi_1 \,\mathtt{R}\, \Phi_2)$.

■

Algorithm 3 Algorithm for computing $\mathsf{Sat}\big(\mathbb{E}_N(\Phi_1 \mathbin{\mathrm{U}} \Phi_2)\big)$

$P_0 \;:=\; \mathsf{Sat}(\Phi_2)$;
$i := 0$;
repeat
  $P_{i+1} \;:=\; P_i \,\cup\, \big(\mathsf{Sat}(\Phi_1) \cap \mathsf{Pre}(P_i, N)\big)$;
  for all states $p \in P_{i+1} \setminus P_i$ **do**
    $\mathfrak{S}(p) \;:=\; \big\{\, c \in \mathsf{CIO} \;:\; \mathsf{active}(c) \cap N = \emptyset \,\vee\, \emptyset \neq \mathsf{Post}[c](p) \subseteq P_i \,\big\}$;
  end for
  $i := i{+}1$;
until $P_i = P_{i-1}$;
for all states $p \in (Q \setminus P_i) \cup \mathsf{Sat}(\Phi_2)$ **do**
  $\mathfrak{S}(p) \;:=\; \mathsf{CIO}_{\surd}$;
end for
return P_i; (* $P_i = \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \mathbin{\mathrm{U}} \Phi_2))$ *)

Algorithm 4 Algorithm for computing $\mathsf{Sat}\big(\mathbb{E}_N(\Phi_1 \mathbin{\mathrm{R}} \Phi_2)\big)$

for all states $p \in Q$ **do**
  $\mathfrak{S}(p) \;:=\; \mathsf{CIO}_{\surd}$;
end for
$P_0 \;:=\; \mathsf{Sat}(\Phi_2)$;
$i := 0$;
repeat
  $P_{i+1} \;:=\; P_i \,\cap\, \big(\mathsf{Sat}(\Phi_1) \cup \mathsf{Pre}(P_i, N)\big)$;
  for all states $p \in P_{i+1} \setminus \mathsf{Sat}(\Phi_1)$ **do**
    $\mathfrak{S}(p) \;:=\; \mathfrak{S}(p) \setminus \{\, c \in \mathsf{CIO}(p) \;:\; \mathsf{Post}[c](p) \not\subseteq P_i \,\}$;
  end for
  $i := i{+}1$;
until $P_i = P_{i-1}$;
return P_i; (* $P_i = \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \mathbin{\mathrm{R}} \Phi_2))$ *)

Proof. Both algorithms rely on the standard iterative approach to compute least and greatest fixed points of monotonic operators. This yields that the returned sets P_i agree with the satisfaction set $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathrm{U}\, \Phi_2))$ and $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathrm{R}\, \Phi_2))$, respectively. It remains to check that the computed strategies are winning.

ad (a). Let j be the number of iterations of the repeat-loop in Algorithm 3. Then:

$$\mathsf{Sat}(\Phi_2) \;=\; P_0 \subsetneq P_1 \subsetneq \ldots \subsetneq P_{j-1} \subsetneq P_j \;=\; \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathrm{U}\, \Phi_2))$$

Let us first observe that $\mathfrak{S}$ is indeed an N-strategy, i.e., $\mathfrak{S}(q)$ contains all $c \in \mathsf{CIO}$ where $\mathsf{active}(c) \cap N = \emptyset$. This condition is obvious for the states $q \in (Q \setminus P_j) \cup \mathsf{Sat}(\Phi_2)$. If q is a state in $P_{i+1} \setminus P_i$, then

$$\{c \in \mathsf{CIO} : \mathsf{active}(c) \cap N = \emptyset\} \;\subseteq\; \mathfrak{S}(q)$$

by the definition of $\mathfrak{S}(q)$.

It remains to check that $\Phi_1 \,\mathrm{U}\, \Phi_2$ holds for all $\mathfrak{S}$-paths that start in a state $q \in P_j$. This is obvious for $q \in \mathsf{Sat}(\Phi_2)$. Suppose $q \in P_{i+1} \setminus P_i$ where $0 \le i < j$. Then, $q \in \mathsf{Pre}(P_i, N)$. By the definition of the Pre-operator and the definition of $\mathfrak{S}(q)$ we obtain:

- $\mathsf{Post}[c](q) \subseteq P_i$ for all $c \in \mathfrak{S}(q)$ and
- $\mathfrak{S}(q) \cap \{c \in \mathsf{CIO}(q) : \mathsf{active}(c) \subseteq N\} \neq \emptyset$.

From this, we get by induction on n that for each finite $\mathfrak{S}$-execution

$$\eta \;=\; q_0 \xrightarrow{c_0} q_1 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n$$

where $q_0 \models \mathbb{E}_N(\Phi_1 \,\mathrm{U}\, \Phi_2)$ and $q_i \not\models \Phi_2$ for $0 \le i \le n$ the following two conditions hold:

- There exist indices $j \ge j_0 > j_1 > j_2 > \ldots > j_n$ such that $q_i \in P_{j_i}$ for $0 \le i \le n$.
- η is not $\mathfrak{S}$-complete, i.e., there is a $c \in \mathfrak{S}(q_n) \cap \mathsf{CIO}(q_n)$ with $\mathsf{active}(c) \subseteq N$.

As $P_i \subseteq \mathsf{Sat}(\Phi_1)$ for $i \ge 1$ and $P_0 = \mathsf{Sat}(\Phi_2)$ we obtain that $\pi \models (\Phi_1 \,\mathrm{U}\, \Phi_2)$ for each $\mathfrak{S}$-path π that starts in a state $q_0 \in P_j = \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathrm{U}\, \Phi_2))$.

ad (b). Let j be the number of iterations of the repeat-loop in Algorithm 4. Then, Algorithm 4 returns the set P_j and we have

$$\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathrm{R}\, \Phi_2)) \;=\; P_j \subsetneq P_{j-1} \subsetneq \ldots \subsetneq P_0 \;=\; \mathsf{Sat}(\Phi_2).$$

We first check that $\mathfrak{S}$ meets the condition required for N-strategies, i.e., $\mathfrak{S}$ does not rule out any non-controllable concurrent I/O-operation. This is obvious for the states $p \in \mathsf{Sat}(\Phi_1) \cup (Q \setminus P_j)$. If $p \in P_j \setminus \mathsf{Sat}(\Phi_1)$ then

$$\{c \in \mathsf{CIO}(p) : \mathsf{Post}[c](p) \subseteq P_j\} \;\subseteq\; \mathfrak{S}(p).$$

Furthermore, $p \in \mathsf{Pre}(P_j, N)$ which yields that $\mathsf{Post}[c](p) \subseteq P_j$ for all $c \in \mathsf{CIO}$ with $\mathsf{active}(c) \cap N = \emptyset$ by condition (1) in Definition 4.3.4.

We now show that $\Phi_1 \,\mathrm{R}\, \Phi_2$ holds for all $\mathfrak{S}$-paths that start in a state $q \in P_j$. For the states $q \in P_j \cap \mathsf{Sat}(\Phi_1)$ we have $q \models \Phi_1 \wedge \Phi_2$, and therefore $\pi \models (\Phi_1 \,\mathrm{R}\, \Phi_2)$ for all paths π starting in q. If $q \in P_j \setminus \mathsf{Sat}(\Phi_1)$ then for all $c \in \mathfrak{S}(q)$ we have $\mathsf{Post}[c](q) \subseteq P_j$ (by definition of $\mathfrak{S}$) and $q \in \mathsf{Pre}(P_j, N)$. The definition of the Pre-operator yields the existence of a concurrent I/O-operation $c \in \mathsf{CIO}(q)$ such that

$$\mathsf{active}(c) \subseteq N \text{ and } \mathsf{Post}[c](q) \subseteq P_j.$$

But then $c \in \mathfrak{S}(q)$, and each $\mathfrak{S}$-execution ending in q is $\mathfrak{S}$-incomplete.

Hence, each $\mathfrak{S}$-path $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots \in \mathsf{Paths}(q_0, \mathfrak{S})$ starting in a state $q_0 \in P_j$ is either

- infinite and consists of states in $P_j \setminus \mathsf{Sat}(\Phi_1)$ or
- has a prefix $q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} q_n$ where $q_0, \ldots, q_{n-1} \models \Phi_2$ and $q_n \models \Phi_1 \wedge \Phi_2$.

In both cases, we have $\pi \models (\Phi_1 \,\mathrm{R}\, \Phi_2)$.

□

It remains to provide algorithms for the computation of the satisfaction sets for ASL state formulas of the form $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ or $\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi$. In what follows, we assume that N is a nonempty subset of $\mathcal{N}$. To compute the satisfaction sets of $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ or $\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi$, we follow an automata-theoretic approach which resembles the standard automata-based LTL model-checking procedure and relies on model checking ASL state formulas of the form $\mathbb{E}_{\tilde{N}} \Diamond \Phi$ and $\mathbb{E}_{\tilde{N}} \Box \Phi$, respectively, in the product of $\mathcal{A}$ and $\mathcal{Z}$.

In the sequel, let $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, z_0, Z_F \rangle$ without loss of generality be a deterministic finite automaton. Given $\mathcal{A}$ and $\mathcal{Z}$, we built the product $\mathcal{A} \odot \mathcal{Z}$, similar to the product of finite automata and the join operator for constraint automata [BSAR06], but with special treatment of the pseudo-transitions with label $\surd$. In fact, the product construction we use here differs from those used in the BTSL model-checking procedure presented in the beginning of this section since in the context of the $\mathbb{E}_N$-operator we have to incorporate the possibilities of the N-agents to enforce termination. For this purpose, the product will use an additional controllable port $A_{\surd}$. Thus, for $\mathcal{A} \odot \mathcal{Z}$ we will ask for $(N \cup \{A_{\surd}\})$-strategies rather than N-strategies. Furthermore, let c_{stop} denote some concurrent I/O-operation with $\mathsf{active}(c_{\mathsf{stop}}) = \{A_{\surd}\}$. The data item $c_{\mathsf{stop}}(A_{\surd})$ is irrelevant. It can be an arbitrary element from the data domain Data. The treatment of ASL state formulas of the form $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ has been illustrated in Figure 4.7.

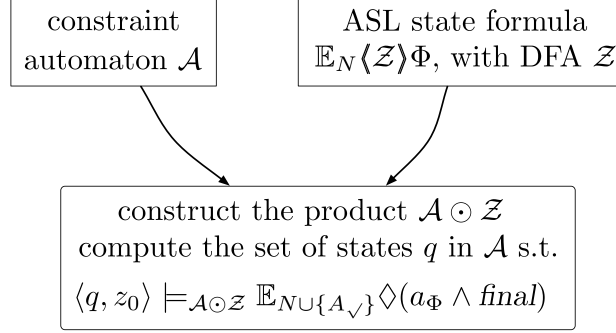


Figure 4.7: Schema for the treatment of $\mathbb{E}_N\langle\mathcal{Z}\rangle\Phi$

Definition 4.3.7 (ASL automata product)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L\rangle$ and $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, z_0, Z_F\rangle$ be as before. Furthermore, let $N \subseteq \mathcal{N}$ a port-set and Φ an ASL state formula. We define the product automaton $\mathcal{A}\odot_{N,\Phi}\mathcal{Z}$, or briefly $\mathcal{A}\odot\mathcal{Z}$ as follows:

$$\mathcal{A}\odot\mathcal{Z} \overset{\text{def}}{=} \langle Q\times Z, \mathcal{N}\cup\{A_{\surd}\}, \longrightarrow, Q_0\times\{z_0\}, \mathcal{Q}_{\odot}, AP', L'\rangle$$

where $A_{\surd}$ is a fresh port-name (not yet contained in $\mathcal{N}$). The transitions in $\mathcal{A}\odot\mathcal{Z}$ for q in $\mathcal{A}$ and state $z \in Z\setminus Z_{\surd}$ are obtained by the following synchronization rule which agrees with the first part of rule (4.1) in the BTSL product (cf. Definition 4.3.1).

$$\frac{q \xrightarrow{c}_{\mathcal{A}} q' \;\wedge\; z \xrightarrow{c}_{\mathcal{Z}} z'}{\langle q, z\rangle \xrightarrow{c}_{\mathcal{A}\odot\mathcal{Z}} \langle q', z'\rangle} \tag{4.2}$$

In addition, we have the following rules for each quiescent state q in $\mathcal{A}$ and state $z \in Z\setminus Z_{\surd}$ which replaces the second part of rule (4.1) for quiescent states in the BTSL product (cf. Definition 4.3.1).

$$\frac{z \xrightarrow{\surd}_{\mathcal{Z}} z' \;\wedge\; q \text{ is quiescent} \;\wedge\; \neg\exists c \in \mathsf{CIO}(q) \text{ s.t. } \mathsf{active}(c) \subseteq N}{\langle q, z\rangle \xrightarrow{c_\emptyset} \langle q, z'\rangle} \tag{4.3}$$

$$\frac{z \xrightarrow{\surd}_{\mathcal{Z}} z' \;\wedge\; q \text{ is quiescent} \;\wedge\; \exists c \in \mathsf{CIO}(q) \text{ s.t. } \mathsf{active}(c)\cap N \neq \emptyset}{\langle q, z\rangle \xrightarrow{c_{\mathsf{stop}}} \langle q, z'\rangle} \tag{4.4}$$

The atomic propositions and labeling function in $\mathcal{A}\odot\mathcal{Z}$ are the same as in the BTSL product $\mathcal{A}\otimes\mathcal{Z}$ and given by the set $AP' = AP\cup\{a_\Phi, \mathit{final}\}$ and the following conditions:

$$\begin{aligned} a_\Phi \in L'(\langle q, z\rangle) \quad &\text{iff} \quad q \models \Phi \\ \mathit{final} \in L'(\langle q, z\rangle) \quad &\text{iff} \quad z \in Z_F \\ a \in L'(\langle q, z\rangle) \quad &\text{iff} \quad a \in L(q). \end{aligned}$$

■

Rule (4.3) in Definition 4.3.7 formalizes the fact that if q is quiescent and there is no $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \subseteq N$ then the opponents of the N-agents may refuse any write or read operation and can therefore enforce dataflow to stop. This is modeled in the product by a transition with the label $c_\emptyset$. Rule (4.4) stands for the fact that whenever q is a quiescent state for which some concurrent I/O-operation c is enabled where the N-ports are involved then the N-agents might decide not to participate in any further I/O-operation. This is modeled in the product by a transition with the label c_{stop} where the new port $A_{\surd}$ is supposed to be controllable.

Properties of the ASL product. As stated in the beginning of Section 4.2, we suppose $\mathcal{Z}$ contains a special subset $Z_{\surd} \subseteq Z$ such that for all states $z \in Z$:

$$z \xrightarrow{\surd}_{\mathcal{Z}} z' \text{ implies } z' \in Z_{\surd}.$$

Thus, for all transitions of the product generated by the rules (4.3) and (4.4) in Definition 4.3.7, a state in $\langle q, z \rangle \in S_{\surd}$

$$S_{\surd} \stackrel{\text{def}}{=} Q \times Z_{\surd}$$

will be entered. For technical reasons we add self-loops with label $c_\emptyset$ for the states in $S_{\surd}$:

$$\langle q, z \rangle \xrightarrow{c_\emptyset} \langle q, z \rangle$$

These transitions ensure that all paths in the product that eventually enter a state $\langle q, z \rangle \in S_{\surd}$ are infinite and repeat state $\langle q, z \rangle$ forever.

The following lemma states some properties of quiescent states in the product. Remember that a quiescent state in a constraint automaton might have outgoing transitions with nonempty port-sets (cf. Definition 4.1.2).

Lemma 4.3.8 (Quiescent states in the ASL product)**.** If $\langle q, z \rangle$ is quiescent in $\mathcal{A} \odot \mathcal{Z}$ then

(a) q is quiescent in $\mathcal{A}$,

(b) there is a $c \in \mathsf{CIO}(q)$ such that $\emptyset \neq \mathsf{active}(c) \subseteq N$, and

(c) c_{stop} is enabled in $\langle q, z \rangle$.

■

Proof. Let $\langle q, z \rangle$ be a quiescent state in the product. Thus, $c_\emptyset \notin \mathsf{CIO}(\langle q, z \rangle)$.

ad (a). Each transition $q \xrightarrow{c_\emptyset}_{\mathcal{A}} p$ in $\mathcal{A}$ can be lifted to a transition in the product:

$$\langle q, z \rangle \xrightarrow{c_\emptyset} \langle p, z' \rangle$$

where $z \xrightarrow{c_\emptyset} z'$. Hence, if state q is not quiescent in $\mathcal{A}$, then $\langle q, z\rangle$ is not quiescent in the product.

ad (b). There is some concurrent I/O-operation $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \subseteq N$, as otherwise rule (4.3) would yield that $c_\emptyset$ is enabled in $\langle q, z\rangle$.

ad (c). We show that $c_{\mathsf{stop}} \in \mathsf{CIO}(\langle q, z\rangle)$. This can be seen as follows:

We have $c_\emptyset \notin \mathsf{CIO}(q)$ (as q is quiescent in $\mathcal{A}$ by part (b)). Suppose by contradiction that $c_{\mathsf{stop}} \notin \mathsf{CIO}(\langle q, z\rangle)$. Then, there is no $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \cap N \neq \emptyset$ (premise of rule (4.4) in Definition 4.3.7). But then

$$\mathsf{active}(c) \cap N = \emptyset \text{ for all } c \in \mathsf{CIO}(q).$$

As $c_\emptyset \notin \mathsf{CIO}(q)$, there is no $c \in \mathsf{CIO}(q)$ such that $\mathsf{active}(c) \subseteq N$. But then rule (4.3) in Definition 4.3.7 yields $c_\emptyset \in \mathsf{CIO}(\langle q, z\rangle)$. This contradicts the assumption that $\langle q, z\rangle$ is quiescent in the product.

□

We will now investigate the relationship of paths in $\mathcal{A}$ and paths in the product $\mathcal{A} \odot \mathcal{Z}$. For this we need the notion of $\mathcal{A}$-projections of paths in the product which are roughly obtained by dropping the $\mathcal{Z}$-components from the states. Moreover, we will need the notion of $\mathcal{Z}$-projection which is defined analogously by dropping the $\mathcal{A}$-components.

Definition 4.3.9 ($\mathcal{A}$-projections, $\mathcal{Z}$-projections)**.** For each transition in the product (obtained by one of the above composition rules) we define its projection to $\mathcal{A}$ as follows:

- If $\langle q, z\rangle \xrightarrow{c} \langle q', z'\rangle$ arises by applying rule (4.2) in Definition 4.3.7 then its $\mathcal{A}$-projection is $q \xrightarrow{c}_{\mathcal{A}} q'$.
- If $\langle q, z\rangle \xrightarrow{c} \langle q, z'\rangle$ (where $z' \in Z_{\surd}$ and $c \in \{c_\emptyset, c_{\mathsf{stop}}\}$) is obtained from rule (4.3) or rule (4.4) in Definition 4.3.7 then the $\mathcal{A}$-projection is $q \xrightarrow{\surd}_{\mathcal{A}} q$.
- The $\mathcal{A}$-projection of the pseudo-transition $\langle q, z\rangle \xrightarrow{\surd} \langle q, z\rangle$ that might appear at the end of a finite path in $\mathcal{A} \odot \mathcal{Z}$ is $q \xrightarrow{\surd}_{\mathcal{A}} q$.

Given a path $\tilde{\pi}$ in the product $\mathcal{A} \odot \mathcal{Z}$ that does not enter a state of the form $\langle q, z\rangle \in S_{\surd}$ then we define the $\mathcal{A}$-projection $\mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$ as the unique path in $\mathcal{A}$ that results by taking the $\mathcal{A}$-projection of all transitions in $\tilde{\pi}$. For a path $\tilde{\pi}$ that eventually enters a state of the form $\langle q, z\rangle$ with $z \in Z_F$ we ignore the (infinite) suffix

$$\langle q, z\rangle \xrightarrow{c_\emptyset} \langle q, z\rangle \xrightarrow{c_\emptyset} \ldots$$

and define $\mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$ as the $\mathcal{A}$-projection of the prefix of π that leads to $\langle q, z\rangle$.

Similarly, we define the $\mathcal{Z}$-projection $\mathsf{proj}_{\mathcal{Z}}(\tilde{\pi})$ as an infinite or finite sequence of elements in $\mathcal{Z}$ of the same length as $\mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$. Then, if $\tilde{\pi}$ starts in a state $\langle q_0, z_0\rangle$ (where z_0 is the initial state of $\mathcal{Z}$) then $\mathsf{proj}_{\mathcal{Z}}(\tilde{\pi})$ is the run for the I/O-stream of $\mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$ in $\mathcal{Z}$. The definitions of the projections are extended for executions (i.e., prefixes of paths) in the obvious way.

■

Lemma 4.3.10 (Projection of paths in the ASL product)**.** If $\tilde{\pi}$ is a path in $\mathcal{A} \odot \mathcal{Z}$ then $\mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$ is a path in $\mathcal{A}$.

■

Proof. By Lemma 4.3.8, all paths in the product are infinite or end in a state $\langle q, z\rangle$ where q is quiescent in $\mathcal{A}$ and c_{stop} is enabled in $\langle q, z\rangle$. The projection of an infinite path $\tilde{\pi}$ in the product that never enters a state in $S_{\surd}$ is an infinite path in $\mathcal{A}$, since all their transitions arise by rule (4.2) (i.e., their labels are concurrent I/O-operations for the original name-set $\mathcal{N}$). The same holds for all finite paths in the product that do not enter $S_{\surd}$. They end in a quiescent state $\langle q, z\rangle$ of the product. But then q is quiescent in $\mathcal{A}$ and the $\mathcal{A}$-projection is a finite path in $\mathcal{A}$. Paths in the product $\mathcal{A} \odot \mathcal{Z}$ that eventually enter a state in $S_{\surd}$ are infinite, but they are projected to finite paths in $\mathcal{A}$.

□

Lemma 4.3.11 (ASL path formulas $\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi$)**.** For each path $\tilde{\pi}$ in the product $\mathcal{A} \odot \mathcal{Z}$ starting in a state $\langle q, z_0\rangle$ we have:

$$\tilde{\pi} \models \Diamond(a_\Phi \wedge \mathit{final}) \quad \text{iff} \quad \mathsf{proj}_{\mathcal{A}}(\tilde{\pi}) \models \langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi.$$

■

Proof. Let $\tilde{\pi}$ be a path in the product starting in a state $\langle q, z_0\rangle$ and let $\pi \stackrel{\text{def}}{=} \mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$ be its $\mathcal{A}$-projection.

"$\Longrightarrow$": Suppose first that $\tilde{\pi} \models \Diamond(a_\Phi \wedge \mathit{final})$. Then, $\tilde{\pi}$ has a finite prefix that leads to a state $\langle p, z\rangle$ where $(a_\Phi \wedge \mathit{final})$ holds. Hence, $p \models \Phi$ and $z \in Z_F$. Let n be the length of this prefix and let $z_0\, z_1\, \ldots\, z_n$ be the sequence of states obtained by the projection $\mathsf{proj}_{\mathcal{Z}}(\tilde{\pi} \downarrow n)$ and $c_1 \ldots c_n = \mathsf{ios}(\tilde{\pi} \downarrow n)$ its I/O-stream. We may suppose that $n \leq |\pi|$. (Note that paths that eventually enter a state $\langle p, z\rangle \in S_{\surd}$ stay in this state $\langle p, z\rangle$ forever.) Then, we have:

- $c_1 \ldots c_n = \mathsf{ios}(\pi \downarrow n)$
- $z_0\, z_1\, \ldots\, z_n$ is the run for $c_1 \ldots c_n$ in $\mathcal{Z}$ and $z_n = z \in Z_F$

But then $c_1 \ldots c_n$ is accepted by $\mathcal{Z}$ and we get $c_1 \ldots c_n \in \mathcal{L}(\mathcal{Z})$. Furthermore, the last state of $\pi \downarrow n$ is p. Since Φ holds in p, this yields $\pi \models \langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi$.

"$\Longleftarrow$": Suppose now that $\pi \models \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$. Then, there is some prefix $\pi \downarrow n$ of π such that its I/O-stream $\mathsf{ios}(\pi \downarrow n)$ belongs to $\mathcal{L}(\mathcal{Z})$ and the last state p of $\pi \downarrow n$ belongs to $\mathsf{Sat}(\Phi)$. Let $z_0, \ldots, z_n$ be the run for $\mathsf{ios}(\pi \downarrow n)$ in $\mathcal{Z}$. Then, $z_n \in Z_F$ and state $\langle p, z_n \rangle$ is the last state of $\tilde{\pi} \downarrow n$. As

$$\langle p, z_n \rangle \models (a_\phi \wedge \mathit{final})$$

we get $\tilde{\pi} \models \Diamond(a_\Phi \wedge \mathit{final})$.

□

We will now investigate the relation between strategies in $\mathcal{A}$ and strategies in the product $\mathcal{A} \odot \mathcal{Z}$.

Lemma 4.3.12 (Correspondence of strategies in $\mathcal{A}$ and $\mathcal{A} \odot \mathcal{Z}$)**.** Let $\mathfrak{S}$ be an N-strategy for $\mathcal{A}$ and $\tilde{\mathfrak{S}}$ an $(N \cup \{A_{\surd}\})$-strategy for $\mathcal{A} \odot \mathcal{Z}$ such that for all finite executions $\tilde{\eta}$ in $\mathcal{A} \odot \mathcal{Z}$ starting in a state $\langle q, z_0 \rangle$ the following conditions hold:

(a) If $c \in \mathsf{CIO}$ is a concurrent I/O-operation then

$$c \in \tilde{\mathfrak{S}}(\tilde{\eta}) \text{ iff } c \in \mathfrak{S}(\mathsf{proj}_{\mathcal{A}}(\tilde{\eta}))$$

(b) $c_{\mathsf{stop}} \in \tilde{\mathfrak{S}}(\tilde{\eta})$ iff $\surd \in \mathfrak{S}(\mathsf{proj}_{\mathcal{A}}(\tilde{\eta}))$

(c) $\surd \notin \tilde{\mathfrak{S}}(\tilde{\eta})$.

Then, the $\mathcal{A}$-projections of the $\tilde{\mathfrak{S}}$-paths starting in a state $\langle q, z_0 \rangle$ are exactly the $\mathfrak{S}$-paths starting in q.

■

Proof. We first show that the $\mathcal{A}$-projection of each $\tilde{\mathfrak{S}}$-path is a $\mathfrak{S}$-path:

- Each $\tilde{\mathfrak{S}}$-execution that enters $S_{\surd}$ via a transition $\langle p, z \rangle \xrightarrow{c_\emptyset} \langle p, z' \rangle$ with $z' \in Z_{\surd}$ is projected to a finite path π that ends with the pseudo-transition

  $$p \xrightarrow{\surd}_{\mathcal{A}} p.$$

  By the premise of rule (4.3) in Definition 4.3.7, we get that $\mathsf{active}(c) \setminus N \neq \emptyset$ for all $c \in \mathsf{CIO}(p)$. If $n = |\pi| - 1$ then the prefix $\pi \downarrow n$ of π leads from some initial state $q_0 \in Q_0$ to state p and constitutes a $\mathfrak{S}$-complete $\mathfrak{S}$-execution. Hence, π is a finite $\mathfrak{S}$-path.

- Each $\tilde{\mathfrak{S}}$-execution that enters $S_{\surd}$ via a transition $\langle p, z \rangle \xrightarrow{c_{\mathsf{stop}}} \langle p, z' \rangle$ with $z' \in Z_{\surd}$ is also projected to a finite path π that ends with the pseudo-transition

  $$p \xrightarrow{\surd}_{\mathcal{A}} p.$$

  Again, let $n = |\pi| - 1$. We regard the prefix $\tilde{\eta} = \tilde{\pi} \downarrow n$ of $\tilde{\pi}$ that leads from the first state $\langle q, z_0 \rangle$ of $\tilde{\pi}$ to $\langle p, z \rangle$. Then, the concurrent I/O-operation c_{stop} belongs to $\tilde{\mathfrak{S}}(\tilde{\eta})$. By the second assumption on the relation between $\tilde{\mathfrak{S}}$ and $\mathfrak{S}$ we get:

$$\surd \in \mathfrak{S}(\eta) \text{ for the } \mathcal{A}\text{-projection } \eta = \mathsf{proj}_{\mathcal{A}}(\tilde{\eta}).$$

State p is quiescent by the premise of rule (4.4) in Definition 4.3.7. Hence, we get the execution $\eta = \pi \downarrow n$ is $\mathfrak{S}$-complete. Therefore, π is a finite $\mathfrak{S}$-path.

- We now regard a $\tilde{\mathfrak{S}}$-complete finite execution $\tilde{\eta}$ that does not visit $S_{\surd}$ and ends in state $\langle p, z\rangle$. Then, $\langle p, z\rangle$ is quiescent and there is no concurrent I/O-operation

$$\tilde{c} \in \tilde{\mathfrak{S}}(\tilde{\eta}) \cap \mathsf{CIO}(\langle p, z\rangle)$$

such that $\mathsf{active}(\tilde{c}) \subseteq N \cup \{A_{\surd}\}$. Hence, there is also no

$$c \in \mathfrak{S}(\eta) \cap \mathsf{CIO}(p)$$

such that $\mathsf{active}(c) \subseteq N$. But then $\eta \overset{\text{def}}{=} \mathsf{proj}_{\mathcal{A}}(\tilde{\eta})$ is a $\mathfrak{S}$-complete execution and therefore the corresponding path π is a $\mathfrak{S}$-path.

- Given an infinite $\tilde{\mathfrak{S}}$-execution $\tilde{\eta}$ that does not enter $S_{\surd}$, its $\mathcal{A}$-projection is an infinite path in $\mathcal{A}$ and therefore a $\mathfrak{S}$-path.

This shows that the projections of all $\tilde{\mathfrak{S}}$-paths are $\mathfrak{S}$-paths.

We now regard a $\mathfrak{S}$-path π in $\mathcal{A}$ and show that it is the $\mathcal{A}$-projection of some $\tilde{\mathfrak{S}}$-path. This is obvious if π is infinite, since then it can be lifted to an infinite $\tilde{\mathfrak{S}}$-path in the product that does not enter $S_{\surd}$. Assume now that the path

$$\pi \;=\; q_0 \xrightarrow{c_1}_{\mathcal{A}} \ldots \xrightarrow{c_n}_{\mathcal{A}} q_n \xrightarrow{\surd}_{\mathcal{A}} q_n$$

is finite and of length $n{+}1$. Let $z_0\, z_1\, \ldots\, z_n\, z_{n+1}$ be the run for the I/O-stream $c_1 \ldots c_n \surd$ of π. Then,

$$\tilde{\eta} \;=\; \langle q_0, z_0\rangle \xrightarrow{c_1} \ldots \xrightarrow{c_n} \langle q_n, z_n\rangle$$

is a $\tilde{\mathfrak{S}}$-execution in the product and its projection $\eta \overset{\text{def}}{=} \mathsf{proj}_{\mathcal{A}}(\tilde{\eta}) = \pi \downarrow n$ is $\mathfrak{S}$-complete. Hence, q_n is quiescent and at least one of the following two conditions (1) or (2) holds:

(1) $\surd \in \mathfrak{S}(\eta)$

(2) there is no CIO $c \in \mathfrak{S}(\eta) \cap \mathsf{CIO}(q_n)$ such that $\mathsf{active}(c) \subseteq N$.

If $\langle q_n, z_n\rangle$ is non-quiescent then (because of rule (4.3) in Definition 4.3.7) $c_\emptyset$ is enabled in $\langle q_n, z_n\rangle$ and we have:

$$\mathsf{Post}[c_\emptyset](\langle q_n, z_n\rangle) = \{\langle q_n, z'\rangle\}$$

where $z_n \xrightarrow{c_{\mathsf{stop}}} z'$. But then π is the projection of the infinite $\tilde{\mathfrak{S}}$-path

$$\tilde{\eta} \xrightarrow{c_\emptyset} \langle q_n, z'\rangle \xrightarrow{c_\emptyset} \langle q_n, z'\rangle \xrightarrow{c_\emptyset} \ldots$$

where $z_0 \xrightarrow{c_{\mathsf{stop}}} z'$.

Let us now assume that $\langle q_n, z_n \rangle$ is quiescent, i.e., $c_\emptyset$ is not enabled in $\langle q_n, z_n \rangle$. Then,

$$\emptyset \neq \mathsf{active}(c) \subseteq N \text{ for all } c \in \mathsf{CIO}(q_n)$$

because otherwise the premise of rule (4.3) in Definition 4.3.7 applies and $\langle q_n, z_n \rangle$ would be non-quiescent.

Suppose first that case (1) applies. Then, $c_{\mathsf{stop}} \in \tilde{\mathfrak{S}}(\tilde{\eta})$ and

$$\tilde{\eta} \xrightarrow{c_{\mathsf{stop}}} \langle q_n, z' \rangle \xrightarrow{c_\emptyset} \langle q_n, z' \rangle \xrightarrow{c_\emptyset} \dots$$

is an infinite $\tilde{\mathfrak{S}}$-path where $z_0 \xrightarrow{c_{\mathsf{stop}}} z'$ and its $\mathcal{A}$-projection is π.

Suppose now that case (2), but not case (1) applies. That is, $\surd \notin \mathfrak{S}(\eta)$ and there is no concurrent I/O-operation $c \in \mathfrak{S}(\eta) \cap \mathsf{CIO}(q_n)$ such that $\mathsf{active}(c) \subseteq N$. Then, $c_{\mathsf{stop}} \notin \tilde{\mathfrak{S}}(\tilde{\eta})$. Hence, there is no concurrent I/O-operation

$$\tilde{c} \in \tilde{\mathfrak{S}}(\tilde{\eta}) \cap \mathsf{CIO}(\langle q_n, z_n \rangle) \text{ such that } \mathsf{active}(\tilde{c}) \subseteq N \cup \{A_\surd\}.$$

But then $\tilde{\eta}$ is $\tilde{\mathfrak{S}}$-complete and

$$\tilde{\eta} \xrightarrow{\surd} \langle q_n, z_n \rangle$$

is a $\tilde{\mathfrak{S}}$-path and its projection is π.

□

The following lemma formalizes the reduction of the model-checking problem for an ASL state formula of the form $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ to the problem of computing satisfaction sets for formulas of the type $\mathbb{E}_{N \cup \{A_\surd\}} \Diamond \Phi$ in the product (which can be treated via Algorithm 3 as presented on Page 74). Furthermore, it asserts the existence of finite-memory winning strategies for objectives of the form $\langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$.

Lemma 4.3.13 (Treatment of $\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$)**.** Let $\mathcal{A}$ be a constraint automaton, and $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, z_0, Z_F \rangle$ a DFA. Furthermore, let q be a state in $\mathcal{A}$, $N \subseteq \mathcal{N}$ and Φ an ASL state formula. Then, the following statements are equivalent:

(a) $q \models \mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ in $\mathcal{A}$

(b) $\langle q, z_0 \rangle \models \mathbb{E}_{N \cup \{A_\surd\}} \Diamond (a_\Phi \wedge \textit{final})$ in $\mathcal{A} \odot \mathcal{Z}$

(c) there exists a finite-memory N-strategy $\mathfrak{S}$ that is winning for $\langle q, \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi \rangle$.

■

Proof. The implication (c) "$\Longrightarrow$" (a) is obvious.

"(a) $\Longrightarrow$ (b)": Suppose that $q \models \mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ and that $\mathfrak{S}$ is an N-strategy for $\mathcal{A}$ that is winning for $\langle q, \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi \rangle$. The goal is to define a corresponding $(N \cup \{A_{\surd}\})$-strategy $\tilde{\mathfrak{S}}$ for $\mathcal{A} \odot \mathcal{Z}$.

Given a finite execution $\tilde{\eta}$ in the product we take its $\mathcal{A}$-projection $\eta \stackrel{\text{def}}{=} \mathsf{proj}_{\mathcal{A}}(\tilde{\eta})$ and define

$$\tilde{\mathfrak{S}}(\tilde{\eta}) \stackrel{\text{def}}{=} \begin{cases} \mathfrak{S}(\eta) & : \text{ if } \surd \notin \mathfrak{S}(\eta) \\ (\mathfrak{S}(\eta) \setminus \{\surd\}) \cup \{c_{\mathsf{stop}}\} & : \text{ otherwise.} \end{cases}$$

Then, $\tilde{\mathfrak{S}}$ and $\mathfrak{S}$ are related as required in Lemma 4.3.12. Furthermore, for each path $\tilde{\pi}$ in the product starting in a state $\langle q, z_0 \rangle$ we have:

$$\tilde{\pi} \models \Diamond(a_\Phi \wedge \mathit{final}) \quad \text{iff} \quad \mathsf{proj}_{\mathcal{A}}(\tilde{\pi}) \models \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$$

as shown in Lemma 4.3.11. Hence, $\tilde{\mathfrak{S}}$ is winning for the state $\langle q, z_0 \rangle$ in the product and the objective $\Diamond(a_\Phi \wedge \mathit{final})$.

"(b) $\Longrightarrow$ (c)": Suppose $\langle q, z_0 \rangle \models \mathbb{E}_{N \cup \{A_{\surd}\}} \Diamond(a_\Phi \wedge \mathit{final})$. Part (a) of Lemma 4.3.6 asserts the existence of a memoryless $(N \cup \{A_{\surd}\})$-strategy $\tilde{\mathfrak{S}}$ for $\mathcal{A} \odot \mathcal{Z}$ that is winning for $\langle \langle q, z_0 \rangle, \Diamond(a_\Phi \wedge \mathit{final}) \rangle$.

We now define a finite-memory N-strategy $\mathfrak{M} = (\mathsf{Modes}, \Delta, \mu, m_0)$ for $\mathcal{A}$ as follows. The set of modes agrees with the state-space of $\mathcal{Z}$, i.e., $\mathsf{Modes} = Z$. The decision function μ is given by:

$$\mu(q, z) \stackrel{\text{def}}{=} \begin{cases} \tilde{\mathfrak{S}}(\langle q, z \rangle) & : \text{ if } c_{\mathsf{stop}} \notin \tilde{\mathfrak{S}}(\langle q, z \rangle) \\ (\tilde{\mathfrak{S}}(\langle q, z \rangle) \setminus \{c_{\mathsf{stop}}\}) \cup \{\surd\} & : \text{ otherwise.} \end{cases}$$

The transition relation Δ is defined by $\Delta(z, q \xrightarrow{c} p) \stackrel{\text{def}}{=} z'$ where $z \xrightarrow{c} z'$.

It remains to show that $\mathfrak{M}$ is winning for $\langle q, \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi \rangle$. In fact, $\tilde{\mathfrak{S}}$ and $\mathfrak{M}$ are related as in Lemma 4.3.12. Again, we make use of the fact that $\tilde{\pi} \models \Diamond(a_\Phi \wedge \mathit{final})$ if and only if $\mathsf{proj}_{\mathcal{A}}(\tilde{\pi}) \models \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$, provided that $\tilde{\pi}$ is a path in the product starting in a state $\langle q, z_0 \rangle$. Thus, we may conclude that $\mathfrak{M}$ is winning for $\langle q, \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi \rangle$.

□

There is an analogous result for ASL state formulas of the type $\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi$ where the objective $[\![\mathcal{Z}]\!] \Phi$ in $\mathcal{A}$ is transformed to the objective $\Box(\mathit{final} \rightarrow a_\Phi)$ in the product. The treatment of ASL state formulas of the form $\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi$ has been illustrated in Figure 4.8.

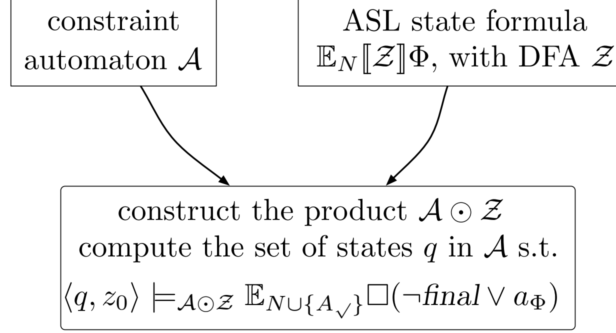


Figure 4.8: Schema for the treatment of $\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$

Lemma 4.3.14 (Treatment of $\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$)**.** Let $\mathcal{A}$ be a constraint automaton, q be a state in $\mathcal{A}$ and $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, z_0, Z_F \rangle$ a DFA. Furthermore, let $N \subseteq \mathcal{N}$ and Φ an ASL state formula. Then, for all states $q \in Q$ the following statements are equivalent:

(a) $q \models \mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$ in $\mathcal{A}$

(b) $\langle q, z_0 \rangle \models \mathbb{E}_{N\cup\{A_{\surd}\}}\Box(\mathit{final} \rightarrow a_\Phi)$ in $\mathcal{A} \odot \mathcal{Z}$

(c) there exists a finite memory N-strategy $\mathfrak{S}$ which is winning for $\langle q, [\![\mathcal{Z}]\!]\Phi \rangle$.

■

Proof. The argument is analogous to the proof of Lemma 4.3.13 and relies on the observation that

$$\tilde{\pi} \models \Box(\mathit{final} \rightarrow a_\Phi) \quad \text{iff} \quad \mathsf{proj}_{\mathcal{A}}(\tilde{\pi}) \models [\![\mathcal{Z}]\!]\Phi$$

for each path $\tilde{\pi}$ in the product starting in a state $\langle q, z_0 \rangle$. This is shown in the following Lemma.

□

Lemma 4.3.15 (ASL path formulas $[\![\mathcal{Z}]\!]\Phi$)**.** For each path $\tilde{\pi}$ in the product starting in a state $\langle q, z_0 \rangle$ we have:

$$\tilde{\pi} \models \Box(\mathit{final} \rightarrow a_\Phi) \quad \text{iff} \quad \mathsf{proj}_{\mathcal{A}}(\tilde{\pi}) \models [\![\mathcal{Z}]\!]\Phi.$$

■

Proof. Let $\tilde{\pi} = \langle q_0, z_0 \rangle \xrightarrow{c_1} \langle q_1, z_1 \rangle \xrightarrow{c_2} \ldots$ be a path in the product starting in a state $\langle q_0, z_0 \rangle$ and let $\pi \stackrel{\text{def}}{=} \mathsf{proj}_{\mathcal{A}}(\tilde{\pi})$ be its $\mathcal{A}$-projection.

"$\Longrightarrow$": Suppose first that $\tilde{\pi} \models \Box(\mathit{final} \to a_\Phi)$. Let $n \leq |\pi|$ such that

$$\mathsf{ios}(\pi \downarrow n) \in \mathcal{L}(\mathcal{Z}).$$

Then, $z_0\, z_1\, \ldots\, z_n$ is the run for $\mathsf{ios}(\pi \downarrow n)$ in $\mathcal{Z}$. Hence, $z_n \in Z_F$ and therefore we have $\langle q_n, z_n \rangle \models \mathit{final}$. As

$$\tilde{\pi} \models \Box(\mathit{final} \to a_\Phi)$$

and $\langle q_n, z_n \rangle$ is the $(n{+}1)$-st state of $\tilde{\pi}$, we have $\langle q_n, z_n \rangle \models a_\Phi$. This yields that $q_n \models \Phi$, and therefore $\pi \models [\![\mathcal{Z}]\!]\Phi$.

"$\Longleftarrow$": Assume $\pi \models [\![\mathcal{Z}]\!]\Phi$. Let $n \leq |\tilde{\pi}|$. The goal is to show that the $(n{+}1)$-st state of $\tilde{\pi}$ satisfies $(\mathit{final} \to a_\Phi)$, i.e.,

$$\langle q_n, z_n \rangle \models (\mathit{final} \to a_\Phi).$$

This is obvious if *final* does not hold for $\langle q_n, z_n \rangle$. Assume now that $\langle q_n, z_n \rangle \models \mathit{final}$. Then, $z_n \in Z_F$.

- If $\langle q_n, z_n \rangle \notin S_{\surd}$ then $n \leq |\pi|$ and $z_0\, z_1\, \ldots\, z_n$ is the run for $\mathsf{ios}(\pi \downarrow n)$ in $\mathcal{Z}$. As $z_n \in Z_F$ we get that

  $$\mathsf{ios}(\pi \downarrow n) \in \mathcal{L}(\mathcal{Z}),$$

  and therefore $q_n \models \Phi$. But then, $\langle q_n, z_n \rangle \models a_\Phi$.

- If $\langle q_n, z_n \rangle \in S_{\surd}$ then there is some $m \leq n$ such that $m \leq |\pi|$ and

  $$\langle q_m, z_m \rangle = \langle q_{m+1}, z_{m+1} \rangle = \ldots = \langle q_n, z_n \rangle.$$

  Then, $z_0\, z_1 \ldots z_m$ is the run for $\mathsf{ios}(\pi \downarrow m)$ in $\mathcal{Z}$. As $z_m = z_n \in Z_F$ we get that

  $$\mathsf{ios}(\pi \downarrow m) \in \mathcal{L}(\mathcal{Z}).$$

  But this yields $q_m \models \Phi$. Hence, $\langle q_m, z_m \rangle = \langle q_n, z_n \rangle \models a_\Phi$.

$\Box$

Thanks to Lemma 4.3.13 and Lemma 4.3.14 the satisfaction sets

$$\mathsf{Sat}(\mathbb{E}_N \langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi) \text{ and } \mathsf{Sat}(\mathbb{E}_N [\![\mathcal{Z}]\!] \Phi)$$

can be computed by means of a reduction to the model-checking problem for the $\mathbb{E}_N$-operator in combination with the eventually- and always-modalities. More precisely, we first build the product $\mathcal{A} \odot \mathcal{Z}$ and then apply the algorithm for until and release, respectively, to compute the satisfaction sets for

$$\mathbb{E}_{N \cup \{A_{\surd}\}} \Diamond(a_\Phi \wedge \mathit{final}) \text{ and } \mathbb{E}_{N \cup \{A_{\surd}\}} \Box(\mathit{final} \to a_\Phi)$$

in the product. Furthermore, the memoryless winning $(N \cup \{A_{\surd}\})$-strategies for the product and the goals

$$\Diamond(a_\Phi \wedge \mathit{final}) \text{ and } \Box(\mathit{final} \to a_\Phi)$$

yield finite-memory winning N-strategies in $\mathcal{A}$ for the objectives $\langle\!\langle \mathcal{Z} \rangle\!\rangle \Phi$ and $[\![\mathcal{Z}]\!]\Phi$, respectively.

$\surd$-free stream expressions. We conclude this section by a simple observation concerning the case α is a $\surd$-free expression (i.e., does not contain a subexpression of the form $\beta; \surd$). In fact, for $\surd$-free expressions, the "best" strategy for the N-agents to ensure $[\![\mathcal{Z}_\alpha]\!]\Phi$ is to stop the dataflow whenever possible. This is formalized in the following lemma:

Lemma 4.3.16 (Winning strategies for $\surd$-free expressions)**.** Let $\mathfrak{S}_\surd$ be the memoryless N-strategy given by $\mathfrak{S}_\surd(q) = \{\surd\} \cup \{c \in \mathsf{CIO} : \mathsf{active}(c) \cap N = \emptyset\}$ for all states q. Then, for each $\surd$-free stream expression α and state q we have:

$$q \models \mathbb{E}_N[\![\mathcal{Z}_\alpha]\!]\Phi \;\text{ iff }\; \mathfrak{S}_\surd \text{ is winning for } \langle q, [\![\mathcal{Z}_\alpha]\!]\Phi\rangle.$$

■

Proof. The implication $\Longrightarrow$ is obvious by the semantics for the modality $\mathbb{E}_N$. Suppose now that $q \models \mathbb{E}_N[\![\mathcal{Z}_\alpha]\!]\Phi$. The goal is to show that $\mathfrak{S}_\surd$ is a winning strategy for $\langle q, [\![\mathcal{Z}_\alpha]\!]\Phi\rangle$. We pick a winning strategy $\tilde{\mathfrak{S}}$ for $\langle q, [\![\mathcal{Z}_\alpha]\!]\Phi\rangle$. That is, $\pi \models [\![\mathcal{Z}_\alpha]\!]\Phi$ for all $\tilde{\mathfrak{S}}$-paths π that start in state q. By definition of $\mathfrak{S}_\surd$ we get that for each incomplete $\mathfrak{S}_\surd$-execution $\eta = q_0 \xrightarrow{c_1} \ldots \xrightarrow{c_i} q_i$ we have:

$$\mathfrak{S}_\surd(q_i) \cap \mathsf{CIO}(q_i) \;\subseteq\; \tilde{\mathfrak{S}}(\eta)$$

Hence, all incomplete executions in $\mathsf{Exec}_{\mathsf{fin}}(q, \mathfrak{S}_\surd)$ are prefixes of $\tilde{\mathfrak{S}}$-executions. Thus, if $\eta \in \mathsf{Exec}_{\mathsf{fin}}(q, \mathfrak{S}_\surd)$ and η is an incomplete $\tilde{\mathfrak{S}}$-execution starting in q then we have:

$$\mathsf{ios}(\eta) \in \mathsf{IOS}(\alpha) \text{ implies } p \models \Phi \text{ where } p \text{ is the last state of } \eta.$$

In particular, this yields $\pi \models [\![\mathcal{Z}_\alpha]\!]\Phi$ for all infinite $\mathfrak{S}_\surd$-paths that start in q. As α is $\surd$-free, none of the I/O-streams in $\mathsf{IOS}(\alpha)$ contains the termination symbol $\surd$. Hence, for each finite $\mathfrak{S}_\surd$-path π we have $\mathsf{ios}(\pi) \notin \mathsf{IOS}(\alpha)$ and therefore $\pi \models [\![\mathcal{Z}_\alpha]\!]\Phi$.

□

Thus, if α is $\surd$-free then the set $\mathsf{Sat}(\mathbb{E}_N[\![\mathcal{Z}_\alpha]\!]\Phi)$ can be computed by considering the sub-automaton $\mathcal{A}'$ of $\mathcal{A}$ that results from the memoryless strategy $\mathfrak{S}_\surd$ and then computing the satisfaction set for $\mathsf{Sat}_{\mathcal{A}'}(\forall[\![\mathcal{Z}_\alpha]\!]\Phi)$ in $\mathcal{A}'$. This can be done by means of a BTSL model checker as described in the beginning of Section 4.3.

In the next Section we will discuss the complexity of solving the ASL (and BTSL) model-checking problem in detail.

4.4. Complexity of the ASL model-checking problem

We first study the asymptotic worst-case time complexity of the ASL model-checking algorithm sketched in Section 4.3. For this, we assume an explicit representation of $\mathcal{A}$ as a directed graph with list representations for the enabled concurrent I/O-constraints $c \in \mathsf{CIO}(q)$ and the successor sets $\mathsf{Post}[c](q)$ for each state q. The size $|\mathcal{A}|$ of $\mathcal{A}$ denotes the number of states plus the total number of transitions.

The time required to derive the predecessor-sets $\mathsf{Pre}(P)$ and $\mathsf{Pre}(P, N)$ respectively from the explicit representation of $\mathcal{A}$ is polynomially bounded in $|\mathcal{A}|$. The precise time complexity depends on the data structures used for P and N and the organization of the additional information of the concurrent I/O-constraints. Such details are irrelevant for the following considerations.

The treatment of the ASL until or release operator on the basis of Algorithms 3 and 4 relies on a standard fixed point computation and requires at most $|Q|$ iterations. Hence, assuming that $\mathsf{Sat}(\Phi_1)$ and $\mathsf{Sat}(\Phi_2)$ have already been computed, then both algorithms run in time $\mathcal{O}(\text{poly}(|\mathcal{A}|))$. We now address the time complexity of the automata-based approach to compute the satisfaction sets of ASL state formulas of the form $\mathbb{E}_N\langle\!\langle \mathcal{Z} \rangle\!\rangle\Phi$ and $\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$. Assuming that $\mathsf{Sat}(\Phi)$ and a DFA $\mathcal{Z}$ are given, the time complexity is polynomial in the size of the product constraint automaton $\mathcal{A} \odot \mathcal{Z}$.

Remark 4.4.1. When starting with a regular stream expression α or an NFA rather than a DFA $\mathcal{Z}$, in the worst-case the size of $\mathcal{Z}$ for α can be exponential in the length of α. These observations yield that the time complexity of the proposed ASL model-checking procedure is polynomial in $|\mathcal{A}|$ and exponential in the length of the given ASL state formula Φ_0. Hence, the model complexity (i.e., for fixed formula) is polynomial, and the ASL model-checking problem (where both the constraint automaton $\mathcal{A}$ and the ASL state formula Φ_0 are viewed as inputs) belongs to the complexity class EXPTIME.

■

Lower complexity bound for ASL model-checking. In fact, we do not expect much better ASL model-checking algorithms, since we will prove the ASL (and BTSL) model-checking problem to be PSPACE-hard.

Theorem 4.4.2 (Lower bound for ASL (and BTSL) model-checking)**.** The ASL (and BTSL) model-checking problems are PSPACE-hard as the following two problems are PSPACE-hard:

(a) given a constraint automaton $\mathcal{A}$ with port-set $N \subseteq \mathcal{N}$, state q_0 in $\mathcal{A}$ and an NFA $\mathcal{Z}$ over the alphabet $\mathsf{CIO}_{\surd}$, check whether $q_0 \models \mathbb{E}_{\mathcal{N}}[\![\mathcal{Z}]\!]\,\mathsf{false}$.

(b) given a constraint automaton $\mathcal{A}$ with port-set $\mathcal{N}$, state q_0 in $\mathcal{A}$ and an NFA $\mathcal{Z}$ over the alphabet $\mathsf{CIO}_{\surd}$, check whether $q_0 \models \exists[\![\mathcal{Z}]\!]\,\mathsf{false}$.

■

Proof. From the results by Chadha et al. [CSV09], it can be derived that the following problem is PSPACE-hard:

- *given:* an NFA $\mathcal{Z}$ over some alphabet Σ that does not accept the empty word
- *question:* does there exists an infinite word $\vartheta = \sigma_1\sigma_2\sigma_3 \ldots \in \Sigma^\omega$ such that no finite prefix of ϑ belongs to the language of $\mathcal{Z}$, i.e., $\sigma_1\sigma_2 \ldots \sigma_n \notin \mathcal{L}(\mathcal{Z})$ for all $n \geq 1$?

We now provide polynomial reductions from the above problem to the model-checking problem for formulas of the type $\mathbb{E}_N[\![\mathcal{Z}]\!]\,\mathsf{false}$ and BTSL formulas of the type $\exists[\![\mathcal{Z}]\!]\,\mathsf{false}$. The letters in the alphabet Σ are treated as concurrent I/O-constraints. Let $\mathcal{A}$ be a constraint automaton with a single state q_0 and the transitions

$$q_0 \xrightarrow{\sigma} q_0 \quad \text{for all } \sigma \in \Sigma.$$

Formally, we may deal with the name-set $\mathcal{N} \stackrel{\text{def}}{=} \{A_\sigma : \sigma \in \Sigma\}$ and introduce self-loops of state q_0 labeled with concurrent I/O-operations σ such that $\mathsf{active}(\sigma) = \{A_\sigma\}$ for all $\sigma \in \Sigma$. In fact, it suffices to restrict Σ to the letters that appear as labels of at least one transition in $\mathcal{Z}$. Then, it is obvious that $\mathcal{A}$ can be constructed in time linear in the size of $\mathcal{Z}$.

Let $\mathcal{Z} \cup tt^* ; \surd$ denote an NFA for the language $\mathcal{L}(\mathcal{Z}) \cup (\mathsf{CIO}^* \surd)$. That is, $\mathcal{Z} \cup tt^* ; \surd$ accepts all finite I/O-streams that end by the termination symbol $\surd$ and all I/O-streams $c_1\, c_2 \ldots c_n \in \mathsf{CIO}^*$ that are accepted by $\mathcal{Z}$, while all other words are rejected by $\mathcal{Z} \cup tt^* ; \surd$. Note that such an NFA can be obtained by adding two states to $\mathcal{Z}$. Hence, we can safely assume that $\mathcal{Z} \cup tt^* ; \surd$ can be constructed from $\mathcal{Z}$ in polynomial time. We now consider the formulas

$$\Psi_1 \stackrel{\text{def}}{=} \mathbb{E}_{\mathcal{N}}[\![\mathcal{Z} \cup tt^* ; \surd]\!]\,\mathsf{false}$$
$$\Psi_2 \stackrel{\text{def}}{=} \exists[\![\mathcal{Z} \cup tt^* ; \surd]\!]\,\mathsf{false}$$

and show the equivalence of the following three statements:

(1) $q_0 \models \Psi_1$

(2) $q_0 \models \Psi_2$

(3) there exists an infinite word $\vartheta \in \Sigma^\omega$ such that no finite prefix of ϑ belongs to $\mathcal{L}(\mathcal{Z})$.

"(1) $\Longrightarrow$ (2)": Suppose $q_0 \models \Psi_1$. Then, there exists a strategy $\mathfrak{S}$ for all ports (i.e., for the name-set $N = \mathcal{N}$) such that $\pi \models [\![\mathcal{Z} \cup tt^* ; \surd]\!]\,\mathsf{false}$ for all $\pi \in \mathsf{Paths}(q_0, \mathfrak{S})$. Hence, we may pick an arbitrary $\mathfrak{S}$-path π that starts in state q_0 to obtain a witness for $q_0 \models \exists[\![\mathcal{Z} \cup tt^* ; \surd]\!]\,\mathsf{false}$.

"(2) $\Longrightarrow$ (3)": Suppose $q_0 \models \Psi_2$. Then, there exists a path π that starts in state q_0 and fulfills the path formula $[\![\mathcal{Z} \cup tt^* ; \surd]\!]\,\mathsf{false}$. This path π must be infinite, because otherwise π would have a finite prefix with an I/O-stream in $\mathsf{IOS}(tt^* ; \surd) \subseteq \mathsf{IOS}(\mathcal{Z} \cup tt^* ; \surd)$ which is impossible as $\pi \models [\![\mathcal{Z} \cup tt^* ; \surd]\!]\,\mathsf{false}$. Say π has the form

$$q_0 \xrightarrow{\sigma_1} q_0 \xrightarrow{\sigma_2} q_0 \xrightarrow{\sigma_3} \ldots$$

and let $\vartheta \stackrel{\text{def}}{=} \sigma_1\sigma_2\sigma_3 \ldots \in \Sigma^\omega$ be the infinite I/O-stream of π. Let $n \in \mathbb{N}$. The I/O-stream of the finite prefix $\pi \downarrow n$ is the finite prefix $\sigma_1 \ldots \sigma_n$ of ϑ. As $\pi \models [\![\mathcal{Z} \cup tt^* \,;\, \surd]\!]\,\mathsf{false}$ we get that $\sigma_1 \ldots \sigma_n \notin \mathcal{L}(\mathcal{Z})$.

"(3) $\Longrightarrow$ (1)": Suppose $\vartheta = \sigma_1\sigma_2\sigma_3 \ldots \in \Sigma^\omega$ is an infinite word such that $\sigma_1 \ldots \sigma_n \notin \mathcal{L}(\mathcal{Z})$ for all $n \in \mathbb{N}$. We now consider the $\mathcal{N}$-strategy $\mathfrak{S}$ given by

$$\mathfrak{S}(\eta) \stackrel{\text{def}}{=} \{\sigma_n\} \text{ if } \eta \text{ is an execution of length } n{-}1.$$

Then, there is a single $\mathfrak{S}$-path in $\mathcal{A}$, namely

$$\pi = q_0 \xrightarrow{\sigma_1} q_0 \xrightarrow{\sigma_2} q_0 \xrightarrow{\sigma_3} \ldots$$

Since $\pi \models [\![\mathcal{Z} \cup tt^* \,;\, \surd]\!]\,\mathsf{false}$ the strategy $\mathfrak{S}$ is winning for $\langle q_0, [\![\mathcal{Z} \cup tt^* \,;\, \surd]\!]\,\mathsf{false}\rangle$. Hence, $q_0 \models \Psi_1$.

As the length of both formulas Ψ_1 and Ψ_2 is polynomial in the size of $\mathcal{Z}$, this completes the proof of Theorem 4.4.2.

□

Part (a) of Theorem 4.4.2 shows that the NFA-based approach for the ASL model-checking problem is computationally hard, even under the assumption that all ports cooperate in a single coalition. For the special case that no port is controllable, the ASL model-checking problem is computationally hard as well, as the ASL formula $\forall_\emptyset[\![\mathcal{Z}]\!]\Phi$ is equivalent the BTSL formula $\exists[\![\mathcal{Z}]\!]\Phi$ that has been shown to be PSPACE-hard in Part (b) of Theorem 4.4.2.

Complexity of ASL (and BTSL) model-checking. By Theorem 4.4.2, we obtain PSPACE-hardness of the ASL (and the BTSL) model-checking problem. At this place it is worth noting that there is a polynomially time-bounded algorithm for the model-checking problem for BTSL formulas of the form $\exists\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi$, provided that $\mathsf{Sat}(\Phi)$ is given. In this case, an automata-based approach with NFA rather than DFA is applicable.

PSPACE is also an upper bound for the BTSL model-checking problem. The algorithms to compute the satisfaction sets of BTSL state formulas of the type $\exists(\Phi_1 \,\mathrm{U}\, \Phi_2)$, $\exists(\Phi_1 \,\mathrm{R}\, \Phi_2)$ and $\exists\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi$ are polynomially time-bounded, and therefore polynomially space-bounded. It remains to show that the satisfaction sets for BTSL state formulas of the type $\exists[\![\mathcal{Z}]\!]\Phi$ can be computed by a polynomial space-bounded algorithm.

Algorithm 5 is a nondeterministic decision procedure for $q_0 \models \exists[\![\mathcal{Z}]\!]a$, where

- $q_0 \in Q_0$ is a state in the constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle$,
- $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ is an NFA, and
- $a \in AP$ an atomic proposition

The decision procedure relies on an on-the-fly construction of the product of $\mathcal{A}$ and the determinized version of $\mathcal{Z}$. The states space in the product is $S = Q \times 2^Z$, where Q is the state space of $\mathcal{A}$ and Z is the state space of $\mathcal{Z}$. The presented algorithm performs a forward search in the product and constructs a witness path $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ in $\mathcal{A}$ such that $\pi \models [\![\mathcal{Z}]\!]a$ holds.

Algorithm 5 Nondeterministic decision procedure for $q_0 \models \exists[\![\mathcal{Z}]\!]a$.

$\langle q, M\rangle := \langle q_0, Z_0\rangle$;

for $i = 1 \ldots (|Q| \cdot 2^{|Z|})$ **do**
  if (($q \not\models a$) and $z \in Z_F$ for some $z \in M$) **then**
    return "$q_0 \not\models \exists[\![\mathcal{Z}]\!]a$";
  end if

  if (q is terminal) **then**
    return "$q_0 \models \exists[\![\mathcal{Z}]\!]a$";
  end if

  guess some transition $q \xrightarrow{c} q'$ in $\mathcal{A}$ or pair $\langle q', c\rangle \stackrel{\text{def}}{=} \langle q, c_{\textsf{stop}}\rangle$ nondeterministically;
  $\langle q, M\rangle := \langle q', \{z' \in Z \mid z \xrightarrow{c}_{\mathcal{Z}} z' \text{ for some } z \in M\}\rangle$;
end for

return "$q_0 \models \exists[\![\mathcal{Z}]\!]a$";

As Algorithm 5 is an on-the-fly variant of Algorithm 2 which applied standard methods for the determinization of $\mathcal{Z}$, the algorithm correctly decides whether $q_0 \models \exists[\![\mathcal{Z}]\!]a$.

The time-complexity of the above algorithm is linear in the number of states in $\mathcal{A}$ and exponential in the number of states in the NFA $\mathcal{Z}$. As it is sufficient to store only the last state of the product that will be visited throughout the execution of the for-loop, the complexity in space is bounded by the number of bits used to store one state of the product only, i.e., $\mathcal{O}(\log_2(|Q| \cdot 2^{|Z|}))$. Hence, the problem to decide "$q_0 \models \exists[\![\mathcal{Z}]\!]a$" can be solved nondeterministically with space linear in the size of $\mathcal{Z}$ and logarithmic in the size of $\mathcal{A}$. With these observations, we obtain:

Lemma 4.4.3 (Upper bound for the complexity of BTSL model checking)**.** The BTSL model-checking problem is in PSPACE.

■

Corollary 4.4.4 (PSPACE-completeness for BTSL)**.** From part (b) of Theorem 4.4.2 and Lemma 4.4.3 we can conclude that the BTSL model-checking problem is PSPACE-complete.

■

For the ASL model checking we present a similar result. We show that PSPACE is also an upper bound for the ASL model-checking problem. The algorithms to compute the satisfaction sets of ASL state formulas of the type $\mathbb{E}_N(\Phi_1 \, \mathrm{U} \, \Phi_2)$ and $\mathbb{E}_N(\Phi_1 \, \mathrm{R} \, \Phi_2)$ are polynomially time-bounded, and therefore polynomially space-bounded.

It remains to show that the satisfaction sets for ASL state formulas of the type $\mathbb{E}_N\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi$ and $\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$ can be computed by a polynomial space-bounded algorithm. We start with formulas of the type $\mathbb{E}_N\langle\!\langle\mathcal{Z}\rangle\!\rangle\Phi$. The result can be established by providing a polynomial reduction to the emptiness problem for alternating finite automata (AFA) with unary alphabet which is known to be PSPACE-complete [Hol95]. More precisely, we provide a polynomial reduction from the ASL model-checking problem

$$\text{“does } q_0 \models \mathbb{E}_N\langle\!\langle\mathcal{Z}\rangle\!\rangle a \text{ hold?”}$$

where q_0 is a state of a constraint automaton $\mathcal{A}$, $\mathcal{Z}$ an NFA, $N \subseteq \mathcal{N}$ a set of port-names and $a \in AP$ an atomic proposition. We combine $\mathcal{A}$ and $\mathcal{Z}$ to obtain an alternating finite automaton that is non-empty if and only if $q_0 \models \mathbb{E}_N\langle\!\langle\mathcal{Z}\rangle\!\rangle a$.

The idea behind the construction of an AFA from $\mathcal{A}$ and $\mathcal{Z}$ is to create a two-player game structure, which separates transitions in $\mathcal{A} \odot \mathcal{Z}$ into three steps. In the first step the N-player may select one of the enabled actions, whereas in the second step the opponent can select a compatible action as well as the successor state in $\mathcal{A}$. In the last step the N-player chooses a successor state in $\mathcal{Z}$.

Definition 4.4.5 (Alternating finite automaton, e.g., [JS07])**.** An alternating finite automaton is a tuple $\mathcal{S} = \langle S, \Sigma, \Delta, s_0, S_F\rangle$ where S is a finite set of states, Σ a finite alphabet,

$$\Delta : S \times \Sigma \to \text{Bool}^+(S)$$

is the transition function, s_0 is the initial state, and $S_F \subseteq S$ the set of accepting states. Here, $\text{Bool}^+(S)$ denotes the set of (positive) boolean formulas that only use $\wedge$ and $\vee$ as boolean connectives and elements of the state space S as variables.

For the acceptance we define the relation $\text{Acc} \subseteq S \times \Sigma^*$ by induction on the length of the input word $\vartheta \in \Sigma^*$. We write $\text{Acc}(s, \vartheta)$ if $\langle s, \vartheta\rangle \in \text{Acc}$.

$$\begin{aligned}
\text{Acc}(s, \varepsilon) &\iff s \in S_F \\
\text{Acc}(s, \nu\vartheta) &\iff v \models \Delta(s, \nu) \text{ for } \nu \in \Sigma \text{ and the boolean assignment } v \text{ satisfying} \\
&\qquad (v(s') = 1 \iff \text{Acc}(s', \vartheta)) \text{ for all } s' \in Q.
\end{aligned}$$

With this definition an AFA $\mathcal{A}$ accepts the language $\mathcal{L}(\mathcal{A}) \stackrel{\text{def}}{=} \{\vartheta \in \Sigma^* \mid \text{Acc}(s_0, \vartheta)\}$.

■

Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton, $q_0 \in Q$ a state in $\mathcal{A}$, $\Phi = \mathbb{E}_N\langle\!\langle\mathcal{Z}\rangle\!\rangle a$, where $N \subseteq \mathcal{N}$ a set of ports, $a \in AP$ an atomic proposition, and $\mathcal{Z} = \langle Z, \mathsf{CIO}_{\surd}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ be an NFA. We construct an AFA $\mathcal{S} = \langle S, \Sigma, \Delta, s_0, S_F\rangle$ as follows:

– $S \stackrel{\text{def}}{=} \{s_0\} \cup (Q \times Z) \cup$

$\{\langle q, z, c\rangle \in Q \times Z \times \mathsf{CIO}_{\surd} \mid c = c_{\mathsf{stop}} \text{ or } c \in \mathsf{CIO}(q) \cap \mathsf{CIO}(z)\} \cup$

$\{\langle q, c, z\rangle \in Q \times \mathsf{CIO}_{\surd} \times Z \mid c = c_{\mathsf{stop}} \text{ or } c \in \mathsf{CIO}(z)\}$,

– $\Sigma \stackrel{\text{def}}{=} \{\nu\}$ where ν is an arbitrary input symbol,

– $s_0 \in S$ the starting state, and

– $S_F \stackrel{\text{def}}{=} \{\langle q, z\rangle \in Q \times Z \mid q \models a \text{ and } z \in Z_F\}$

– As the input alphabet Σ is a singleton set, the transition function can now be interpreted as a function $\Delta : S \to \text{Bool}^+(S)$ which is given by the following definition

$$\Delta(s) \stackrel{\text{def}}{=} \begin{cases} \bigwedge_{s' \in \lambda(s)} s' & \text{if } s \in \{s_0\} \cup (Q \times Z \times \mathsf{CIO}_{\surd}) \\ \bigvee_{s' \in \lambda(\langle q,z\rangle)} s' & \text{if } s \in (Q \times Z) \cup (Q \times \mathsf{CIO}_{\surd} \times Z) \end{cases}$$

Here, λ is a total function $\lambda : S \to 2^S$ which is given by

$$\begin{aligned} \lambda(s_0) &\stackrel{\text{def}}{=} \{\langle q_0, z\rangle \in S \mid z \in Z_0\} \\ \lambda(\langle q, z\rangle) &\stackrel{\text{def}}{=} \{\langle q, z, c\rangle \in S \mid c \in \mathsf{CIO}(\langle q, z\rangle) \text{ in } \mathcal{A} \odot \mathcal{Z}\} \\ \lambda(\langle q, z, c\rangle) &\stackrel{\text{def}}{=} \{\langle q', c', z\rangle \in S \mid \langle q, z\rangle \xrightarrow{c'}_{\mathcal{A} \odot \mathcal{Z}} \langle q', z'\rangle \text{ with } c' \in \text{compAct}(c, N)\} \\ \lambda(\langle q, c, z\rangle) &\stackrel{\text{def}}{=} \{\langle q, z'\rangle \in S \mid z \xrightarrow{c} z'\} \end{aligned}$$

where $\text{compAct}(c, N)$ stands for the set of all c-compatible actions when $N \subseteq \mathcal{N}$ is the set of controllable ports, i.e.,

$$\begin{aligned} \text{compAct}(c, N) \stackrel{\text{def}}{=} & \{c\} \\ \cup & \{c' \in \mathsf{CIO} \mid \mathsf{active}(c') \cap N = \emptyset\} \\ \cup & \{c_{\mathsf{stop}} \mid \text{if } \mathsf{active}(c) \setminus N \neq \emptyset\} \end{aligned}$$

The structure of the AFA above is constructed in such a way that the states $s \in (Q \times Z) \cup (Q \times \mathsf{CIO}_{\surd} \times Z)$ where the N-player moves the transition function has a disjunctive format, i.e., $\Delta(s) = s_1 \vee \ldots \vee s_n$, and whenever the opponent moves in a state $s \in \{s_0\} \cup (Q \times Z \times \mathsf{CIO}_{\surd})$ the transition function has a conjunctive format, i.e., $\Delta(s) = s_1 \wedge \ldots \wedge s_n$. The accepted language of the constructed AFA will be non-empty if the N-player can choose (at least) one action and later a successor state in $\mathcal{Z}$ in any of his game configurations such that all of the opponents moves lead to a state $s \in S_F$. Hence, we have that

$$\mathcal{L}(\mathcal{S}) \neq \emptyset \text{ iff } q_0 \models \mathbb{E}_N\langle\!\langle \mathcal{Z} \rangle\!\rangle a.$$

Clearly, the construction of $\mathcal{S}$ can be performed in polynomial time and space.

It remains to provide a similar reduction for ASL formulas of the type $\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$. More precisely, we provide a polynomial reduction from the ASL model-checking problem

"does $q_0 \models \mathbb{E}_N[\![\mathcal{Z}]\!]a$ hold?"

where q_0 is a state of a constraint automaton $\mathcal{A}$, $\mathcal{Z}$ an NFA, $N \subseteq \mathcal{N}$ a set of port-names and $a \in AP$ an atomic proposition. Here we combine $\mathcal{A}$ and $\mathcal{Z}$ to obtain an alternating finite automaton that is empty if and only if $q_0 \models \mathbb{E}_N[\![\mathcal{Z}]\!]a$.

Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton, $q_0 \in Q$ a state in $\mathcal{A}$, $\Phi = \mathbb{E}_N[\![\mathcal{Z}]\!]a$, where $N \subseteq \mathcal{N}$ a set of ports, $a \in AP$ an atomic proposition, and $\mathcal{Z} = \langle Z, \mathsf{CIO}_{\surd}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ be an NFA. We construct an AFA $\mathcal{S} = \langle S, \Sigma, \Delta, s_0, S_F\rangle$ as follows:

– $S \stackrel{\text{def}}{=} \{s_0\} \cup (Q \times Z) \cup \{\langle q, z, c\rangle \in Q \times Z \times \mathsf{CIO}_{\surd} \mid c = c_{\mathsf{stop}} \text{ or } c \in \mathsf{CIO}(q) \cap \mathsf{CIO}(z)\}$,

– $\Sigma \stackrel{\text{def}}{=} \{\nu\}$ where ν is an arbitrary input symbol,

– $s_0 \in S$ the starting state, and

– $S_F \stackrel{\text{def}}{=} \{\langle q, z\rangle \in Q \times Z \mid q \not\models a \text{ and } z \in Z_F\}$

– As the input alphabet Σ is a singleton set, the transition function can now be interpreted as a function $\Delta : S \to \text{Bool}^+(S)$ which is given by the following definition

$$\Delta(s) \stackrel{\text{def}}{=} \begin{cases} \bigvee_{s' \in \lambda(s)} s' & \text{if } s \in \{s_0\} \cup (Q \times Z \times \mathsf{CIO}_{\surd}) \\ \bigwedge_{s' \in \lambda(\langle q, z\rangle)} s' & \text{if } s \in (Q \times Z) \end{cases}$$

Here, λ is a total function $\lambda : S \to 2^S$ which is given by

$$\begin{aligned} \lambda(s_0) &\stackrel{\text{def}}{=} \{\langle q_0, z\rangle \in S \mid z \in Z_0\} \\ \lambda(\langle q, z\rangle) &\stackrel{\text{def}}{=} \{\langle q, z, c\rangle \in S \mid c \in \mathsf{CIO}(\langle q, z\rangle) \text{ in } \mathcal{A} \odot \mathcal{Z}\} \\ \lambda(\langle q, z, c\rangle) &\stackrel{\text{def}}{=} \{\langle q', z'\rangle \in S \mid \langle q, z\rangle \xrightarrow{c'}_{\mathcal{A} \odot \mathcal{Z}} \langle q', z'\rangle \text{ with } c' \in \text{compAct}(c, N)\} \end{aligned}$$

where $\text{compAct}(c, N)$ again stands for the set of all c-compatible actions.

The accepted language of the constructed AFA will be empty if the N-player can choose (at least) one action in any of his game configurations such that none of the opponents moves leads to a state $s \in S_F$. Hence, we have that

$$\mathcal{L}(\mathcal{S}) = \emptyset \text{ iff } q_0 \models \mathbb{E}_N[\![\mathcal{Z}]\!]a.$$

As there is a polynomially space-bounded algorithm for checking emptiness of AFA with unary alphabet, we obtain the following lemma:

Lemma 4.4.6 (Upper bound for the complexity of ASL model checking)**.** The ASL model-checking problem is in PSPACE.

■

Corollary 4.4.7 (PSPACE-completeness for ASL)**.** From part (a) of Theorem 4.4.2 and Lemma 4.4.6 we can conclude that the ASL model-checking problem is PSPACE-complete.

■

Complexity of the ATL-fragment. We will conclude this section with a short inspection of the complexity of the ATL-model checking problem. By the ATL-fragment of ASL, we mean the sublogic of ASL without stream expressions, i.e., path formulas of the ATL-fragment are generated by the standard path modalities $\mathtt{U}$, $\mathtt{R}$ and $\mathtt{X}$, but not $\langle\mathcal{Z}\rangle$ or $[\![\mathcal{Z}]\!]$. The difference to standard ATL [AHK02] is that in the classical setting logic reasons about infinite behavior only. ATL as introduced in [AHK02] does not have a notion for quiescence and finite behavior.

The worst complexity of our model-checking algorithm for the ATL-fragment is roughly the same as for standard ATL, i.e., linear in the size of $\mathcal{A}$ and the length of the given ASL state formula, when appropriate data structures for the internal representation of the constraint automaton are assumed.

4.5. ASL with fairness

Several extensions of ASL could be considered. We conclude this chapter with a few remarks on introducing fairness for ASL, as this concept is of special interest for our game-based interpretation of constraint automata. In this section we provide a preview on the most important aspects a reasonable notion of ASL fairness must address and examine the relationship to controller synthesis. The concept of fairness, implicitly introduced by Dijkstra [Dij65, Dij68], is often necessary to establish liveness properties [Fra86, Kwi89, VVK05] for nondeterministic transition systems and models with interleaving processes. In such nondeterministic system models fairness serves to rule out pathological behaviors that exists due to unfair resolution of nondeterministic choices. There exists three different types of fairness conditions: for *unconditional fairness* it is required that every process is scheduled infinitely often. For *strong fairness* we require that every process that is enabled infinitely often acts infinitely often, while for *weak fairness* only processes that are continuously enabled need to get the turn infinitely often. The notions of *weak* and *strong fairness* have first been introduced for shared variable concurrent programs [LPS81] and later considered in the context of transition systems [QS83].

In addition to the standard notion of process fairness, our special interpretation of constraint automata as multi-player games demands for some additional ASL fairness assumptions that go beyond the standard notion of process fairness. The goal is to rule out “unfair behavior” which is imposed by schedulers that continuously ignore the choice of an ASL strategy.

To illustrate the need for this special form of fairness assumptions, we revisit Example 4.2.4. One would expect that the ASL state formula $\mathbb{E}_A\Diamond\neg\exists\,\mathtt{X}\,\mathsf{true}$ would be fulfilled, since the memoryless strategy $\mathfrak{S}$, which tries to write on A whenever q_1 is reached during an execution should be winning for $\langle q_0, \Diamond\neg\exists\,\mathtt{X}\,\mathsf{true}\rangle$. However, $\mathfrak{S}$ cannot rule out the case where only the empty transition fires, while the write request at port A never gets granted. This, however, is a pathological case which can be avoided with appropriate fairness assumptions.

In the following definition we introduce one reasonable notion of strong fairness for paths with respect to a port-set N and a given N-strategy $\mathfrak{S}$.

Definition 4.5.1 ($\langle N, \mathfrak{S}\rangle$-fairness)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton, $N \subseteq \mathcal{N}$ a port-set, $\mathfrak{S}$ an N-strategy, and $\pi = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ a $\mathfrak{S}$-path in $\mathcal{A}$. Then π is called *(strongly)* $\langle N, \mathfrak{S}\rangle$*-fair* if either π is finite or for all $c \in \mathsf{CIO}$ we have:

$$\overset{\infty}{\exists}\, i \geq 0.\ c \in \mathsf{CIO}(q_i) \cap \mathfrak{S}(\pi \downarrow i) \text{ and } \emptyset \neq \mathsf{active}(c) \subseteq N \text{ implies } \overset{\infty}{\exists}\, i \geq 0.\ c_i = c,$$

where $\overset{\infty}{\exists}\, i$ means "there exist infinitely many i". If N and $\mathfrak{S}$ are understood from the context we simply talk about fairness rather than $\langle N, \mathfrak{S}\rangle$-fairness.

■

Note that the very special notion of fairness introduced in this section is an additional concept to standard process fairness. Compared to standard process our notion of fairness is different as it does not impose any constraint on the specific write and read operations of the opponents of N. Instead, it just requires that any I/O-operation c that is fully controllable by N and offered infinitely often by the given N-strategy $\mathfrak{S}$ will not be ignored forever.

Example 4.5.2 (Fair und unfair paths)**.** Dealing with $N = \{A\}$ and the memoryless N-strategy $\mathfrak{S}$ that attempts to write on A whenever the current state is q_1 in Example 4.2.4, the path that alternates between states q_0 and q_1 via the empty I/O-operation $c_\emptyset$ is not $\langle N, \mathfrak{S}\rangle$-fair.

For the system in Example 4.2.5 and $N = \{A^{out}, A_1^{in}, A_2^{in}\}$, i.e., the port-set of component C_A, and the memoryless N-strategy $\mathfrak{S}$ that does not rule out any I/O-operation (i.e., $\mathfrak{S}(q) = \mathsf{CIO}_{\surd}$ for all $q \in Q$), the $\mathfrak{S}$-path that runs forever through the cycle

$$q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} q_3 \xrightarrow{c_3} q_0$$

with $\mathsf{active}(c_1) = \{A^{out}\}$, $\mathsf{active}(c_2) = \{B_1^{in}\}$, and $\mathsf{active}(c_3) = \{A_1^{in}, B_1^{out}\}$ is $\langle N, \mathfrak{S}\rangle$-fair, although the transition from q_0 to q_2 is never taken.

It should be noticed that $\langle N, \mathfrak{S}\rangle$-fairness asserts that any controllable I/O-operation c that is infinitely often enabled and offered by $\mathfrak{S}$ will be taken infinitely often, but $\langle N, \mathfrak{S}\rangle$-fairness does not impose a fair resolution of the choice between several c-transitions.

■

Compositional controller synthesis. The notion of fairness needed in the context of ASL is closely related to our work on compositional controller synthesis [BKK11a] which is applicable to various automata types. With constraint automata the presented approach relies on a similar game-based interpretation as for ASL.

In ASL we were asking for the existence or absence of a winning strategy for a temporal property, whereas the question in controller synthesis is whether or not there is a way of connecting components to make a certain temporal objective hold. The orchestrating component(s) and network ensuring a given property to hold is called a controller. We studied the problem of how to synthesis such controllers exogenously within our Reo and constraint automata framework. Starting with an ASL formula, the goal is to derive a constraint automaton from finite-memory winning N-strategy which can then serve as a controller for

the part of the system that is controllable, namely the N-components. In [ABdB$^+$05] an algorithm has been presented that constructs a Reo network from a given constraint automaton. This approach is compositional and relies on a preprocessing step that generates an ω-regular expression from the given constraint automaton. We presented an alternative approach [BKK11c, BKK11b] for the generation of a Reo network directly from the constraint automaton $\mathcal{A}$ which reuses some ideas of [ABdB$^+$05], but avoids the potential exponential blow-up in the construction of an ω-regular expression.

In the compositional controller synthesis setting as presented in [BKK11a] our starting point is a temporal logic formula of the form $\Phi = \Phi_1 \wedge \ldots \wedge \Phi_n$. The goal is to subsequently create the controllers $\mathcal{C}_i$ for each of the sub-formulas Φ_i and to synthesize and connect the controllers with the controlled parts of the system exogenously. Applying the presented framework to the setting of Reo and constraint automata, the synthesized controllers can be again regarded as constraint automata (with some additional fairness constraints). These automata can then likewise be transformed into Reo networks using the (controller) synthesis construction presented in [BKK11b]. In the context of the compositional controller synthesis most-general controllers that preserve as much behavior of the controlled system as possible are of special interest. To support compositional controller synthesis, where we gradually create and compose controllers for all sub-formulas, the controllers have to be most-general in the sense that they capture *all possible* ways in which a given sub-formula may be enforced in the system. In [BKK11a] we presented a compositional controller synthesis approach covering regular safety and co-safety objectives for a partial information and partial control setting. A detailed discussion as well as further extensions towards the full set of ω-regular languages will be discussed in the upcoming thesis [Kle12]. In the next chapter, we present our implementation of the Vereofy tool set, which already includes a prototype implementation for the synthesis of a Reo network from a given constraint automaton. Algorithms addressing the compositional controller synthesis problem are also planned to be integrated into the Vereofy tool set.

5 Implementation

The presented approach for modeling, verification and synthesis of the Chapters 3 and 4 has been implemented and integrated (along with other verification formalisms [BBKK09b, BKK11b]) in our model-checking tool set Vereofy [BKK12]. Vereofy[1] is jointly developed by members of our group at Technische Universität Dresden. To overcome the state-space explosion problem the constraint automata for components and the coordinating network are represented symbolically. For this purpose we rely on well-known concepts to encode automata as switching functions and represent them by *binary decision diagrams* (BDDs). (See e.g. [Bry86, McM93, HS96, Min96, DB98, MT98, Weg00].) In this chapter we present the relevant details of the BDD encoding along with further important implementation details of Vereofy. The tool set Vereofy has successfully been used in two major case studies [BBK$^+$10, KKSB11] within the former EU project CREDO [GJA$^+$10] and served well for the evaluation of our hierarchical modeling approach and the verification engines. At the end of this chapter our approach is evaluated by means of various case studies carried out with Vereofy.

Organization. Section 5.1 provides the relevant implementation details about Vereofy. In Section 5.2 we present the details of the BDD-based model checking of ASL formulas and Section 5.3 gives insights some of the case studies we carried out for the evaluation our modeling and verification approach.

5.1. The model-checking tool Vereofy

Vereofy is a tool set for the formal verification of component-based systems described in terms of Reo and constraint automata. The tool set uses the two input languages CARML and RSL for modeling component-based systems. Vereofy itself is written in C++ and runs on Mac OS X, Linux, and Windows. Vereofy can be used as a standalone application or as an Eclipse plugin for the Extensible Coordination Tools (ECT)[2] [Kra11], a graphical user interface developed at the Centrum Wiskunde & Informatica (CWI) Amsterdam[3] supporting Reo and constraint automata. The Vereofy plugin in the Extensible Coordination Tools supports conversion between graphical Reo connectors and RSL scripts, and vice versa. Figure 5.1 provides an overview on the main elements of Vereofy.

[1] `http://www.vereofy.de/`
[2] `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools`
[3] `http://www.cwi.nl/`

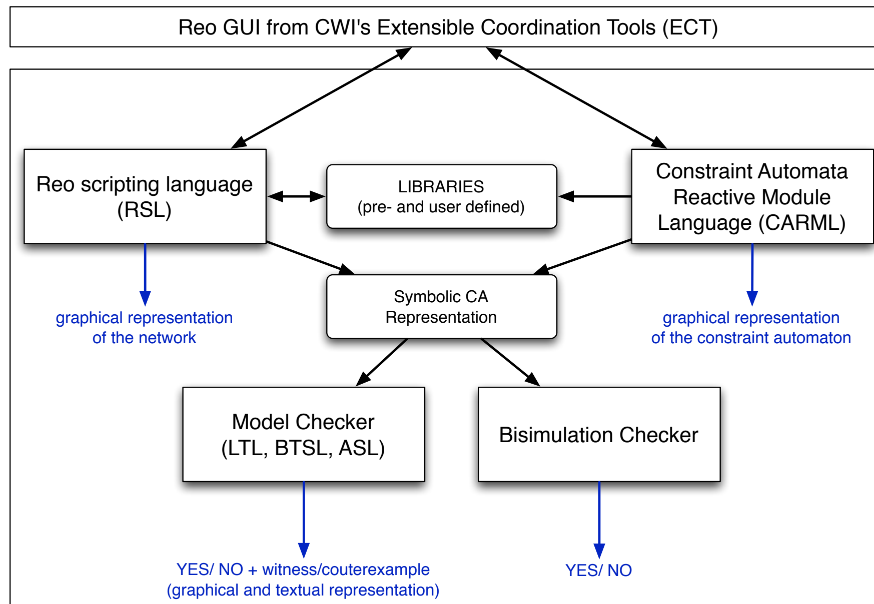


Figure 5.1: Main elements of the Vereofy tool set

Main features. For the modeling we use CARML and RSL as described in Chapter 3.2. In the implementation we provide some additional language features such as functions, enumerator concepts, conditional code inclusion, labels for transitions, etc. Global variables are not yet fully supported in the currently released version of Vereofy. For details about additional language and tool features we refer to the Vereofy manual [BKK12].

Vereofy provides a library of Reo channels, connectors and useful components that can be used to compose more complex connectors and components. A user may define his own libraries of channels, connectors, and components that are then available for further composition. For debugging, Vereofy allows to write the structure of a Reo circuit constituted by its components and connectors to a GraphViz[4] file. For constraint automata (with a reasonably small number of states and ports) Vereofy supports writing them to GraphViz files as well.

The constraint automata for components and the orchestrating network are represented symbolically by means of switching functions stored in binary decision diagrams (BDDs). A detailed description of the encoding of constraint automata in terms of switching functions as well as symbolic reformulations of the algorithms presented in Chapter 4 are be provided in the next section. For the implementation we use the external BDD library JINC[5] [Oss10].

[4] `http://www.graphviz.org/`
[5] `http://jossowski.de/projects/jinc/jinc.html`

Vereofy supports symbolic model checking for ASL (and BTSL) formulas (cf. Chapter 4) as well as model checking for linear-time properties and equivalence checking using bisimulation [BBKK09b, BKK11b]. The syntax of ASL/BTSL path formulas of the form $\langle \mathcal{Z} \rangle \Phi$ and $[\![\mathcal{Z}]\!]\Phi$ uses regular I/O-stream expressions α, from which a symbolic NFA $\mathcal{Z}_\alpha = \mathcal{Z}$ is built.

The BTSL model checker allows computing and printing witnesses and counterexamples for BTSL formulas. This option is activated in Vereofy by the command line option btsl-witness-level. Using the Vereofy Eclipse plugin for the Extensible Coordination Tools (ECT) one can explore counterexamples and witnesses of BTSL formulas in the graphical user interface by skimming through the detailed state information (i.e., the evaluation of the state variables) and the observable dataflow along the counterexample or witness path. For ASL a witness for a formula of the form $\mathbb{E}_N\varphi$ is a strategy $\mathfrak{S}$ such that when the strategy is applied, all remaining $\mathfrak{S}$-paths in the constraint automaton $\mathcal{A}$ satisfy φ. Vereofy allows to compute such a winning finite-memory strategy as suggested by the model checking algorithms presented in Section 4.3 and Section 5.2. This option is activated in Vereofy by the command line option compute-asl-strategy. The computed strategy $\mathfrak{S}$ can be represented by a constraint automaton which Vereofy can write to a GraphViz file for debugging. Moreover, Vereofy allows to store the strategy symbolically, apply the computed strategy $\mathfrak{S}$ to $\mathcal{A}$, and rerun the model checker with the modified system and modified BTSL (or CTL) formula to validate $\mathfrak{S}$ to be a winning. This check is activated in Vereofy by the command line option check-result. E.g., when starting with a constraint automaton $\mathcal{A}$ and an ASL state formula of the form $\mathbb{E}_N\varphi$, Vereofy may successfully constructs an N-strategy $\mathfrak{S}$ that is winning and stored in terms of a constraint automaton $\mathcal{A}_\mathfrak{S}$. Then Vereofy builds the synchronous product of $\mathcal{A}$ and $\mathcal{A}_\mathfrak{S}$ and starts the BTSL model checking procedure to check whether $\forall\varphi$ holds in the product. This check has been carried out (together with some additional sanity checks) for all ASL formulas presented later for the case studies in this chapter to verify the result of the model checker.

Vereofy also provides some heuristics and methods that should improve the ASL and BTSL model checking. For ASL and BTSL path formulas of the form $\langle \mathcal{Z} \rangle$ and $[\![\mathcal{Z}]\!]$ we start with a regular I/O-stream expression α and create an NFA $\mathcal{Z}$ with $\mathcal{L}(\mathcal{Z}) = \mathsf{IOS}(\alpha)$. For the NFA we use a data structure that allows to store the states of $\mathcal{Z}$ explicitly and represent the transition labels (i.e., data constraints) symbolically in terms of switching functions. For BTSL state formulas of the form $\exists[\![\mathcal{Z}]\!]\Phi$ as well as ASL state formulas of the form $\mathbb{E}_N\langle \mathcal{Z} \rangle\Phi$ and $\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi$ and their duals $\mathbb{A}_N[\![\mathcal{Z}]\!]\Phi$ and $\mathbb{A}_N\langle \mathcal{Z} \rangle\Phi$ this NFA is determinized and minimized to reduce the actual amount of states. For this, Vereofy provides algorithms based on the ideas of Hopcroft [Hop71] and Brzozowski [Brz62], where Hopcroft is selected by default. The minimization method can be selected in Vereofy by using the command line option minimize-dfa. In the very last step the NFA is transferred into a new data structure that is fully symbolic and allows to encode states and transitions of $\mathcal{Z}$ in terms of switching functions. This symbolic representation of $\mathcal{Z}$ is used throughout the model checking procedures.

The model checking engine for ASL and BTSL uses an early termination mechanism allowing fixpoint computation to be stopped before the actual fixpoint is reached. Early termination will only be applied if the current formula under inspection is a top-level formula. The

termination succeeds for greatest fixpoints if it is clear from the current satisfaction set that shrinking the satisfaction set further cannot change satisfiability of the formula. For least fixpoints the condition is that expanding the satisfaction set cannot change satisfiability of the formula.

In the next section we show the symbolic representation for constraint automata as well as symbolic formulations of the model-checking algorithms, which then work on the symbolic rather than an explicit representation of constraint automata. For this, we adapted standard concepts of symbolic model checking with BDDs for labeled transitions systems [BCM+92, CGP99, McM93, Som99] to our framework of constraint automata and the alternating-time logic ASL.

5.2. BDD-based model checking

The implementation of our modeling and verification approach in the verification tool set Vereofy including the ASL model-checking procedures as they were introduced in the previous chapter is based on a symbolic representation of constraint automata in terms of switching functions. The data structures used to store switching functions are *ordered binary decision diagrams* (OBDDs) [Bry86]. This section summarizes the main aspects of our symbolic encoding of constraint automata, namely states, transition relation (and labeling function), with the help of switching functions. Moreover we present symbolic versions of the product construction for constraint automata and of the algorithms introduced in Section 4.3.

Constraint automata encoding. To represent a constraint automaton $\mathcal{A}$ by a BDD, we fix a binary encoding of the states. For a constraint automata $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L \rangle$ we embed Q into $\{0,1\}^n$ by an injective function

$$bin : Q \to \{0,1\}^n$$

where $n = \lceil \log |Q| \rceil$. We choose boolean state variables $v_1, \ldots, v_n$ and then identify each state q with the evaluation for $v_1, \ldots, v_n$ given by $bin(q)$. We write $\bar{q}$ for the variable tuple $(v_1, \ldots, v_n)$. In the same way, we may encode the data items by bit tuples. For simplicity, we assume here the boolean data domain $\mathsf{Data} = \{0,1\}$ and treat the symbols d_A and ports $A \in \mathcal{N}$ as boolean variables. In the sequel, let $\mathcal{N} = \{A_1, \ldots, A_k\}$ and $d_i = d_{A_i}$, $i = 1, \ldots, k$. We write $\bar{c}$ for the variable tuple $(A_1, \ldots, A_k, d_1, \ldots, d_k)$.

The transition relation $\longrightarrow_{\mathcal{A}}$ can be identified with its characteristic function and viewed as a switching function

$$T_{\mathcal{A}} : Eval(\bar{q}, \bar{c}, \bar{q}') \to \{0,1\},$$

where $Eval(\bar{q}, \bar{c}, \bar{q}')$ denotes the set of evaluations for the variable tuples $\bar{q}, \bar{c}$ and $\bar{q}'$. The tuple $\bar{q} = (v_1, \ldots, v_n)$ encodes the starting state, $\bar{q}' = (v_1', \ldots, v_n')$ the target state, while $\bar{c}$ serves to represent the concurrent I/O-operation. For instance, the transition relations of the constraint automata for a synchronous channel and a synchronous drain with source port A

and sink port B are given by:

$$\begin{array}{lcl} T_{\text{sync_channel}}(v_1, A, B, d_A, d_B, v_1') & = & v_1 \wedge A \wedge B \wedge (d_A \leftrightarrow d_B) \wedge v_1' \\ T_{\text{sync_drain}}(v_1, A, B, d_A, d_B, v_1') & = & v_1 \wedge A \wedge B \wedge v_1' \end{array}$$

For a FIFO1 channel we have to encode three states, say

$$bin(q^{(\emptyset)}) = 00,\ bin(q^{(1)}) = 11 \text{ and } bin(q^{(0)}) = 10,$$

and then may represent the automaton by

$$\begin{array}{lcl} T_{\text{FIFO1}}(v_1, v_2, A, B, d_A, d_B, v_1', v_2') & = & \overbrace{(\neg v_1 \wedge \neg v_2 \wedge A \wedge \neg B \wedge (q_2' \leftrightarrow d_A) \wedge v_1')}^{\text{write into FIFO1}} \\ & \vee & \underbrace{(v_1 \wedge \neg A \wedge B \wedge (v_2 \leftrightarrow d_B) \wedge \neg v_1' \wedge \neg v_2')}_{\text{read from the FIFO1}} \end{array}$$

Beside the transition relation, we also need a BDD representation of the initial states Q_0, quiescent states $\mathcal{Q}$ and the labeling function L. This can be done by representing the characteristic function of Q_0, $\mathcal{Q}$ and

$$\mathsf{Sat}(a) = \{q \in Q \mid a \in L(q)\}$$

by a BDD for the induced function SAT_{Q_0}, $SAT_{\mathcal{Q}}$, $SAT_a : Eval(\bar{q}) \to \{0, 1\}$. In the sequel, we use the shorthand notation $quie_{\mathcal{A}}$ for the quiescent states in $\mathcal{A}$, i.e., the characteristic function $SAT_{\mathcal{Q}}$.

The BDD representation for the transition relation of a constraint automaton can be constructed in a compositional manner, by mimicking Reo's composition operators using the corresponding operators on constraint automata and applying the analogous symbolic operations for manipulating switching functions. Let $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \longrightarrow_i, Q_{0,i}\rangle$ for $i \in \{1, 2\}$ be two constraint automata without labeling and quiescence and $T_i(V_i)$ their switching function representations, where $V_i = (\bar{q}_i, \bar{c}_i, \bar{q}_i{}')$. The rules of the constraint automata product for the transition relation from Definition 2.1.8 can be expressed symbolically by:

$$T_{\mathcal{A}_1 \bowtie \mathcal{A}_2}(V_1, V_2) = \underbrace{(T_1(V_1) \wedge T_2(V_2))}_{\text{synchronous part}} \vee \underbrace{(T_1(V_1) \wedge id_{\mathcal{A}_2}(V_2) \vee (id_{\mathcal{A}_1}(V_1) \wedge T_2)(V_2))}_{\text{asynchronous part}}$$

where

$$id_{\mathcal{A}}(v_1, \ldots, v_n, A_1, \ldots, A_k, d_1, \ldots, d_k, v_1', \ldots, v_n') = \bigwedge_{j=1,\ldots,n} (v_j \leftrightarrow v_j') \wedge \bigwedge_{A \in \mathcal{N}_{\mathcal{A}}} \neg A.$$

The boolean function $id_{\mathcal{A}}$ describes that $\mathcal{A}$ does not change its state and no port is active, i.e., in the product we freeze the state of one constraint automaton and prohibit any port activity, while the other automaton moves.

Symbolic BTSL model checking. Similarly, the rules for the BTSL product $\mathcal{A} \otimes \mathcal{Z}$ (cf. Definition 4.3.1) and ASL product $\mathcal{A} \odot \mathcal{Z}$ (cf. Definition 4.3.7) can be expressed symbolically. For the BTSL product we need to include a boolean variable $v_{\surd}$ (which is not yet part of the encoding of $\mathcal{A}$) for the $\surd$-transitions in the finite automaton $\mathcal{Z}$ and in the product. Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton. Furthermore let $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ be a finite automaton as in Definition 4.3.1 and let $T_{\mathcal{A}}(V_{\mathcal{A}})$ and $T_{\mathcal{Z}}(V_{\mathcal{Z}})$ be the switching function representation of their transition relations. The symbolic way of expressing the BTSL product is

$$\begin{aligned} T_{\mathcal{A}\otimes\mathcal{Z}}(V_{\mathcal{A}}, V_{\mathcal{Z}}) \;&=\; \overbrace{(T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge T_{\mathcal{Z}}(V_{\mathcal{Z}}) \wedge \neg v_{\surd})}^{\text{synchronous part}} \\ &\vee\; \underbrace{(quie_{\mathcal{A}}(\bar{q}) \wedge T_{\mathcal{Z}}(V_{\mathcal{Z}}) \wedge v_{\surd} \wedge id_{\mathcal{A}}(V_{\mathcal{A}}))}_{\text{quiescent part}}. \end{aligned}$$

Symbolic reformulations of Algorithm 1 and 2 are shown in Algorithm 6 and 7 where it is assumed that the BDD SAT_{Ψ} for $\mathsf{Sat}(\Psi)$ has already been constructed. BDD representations SAT_{Ψ} for the satisfaction sets $\mathsf{Sat}(\Psi)$ of the subformulas Ψ of Φ are obtained by reformulating the model-checking algorithms in a symbolic way with boolean operators and applying the corresponding BDD synthesis algorithms.

We use the variable tuple $\bar{q} = (v_1, \dots, v_n)$ to encode the states in $\mathcal{A}$ and $\bar{z} = (z_1, \dots, z_m)$ for the states in $\mathcal{Z}$. Subsets V of $Q \times Z$ are encoded by the variables in $\bar{q}$ and $\bar{z}$. The sets Z_0 and Z_F are represented by BDDs with the variables $\bar{z}$. The symbolic fixpoint computation makes use of a symbolic representation of the predecessors $\mathsf{Pre}(P)$:

$$\Lambda_{\mathsf{Pre}(P)}(\bar{q}) \;=\; \exists \bar{q}'\, \exists \bar{c}.\, [T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge P(\bar{q}' \leftarrow \bar{q})]$$

where the notation $P(\bar{q}' \leftarrow \bar{q})$ means that the variables $\bar{q}$ of P are renamed into their primed copies and $\exists \bar{v}.\,[f] = (f|_{v_1=0} \vee f|_{v_1=1}) \vee (f|_{v_2=0} \vee f|_{v_2=1}) \vee \dots \vee (f|_{v_n=0} \vee f|_{v_n=1})$ denotes the existential quantification for the variable tuple $\bar{v} = (v_1, \dots, v_n)$ and switching function f. The switching function $f|_{v=c}$ is called the *cofactor* of f and results from f by evaluating variable v with the constant value $c \in \{0, 1\}$.

Algorithm 6 Symbolic computation of $\mathsf{Sat}\big(\exists\langle\mathcal{Z}\rangle\Phi\big)$ for NFA $\mathcal{Z}$

generate BDD representations for the sets Z_0 and Z_F and for $T_{\mathcal{A}\otimes\mathcal{Z}}$;
$P := SAT_{\Phi} \wedge Z_F$;

repeat
  $\mathcal{V} := P$;
  $P := P \vee \exists\langle\bar{q}', \bar{z}'\rangle\, \exists\bar{c}.\, [T_{\mathcal{A}\otimes\mathcal{Z}}(\bar{q}, \bar{z}) \wedge P(\langle\bar{q}', \bar{z}'\rangle \leftarrow \langle\bar{q}, \bar{z}\rangle)]$;
until $(\mathcal{V} = P)$;

return $\exists\bar{z}.\,[P \wedge Z_0]$; (* symbolic representation of $\mathsf{Sat}(\exists\langle\mathcal{Z}\rangle\Phi)$ by $SAT_{\exists\langle\mathcal{Z}\rangle\Phi}$ *)

Algorithm 7 Symbolic computation of $\mathsf{Sat}\big(\exists[\![\mathcal{Z}]\!]\Phi\big)$ for DFA $\mathcal{Z}$

generate BDD representations for the sets Z_0 and Z_F and for $T_{\mathcal{A}\otimes\mathcal{Z}}$;
$P := \neg Z_F \vee SAT_\Phi$;

repeat
  $\mathcal{V} := P$;
  $P := P \wedge \exists\langle\bar{q}',\bar{z}'\rangle\,\exists\bar{c}.\,[T_{\mathcal{A}\otimes\mathcal{Z}}(\bar{q},\bar{z}) \wedge P(\langle\bar{q}',\bar{z}'\rangle \leftarrow \langle\bar{q},\bar{z}\rangle)]$;
until $(\mathcal{V} = P)$;

return $\exists\bar{z}.\,[P \wedge Z_0]$; $\big($* symbolic representation of $\mathsf{Sat}(\exists[\![\mathcal{Z}]\!]\Phi)$ by $SAT_{\exists[\![\mathcal{Z}]\!]\Phi}$ *$\big)$

Symbolic ASL model checking. In the case of the ASL product we use the variable $v_{\surd}$ for the encoding of $\surd$-transitions in the finite automaton $\mathcal{Z}$ and the variable v_{stop} to encode port activity at the additional port $A_{\surd}$ in the product. Moreover we add characteristic functions

$$\mathit{cont}^N_{\mathcal{A}},\ \mathit{prev}^N_{\mathcal{A}}:\ \mathit{Eval}(\bar{q}) \longrightarrow \{0,1\}$$

to characterize the states in $\mathcal{A}$ which have *controllable transitions* (i.e., $\mathsf{active}(c) \subseteq N$), and the states which have *preventable transitions* (i.e., $\mathsf{active}(c) \cap N \neq \emptyset$), respectively. These functions are given by

$$\begin{aligned}
\mathit{cont}^N_{\mathcal{A}}(\bar{q}) &= \exists\bar{c}\,\exists\bar{q}'.\,[T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \bigvee_{A\in N} A \wedge \bigwedge_{A\in\mathcal{N}_{\mathcal{A}}\setminus N} \neg A] \\
\mathit{prev}^N_{\mathcal{A}}(\bar{q}) &= \exists\bar{c}\,\exists\bar{q}'.\,[T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \bigvee_{A\in N} A]
\end{aligned}$$

Let $\mathcal{A} = \langle Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, \mathcal{Q}, AP, L\rangle$ be a constraint automaton. Furthermore let $\mathcal{Z} = \langle Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, Z_0, Z_F\rangle$ be a finite automaton as in Definition 4.3.1. Let $T_{\mathcal{A}}(V_{\mathcal{A}})$ and $T_{\mathcal{Z}}(V_{\mathcal{Z}})$ be the switching function representation of their transition relations. As $v_{\surd}$ will not be needed in the ASL product of $\mathcal{A}$ and $\mathcal{Z}$, we define the symbolic representation of the ASL product to be of the form

$$T_{\mathcal{A}\odot\mathcal{Z}}(V_{\mathcal{A}\odot\mathcal{Z}}) \stackrel{\text{def}}{=} \exists v_{\surd}.\,[T_\gamma(V_{\mathcal{A}\odot\mathcal{Z}})]$$

where $V_{\mathcal{A}\odot\mathcal{Z}}$ consists of all variables in $V_{\mathcal{A}}$, $V_{\mathcal{Z}}$ and the additional variable v_{stop} and where

$$\begin{aligned}
T_\gamma(V_{\mathcal{A}\odot\mathcal{Z}}) &= \underbrace{(T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge T_{\mathcal{Z}}(V_{\mathcal{Z}}) \wedge \neg v_{\surd} \wedge \neg v_{\mathsf{stop}})}_{\text{synchronous part}} \\
&\vee \underbrace{(\mathit{quie}_{\mathcal{A}}(\bar{q}) \wedge \neg\mathit{cont}^N_{\mathcal{A}}(\bar{q}) \wedge T_{\mathcal{Z}}(V_{\mathcal{A}}) \wedge v_{\surd} \wedge \neg v_{\mathsf{stop}} \wedge \mathit{id}_{\mathcal{A}}(V_{\mathcal{A}}))}_{\text{quiescent part (1)}} \\
&\vee \underbrace{(\mathit{quie}_{\mathcal{A}}(\bar{q}) \wedge \mathit{prev}^N_{\mathcal{A}}(\bar{q}) \wedge T_{\mathcal{Z}}(V_{\mathcal{Z}}) \wedge v_{\surd} \wedge v_{\mathsf{stop}} \wedge \mathit{id}_{\mathcal{A}}(V_{\mathcal{A}}))}_{\text{quiescent part (2)}}.
\end{aligned}$$

In the ASL case we are not only interested in whether or not a strategy does exist, but also in the concrete strategy. For this, we introduce the switching function

$$S_{P,N}(\bar{q},\bar{c}) : \mathit{Eval}(\bar{q},\bar{c}) \to \{0,1\}$$

keeping track of all controllable transitions that enforce moving to a state in P:

$$S_{P,N}(\bar{q},\bar{c}) = \exists \bar{q}'.\,[T^{cont}_{\mathcal{A}}(V_{\mathcal{A}}) \wedge P(\bar{q}' \leftarrow \bar{q})] \wedge \neg\exists \bar{q}'.\,[T^{cont}_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \neg P(\bar{q}' \leftarrow \bar{q})]$$

where

$$T^{cont}_{\mathcal{A}}(V_{\mathcal{A}}) = T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \bigvee_{A \in N} A \wedge \bigwedge_{A \in \mathcal{N}_{\mathcal{A}} \setminus N} \neg A$$

describes the controllable fragment of the transition relation. The symbolic realization of Algorithms 3 and 4 for computing the satisfaction sets

$$\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2)) \text{ and } \mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{R}\, \Phi_2)),$$

and hence the computation of $\mathsf{Sat}(\mathbb{E}_N\langle\!\langle \mathcal{Z} \rangle\!\rangle\Phi)$ and $\mathsf{Sat}(\mathbb{E}_N[\![\mathcal{Z}]\!]\Phi)$, depends on the symbolic representation of the predecessors $\mathsf{Pre}(P,N)$:

$$\Lambda_{\mathsf{Pre}(P,N)}(\bar{q}) \;=\; \underbrace{\neg\exists\bar{c}\exists\bar{q}'.\,[T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \bigwedge_{A \in N} \neg A \wedge \neg P(\bar{q}' \leftarrow \bar{q})]}_{\text{condition (1) in Definition 4.3.4}} \wedge \underbrace{\exists \bar{c}.\,\big[\exists \bar{q}'.\,[S_{P,N}(\bar{q},\bar{c})]\big]}_{\text{condition (2) in Definition 4.3.4}}$$

Algorithms 8 and 9 show symbolic realizations of Algorithms 3 and 4.

Algorithm 8 Symbolic computation of $\mathsf{Sat}\big(\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2)\big)$

$P := SAT_{\Phi_2}$;
$T_{\mathfrak{S}} := \exists \bar{q}'.\,\big[T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \bigwedge_{A \in N} \neg A\big]$;

repeat
  $\mathcal{V} := P$;
  $P' := (SAT_{\Phi_1} \wedge \Lambda_{\mathsf{Pre}(P,N)})$;
  $P := P \vee P'$;
  $T_{\mathfrak{S}} := T_{\mathfrak{S}} \vee (P \wedge \neg(\exists \bar{c}\exists \bar{q}'.\,[T_{\mathfrak{S}}]) \wedge S_{P',N}(\bar{q},\bar{c}))$;
until $(\mathcal{V} = P)$;

return P; (* symbolic representation of $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2))$ by $SAT_{\mathbb{E}_N(\Phi_1 \,\mathtt{U}\, \Phi_2)}$ *)

Algorithm 9 Symbolic computation of $\mathsf{Sat}\big(\mathbb{E}_N(\Phi_1 \mathrel{\mathrm{R}} \Phi_2)\big)$

$P := SAT_{\Phi_2};$
$T_{\mathfrak{S}} := \exists \bar{q}'.\, \big[T_{\mathcal{A}}(V_{\mathcal{A}}) \wedge \bigwedge_{A \in N} \neg A\big];$

repeat
  $\mathcal{V} := P;$
  $P := P \wedge (SAT_{\Phi_1} \vee \Lambda_{\mathsf{Pre}(P,N)});$
until $(\mathcal{V} = P);$

$T_{\mathfrak{S}} := T_{\mathfrak{S}} \vee (P \wedge S_{P,N}(\bar{q}, \bar{c}));$

return P; (* symbolic representation of $\mathsf{Sat}(\mathbb{E}_N(\Phi_1 \mathrel{\mathrm{R}} \Phi_2))$ by $SAT_{\mathbb{E}_N(\Phi_1 \mathrel{\mathrm{R}} \Phi_2)}$ *)

Implementation details. The BDD library JINC [Oss10] has its own memory management that allocates the main memory in chunks. When temporary functions become useless, i.e., there is no reference to the corresponding BDD node, the nodes are kept in memory to be referenced again. The number of "dead nodes" kept in memory before freeing the memory for other nodes is called *garbage collection delay* (GCD) and can be set from outside of the BDD library. In Vereofy we use a GCD of 1500 BDD nodes in the composition phase and 200000 BDD nodes in the analysis phase. Vereofy allows to modify these two values by the command line options gcd-building and gcd-checking. JINC uses computed tables for the Boolean and other BDD operators that allow storing the result of conjunctions, disjunctions, etc. of two functions. Whenever the same operator is applied to the same functions the result will be taken from the computed table. Vereofy uses the default values of JINC, i.e., 1048576 entries for each operator. The values can be changed by providing new values in the configuration file of JINC.

The implementation of Vereofy provides various heuristics that help in reducing size of the BDD, i.e., the number of BDD nodes needed to store constraint automata symbolically. The memory to store the symbolic representation of a constraint automaton is dominated by the number of BDD nodes for the encoding of its transition relation. Each BDD node requires 40 bytes in the main memory.

A heuristic is used to identify dataflow locations in the Reo network that are always guaranteed to act the exact same way, i.e, they are either both active or passive and if active the same data is observable. For these ports there is only a single BDD variable encoding their activity and only one set of data bits required for their encoding. This heuristic can be enabled or disabled in Vereofy by using the command line options optimize-ports and no-shared-data.

The size of the entire BDD structure crucially depends on the order of the BDD variables. It is well known that for a given function the number of BDD nodes may vary from linear to exponential. The initial variable ordering is created in the building phase, in which the BDD representation of a Reo circuit is built by gradually building and composing the BDD representation of all components, channels, connectors and Reo nodes. The traversal of the Reo circuit is done in depth-first-search manner such that the distance of BDD variables of

components and connectors interacting with each other is minimized. When creating new BDD variables encoding states, ports, or observable data the order in which the variables appear in the BDD structure is chosen according to the same rule.

This initial order can be further optimized by applying a heuristic for dynamic reordering of the BDD that allow to compute more reasonable variable orderings. For this, a variant of the *sifting* [Rud93, PS95] algorithm, that tries to minimize the number of BDD nodes by swapping neighboring variable levels is implemented in JINC. The computed variable ordering can be stored and loaded for subsequent compositions of the same system model. Sifting uses a *max-growth* constant to limit the space in which we allow the BDD to grow during the swap of variable levels. The max-growth value can be set in Vereofy via the command line option dynamic-reorder-max-growth. We equipped this standard version of sifting with a new concept of a *reorder-timeout*, limiting the number of milliseconds in which the BDD size cannot be decreased, before considering the current swap of a variable level to be worthless. The timeout can be set in Vereofy using the command line option dynamic-reorder-timeout. This allows to limit the overall time spent on the sifting considerably. It turns out that, even for small values of the timeout value (100 to 300ms), the grade of the solution, i.e., the number of BDD nodes needed to represent a system model, does not suffer much from spending only a small fraction of time on reordering. The dynamic reordering can be triggered by Vereofy in two different ways. The reordering can either be executed whenever the number of BDD nodes has doubled after the last time sifting was executed. Vereofy supports this by the command line option dynamic-reorder. The reordering can also be triggered at designated points in the building procedure. This option is activated in Vereofy by using the command line argument selective-reorder.

Precomputing the reachable part of the state space before the model checking allows a more compact representation of some of the boolean functions during the model checking as they can be modified for unreachable states using the BDD constrain operator [CBM90] to end up with more compact BDD representations. The precomputation is enabled by using the command line option precompute-reachable. In the building phase of a system, one can use the options constrain-to-reachable-components and constrain-to-reachable-circuits to constrain each instance of a CARML module and RSL circuit with respect to its reachable states before the composition moves on.

In the next section we present a few examples and practical results that illustrate applicability of our approach for different system types and evaluate the implementation of our modeling formalism and verification algorithms realized in Vereofy. The Vereofy models presented throughout this section make use of some language features of CARML and RSL that are not yet released in the current version and will be part of an upcoming releases of Vereofy. The full CARML / RSL code of the models is available online on:

`http://wwwtcs.inf.tu-dresden.de/~klueppel/PhD/examples.zip`:

All computations were performed on a dual processor server with two Intel(R) Xeon(R) L5630 CPUs at 2.13GHz, and 32 GB RAM running on Debian GNU/Linux 6.0.3 and kernel 2.6.32-5-amd64.

5.3. Tool evaluation

The first class of system models that we will consider for a simple case study are classical board games. For this, we will stay in the component-based setting and have one component for the game arena, holding the current configuration of the game, and another component accepting the legal moves of the players. The latter will be called *Rules* and serves to model the rules of the game. For the players themselves no component will be introduced and a priori we do not make any assumptions regarding their behavior. Instead, we model an open system where players can connect to the open ports of our model. The structure of a two-player game in the component-based setting is depicted in Figure 5.2.

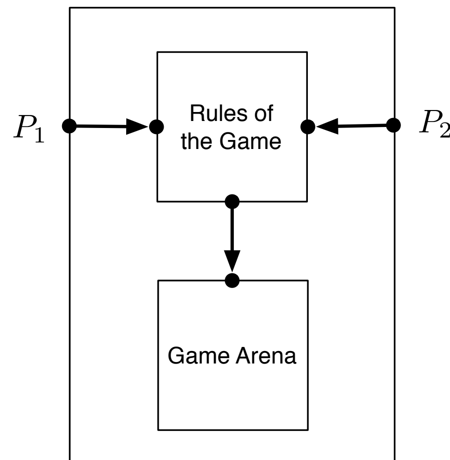


Figure 5.2: Component model for a two-player game

Players can connect to the interface ports P_1 and P_2 provided by the *Rules* component, which will accept legal moves on its input ports in the right order, e.g. alternately or concurrent. The moves of a player are immediately carried out by a synchronous write operation on the game arena, which updates the current configuration of the game. Moves that will not be legal due to the current game configuration will be completely blocked and not be accepted by the *Rules* component. For example, in some games the arena itself does not accept a move of a player if the field on the game board is already occupied. This illustrates that some of the intelligence deciding whether or not a move is legal relies on the current game configuration and thus on the corresponding component, while others depend on the context-independent rules of the game. This is elegantly captured within our component-based model.

Remark 5.3.1 (Turn-based games and ASL)**.** In ASL each component or player has the possibility to refuse interaction with the other components at any time. This is not a natural interpretation when reasoning about turn-based games. Intuitively, this means that each player has the possibility of leaving the game board at any time, even before the game is over. To eliminate this phenomena, it is sufficient to hide all opponent moves in the constraint automata in a structure-preserving way. Hence, when asking for a strategy for

the first player, we will hide the moves of the second player and vice versa. This results in $c_\emptyset$-transitions for all opponent moves which can neither be controlled by the player nor refused by the opponent.

■

Our first example of a game is the famous *Tic-Tac-Toe*, also called *noughts and crosses*. It has been selected to show that our approach is also applicable for solving questions typically raised in the context of turn-based games.

5.3.1. Tic-Tac-Toe

Tic-Tac-Toe is a well known two-player, turn-based game with complete information. The game arena is an $n \times n$ grid board where the players called player X and player O alternately mark the fields with their own symbols. Initially the board is empty and usually player X goes first. The goal of the game for both players is to place three of their own marks in a horizontal, vertical, or diagonal row. The game stops if either one of the two players wins or there is no unmarked field left on the grid. The latter is called a draw. Each play consists of finitely many steps and thus the number of possible plays is finite. In the 3×3 case the number of reachable game configurations is 5478 and the number of possible plays is 255.168, from which 131.184 are finished plays won by player X, 77.904 are won by player O, and 46.080 are draws[6]. Tic-Tac-Toe belongs to the class of solved games and it is well known that neither of the players has a winning strategy when their opponent plays optimally.

We present a parameterized model of the game with grids of size $n \times n$ to see how good the ASL model checking scales. The set of considered properties includes the standard questions studied for Tic-Tac-Toe in the literature (see e.g. [RvdHW09]) and also some non-standard questions which have not been addressed previously. E.g., we will see that player O has a strategy to avoid a draw, i.e., player O can enforce a play that is winning for either player O or player X. When scaling the size of the game board to size $n \times n$ in our model the players need n in a row to win the game.

The Vereofy model for the Tic-Tac-Toe game. Figure 5.3 shows the declarations of constants and types within the RSL main program for the Tic-Tac-Toe game. The declaration part also contains some function definitions for the computation of field indices and to get the current configuration of the game arena at a certain position. Figure 5.4 shows a CARML module for the rules of the Tic-Tac-Toe game. Here, the basic rules are that the moves of player O and player X do alternate and players can only place their own symbol on the game board. Figure 5.5 shows the CARML code for the game arena of Tic-Tac-Toe. The arena accepts incoming put operations as long as the addressed field is still empty and the game is not yet over. The latter requires to check the winning condition of player O and player X. A sequence of atomic propositions at the beginning of the module definition serves for this purpose.

[6] The number of different configurations and plays can significantly be reduced due to the symmetry.

```
CONST arena_size_default = 3;
CONST arena_size = arena_size_default;
TYPE index_t = int(0,arena_size-1);

CONST field_size = (arena_size * arena_size);
TYPE field_index_t = int(0,field_size-1);

TYPE symbol_t   = enum{empty,cross,circle};
TYPE position_t = int(0,arena_size-1);

TYPE put_t = struct{
  symbol_t symbol;
};

#include "builtin"

TYPE row_index_t    = index_t;
TYPE column_index_t = index_t;

FUNCTION field_index_t value(row_index_t row, column_index_t col)
  = (arena_size*row)+col;

FUNCTION symbol_t get(
  symbol_t[field_size] field, row_index_t row, column_index_t col)
  = field[value(row,col)];
```

Figure 5.3: RSL declarations for the Tic-Tac-Toe game

```
MODULE TTT_ROTG{
  in: put_t PlayerXput;
  in: put_t PlayerOput;
  out: put_t put_out;

  var: enum{PlayerX,PlayerO} turn := PlayerX;

  turn == PlayerX
   -[ {PlayerXput,put_out}
       & #put_out == #PlayerXput
       & #PlayerXput.symbol == cross ]->
  turn := PlayerO;

  turn == PlayerO
   -[ {PlayerOput,put_out}
       & #put_out == #PlayerOput
       & #PlayerOput.symbol == circle ]->
  turn := PlayerX;
}
```

Figure 5.4: CARML module for the rules of the Tic-Tac-Toe game

```
MODULE TTT_Arena{
  in: put_t put;

  var: symbol_t[field_size] arena := empty;

  ap: cross_wins_h
    <=> OR(j in index_t; AND(i in index_t; get(arena,j,i)==cross));

  ap: cross_wins_v
    <=> OR(j in index_t; AND(i in index_t; get(arena,i,j)==cross));

  ap: cross_wins_d1
    <=> AND(i in index_t; get(arena,i,i)==cross);

  ap: cross_wins_d2
    <=> AND(i in index_t; get(arena,arena_size-1-i,i)==cross);

  ap: cross_wins_d
    <=> cross_wins_d1 | cross_wins_d2;

  ap: cross_wins
    <=> cross_wins_h | cross_wins_v | cross_wins_d;

  ap: circle_wins_h
    <=> OR(j in index_t; AND(i in index_t; get(arena,j,i)==circle));

  ap: circle_wins_v
    <=> OR(j in index_t; AND(i in index_t; get(arena,i,j)==circle));

  ap: circle_wins_d1
    <=> AND(i in index_t; get(arena,i,i)==circle);

  ap: circle_wins_d2
    <=> AND(i in index_t; get(arena,arena_size-1-i,i)==circle);

  ap: circle_wins_d
    <=> circle_wins_d1 | circle_wins_d2;

  ap: circle_wins
    <=> circle_wins_h | circle_wins_v | circle_wins_d;

  ap: someone_wins   <=> circle_wins | cross_wins;
  ap: isFull         <=> AND(i in field_index_t; arena[i]!=empty);
  ap: game_over      <=> someone_wins | isFull;
  ap: running_game   <=> !game_over;

  running_game
   -[ {put} & IF(get(arena,#put.yposition,#put.xposition) == empty) ]->
  arena[value(#put.yposition,#put.xposition)] := #put.symbol;
}
```

Figure 5.5: CARML module for the game arena of the Tic-Tac-Toe game

The Tic-Tac-Toe game itself is composed with the help of an RSL script as shown in Figure 5.6. The script creates an instance of the game arena and of the *Rules* component and joins their interface ports. Additionally, some of the atomic propositions defined for the game arena are lifted to the game level. These atomic propositions are now available for use within ASL formulas. Moreover the script provides a mechanism to hide the moves of player X and/or player O (compare Remark 5.3.1) by setting corresponding command line flags of Vereofy. Finally, an alias is set, which identifies the Tic-Tac-Toe game as the main system to be analyzed by default.

```
CIRCUIT TTT_Game{
  theArena = new TTT_Arena(NULL);
  theRules = new TTT_ROTG(PlayerX,PlayerO;NULL);
  join(theArena.source,theRules.sink);

  AP("cross_wins",  "theArena.cross_wins");
  AP("circle_wins", "theArena.circle_wins");
  AP("winning",     "theArena.someone_wins");
  AP("draw",        "theArena.isFull & !winning");
  AP("game_over",   "theArena.game_over");

  @if +hide_PlayerO_moves
    PlayerO = NULL;
  @endif

  @if +hide_PlayerX_moves
    PlayerX = NULL;
  @endif
}

ALIAS main = TTT_Game;
```

Figure 5.6: RSL circuit composing the Tic-Tac-Toe game

Model statistics. We will now present some statistics regarding the size of the composed system model for the Tic-Tac-Toe game as well as the time needed to parse the RSL main program and build the symbolic representation of the model. Table 5.1 shows the statistics for various sizes of the game board.

board size	reachable states	BDD variables	BDD nodes	build time (s)
3×3	$5.4 \cdot 10^3$	78 (19)	4076	0.1
4×4	$9.7 \cdot 10^6$	120 (33)	50121	0.3
5×5	$1.6 \cdot 10^{11}$	180 (51)	359207	2.6
6×6	$2.4 \cdot 10^{16}$	246 (73)	2106808	23
7×7	$3.3 \cdot 10^{22}$	324 (99)	11803721	970

Table 5.1: Composition time and size of the system model of the Tic-Tac-Toe game

The first column of the table shows the size of the game board, which corresponds to the `arena_size` constant defined in the RSL main program declaration part. By passing this constant value via the command line, it takes precedence over the previous definition in the source code of the model. In the second column the number of reachable states of the composed system is shown. This value agrees with the number of possible game configurations. The third column shows the number of BDD variables used by Vereofy to encode all constraint automata including their ports and data bits for the encoding of the state space. The fourth column contains the number of BDD nodes used to encode the transition relation for the selected BDD variable ordering of the underlying constraint automaton. The last column shows the time spent to compose the entire system model for a variable ordering that was previously computed, stored, and then loaded before starting the composition procedure.

Model checking for the Tic-Tac-Toe game. We will now provide the model checking results for the Tic-Tac-Toe game. First we will consider a set of CTL formulas before providing some results for properties specified in terms of BTSL and ASL.

CTL model checking. For the sublogic CTL we will consider the formulas:

$$\begin{aligned}
\Phi_1 &= \exists((\text{“}\overline{\text{winning}}\text{”} \wedge \text{“}\overline{\text{draw}}\text{”} \wedge \text{“}\overline{\text{game over}}\text{”}) \mathbin{\mathtt{U}} \text{“cross wins”}) \\
\Phi_2 &= \exists((\text{“}\overline{\text{winning}}\text{”} \wedge \text{“}\overline{\text{draw}}\text{”} \wedge \text{“}\overline{\text{game over}}\text{”}) \mathbin{\mathtt{U}} \text{“circle wins”}) \\
\Phi_3 &= \exists((\text{“}\overline{\text{winning}}\text{”} \wedge \text{“}\overline{\text{draw}}\text{”} \wedge \text{“}\overline{\text{game over}}\text{”}) \mathbin{\mathtt{U}} \text{“draw”}) \\
\Phi_4 &= \forall((\text{“}\overline{\text{winning}}\text{”} \wedge \text{“}\overline{\text{draw}}\text{”} \wedge \text{“}\overline{\text{game over}}\text{”}) \mathbin{\mathtt{U}} \text{“game over”}) \\
\Phi_5 &= \exists\Box\, \text{“}\overline{\text{winning}}\text{”} \\
\Phi_6 &= \forall\Box((\text{“winning”} \vee \text{“draw”}) \implies \neg\exists \mathtt{X}\, \mathsf{true})
\end{aligned}$$

Here, by “$\overline{a}$” we mean the negation of an atomic proposition $a \in AP$. The above CTL formulas reason about the visited states along paths in the constraint automaton, i.e., plays

in the Tic-Tac-Toe game. The first three properties are liveness properties stating that configurations where player X wins, player O wins, or the game ends in a draw are reachable. Φ_4 states that all plays end in a configuration where the game is over. Φ_5 states the existence of a path that is globally not visiting a winning configuration. In Φ_6 it is stated that there is always possible a next step, if the current state is neither a winning state nor a state in which the game is a draw. Here, the sub formula $\exists \mathrm{X}\, \mathsf{true}$ states the existence of a next step, i.e., characterizes the set of states that are not terminal states. All CTL formulas are defined such that they are satisfied for the system model of size $3{\times}3$. Indeed they hold for all board sizes that have been investigated within this case study.

We successfully checked the above properties with Vereofy. Figure 5.7 shows the output when executing Vereofy for the Tic-Tac-Toe game and formula Φ_4. After building the symbolic representation of the constraint automaton and checking Φ_4 (and its subformulas recursively), a witness path constituted by the configurations of the game board and the moves of the players is printed. The game configuration (which is indeed a draw) that is reached in the witness path (cf. output presented in Figure 5.7) is depicted in Figure 5.8.

Table 5.2 shows the time in seconds spent to check the given properties. The table consists of three sub-tables, where the first shows the time when the reachable fragment of the state space has been computed before starting the model checking. The time shown in the first sub-table includes the time needed for the precomputation of the reachable game configurations. The second sub-table shows the times when precomputation has been disabled and replaced by a dynamic reordering during the model checking procedure again using a time and spaced bounded variant of sifting. The third sub-table shows the times when neither the reachable fragment has been computed in advance, nor dynamic reordering was carried out. The Vereofy process was executed with a limited memory of 20GB (which was never exceeded throughout our experiments) and a limited time of 2, 3, or 4 hours. Once the resources were exhausted either in time or space the process has been killed which is indicated by the entries in the tables that are marked red.

```
klueppel@yoda:~/work/mydiss/code/TicTacToe$ vereofy-run --input=TicTacToe.rsl --formula='ASL<<EU["!win
ning & !draw & !game_over","draw"]>>' --btsl-witness-level=1

Vereofy Model Checker (internal Nov 10 2011 22:56:44 [JINC_new ADD]),  http://www.vereofy.de
Copyright (C) 2008-2011, Vereofy Group, TU Dresden
Generation of CA finished, time = 0.020s
CA initialStates() = 21 BDD Nodes
CA delta() = 5018 BDD Nodes
Checking formula ASL<<EU["!winning & !draw & !game_over","draw"]>>

checking: EU[!winning & !draw & !game_over,draw]
.early termination is enabled for EU[!winning & !draw & !game_over,draw]

checked: !winning & !draw & !game_over: 33.3333%

checked: draw: 66.6667%
 step 0, 38994 bdd nodes
 step 1, 40684 bdd nodes
 step 2, 45445 bdd nodes
 step 3, 54131 bdd nodes
 step 4, 64762 bdd nodes
 step 5, 75323 bdd nodes
 step 6, 83347 bdd nodes
 step 7, 88426 bdd nodes
 step 8, 90749 bdd nodes

checked: EU[!winning & !draw & !game_over,draw]: 100%
.Time: 0.050 sec (elapsed), 0.176011 sec (CPUTime)

status: 1 formula(s), 3 subformula(s), 94131 BDD nodes.

Formula: ASL<<EU["!winning & !draw & !game_over","draw"]>>
Result: PASSED

Witness for EU[!winning & !draw & !game_over,draw]:
theArena[0].arena={empty,empty,empty,empty,empty,empty,empty,empty,empty} theRules[0].turn=PlayerX
  {PlayerX[0]={cross,2,2}}
theArena[0].arena={empty,empty,empty,empty,empty,empty,empty,empty,cross} theRules[0].turn=PlayerO
  {PlayerO[0]={circle,1,2}}
theArena[0].arena={empty,empty,empty,empty,empty,empty,empty,circle,cross} theRules[0].turn=PlayerX
  {PlayerX[0]={cross,0,2}}
theArena[0].arena={empty,empty,empty,empty,empty,empty,cross,circle,cross} theRules[0].turn=PlayerO
  {PlayerO[0]={circle,2,1}}
theArena[0].arena={empty,empty,empty,empty,empty,circle,cross,circle,cross} theRules[0].turn=PlayerX
  {PlayerX[0]={cross,1,1}}
theArena[0].arena={empty,empty,empty,empty,cross,circle,cross,circle,cross} theRules[0].turn=PlayerO
  {PlayerO[0]={circle,2,0}}
theArena[0].arena={empty,empty,circle,empty,cross,circle,cross,circle,cross} theRules[0].turn=PlayerX
  {PlayerX[0]={cross,0,1}}
theArena[0].arena={empty,empty,circle,cross,cross,circle,cross,circle,cross} theRules[0].turn=PlayerO
  {PlayerO[0]={circle,0,0}}
theArena[0].arena={circle,empty,circle,cross,cross,circle,cross,circle,cross} theRules[0].turn=PlayerX
  {PlayerX[0]={cross,1,0}}
theArena[0].arena={circle,cross,circle,cross,cross,circle,cross,circle,cross} theRules[0].turn=PlayerO
klueppel@yoda:~/work/mydiss/code/TicTacToe$ []
```

Figure 5.7: Executing Vereofy for the Tic-Tac-Toe game and formula Φ_4

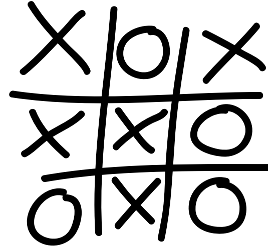


Figure 5.8: Tic-Tac-Toe draw configuration as reached in the witness path in Figure 5.7

reachable fragment, no dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6
3×3	0.04	0.04	0.04	0.04	0.05	0.02
4×4	5.71	8.56	6.91	1.33	6.18	0.53
5×5	4205	3519	3133	69	2413	23
6×6	>3h	>3h	>3h	10603	>3h	1690
7×7	>3h	>3h	>3h	>3h	>3h	>3h

full game arena, with dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6
3×3	1.37	1.37	1.34	1.33	1.40	0.42
4×4	48	68	104	44	95	8.54
5×5	3984	4756	>3h	1117	>3h	27
6×6	>3h	>3h	>3h	>3h	>3h	68
7×7	>3h	>3h	>3h	>3h	>3h	461

full game arena, no dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6
3×3	0.04	0.03	0.05	0.04	0.05	0.01
4×4	10	12	50	5.38	36	0.03
5×5	3277	3684	>3h	751	>3h	0.37
6×6	>3h	>3h	>3h	>3h	>3h	4.9
7×7	>3h	>3h	>3h	>3h	>3h	275

Table 5.2: Time in seconds for CTL model checking of Tic-Tac-Toe

As one can see, the runtime of the model checker is dominated by the precomputation of the reachable fragment (where carried out). For a board size of 6×6 the precomputation takes almost 30 minutes, whereas for board size 7×7 it takes weeks to finish. In case of Φ_4 the precomputation of the reachable fragment allowed to improve the runtime of the symbolic model checking algorithms, but this does not hold in general and depends on the investigated system model and the current formula. Here, for some of the selected formulas such as Φ_1 and Φ_2 the precomputation required some additional time and the time for model checking reduced about the same order. For other CTL formulas such as Φ_5 and Φ_6 the model checking does not benefit from the precomputation at all. The dynamic reordering has no positive effect on the runtime here although it helps to reduce the required memory.

BTSL model checking. For BTSL we will consider the following formulas:

$$
\begin{aligned}
\Phi_1 &= \neg\exists\langle tt^*; data_{PlayerX} = \text{circle}\rangle \mathsf{true} \\
\Phi_2 &= \neg\exists\langle tt^*; data_{PlayerO} = \text{cross}\rangle \mathsf{true} \\
\Phi_3 &= \neg\exists\langle tt^*; PlayerX; PlayerX\rangle \mathsf{true} \\
\Phi_4 &= \neg\exists\langle tt^*; PlayerO; PlayerO\rangle \mathsf{true} \\
\Phi_5 &= \neg\exists\langle tt^*; PlayerX\rangle(\text{“}\overline{\text{cross wins}}\text{”} \wedge \text{“}\overline{\text{draw}}\text{”} \wedge \neg\exists\langle PlayerO\rangle \mathsf{true}) \\
\Phi_6 &= \neg\exists\langle tt^*; PlayerO\rangle(\text{“}\overline{\text{circle wins}}\text{”} \wedge \text{“}\overline{\text{draw}}\text{”} \wedge \neg\exists\langle PlayerX\rangle \mathsf{true})
\end{aligned}
$$

The above BTSL formulas now reason about states and the observable dataflow at the two existing ports $PlayerX$ and $PlayerO$. Again, all BTSL formulas are defined in a way such that they are satisfied for the system model. For $\Phi_1, \dots, \Phi_4$ this means that there is not a single play in the game with the specified prefix, i.e., where one of the players puts the wrong symbol on the game board or has the ability to make two back-to-back moves. Formula Φ_5 states that there should not be play in which player X made the last move, the reached configuration is neither winning for player X nor a draw and the opponent cannot place his next mark on the game board. Formula Φ_6 formulates an analog proposition for player O. Table 5.3 depicts the time in seconds to check the given BTSL formula.

reachable fragment, no dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6
3×3	0.02	0.02	0.02	0.02	0.02	0.03
4×4	0.41	0.41	0.45	0.46	0.54	0.53
5×5	20	21	22	21	22	22
6×6	1651	1646	1652	1661	1687	1670
7×7	$>$3h	$>$3h	$>$3h	$>$3h	$>$3h	$>$3h

full game arena, with dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6
3×3	0.42	0.42	0.42	0.42	0.42	0.43
4×4	8.58	8.58	8.53	8.54	8.56	8.59
5×5	27	28	27	27	28	27
6×6	68	67	69	70	72	72
7×7	457	459	472	477	655	600

full game arena, no dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6
3×3	0.01	0.01	0.01	0.01	0.01	0.01
4×4	0.02	0.02	0.04	0.04	0.07	0.07
5×5	0.21	0.23	0.39	0.41	0.64	0.65
6×6	3.08	3.3	5.17	5.33	9.3	9.08
7×7	229	255	268	286	538	740

Table 5.3: Time in seconds for BTSL model checking of Tic-Tac-Toe

As Table 5.3 shows that the selected BTSL formulas can be checked efficiently without dynamic reordering and precomputation of the reachable states. This is due to the fact that we quantity over prefixes that cannot occur, as for $\Phi_1, \ldots, \Phi_4$ or the combination of states and prefix cannot occur. For Φ_5 and Φ_6 this means that they are a little more time-consuming.

ASL model checking. For ASL we will consider the following formulas:

$$
\begin{aligned}
\Phi_1 &= \mathbb{E}_{\{PlayerX, PlayerO\}} \Diamond \text{“draw”} \\
\Phi_2 &= \neg\mathbb{E}_{\{PlayerX\}} \Diamond \text{“cross wins”} \\
\Phi_3 &= \neg\mathbb{E}_{\{PlayerX\}} \Diamond \text{“draw”} \\
\Phi_4 &= \mathbb{E}_{\{PlayerX\}} \Diamond (\text{“game over”} \wedge (\text{“}\overline{\text{cross wins}}\text{”} \implies \text{“draw”})) \\
\Phi_5 &= \mathbb{E}_{\{PlayerX\}} \Box (\text{“game over”} \implies (\text{“}\overline{\text{cross wins}}\text{”} \implies \text{“draw”})) \\
\Phi_6 &= \neg\mathbb{E}_{\{PlayerO\}} \Diamond \text{“circle wins”} \\
\Phi_7 &= \neg\mathbb{E}_{\{PlayerO\}} \Diamond \text{“draw”} \\
\Phi_8 &= \mathbb{E}_{\{PlayerO\}} \Diamond (\text{“game over”} \wedge (\text{“}\overline{\text{circle wins}}\text{”} \implies \text{“draw”})) \\
\Phi_9 &= \mathbb{E}_{\{PlayerO\}} \Box (\text{“game over”} \implies (\text{“}\overline{\text{circle wins}}\text{”} \implies \text{“draw”})) \\
\Phi_{10} &= \mathbb{E}_{\{PlayerO\}} \Diamond \text{“someone wins”}
\end{aligned}
$$

The above ASL formulas now reason about strategies rather than only states and dataflow. For different sets of controllable ports the formulas state whether or not certain liveness and safety properties can be enforced. In case of ASL our formulas are defined such that they are satisfied for the system model of size $3{\times}3$. As we found out with the help of Vereofy, the formulas Φ_3 and Φ_{10} do not hold for board size $4{\times}4$, which is why the corresponding fields in Table 5.4 are highlighted. The table depicts the time in seconds to check the given ASL formula.

One can see that the ASL model checking is consuming more time than needed for the presented CTL and BTSL formulas and applicable up to the board size of $5{\times}5$ only. We claim that this is due to the fact that far more memory is needed during the fixpoint computations as the transition relation has to be decided into controllable and (un)preventable transitions and hence more (temporary) BDD nodes are introduced into the BDD structure. Furthermore we claim that this may change when adding regular expressions for reasoning about the dataflow. Indeed, the above ASL formulas are ATL formulas only. At the end of this subsection we now compare the memory and time-usage during the fixpoint computation for different formula types more systematically.

reachable fragment, no dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6	Φ_7	Φ_8	Φ_9	Φ_{10}
3×3	0.09	0.10	0.03	0.11	0.03	0.09	0.05	0.13	0.03	0.15
4×4	436	78	491	94	1.1	44	286	240	1.02	516
5×5	>3h	>3h	>3h	>3h	287	>3h	>3h	>3h	305	>3h
6×6	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h

full game arena, with dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6	Φ_7	Φ_8	Φ_9	Φ_{10}
3×3	1.85	1.47	1.31	1.77	0.38	1.50	1.35	1.88	0.42	1.75
4×4	288	164	797	453	18	165	790	452	18	936
5×5	>3h	>3h	>3h	>3h	60	>3h	>3h	>3h	60	>3h
6×6	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h

full game arena, no dynamic reordering:

board size	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5	Φ_6	Φ_7	Φ_8	Φ_9	Φ_{10}
3×3	0.12	0.08	0.06	0.16	0.02	0.07	0.05	0.16	0.01	0.15
4×4	1105	149	1207	533	0.47	147	1206	532	0.43	1575
5×5	>3h	>3h	>3h	>3h	135	>3h	>3h	>3h	95	>3h
6×6	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h	>3h

Table 5.4: Time in seconds for ASL model checking of Tic-Tac-Toe

Memory-usage. To analyze the memory-usage of the different Pre-operators, we now look at the Tic-Tac-Toe game with board size of $3{\times}3$ and $4{\times}4$, fix the variable ordering and measure the number of BDD nodes kept in memory for the different Pre-operators applied when checking the following CTL, BTSL, ATL, and ASL formulas which have a similar structure:

$$\begin{aligned} \text{CTL formula } \Phi_1 &= \exists\Diamond\text{``draw''} \\ \text{BTSL formula } \Phi_2 &= \exists\langle\mathcal{Z}_k\rangle\text{``draw''} \\ \text{ATL formula } \Phi_3 &= \mathbb{E}_{\{PlayerX,PlayerO\}}\Diamond\text{``draw''} \\ \text{ASL formula } \Phi_4 &= \mathbb{E}_{\{PlayerX,PlayerO\}}\langle\mathcal{Z}_k\rangle\text{``draw''} \end{aligned}$$

where $\mathcal{Z}_k$ is an NFA accepting the language

$$\mathcal{L}(\mathcal{Z}_k) = \big\{c_1 \dots c_k \bigm| \mathsf{active}(c_1) = \{PlayerX\} \text{ and } \mathsf{active}(c_2) = \{PlayerO\}\big\} \tag{5.1}$$

Here, we set $k \stackrel{\text{def}}{=} n^2$ if the selected board size equals $n{\times}n$ as k is the maximum number of moves that the two players can make. Hence, the fixpoint computation for the selected formulas requires exactly k Pre-image steps before termination for all the selected formulas. Figure 5.9 shows the number of BDD nodes (including all temporal BDD nodes) that are stored for each of the four formulas when looking at a board size of $3{\times}3$. The first plot illustrates the memory usage when the precomputation of the reachable states is disabled, while the second plot shows the same progression, but here the precomputation is enabled. One can see that storing the characteristic function of the reachable fragment requires some additional memory, as the starting point for all four plots is higher in the second diagram. Moreover, the alternating-time logics seem to require about twice as much memory as their branching-time counterparts. This is due to the more complex Pre-operator, which requires the controllable and the (un)preventable parts of the transition relation to be stored separately. Here, another interesting observation is, that using an NFA $\mathcal{Z}$ of the form describes in equation (5.1) in the BTSL and ASL formula can reduce the actual amount of memory that is needed in comparison with their CTL and ATL counterpart formulas, although new BDD variables for the state space of $\mathcal{Z}$ have been added to the BDD structure. This does not hold in general, as the structure of the NFA and hence the model checking can be arbitrary complex.

Figure 5.10 shows the same progression for a board size of $4{\times}4$. In these plots we can observe some similar behavior to the $3{\times}3$ case, but at some point the number of BDD nodes breaks down to some lower value. This behavior can be observed, because at this point of the fixpoint computation a threshold of unused temporary BDD nodes has been reached and the garbage collection is carried out. Executing the garbage collection at this point consumes some additional time and reduces the number of BDD nodes dramatically. In Figure 5.11 we plot the progression in the $4{\times}4$ case when setting the garbage collection delay (gcd), i.e., the number of unused BDD nodes kept in memory before starting the garbage collection procedure, to "infinity". When selecting a value that will never be reached (indicated by gcd=inf), one can observe the same behavior as in the $3{\times}3$ case (cf. Figure 5.9) where the garbage collection has not been started during the fixpoint computation either.

Again, the alternating-time logics require more memory as their branching-time counterparts and using the BTSL and ASL formula rather than a similar CTL and ATL formula

reduces the actual amount of memory needed during the fixpoint computation. The latter result crucially depends on the structure of the NFA and the system itself and cannot be generalized as the BTSL and ASL model checking problem can become arbitrary difficult by increasing the complexity of the NFA.

As the above results include the temporary BDD nodes, we close this paragraph by comparing the memory usage during the fixpoint computation for the four formulas when setting the garbage collection delay to value zero. Hence, the number of BDD nodes plotted in Figures 5.12 and 5.13 show the progression for the board sizes 4×4 and 3×3 when all the temporary BDD nodes are instantaneously removed from the memory once they get unused. This reduces the memory usage, requires some additional time for collecting the unused BDD nodes, and slows down the entire fixpoint computation as temporal results cannot be reused in terms of a computed table.

Overall, the two figures show that the plots for BTSL and ASL as well as CTL and ATL have a very similar shape. We observe here, that the actual amount of memory (which does not include the BDD nodes of temporal intermediate results) is larger for the BTSL and ASL formulas than for the CTL and ATL formulas. This is because, after each image step we look at the result characterized by boolean functions over the BDD variables for the state space of the product of $\mathcal{A}$ and $\mathcal{Z}_k$ rather than variables for the state space of $\mathcal{A}$ only.

Time and memory correlation. As all the model checking algorithms introduced in the previous chapters use BDDs as data structure predominantly, we do expect the runtime of the model checking to be correlated with the size of the BDD structure. Figure 5.14 shows the overall runtime of the model checking procedures for the formulas $\Phi_1, \ldots, \Phi_4$ as introduced above. The first graph plots the time in seconds needed to verify the properties for the board size 3×3 whereas the second graph shows the overall runtime in seconds in the 4×4 case. The plots are again given for the two cases, when precomputing the reachable fragment is enabled or disabled. The garbage collection delay has again been set to “infinity” to ensure that the garbage collection is disabled and does not consume any additional time.

The correlation between the allocated memory and the runtime can be seen at different positions. The first is, that having the reachable fragment of the state space precomputed, allows to have a more compact representation of the intermediate results (by constraining with respect to the reachable states). Hence, the overall runtime for the selected type of formulas (i.e., liveness property whose satisfaction set is characterized by a least fixpoint) decreases along with the allocated memory (compare, e.g., Figure 5.12). Another point is that the verification of the alternating-time formulas requires more time than the branching-time properties, as more temporary boolean functions must be computed and stored (cf. Figure 5.11). Hence, the runtime is dominated by the size of the temporary functions. Another important influence comes from the size of the boolean functions storing the intermediate results during the model checking. This can be seen when comparing the time for model checking of the ASL and the ATL formulas in the Tic-Tac-Toe game of size 4×4. Here, checking the ASL formula requires more time although the overall allocated memory for checking is much larger when checking the ATL formula. This is because for the ATL formula the functions storing intermediate results can be represented by fewer BDD nodes (cf. Figure 5.12).

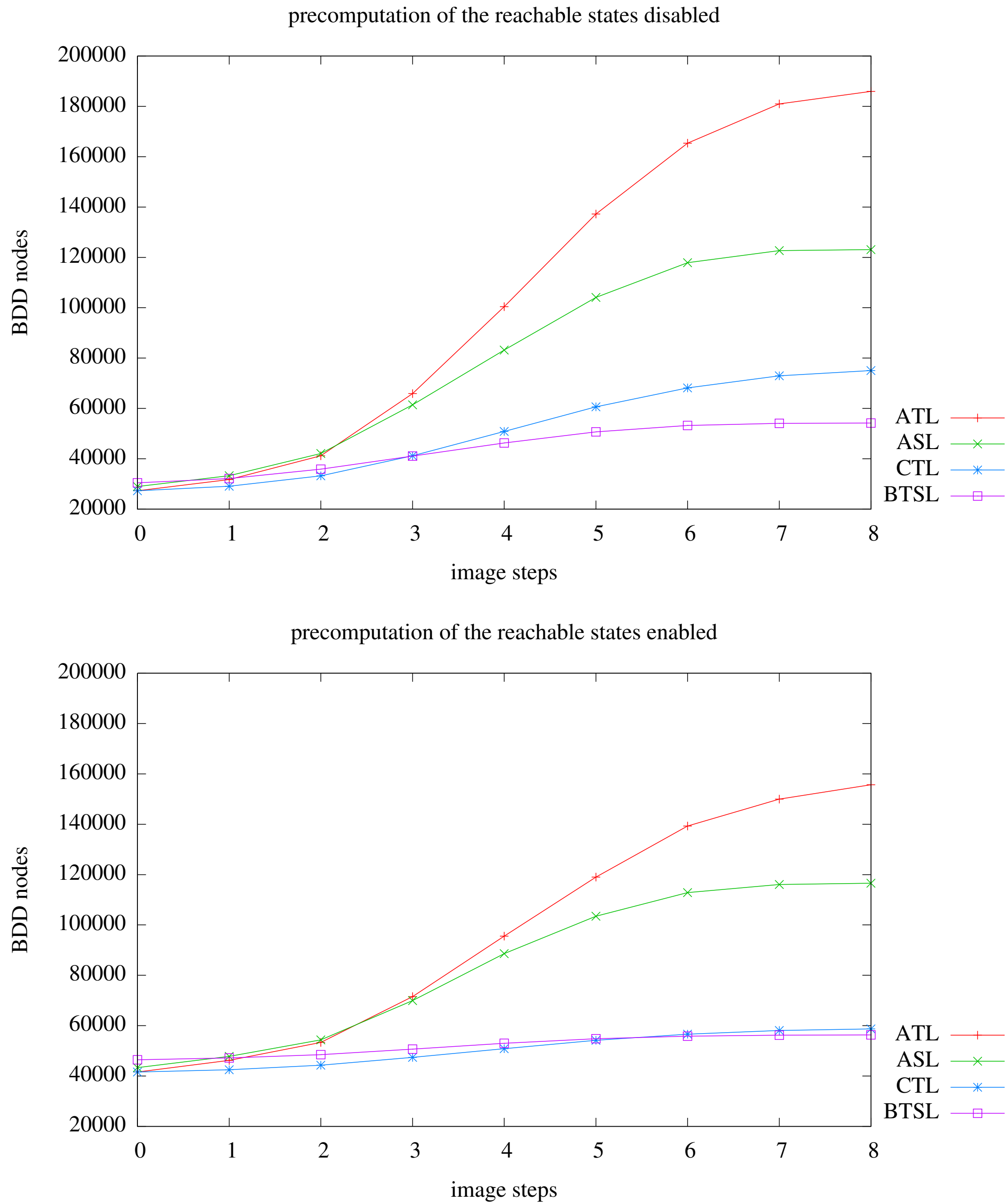


Figure 5.9: BDD nodes during fixpoint computation for Tic-Tac-Toe 3×3 (gcd=default)

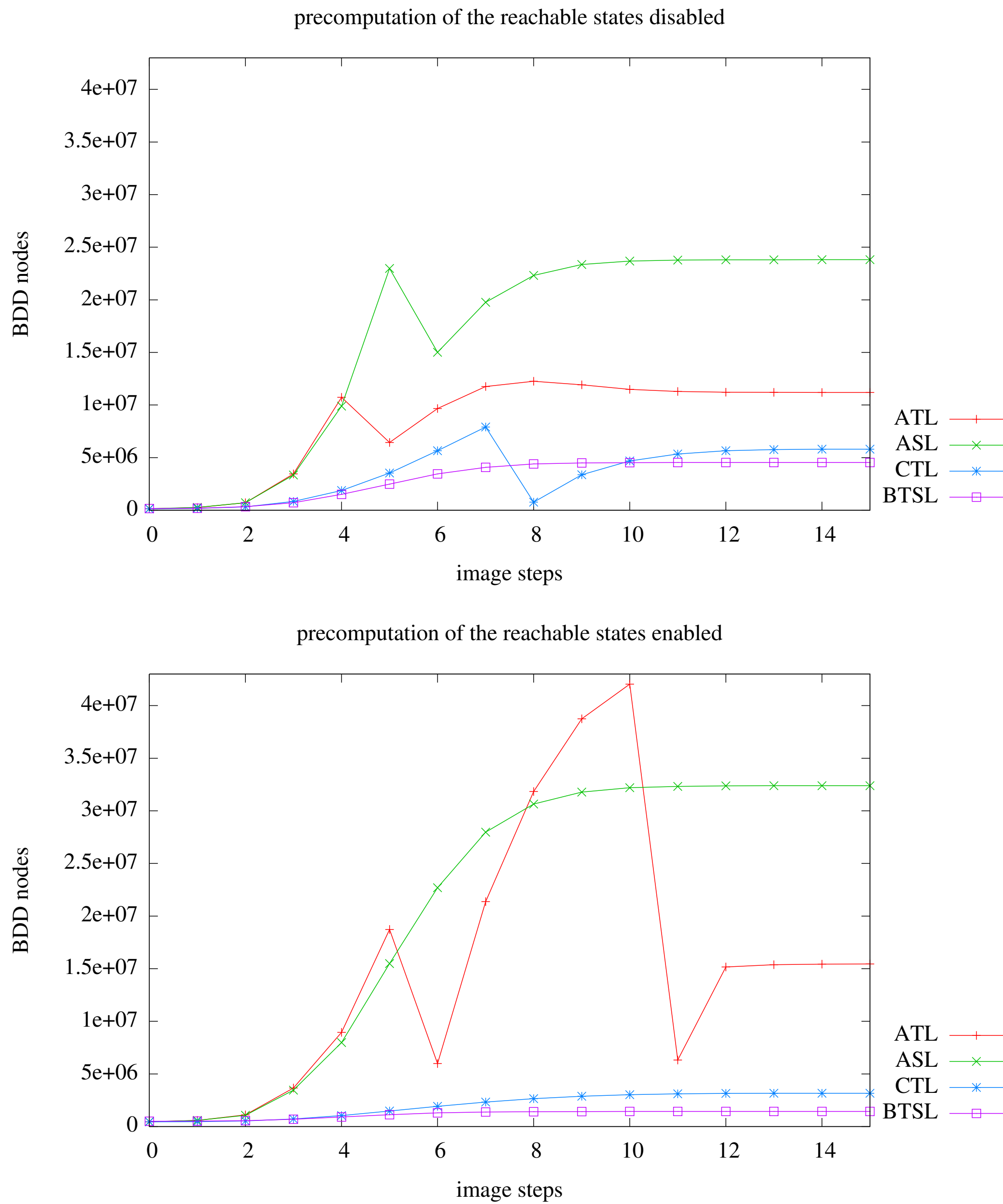


Figure 5.10: BDD nodes during fixpoint computation for Tic-Tac-Toe 4×4 (gcd=default)

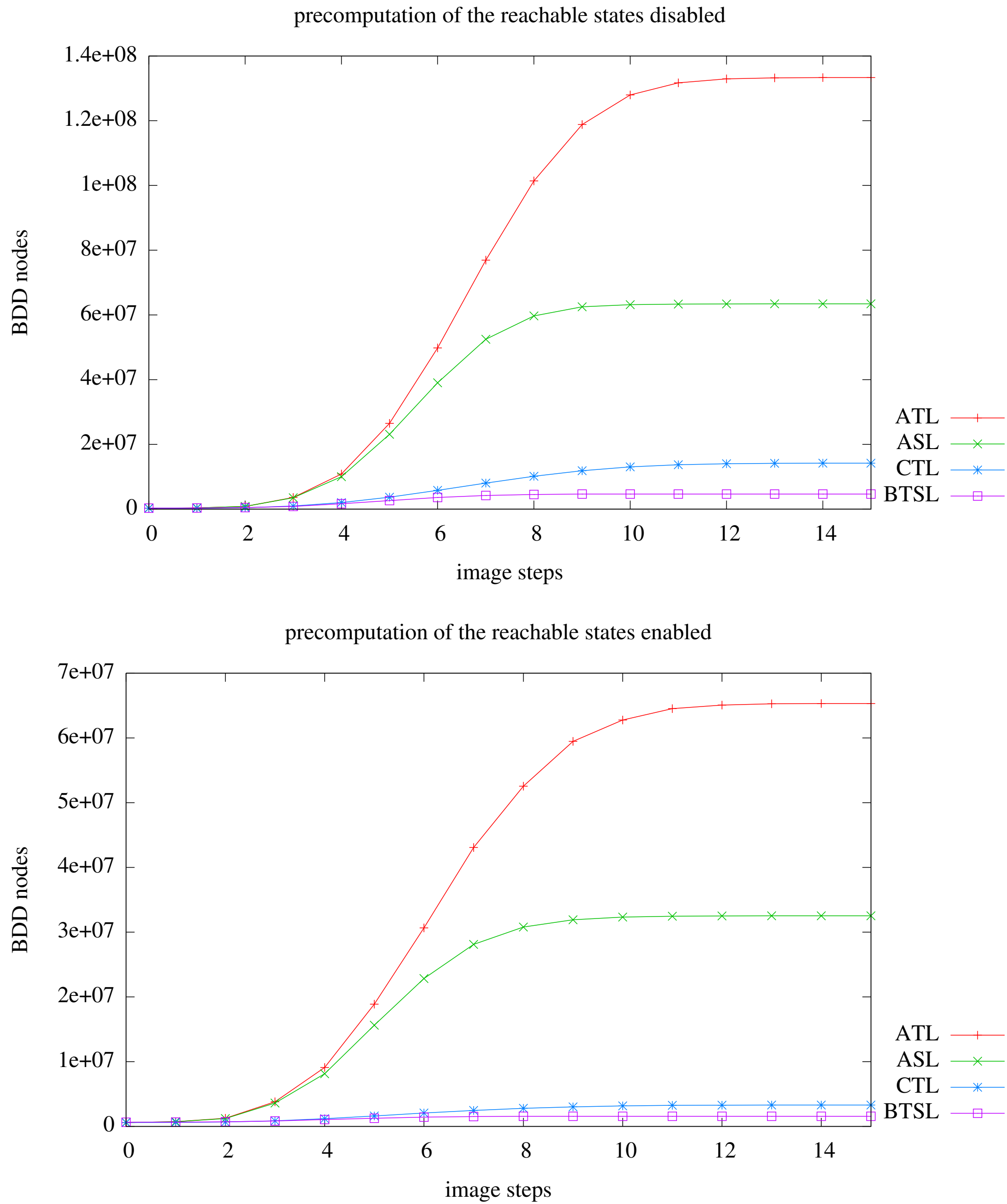


Figure 5.11: BDD nodes during fixpoint computation for Tic-Tac-Toe 4×4 (gcd=inf)

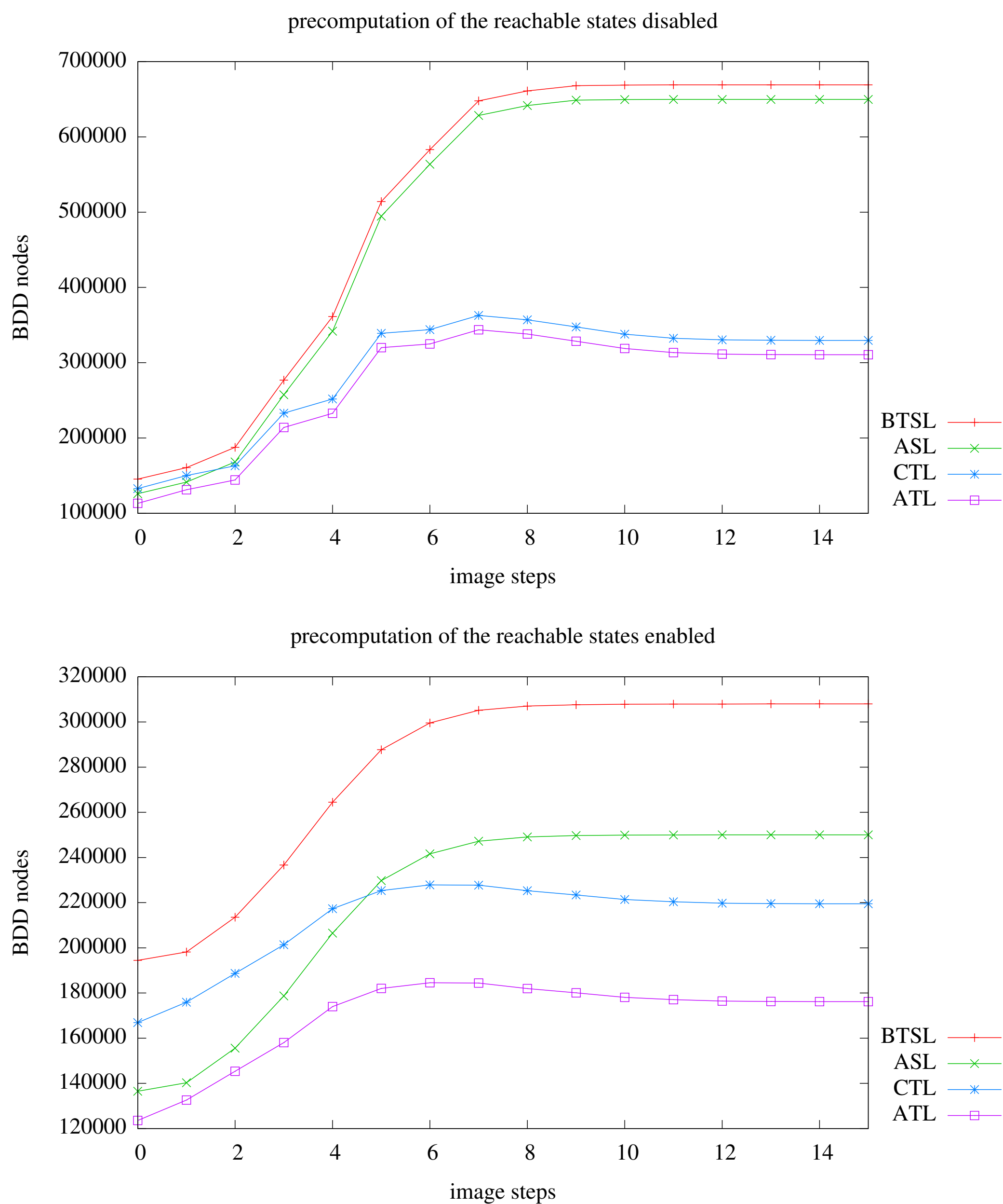


Figure 5.12: BDD nodes during fixpoint computation for Tic-Tac-Toe 4×4 (gcd=0)

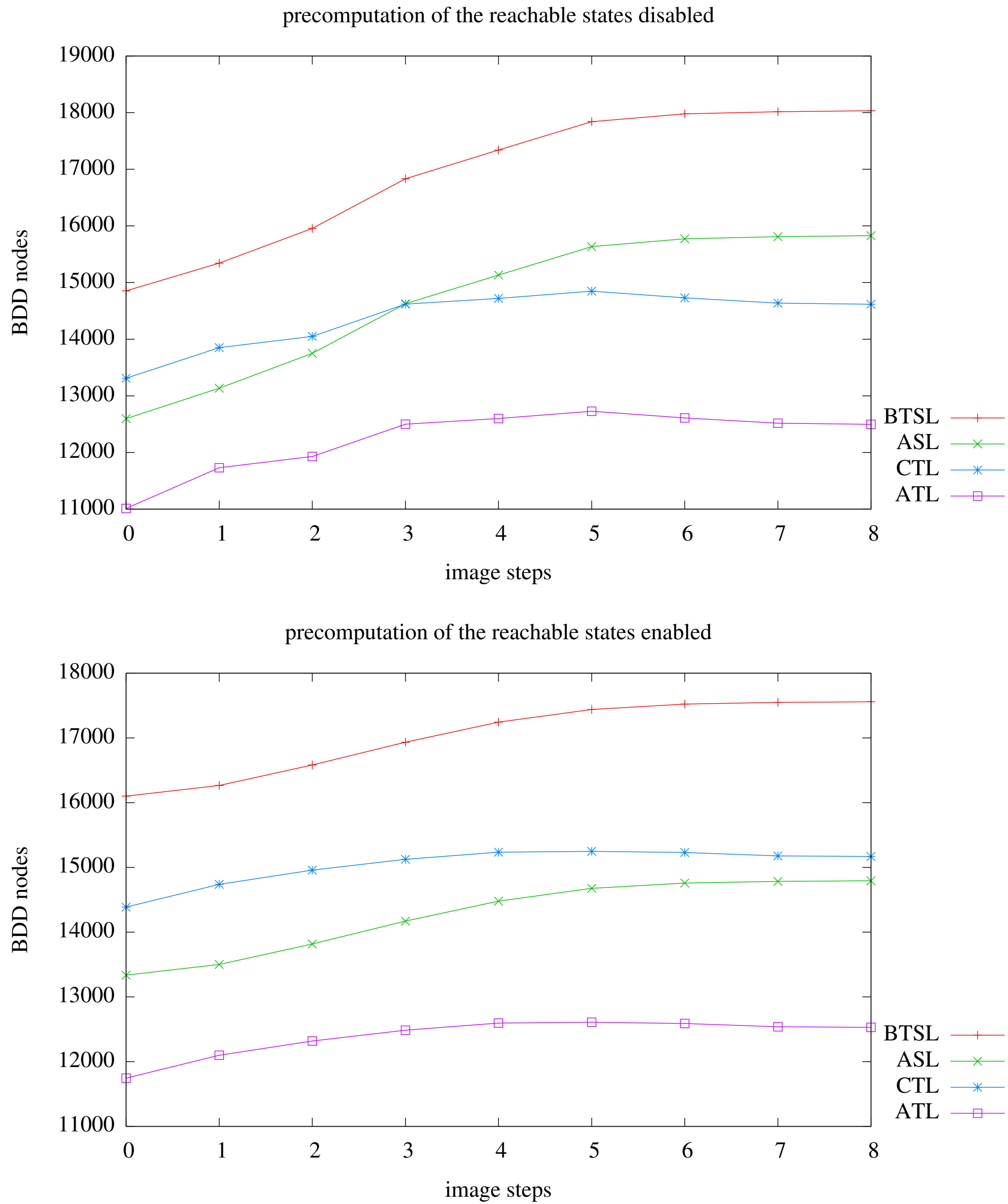


Figure 5.13: BDD nodes during fixpoint computation for Tic-Tac-Toe 3×3 (gcd=0)

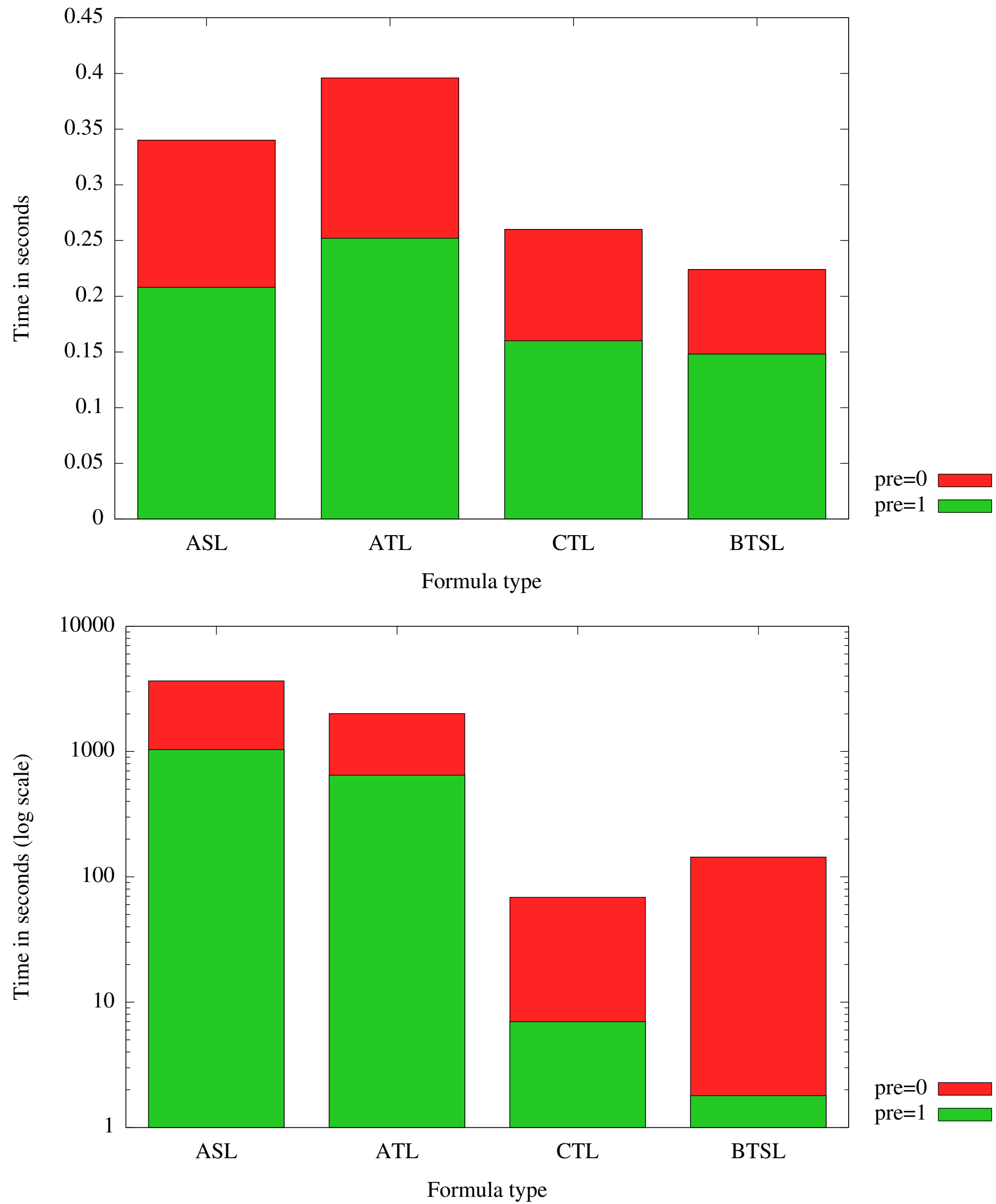


Figure 5.14: Time needed for fixpoint computation in Tic-Tac-Toe (gcd=inf)

5.3.2. Other game examples

We used Vereofy to build and check a number of other games such as Teeko, connect-k, and the boss-puzzle. The structure of our models of these games all follow the component model illustrated in Figure 5.2 and varies in the size of the game arena and the complexity of the game rules. It turned out that we could compose and verify system models that have about the same order of magnitude of reachable states as for the Tic-Tac-Toe game with our approach. E.g., we built the connect-k for a board size of 6×5 in 752 seconds. Computing the number of reachable states $(2.82 \cdot 10^9)$ consumed 6256 seconds.

The full models and some example properties are included in the zip archive available at:

`http://wwwtcs.inf.tu-dresden.de/~klueppel/PhD/examples.zip`

5.3.3. A dining philosophers example

In our next case study, which constitutes a very simple protocol, we revisit a variant of the prominent dining philosophers originated by Dijkstra [Dij68]. A number of philosophers are sitting at a round table. Their main activity is thinking, while from time to time they get hungry and want to eat some of the food placed in a bowl in the center of the table. To take some food out of the bowl, a philosopher needs exactly two pieces of cutlery, each of which is situated between any two neighboring philosophers. Hence, two neighboring philosophers share a single piece of cutlery, which can only be used by one of them at the same time. Hence, neighboring philosophers never eat at the simultaneously.

Our Vereofy model consists of a CARML module for each philosopher. The component modeling the behavior of each philosopher has four output ports to trigger the subsequent take and release actions of the cutlery pieces on their left and on their right. Furthermore, we provide CARML modules for the cutlery pieces, which can either be available or taken and they have two input ports for accepting take and release actions of the philosophers. The entire system is composed of these components and synchronous Reo channels. The channels connect the output ports of the philosopher components with the input port of the cutlery components. Here, a Reo standard node is created, ensuring mutually exclusive take actions. Figure 5.15 shows our component model for the dining philosophers example.

The constraint automata for the philosophers and the cutlery are shown in Figure 5.16. For the data domain Data of our Vereofy model we just require a single data item, i.e., the domain can be a singleton such as $\mathsf{Data} = \{0\}$.

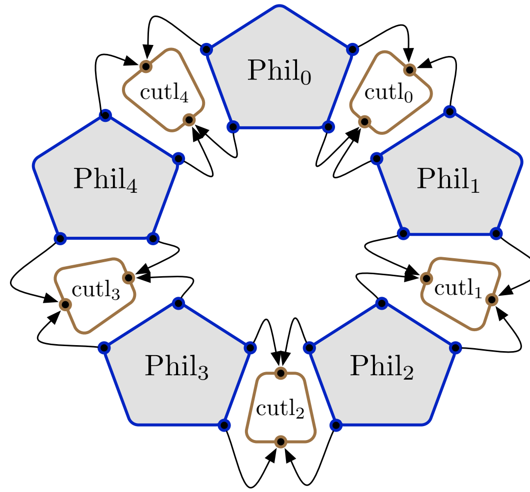


Figure 5.15: Component model for the dining philosophers

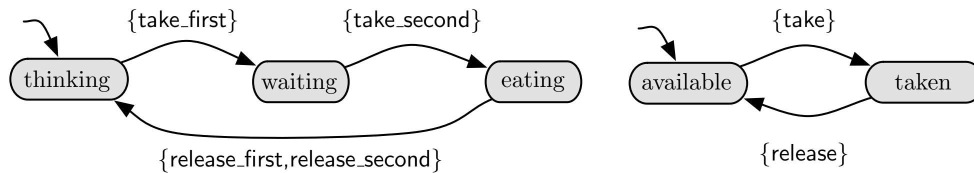


Figure 5.16: Constraint automata for the dining philosophers components

It is well known that a global deadlock may happen, e.g., when all philosophers simultaneously pick up the cutlery on their left (or in arbitrary order) and then wait for the second piece. Furthermore, there does not exist a fully symmetric deterministic solution without global control that can avoid this kind of deadlocks (e.g., see [Lyn96]). There are different ways to design a protocol for the philosophers which avoids these kinds of deadlock situations. One way is to break the symmetry by letting one of the philosophers take the cutlery on its right before taking the piece on the left whereas the other philosophers take the cutlery on the left first. To achieve this in our exogenous component model we will just connect the synchronous channels of one of the philosophers accordingly (see Figure 5.17), rather than modifying the specification of one of the philosophers [Arb05].

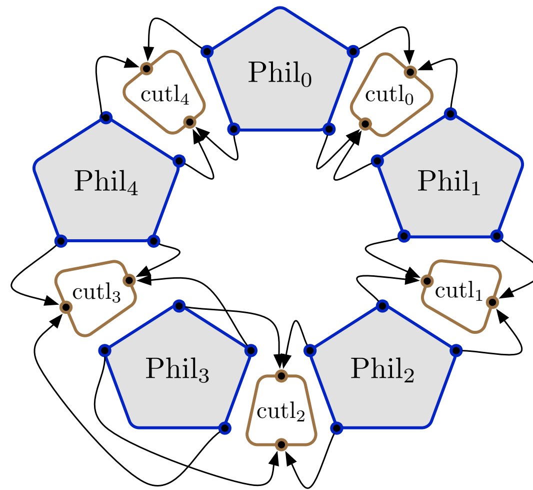


Figure 5.17: Alternative component model for the dining philosophers without deadlock

We provided RSL scripts for both variants of the protocol – with and without deadlocks – and also variants of the philosopher having four states rather than three and releasing the cutlery in two steps rather than in one. All variants are included in the zip archive which is available on the web:

`http://wwwtcs.inf.tu-dresden.de/~klueppel/PhD/examples.zip`.

Model statistics. We will again first present some statistics regarding the size of the composed system model as well as the time needed to parse the RSL main program and build the symbolic representation of the model. We show here the numbers for the model variant that includes a potential deadlock and where releasing the cutlery is done within a single step. Table 5.5 shows the statistics for an increasing number of philosophers. Here, we stopped at 1000 philosophers, where a peak number of 58741573 BDD nodes (almost 2.2GB) was needed to compose the system and compute the reachable part. The value "inf" for the number of reachable states indicated that the maximum value for floating-point representation was exceeded and the actual value could not be computed. This amount of memory is needed when computing the reachable states using the default garbage collection delay. When disabling the garbage collection entirely (gcd=inf) the maximum number of BDD nodes reached 571878848 which is almost 21GB.

philosophers	**reachable states**	**BDD vars**	**BDD nodes**	**build time (s)**		
5	82	75	539	2.51	0.02	0.02
6	198	90	692	3.02	0.03	0.04
7	478	105	835	3.52	0.04	0.03
8	1154	120	983	4.01	0.03	0.03
9	2786	135	1131	4.53	0.04	0.04
10	6726	150	1284	5.08	0.06	0.06
20	$4.5 \cdot 10^{7}$	300	2759	11.0	0.16	0.13
30	$3.0 \cdot 10^{11}$	450	4243	16.0	0.39	0.41
40	$2.0 \cdot 10^{15}$	600	5719	20.0	0.52	0.54
50	$1.4 \cdot 10^{19}$	750	7199	26.0	0.8	0.89
60	$9.3 \cdot 10^{22}$	900	8684	32.0	1.93	2.06
70	$6.2 \cdot 10^{26}$	1050	10159	37.0	1.78	1.92
80	$4.2 \cdot 10^{30}$	1200	11639	42.0	2.32	2.57
90	$2.8 \cdot 10^{34}$	1350	13119	48.0	3.08	3.34
100	$1.9 \cdot 10^{38}$	1500	14599	54.0	3.85	4.17
150	$2.6 \cdot 10^{57}$	2250	21999	83.0	9.52	10.0
200	$3.6 \cdot 10^{76}$	3000	29404	121.0	28.0	30.0
250	$4.9 \cdot 10^{95}$	3750	36799	147.0	29.0	32.0
300	$6.8 \cdot 10^{114}$	4500	44199	183.0	44.0	49.0
350	$9.4 \cdot 10^{133}$	5250	51599	219.0	62.0	68.0
400	$1.3 \cdot 10^{153}$	6000	59004	284.0	127.0	142.0
450	$1.8 \cdot 10^{172}$	6750	66399	298.0	105.0	119.0
500	$2.4 \cdot 10^{191}$	7500	73799	341.0	134.0	151.0
600	$4.6 \cdot 10^{229}$	9000	88599	432.0	202.0	228.0
700	$8.8 \cdot 10^{267}$	10500	103404	621.0	428.0	502.0
800	$1.7 \cdot 10^{306}$	12000	118204	758.0	695.0	701.0
900	"inf"	13500	132999	760.0	477.0	571.0
1000	"inf"	15000	147799	887.0	724.0	743.0

Table 5.5: Composition time and size of the dining philosophers

Again, for the numbers presented in Table 5.5 a variable ordering has been computed, stored and loaded before executing the build procedure again, but this time we applied the dynamic reordering of variables at designated points during the build process. The column for the composition time shows three time values for three different values of the garbage collection delay. The first value is for value 0, the second for the default value, and the last for setting the garbage collection delay to “inf”.

One can see that the number of reachable states grows exponentially, while at the same time the number of BDD nodes needed to represent the system behavior grows only linearly. For the composition time displayed in the last column of the table we observe that the time is not increasing strictly monotonously for all selected values of the garbage collection delay. Figure 5.18 shows a plot for the build times. It seems that there small irregularities for the composition of 200, 400, and 800 philosophers. As the effect seems to be independent from the chosen garbage collection delay and hence independent from the size of temporary functions, we claim that the effect traces back to the quality of the computed variable ordering.

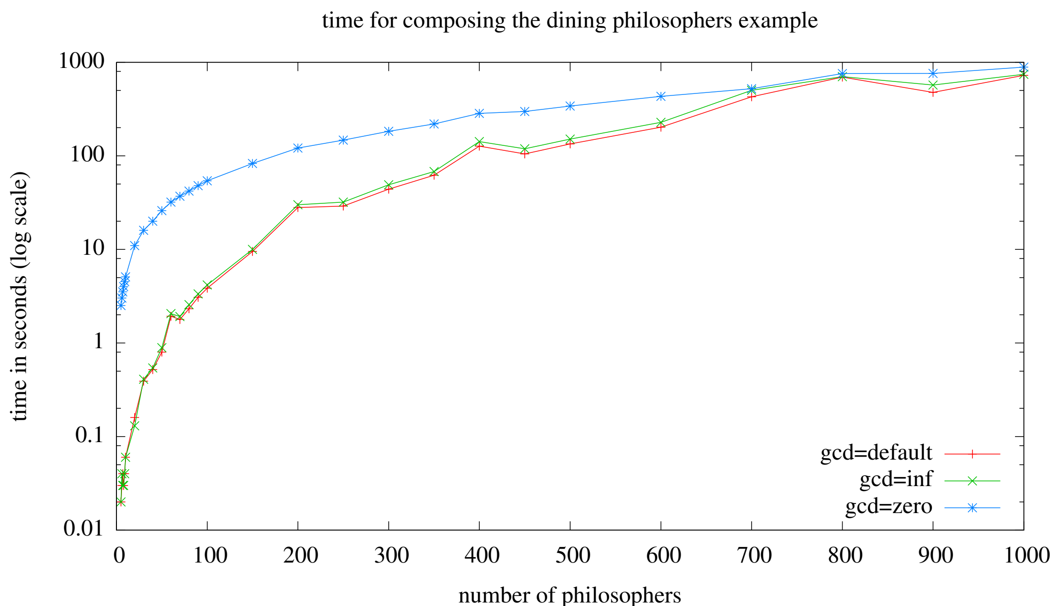


Figure 5.18: Composition time in seconds for the dining philosophers protocol

Similar irregularities can also be observed when looking at the peak number of BDD nodes after precomputing the reachable states, which took less than a second even for 1000 philosophers. Figure 5.19 shows the memory usage in terms of the peak number of BDD nodes in memory after precomputing the reachable states.

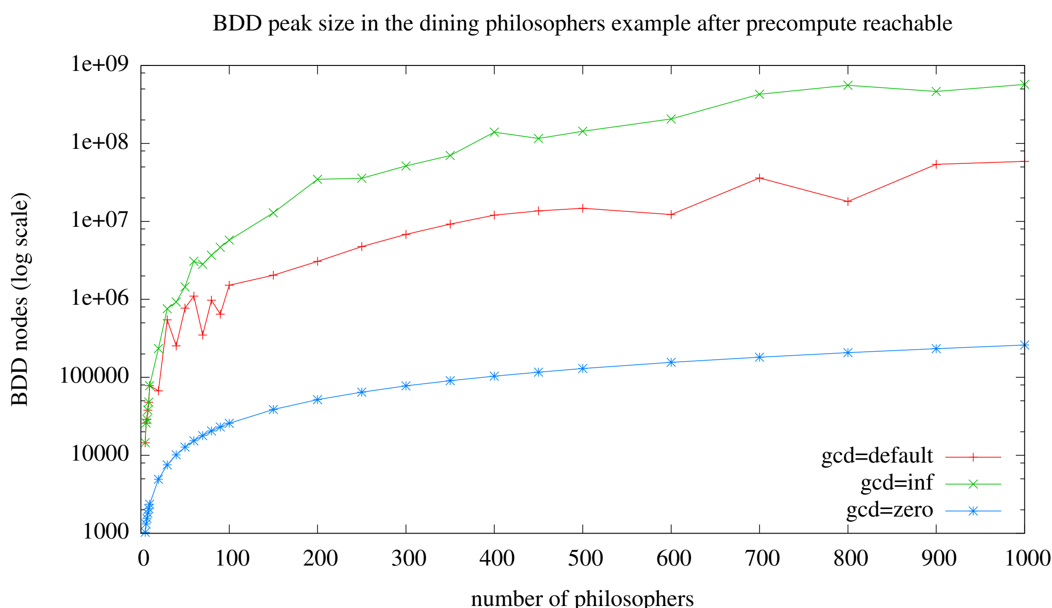


Figure 5.19: BDD nodes: precomputing reachable states of the dining philosophers

Model checking for the dining philosophers. The formulas listed below represent some of the properties that have been verified with the help of Vereofy for the dining philosophers protocol of various input sizes.

$$\Phi_1 = \forall\Box(\exists \mathsf{X}\, \mathsf{true})$$
$$\Phi_2 = \exists\Diamond\text{“all phils are waiting”}$$
$$\Phi_3 = \exists\langle \text{take_first[0]; take_second[0]}\rangle\text{“phil[0] is eating”}$$
$$\Phi_4 = \exists[\![\text{step}^+]\!]\text{“phil[0] is waiting”}$$
$$\Phi_5 = \exists(\text{“phil[0] is thinking”}\ \mathsf{U}\ \exists[\![\text{step}^*]\!]\text{“phil[0] is waiting”}$$
$$\Phi_6 = \mathbb{A}_{\text{phil[0],phil[2]}}\Diamond\text{“phil[1] is eating”}$$
$$\Phi_7 = \mathbb{A}_{\text{phil[0],phil[2]}}\langle \text{step; step}^+\text{; take_first[1]; take_second[1]}\rangle\text{“phil[1] is eating”}$$

The above ASL and BTSL formulas reason about different properties of the dining philosophers protocol. Φ_1 is a CTL formula characterizing the absence of a deadlock. Φ_2 states that there is a path to a configuration where all philosophers are in their waiting state. The formula Φ_3 states the existence of a path where the first philosopher immediately takes the two pieces of cutlery and starts eating. Φ_4 reasons about the existence of a path where after k steps the first philosopher is globally waiting for $k \geq 1$. The formula Φ_5 is a similar formula, but here along the path the first philosopher can stay at its initial location until in the next step he is waiting forever. The last two formulas are ATL and ASL formulas, where $\mathbb{A}_{\text{phil[0],phil[2]}}$ serves a shorthand notation for

$$\mathbb{A}_{\{\text{take_first[0],take_second[0],take_first[2],take_second[2]}\}}$$

i.e., all ports of the first and third philosopher. Both formulas state similar properties – Φ_7 includes an existential quantification over the observable actions and Φ_6 does not. We expect Φ_6 and Φ_7 to hold, as none of the neighboring philosophers can distract the philosophers in between them, because the nondeterminism may be resolved in a way such that the philosopher is the one that takes the cutlery.

Table 5.6 shows the results of the model checking without precomputing the reachable states for the model variant with three states per philosopher and no deadlock. Formula Φ_2 specifies the property that the system can get into a deadlock situation, where all philosophers are waiting. It can be seen in Table 5.6 that falsifying Φ_2 gets expensive such that the model checking runs into a timeout for the deadlock-free model. For Φ_2 Table 5.6 contains also the time for model checking for the model variant where a deadlock can occur and also the time for model checking the deadlock-free model when the set of reachable states has been precomputed in advance. Precomputing the reachable states took less than a second even for 1000 philosophers. Hence, precomputing the reachable states for this variant of the dining philosophers is rather cheap and the effort is worth doing it. Also the formulas Φ_4 and Φ_5 benefit from precomputing the reachable states. As for Φ_2 the model checking can be carried out in less than a second even for 1000 philosophers.

philosophers	Φ_1	Φ_2	(dead)	(pre)	Φ_3	Φ_4	Φ_5	Φ_6	Φ_7
5	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
6	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
7	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
8	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01
9	0.03	0.03	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10	0.03	0.04	0.01	0.01	0.01	0.01	0.01	0.01	0.02
20	0.09	0.5	0.01	0.01	0.01	0.01	0.01	0.01	0.02
30	0.19	2.12	0.01	0.01	0.01	0.03	0.04	0.02	0.06
40	0.26	6.65	0.02	0.01	0.01	0.04	0.06	0.02	0.08
50	0.34	14	0.01	0.01	0.01	0.07	0.07	0.02	0.09
60	0.41	26	0.01	0.02	0.01	0.09	0.1	0.02	0.1
70	0.54	47	0.01	0.02	0.01	0.16	0.15	0.03	0.13
80	0.6	77	0.01	0.02	0.01	0.18	0.19	0.03	0.14
90	0.78	121	0.01	0.02	0.01	0.28	0.28	0.05	0.19
100	0.87	183	0.01	0.03	0.02	0.37	0.36	0.04	0.23
150	1.28	1273	0.03	0.06	0.02	0.95	0.93	0.09	0.39
200	2.01	6413	0.04	0.08	0.02	1.77	1.82	0.12	0.53
250	2.23	>3h	0.03	0.08	0.03	2.2	2.16	0.13	0.55
300	2.53	>3h	0.05	0.13	0.05	6.69	7.18	0.22	0.77
350	3.04	>3h	0.06	0.12	0.05	13	14	0.2	0.85
400	3.34	>3h	0.07	0.14	0.06	36	71	0.24	0.95
450	3.9	>3h	0.1	0.11	0.04	472	471	0.22	0.97
500	11	>4h	0.1	0.22	0.07	817	820	0.35	1.34
600	17	>4h	0.14	0.29	0.1	1725	1738	0.43	1.81
700	24	>4h	0.16	0.31	0.14	3057	3086	0.47	2.05
800	31	>4h	0.21	0.38	0.15	4938	4957	0.58	2.43
900	18	>4h	0.17	0.39	0.13	9223	8486	0.81	3.51
1000	13	>4h	0.3	0.37	0.16	>4h	>4h	0.73	3.06

Table 5.6: Time in seconds for model checking of a dining philosophers protocol

5.3.4. Other protocol examples

We used Vereofy to build and check a number of other protocols such as a mutual exclusion protocol using semaphores [BBKK09a], a peer-to-peer network [GJA+10] and a sensor network protocol [BBK+10]. These applications illustrate already that our modeling and verification approach is applicable to systems of practical relevance. From the case studies of more complex and more data-intensive protocols one could see that the composition and analysis becomes more and more difficult.

In the last paragraph of this section we will give a brief overview on a case study of an industrial communication system – called the ASK system – that has an impressive size and is very data-intensive. Nevertheless, the case studies is a perfect example for our holistic approach. The entire ASK system model has an hierarchical structure that covers various layers of the hardware/software stack and the RSL code of the Vereofy model contains various Boolean flags that allow to compose the system model in different levels of detail. The work presented in this section has partially been published in [KKSB11].

5.3.5. The ASK communication system

ASK is an industrial software system for connecting people to each other. The system uses intelligent matching functionality in order to find effective connections between requesters and responders in a community. ASK has been developed by Almende[7], a Dutch research company focusing on the application of self-organization techniques in human organizations and agent-oriented software systems. The system is marketed by ASK Community Systems[8]. ASK provides mechanisms for matching users requiring information or services with potential suppliers. Based on information about earlier established contacts and feedback of users, the system learns to bring people into contact with each other in the most effective way. Typical applications for ASK are workforce planning, customer service, knowledge sharing, social care and emergency response. Customers of ASK include the European mail distribution company TNT Post, the cooperative financial services provider Rabobank and the world's largest pharmaceutical company Pfizer. The amount of people using a single ASK configuration varies from several hundreds to several thousands. Customers of ASK include TNT Post, Rabobank and Pfizer. The number of people using a single ASK system varies from several hundreds to several thousands.

System Architecture. The ASK system can be technically divided into three parts: the *web front-end*, the *database* and the *ASK core*. The *web front-end* acts as a configuration dashboard, via which typical domain data like users, groups, phone numbers, mail addresses, interactive voice response menus, services and scheduled jobs can be created, edited and deleted. This data is stored in a database, one for each deployment instance of the ASK system. The feedback of users and the knowledge derived from earlier established contacts are also stored in this database. Central to our case study, however, is the *ASK core* (see Figure 5.20), the "communication engine" of the system, which consists of the follow-

[7] `http://www.almende.com/`
[8] `http://www.ask-cs.com/`

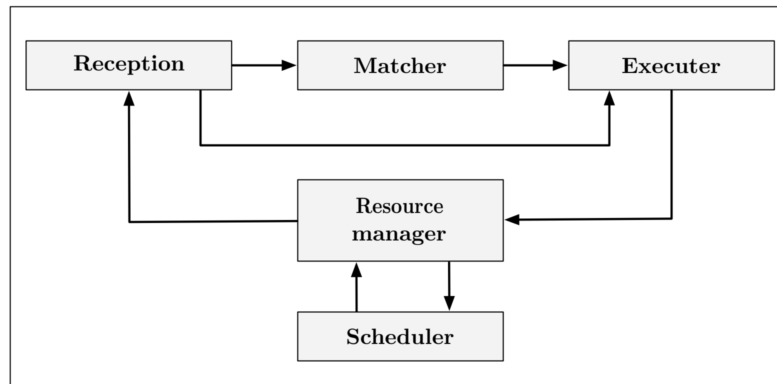


Figure 5.20: ASK core overview

ing components: *Reception*, *Matcher*, *Executer*, *Resource Manager* and *Scheduler*. These components handle all communication with and between users of ASK, provide matching functionality and schedule outbound communication and other kinds of jobs.

The "heartbeat" of the *ASK core* is the *Request loop*, indicated with thick arrows. Requests flow through the request loop until they are fully completed and removed. Requests are primarily initiated by the *Resource Manager* and normally removed by the *Executer*. The request loop itself is *medium and resource independent*. Within the *Resource Manager* component, the loop is separated from the level of media-specific resources needed for fulfilling the request. Telephony connections are handled inside the *Resource Manager* with the help of *Asterisk* (IAX), a telephony development tool kit. In particular, the individual components in the *ASK core* perform the following tasks: The *Reception* coordinates the communication process, i.e., the steps to be taken by ASK in order to fulfill a request, while the *Matcher* seeks appropriate context-based participants for a request. The *Executer* determines the best way in which the matching participants can be connected at the time of the request and the *Resource Manager* effectuates and maintains the actual connection between participants. Finally, the *Scheduler* schedules jobs, in particular requests for (outbound) communication between the ASK system and its users.

Vereofy model of the ASK system. The Vereofy model of the ASK system is hierarchically structured according to the levels shown in Figure 5.21. The top most level is called the *context level*. It contains the *ASK core*, which can be interpreted as an open system or as a closed system when connected to driver components modeling the database and the communication with the outside world. The *system level* is constituted by the *Reception*, *Matcher*, *Executer*, *Resource Manager*, and *Scheduler* processes. Each of the processes contains a workers pool and a task queue which constitute the *process level*. The workers in the original ASK system are called *Monks* and can apply for various types of tasks. The tasks constitute the *thread level*. Figure 5.22 illustrates the idea of refining the *ASK core* with the help of Reo into components and their orchestrating network. The components model is structured according to the request loop of the *ASK core* (cf. Figure 5.20), but with the Reo network in between the components the interaction becomes explicit. For each of

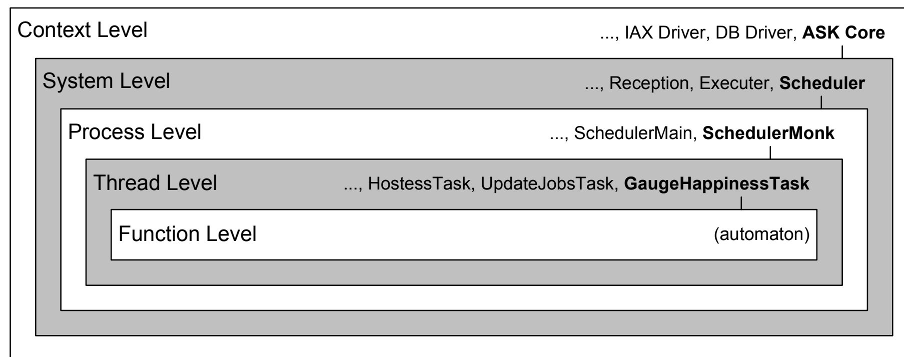


Figure 5.21: Abstraction levels in the ASK model and components contained in the level

the above levels RSL scripts are provided that refine into concepts of the subjacent level. At the bottom-most level, the *function level*, constraint automata provide the operational behavior for each of the tasks. For each of the levels or layers in this model appropriate methods are needed to translate the "implementation concepts" of the system (processes, threads, shared data structures, functions, function calls, etc.) into the "modeling concepts" of Reo (channels, connectors, and nodes) and constraint automata (states, transitions, and constraints). For this purpose, the data domain for the messages in the model is structured in such a way as to capture the essential information present in the implementation, e.g. function parameters or return values. Threads, tasks and shared variables are modeled as components with the appropriate connector that orchestrates the components, while the various queues in the ASK system are mapped to FIFO channels. More complex data structures, such as a hash table for mapping addresses are modeled as well by specifying their behavior as constraint automata. A detailed description of the model for ASK system is given in [KKSB11] and the complete and updated version of the CARML and RSL code together with example properties is available online on

`http://wwwtcs.inf.tu-dresden.de/~klueppel/PhD/ask_typed.zip`

This variant of the model is an updated version of the model presented in [KKSB11]. Instead of using one large data domain to encode all possible messages in a single tagged union, which consists of all possible message types plus an additional tag indicating the type, the latest version of the model is now fully typed. That is, all I/O-ports of components are associated with an individual type and hence allow sending and receiving messages of a certain data type only. In earlier versions this was modeled via additional checks in all the transition definitions prohibiting messages which were not tagged in the expected way. With the fully typed version of the model we expect more a compact BDD representation without changing the operational behavior.

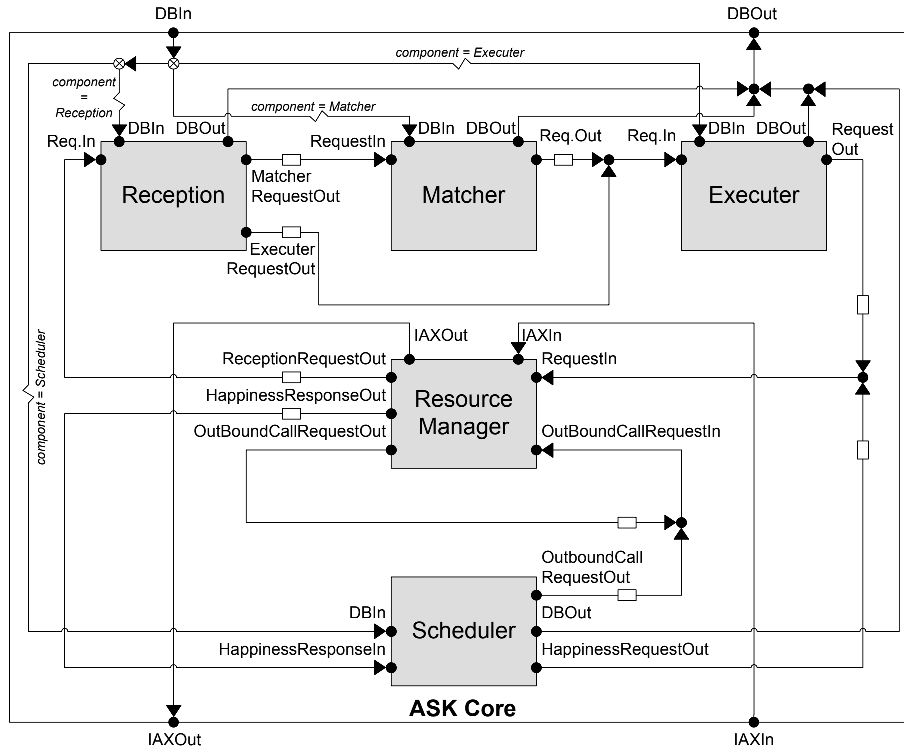


Figure 5.22: Reo network for the ASK core

Model statistics and verification. The ASK system is the largest case study we did with Vereofy. The entire model for the ASK system consists of more than 2000 lines of CARML and RSL code. At the same time it is the most complex one with respect to its data-intensity and the ASK system provides a challenging test case for the our tool set.

Overall, the latest version of the ASK model in its simplest version, where all parameters (e.g., the sizes of task queues and workers) have been set to small reasonable values, consists of 328 component instances and channels (with 63 different types) from which 73 constitute the *Scheduler*, 38 the *Matcher*, 40 the *Reception*, and 141 the *Resource Manager*. The remaining channels form the coordinating network. The symbolic representation of the ASK model at full detail uses 323 boolean variables to encode the state space of the composite system. On each of the levels (cf. Figure 5.21) various kinds of messages and data need to be handled, e.g., requests, tasks, telephone calls, or database communication messages – each of them with different parameters. There are 1007 distinct communication points in the model, i.e., interface ports or Reo nodes.

Model checking for the ASK system. Modeling the ASK system gives rise to a variety of important verification issues such as various liveness and safety properties ensuring the correct functioning of the individual system components in isolation, the coordination employed to model the real-world concepts in the Reo network, and the components acting together in concert [KKSB11]. The majority of the properties for verifying the correct

functioning of the modeling concepts on the various levels could be specified with BTSL and LTL formulas. On the context level there are also a few ASL properties of interest. For small number of tasks and a small capacity of the task queue we were able to build (and verify) the ASK system model in full detail, but only up to the system level. Due to the complexity of the model, a reasonable variant of the *ASK core* (i.e., the context level) could neither be built nor analyzed. With the model version presented in [KKSB11] we were able to build the *Scheduler* with two monks and a task queue capacity of 3. The structure of the Vereofy model for the *Scheduler* is shown in Figure 5.23. The largest components of the

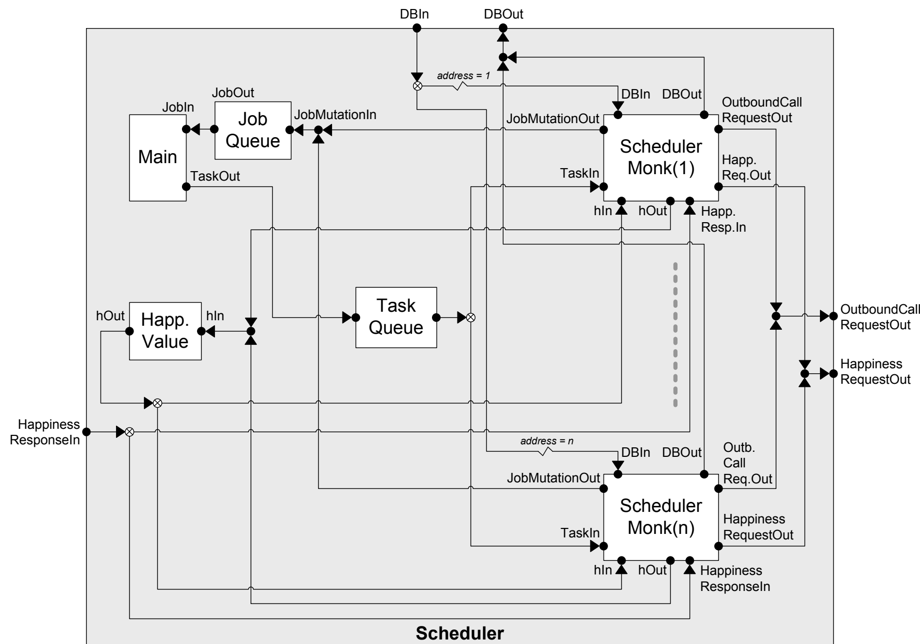


Figure 5.23: Reo network for the scheduler process

ASK system that we could build in full detail had about 10^{10} states. Although we could not compose the *Resource Manager* or the *ASK core* itself, the updated model variant with fully typed I/O-ports now allows to build and verify processes with more monks and larger task queues. Table 5.7 shows the times for composition and the reachable states for the scheduler process with different task queue capacities (tq1, . . ., tq12) and number of monks. Again, the variable ordering was computed and stored in advance.

One can see that the time to compose the *Scheduler* is not strictly increasing with a growing capacity of the task queue. It can be seen in Figure 5.24 that the time for the composition is correlated with the number of BDD nodes. Hence, we claim that better variable orderings exists for some input sizes and that there are more compact representations of the *Scheduler*.

	two monks		three monks	
	build time	**reach. states**	**build time**	**reach. states**
tq1	$5.3s$	$5.66 \cdot 10^{8}$	$8.6s$	$3.59 \cdot 10^{10}$
tq2	$5.9s$	$1.04 \cdot 10^{11}$	$10s$	$6.67 \cdot 10^{12}$
tq3	$8.9s$	$1.89 \cdot 10^{13}$	$30s$	$1.23 \cdot 10^{15}$
tq4	$8.9s$	$3.42 \cdot 10^{15}$	$31s$	$2.26 \cdot 10^{17}$
tq5	$12s$	$6.14 \cdot 10^{17}$	$34s$	$4.10 \cdot 10^{19}$
tq6	$14s$	$1.10 \cdot 10^{20}$	–	–
tq7	$14s$	$1.94 \cdot 10^{22}$	–	–
tq8	$13s$	$3.44 \cdot 10^{24}$	–	–
tq9	$20s$	$6.05 \cdot 10^{26}$	–	–
tq10	$31s$	$1.06 \cdot 10^{29}$	–	–
tq11	$20s$	$1.85 \cdot 10^{31}$	–	–

Table 5.7: Composition time and reachable states for the scheduler process

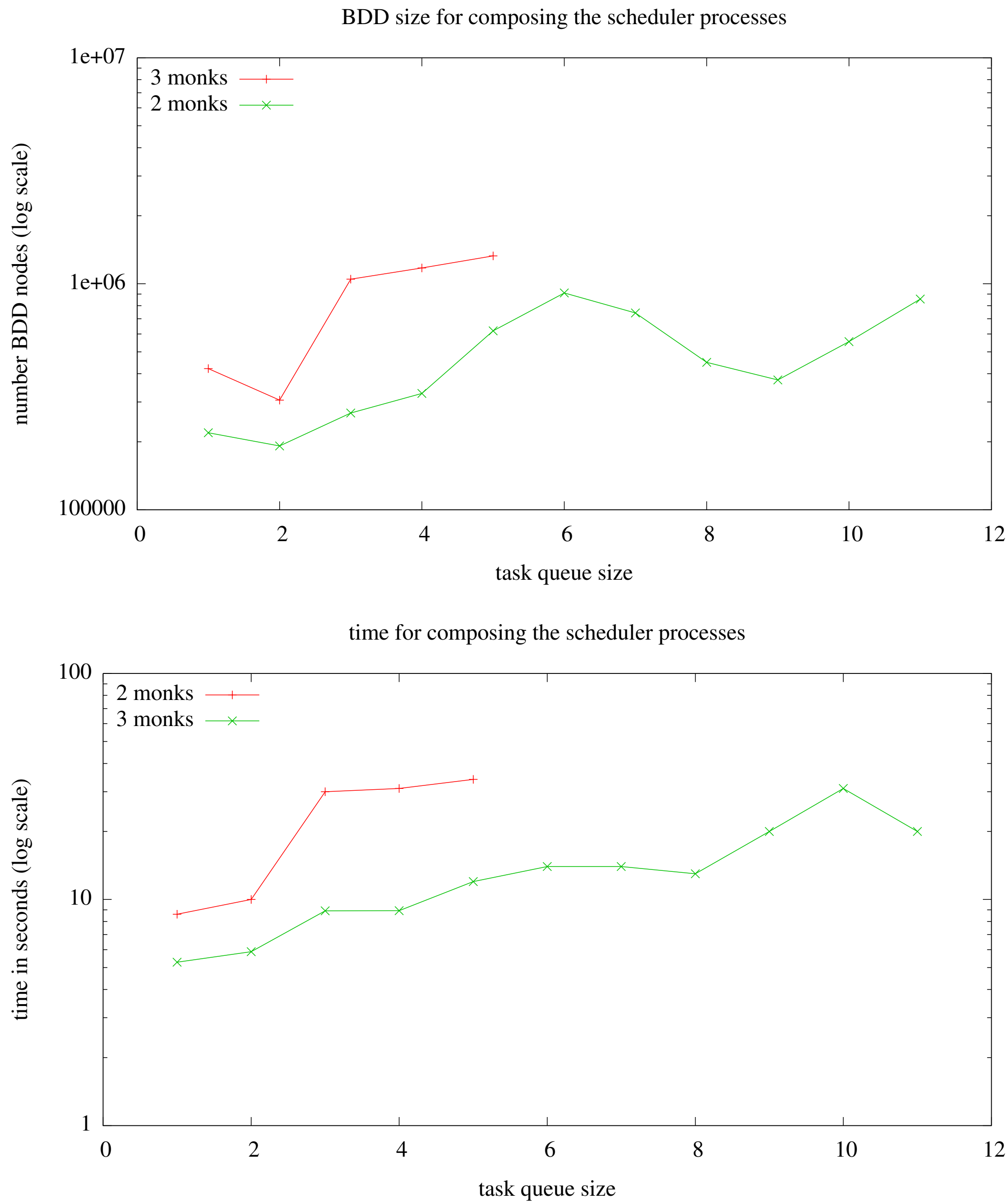


Figure 5.24: Time and memory usage for the scheduler process

The same effect becomes even more visible when looking at the *Reception* (and *Matcher*). Table 5.8 shows the composition time and number of reachable states of the *Reception* for a growing number of monks and increasing task queue capacities. In Figure 5.25 the time and memory usage for the *Reception* are plotted.

	two monks		**three monks**		**four monks**	
	build time	**reachable**	**build time**	**reachable**	**build time**	**reachable**
tq1	$0.4s$	$1.79 \cdot 10^{07}$	$0.7s$	$1.19 \cdot 10^{10}$	$1.9s$	$7.92 \cdot 10^{12}$
tq2	$0.4s$	$3.18 \cdot 10^{09}$	$0.8s$	$2.12 \cdot 10^{12}$	$2.8s$	$1.40 \cdot 10^{15}$
tq3	$0.5s$	$5.64 \cdot 10^{11}$	$1.3s$	$3.77 \cdot 10^{14}$	$3.0s$	$2.48 \cdot 10^{17}$
tq4	$0.7s$	$1.00 \cdot 10^{14}$	$3.0s$	$6.71 \cdot 10^{16}$	$8.8s$	$4.41 \cdot 10^{19}$
tq5	$2.0s$	$1.78 \cdot 10^{16}$	$8.2s$	$1.20 \cdot 10^{19}$	$18s$	$7.83 \cdot 10^{21}$
tq6	$2.3s$	$3.17 \cdot 10^{18}$	$29s$	$2.13 \cdot 10^{21}$	$47s$	$1.39 \cdot 10^{24}$
tq7	$6.0s$	$5.64 \cdot 10^{20}$	$45s$	$3.81 \cdot 10^{23}$	$283s$	$2.48 \cdot 10^{26}$
tq8	$5.8s$	$1.00 \cdot 10^{23}$	$37s$	$6.81 \cdot 10^{25}$	$194s$	$4.43 \cdot 10^{28}$
tq9	$1.3s$	$1.79 \cdot 10^{25}$	$41s$	$1.22 \cdot 10^{28}$	$8.5s$	$7.90 \cdot 10^{30}$
tq10	$2.2s$	$3.19 \cdot 10^{27}$	$59s$	$2.18 \cdot 10^{30}$	$158s$	$1.41 \cdot 10^{33}$
tq11	$1.6s$	$5.70 \cdot 10^{29}$	$47s$	$3.90 \cdot 10^{32}$	$6.9s$	$2.52 \cdot 10^{35}$
tq12	$2.7s$	$1.08 \cdot 10^{32}$	$1.9s$	$6.99 \cdot 10^{34}$	$4.8s$	$4.51 \cdot 10^{37}$

Table 5.8: Composition time and reachable states for the reception process

Summary. The case studies presented in this chapter show the applicability of our modeling and verification approach to a large class of system models including systems of industrial relevance. For systems which are regularly structured and are not data-intensive such as the dining philosophers protocol, our model checking is applicable and can treat system models with 10^{300} states and even beyond. For systems that are more data-intensive and more complex the approach can still be applied to system models with about 10^{30} and more states. With a growing complexity of the systems and data-intensive protocols the need for advanced abstraction techniques and heuristics grows as well.

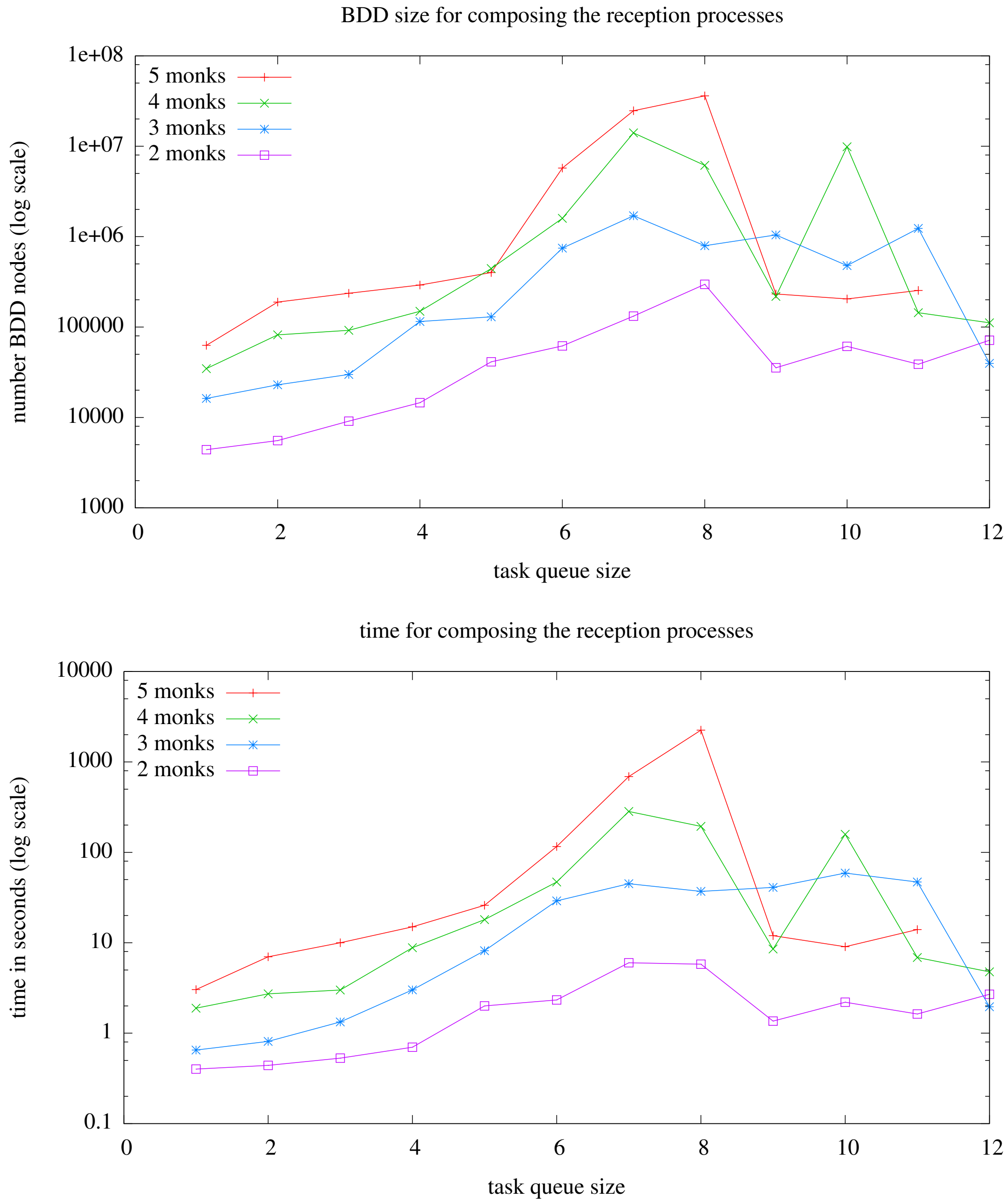


Figure 5.25: Time and memory usage for the reception process

6 | Conclusion

The goal of this thesis was to provide an approach for modeling, analysis, and synthesis which targets towards a closer integration in the system design phase of complex parallel (and distributed) hardware and software systems, as the proposed approach involves input languages that are expressive and simple, allow a clear separation of computation and coordination, support compositional and hierarchical modeling, allow specifying systems at any layer of the hardware/software stack, have clear formal semantics eligible to formal methods as well as temporal logics and model checking algorithms that treat exogenous coordination in component-based systems as a first class citizen, allow reasoning about aspects of coordination and dataflow, allow reasoning about alternating-time aspects such as controllability, are decidable within reasonable time and space bounds. For this, we decided to use an exogenous modeling approach for component-based system models which is inspired by the coordination language Reo [Arb04] and yields an elegant declarative framework for the compositional construction of component connectors by creating channels and glueing their channels ends, the I/O-ports of components, or sub-connectors together. It facilitates hierarchical modeling and reusability and exchangeability of components and the orchestrating network and does align perfectly in the refinement procedures known from software engineering.

CARML and RSL. For the modeling of exogenously coordinated component-based systems, we proposed a hybrid modeling framework based on two newly introduced input languages that allows for the compositional and hierarchical design and supports reusability of components and coordination units. The first is the guarded command language CARML which mainly serves to specify the I/O-ports of components and their stepwise "observable" behavior at their interface ports. The second specification language RSL is a scripting language which combines the major features of the exogenous coordination language Reo with concepts to specify connectors with dynamically changing network topologies and some features of other languages (such as data types). The two languages both rely on constraint automata as their operational semantic and indeed, RSL treats components, channels and component connectors in the same way. This means that a channel is viewed as a primitive component connector and any component connector can use components or other connectors as building blocks via the instantiation mechanism.

BTSL and ASL. For reasoning about the interface behavior of the components, the coordination mechanisms implemented in terms of components connectors and the overall behavior of the composite system we introduced the two temporal logics BTSL and ASL. The branching-time logic BTSL combines the standard CTL operators with an existential path modality $\langle \mathcal{Z} \rangle$ and its dual universal modality $[\![\mathcal{Z}]\!]$ that allow reasoning about the observable dataflow at any dataflow location (i.e., I/O-ports of the components and component connectors as well as at internal nodes of a Reo network), by means of a finite automaton $\mathcal{Z}$. The

alternating-time temporal logic ASL is based on a multi-player game interpretation of constraint automata. The logic ASL combines features of standard alternating-time logics such as ATL with the concept of having finite automata $\mathcal{Z}$ as in BTSL to reason about the observable dataflow. For existential quantification over N-strategies we provided the additional modality $\mathbb{E}_N$ whereas $\mathbb{A}_N$ serves as the dual universal modality operator. Combining the BTSL and the ASL modalities yields a rather powerful logic to specify classical temporal safety and liveness properties, to reason about dataflow and to express game-theoretic properties of the components viewed as players in the game structure of a constraint automaton. We proposed the model checking algorithms for ASL and BTSL formulas, investigated their complexity and showed that ASL as well as BTSL model checking is PSPACE complete.

Vereofy. We combined the prototype implementations of our very first BTSL model checker, an LTL model checker [BBKK09b] and a bisimulation checker [BB08] for the Reo and constraint automaton framework that have been developed in our group into a single tool set called Vereofy. The tool set uses BDDs to store the characteristic boolean functions for constraint automata that were specified with the help of CARML and RSL. Vereofy supports symbolic branching-time, linear-time and alternating-time model checking as well as bisimulation checking. The wide range of application areas of Reo (such as modeling of compliance-aware business processes [AKM08], long-run business transactions [KA09], orchestration of web services [DA04, LPA04, MA07, MA10], etc.) makes our modeling and verification approach with the tool set Vereofy applicable for many purposes. In this thesis we presented some of the experimental studies that we did with the Vereofy tool set that illustrate the applicability of the proposed approach to a wide area of systems. The class of the investigated system models reaches from very small toy examples and games to complex industrial case studies such as a sensor network [BBK+10] or a communication platform [KKSB11].

Future work. In future work we will try to eliminate some of the minor weaknesses and limitations of the current approach. For example, improving the modeling approach for dynamic system behavior to support more flexible reconfiguration and runtime adaptation operations. Furthermore we aim to have a more flexible notion of quiescence. In the current approach any component can at any time refuse interacting with its environment. In this context we require to have a finer notion where for example some components are declared to be “passive” and cannot refuse communication. Another way would be to declare some locations, i.e., local states of components, to be safe for stopping. Another natural improvement would be to weaken some of the assumptions made on the available knowledge that our N-strategies can rely on. The notion of strategies studied in this thesis relies on the assumption that the strategies have complete information on the system history. In future work we will switch to observation-based strategies [APR91, CDHR07, vdHRW05, vdHW03, Rei84, WDR06, Sch04b] and include knowledge and epistemic aspects [RS02, Jam03, vdHWJ05, RP80, Rei79]. These changes usually entail an increased complexity as solving the underlying games requires power-set constructions as in [Rei84].

The linear-time perspective. In future work we aim to have a closer integration of the linear-time, branching-time, and alternating-time model checking engines of Vereofy and study the relationship between compositional controller synthesis [BKK11a, Kle12] and the proposed alternating-time logic ASL. The latter is a very promising approach for addressing fairness issues in ASL model checking and providing more expressive logics and model checking algorithms, e.g., for ASL* where ASL is combined with the linear-time modalities.

Quantitative aspects and QoS. As QoS measures become an important aspect for dynamic reconfiguration, runtime adaptations and in the context of contract negotiation for non-functional properties [MS08], we also plan to provide extensions of our formal modeling and verification approach by quantitative aspects such as real-time, probabilities, and other quantitative measures for reasoning about QoS aspects and important system properties, e.g., energy, throughput, speed, reliability, etc. For some of these quantitative measures there exists already some work on extensions of Reo, e.g. for real-time [Kem06, Kem11] and probabilities [Bai05, ACvdM+09] and for others new operational model must be developed that potentially support combinations of the above measures. Hence, one important goal of future work is to extend CARML and RSL to allow modeling the quantitative behavior of channels, connectors, and components with respect to the QoS measures mentioned above. The main challenge here is to maintain the compositionality of the underlying quantitative automata models that must still permit the application of model checking techniques. For the verification we plan to use external tools like the real-time model checker UPPAAL [LPY97] and the probabilistic model checker PRISM [KNP04] which is based on MTBDDs (multi-terminal BDDs) or MRMC [KZH+09] – an explicit-state model checker. Where adequate, new temporal logics providing logical operators that fit with our modeling language and automata models and corresponding model checking algorithms need to be designed and integrated into our existing approach.

Parallel skeletons. We also plan to investigate the application spectrum of our framework in the area of algorithmic skeletons [Col89, Col88]. With algorithmic skeleton frameworks (ASKF) [GVL10] one can describe the high-level structure of a parallel program rather than the implementation code for a certain platform. There exist three classes of skeletons: *data-parallel skeletons* (e.g., fork, map, reduce etc.), *task-parallel skeletons* (e.g., sequential, if, for, while, etc.), and *resolution* (e.g., divide-and-conquer, branch-and-bound, dynamic-programming). The skeletons allow to compose programs from abstract building blocks that cover commonly used patterns of parallel (and sequential) computation, synchronous and asynchronous communication and interaction. Apart from a language independent formalization of programs that can be compiled and optimized for different (heterogenous and highly adaptable) hardware architectures before or in runtime, one of the major conceptional goals of using skeletons is to provide a clear separation of coordination and computation. As this is also the major motivation for applying exogenous coordination, we are going to study the applicability of the Reo and our modeling languages for ASKF and work on language extensions enabling to provide a collection of important and typical skeletons that can be bundled into a CARML / RSL-library available in Vereofy.

Complexity. With the growing complexity of the systems, their quantitative behavior in various dimensions and hence the system models the demand for advanced abstraction and reduction techniques that tackle the state explosion problem grows rapidly. Hence, new techniques have to be introduced and well known techniques such as partial order reduction [Val92, God96, Pel93],symmetry reduction [CEFJ96], slicing for concurrent programs [Wei84, Kri03], (learning-based) assume-guarantee reasoning [Pnu85, HQR98], iterative (counterexample-guided) abstraction-refinement algorithms [GS97, CGJ$^+$00] or SAT-based model checking [BCCZ99, WBCG00] have to be adapted and integrated into our modeling and verification framework.

Bibliography

[ABdB+05] F. Arbab, C. Baier, F. de Boer, J.J.M.M. Rutten, and M. Sirjani. Synthesis of Reo Circuits for Implementation of Component Connector Automata Specifications. In *Proceedings of the 7th International Conference on Coordination Models and Languages (COORDINATION'05)*, volume 3454 of *Lecture Notes in Computer Science*, pages 236–251. Springer, 2005.

[ABdBR04] F. Arbab, C. Baier, F. S. de Boer, and J.J.M.M. Rutten. Models and Temporal Logics for Timed Component Connectors. *Proceedings 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)*, 0:198–207, 2004.

[ACvdM+09] F. Arbab, T. Chothia, R. van der Mei, S. Meng, Y.J. Moon, and C. Verhoef. From Coordination to Stochastic Models of QoS. In *Proceedings of the 11th International Conference on Coordination Models and Languages (COORDINATION'09)*, volume 5521 of *Lecture Notes in Computer Science*, pages 268–287. Springer, 2009.

[AH99] R. Alur and T.A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AHK02] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[AHM+98] R. Alur, T.A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 1998.

[AK86] K.R. Apt and D. Kozen. Limits for the automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[AKM08] F. Arbab, N. Kokash, and S. Meng. Towards Using Reo for Compliance-Aware Business Process Modeling. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08)*, volume 17 of *Communications in Computer and Information Science*, pages 108–123. Springer, 2008.

[ALSN01] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. *PICCOLA—a small composition language, Formal methods for distributed processing*, pages 403–426. Cambridge University Press, New York, NY, USA, 2001.

[APR91] S. Azhar, G. Peterson, and J. Reif. On Multiplayer Non-Cooperative Games of Incomplete Information: Part 1 & 2. Technical report, Durham, NC, USA, 1991.

[AR02] F. Arbab and J.J.M.M. Rutten. A Coinductive Calculus of Component Connectors. In *Proceedings of the International Workshop on Algebraic Development Techniques (WADT'02)*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer, 2002.

[Arb96] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In *Proceedings of the 1st International Conference on Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer, 1996.

[Arb04] F. Arbab. Reo: A Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[Arb05] F. Arbab. Abstract Behavior Types: a foundation model for components and their composition. *Science of Computer Programming*, 55(13):3–52, 2005.

[Arn94] A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.

[Bai05] C. Baier. Probabilistic Models for Reo Connector Circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BB08] T. Blechmann and C. Baier. Checking Equivalence for Reo Networks. *Electronic Notes in Theoretical Computer Science*, 215:209–226, 2008.

[BBF+01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[BBK+10] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, and W. Leister. Design and Verification of Systems with Exogenous Coordination using Vereofy. In *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)*, volume 6416 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2010.

[BBKK09a] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. A Uniform Framework for Modeling and Verifying Components and Connectors. In *Proceedings of the 11th International Conference on Coordination Models and Languages (COORDINATION'09)*, volume 5521 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2009.

[BBKK09b] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. Formal Verification for Components and Connectors. In *Proceedings of the 7th International Symposium on Formal Methods for Components and Objects (FMCO'08)*, volume 5751 of *Lecture Notes in Computer Science*, pages 82–101. Springer, 2009.

[BC11] S. Borkar and A.A. Chien. The future of microprocessors. *Communications of the ACM*, 54:67–77, May 2011.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer.

[BCM$^+$92] J. Burch, E. M. Clarke, K.L. McMillan, D.L. Dill, and L. Hwang. Symbolic Model Checking 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BK08] C. Baier and J.P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BKK11a] C. Baier, J. Klein, and S. Klüppelholz. A Compositional Framework for Controller Synthesis. In *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR'11)*, volume 6901 of *Lecture Notes in Computer Science*, pages 512–527. Springer, 2011.

[BKK11b] C. Baier, J. Klein, and S. Klüppelholz. Modeling and Verification of Components and Connectors. In *Proceedings of the 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM'11)*, volume 6659 of *Lecture Notes in Computer Science*, pages 114–147. Springer, 2011.

[BKK11c] C. Baier, J. Klein, and S. Klüppelholz. Synthesis of Reo Connectors for Strategies and Controllers. In *Proceedings of the International Workshop on Logic, Agents, and Mobility (LAM'11)*, 2011.

[BKK12] T. Blechmann, J. Klein, and S. Klüppelholz. *Vereofy User Manual*. Technische Universität Dresden, 2008–2012. http://www.vereofy.de.

[BL69] J.R. Büchi and L. Landweber. Solving Sequential Conditions by Finite-State Strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[BPe01] J.A. Bergstra, A. Ponse, and S.A. Smolka (editors). *Handbook of Process Algebra*. Elsevier Publishers B.V., 2001.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[Brz62] J.A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561, Polytechnic Institute of Brooklyn, 1962. Polytechnic Press.

[BSAR06] C. Baier, M. Sirjani, F. Arbab, and J.J.M.M. Rutten. Modeling Component Connectors in Reo by Constraint Automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[CBM90] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, Lecture Notes in Computer Science, pages 365–373, New York, NY, USA, 1990. Springer.

[CCA04] D. Clarke, D. Costa, and F. Arbab. Modelling Coordination in Biological Systems. In *Proceedings of the 1st International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'04)*, volume 4313 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 2004.

[CCA07] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007. Special Issue on the 4th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA '05).

[CCF$^+$10] Y.-F. Chen, E.M. Clarke, A. Farzan, F. He, M.-H. Tsai, Y.K. Tsay, B.-Y. Wang, and L. Zhu. Comparing learning algorithms in automated assume-guarantee reasoning. In *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)*, volume 6416 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2010.

[CCGR00] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CDHR07] K. Chatterjee, L. Doyen, T.A. Henzinger, and J.F. Raskin. Algorithms for Omega-Regular Games with Imperfect Information. *Logical Methods in Computer Science*, 3(3), 2007.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1996.

[CES86] E.M. Clarke, E.A. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Cha84] D.M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, CA., 1984.

[CK96] E.M. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[Col88] M. Cole. *Algorithmic skeletons: a structured approach to the management of parallel computation*. PhD thesis, University of Edinburgh, Computer Science Department, 1988.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.

[Cos10] D. Costa. *Formal Models for Context Dependent Connectors*. Phd thesis, Theoretical Computer Science, Free University of Amsterdam, The Netherlands, 2010.

[CPLA11] D. Clarke, J. Proença, A. Lazovik, and F. Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76(8):681–710, 2011.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In *International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer, 1989.

[CS00] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (Volume II)*, chapter 24, pages 1635–1790. Elsevier Publishers B.V., 2000.

[CSV09] R. Chadha, A.P. Sistla, and M. Viswanathan. On the expressiveness and complexity of randomization in finite state monitors. *Journal of the ACM*, 56(5):1–44, 2009.

[CSZ04] S. Capizzi, R. Solmi, and G. Zavattaro. From Endogenous to Exogenous Coordination Using Aspect-Oriented Programming. In *Proceedings of the 6th International Conference on Coordination Models and Languages (Coordination'04)*, volume 2949 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 2004.

[CW96] E.M. Clarke and J. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DA04] N.K. Diakov and F. Arbab. Compositional Construction of Web Services Using Reo. In *Proceedings of the 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure*, WSMAI'04, pages 49–58. INSTICC Press, 2004.

[DB98] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation*. Kluwer Academic Publishers, 1998.

[DG95] D. Dams and J.F. Groote. Specification and Implementation of Components of a μCRL Toolbox. Logic Group Preprint Series 152. Technical report, 1995.

[Dij65] E.W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[Dij68] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[FL79] M.J. Fischer and R.J. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 8:194–211, 1979.

[Fra86] N. Francez. *Fairness*. Springer, 1986.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel Programming in Linda. In *Proceedings of the International Conference on Parallel Processing (ICPP'85)*, pages 255–263, 1985.

[GJA+10] I. Grabe, M. M. Jaghoori, B. Aichernig, C. Baier, T. Blechmann, F. de Boer, A. Griesmayer, E.B. Johnsen, J.Klein, S. Klüppelholz, M. Kyas, W. Leister, R. Schlatte, A. Stam, M. Steffen, S. Tschirner, X. Liang, and W. Yi. Credo Methodology - Extended Version. In *Proceedings of the 8th International Symposium on Formal Methods for Components and Objects (FMCO'09)*, volume 6286 of *Lecture Notes in Computer Science*, pages 41–69. Springer, 2010.

[GLMS11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In P. Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.

[God96] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[GS97] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, 1997. Springer.

[GS03] G. Gößler and J. Sifakis. Component-Based Construction of Deadlock-Free Systems: Extended Abstract. In *Proceedings of the Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, pages 420–433, 2003.

[GS05] G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[GS07] J. Guillen-Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Phd thesis, University of Leiden, 2007.

[GSAdBB05] J. Guillen-Scholten, F. Arbab, F.S. de Boer, and M.M. Bonsangue. MoCha-pi: an exogenous coordination calculus based on mobile channels. In *Proceedings of the ACM symposium on Applied computing (SAC'05)*, pages 436–442. ACM, 2005.

[GVL10] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[Hal02] A. Hall. Correctness by Construction: Integrating Formality into a Commercial Development Process. In *Proceeding of Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe (FME)*, volume 2391 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2002.

[Hal05] A. Hall. Realising the Benefits of Formal Methods. In *Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM'05), Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2005.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HC02] A. Hall and R. Chapman. Correctness by construction: developing a commercial secure system. *Software, IEEE*, 19(1):18–25, 2002.

[HMU06] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol90] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1990.

[Hol93] G.J. Holzmann. Design and Validation of Protocols: A Tutorial. *Comp. Networks and ISDN Systems*, 25(9):981–1017, 1993.

[Hol95] M. Holzer. On Emptiness and Counting for Alternating Finite Automata. In *Developments in Language Theory*, pages 88–97. World Scientific Verlag, 1995.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hol03] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[Hop71] J.E. Hopcroft. *An n log n algorithm for minimizing states in a finite automaton*. Stanford University, Stanford, CA, USA, 1971.

[HQR98] T.A. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 1998.

[HR99] M. Huth and M.D. Ryan. *Logic in Computer Science – Modelling and Reasoning about Systems*. Cambridge University Press, 1999.

[HS96] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, 1996.

[JA11] S.-S. T. Q. Jongmans and F. Arbab. Correlating Formal Semantic Models of Reo Connectors: Connector Coloring and Constraint Automata. In *Proceedings of the 4th Interaction and Concurrency Experience (ICE'11)*, volume 59 of *Electronic Proceedings in Theoretical Computer Science*, pages 84–103, 2011.

[Jam03] W. Jamroga. Some Remarks on Alternating Temporal Epistemic Logic. In *Proceedings of the Workshop on Formal Approaches to Multi-Agent Systems*, pages 133–140, 2003.

[JS07] P. Jancar and Z. Sawa. A note on emptiness for alternating finite automata with a one-letter alphabet. *Information Processing Letters*, 104(5):164–167, 2007.

[KA09] N. Kokash and F. Arbab. Applying Reo to service coordination in long-running business transactions. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1381–1382. ACM, 2009.

[KB09] S. Klüppelholz and C. Baier. Symbolic model checking for channel-based component connectors. *Science of Computer Programming*, 74(9):688–701, 2009. Special Issue on the Fifth International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06).

[KB10] S. Klüppelholz and C. Baier. Alternating-time stream logic for multi-agent systems. *Science of Computer Programming*, 75(6):398–425, 2010. 10th International Conference on Coordination Models and Languages (COORD'08).

[Kel76] R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[Kem06] S. Kemper. SAT-based Verification for Abstraction Refinement. Master thesis, University of Oldenburg, January 2006.

[Kem11] S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), 2011.

[KKdV12] N. Kokash, C. Krause, and E.P. de Vink. Reo + mCRL2: A Framework for Model-Checking Dataflow in Service Compositions. *Formal Aspects of Computing*, 24(2):187–216, 2012.

[KKSB11] J. Klein, S. Klüppelholz, A. Stam, and C. Baier. Hierarchical modeling and formal verification. An industrial case study using Reo and Vereofy. In *Formal Methods for Industrial Critical Systems (FMICS 2011)*, volume 6959 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2011.

[Kle12] J. Klein. *Compositional Controller Synthesis*. Phd thesis, Technische Universität Dresden, forthcoming 2012.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KNP04] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[Kra11] C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, The Netherlands, 2011.

[Kri03] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 28 of *ESEC/FSE-11*, pages 178–187, New York, NY, USA, September 2003. ACM.

[Kro99] T. Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999.

[Kur94] R. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[Kwi89] M. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.

[KZH+09] J.-P. Katoen, I. Zapreev, E.M. Hahn, H. Hermanns, and D.N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. In *Proceedings*

of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems, QEST'09, pages 167–176, Washington, DC, USA, 2009. IEEE Computer Society.

[LPA04] T. Lemniotes, G.A. Papadopoulos, and F. Arbab. Coordinating Web Services Using Channel Based Communication. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC'04, pages 486–491. IEEE Computer Society, 2004.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: the ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 1981.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[MA07] S. Meng and F. Arbab. Web services choreography and orchestration in Reo and constraint automata. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC'07, pages 346–353. ACM, 2007.

[MA10] S. Meng and F. Arbab. A Model for Web Service Coordination in Long-Running Transactions. In *Proceedings of the 5th IEEE International Symposium on Service-Oriented System Engineering*, SOSE'10, pages 121–128, 2010.

[Mar75] D.A. Martin. Borel Determinacy. *Annals of Mathematics*, 102:363–371, 1975.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MCM08] M. Majster-Cederbaum and C. Minnameier. Everything Is PSPACE-Complete in Interaction Systems. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC'08)*, volume 5160 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 2008.

[Mer01] S. Merz. Model checking: a tutorial. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modelling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer, 2001.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[Min96] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–40, 1992.

[MS08] M. Mulugeta and A. Schill. Balancing Agility and Formalism in Software Engineering. volume 5082, chapter A Framework for QoS Contract Negotiation in Component-Based Applications, pages 238–251. Springer, 2008.

[MT98] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

[Oss10] J. Ossowski. *JINC - A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation*. Phd thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2010.

[Pel93] D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[Pet77] C.A. Petri. Non-sequential processes. GMD-ISF Report ISF-77-05, GMD, Sankt Augustin, 1977.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer, New York, NY, USA, 1985.

[PS95] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, ICCAD '95, pages 74–77, Washington, DC, USA, 1995. IEEE Computer Society.

[QS82] J. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[QS83] J.-P. Queille and J. Sifakis. Fairness and related properties in transition systems. A temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, 1983.

[Rei79] J.H. Reif. Universal Games of Incomplete Information. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 288–308, 1979.

[Rei84] J.H. Reif. The Complexity of Two-Player Games of Incomplete Information. *Journal of Computer and System Sciences*, 29(2):274–301, October 1984.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., 1999.

[RP80] J.H. Reif and G.L. Peterson. A Dynamic Logic of Multiprocessing with Incomplete Information. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 193–202, 1980.

[RS02] M. Ryan and P.-Y. Schobbens. Agents and Roles: Refinement in Alternating-Time Temporal Logic. In *Proceedings of the 8th International Workshop on Intelligent Agents*, volume 2333 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 2002.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[RvdHW09] J. Ruan, W. van der Hoek, and M. Wooldridge. Verification of Games in the Game Description Language. *Journal of Logic and Computation*, 19:1127–1156, 2009.

[Sch04a] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.

[Sch04b] P.-Y. Schobbens. Alternating-time logic with imperfect recall. *Electronic Notes in Theoretical Computer Science*, 85(2):82–93, 2004.

[SLN01] J.G. Schneider, M. Lumpe, and O. Nierstrasz. Agent coordination via scripting languages. pages 153–175. Springer, London, UK, 2001.

[Som99] F. Somenzi. Binary decision diagrams. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[Ste97] P. Stevens. *The Edinburgh Concurrency Workbench (Version 7.1). User manual*, 1997.

[Val92] A. Valmari. A Stubborn Attack on State Explosion. *Formal Methods in System Design*, 1:297–322, 1992.

[Var96] M. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In *Proceedings of the 8th Banff Workshop on Higher Order: Logics for Concurrency - Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996.

[vdHRW05] W. van der Hoek, M. Roberts, and M. Wooldridge. Knowledge and social laws. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 674–681. ACM, 2005.

[vdHW03] W. van der Hoek and M. Wooldridge. Cooperation, Knowledge, and Time: Alternating-time Temporal Epistemic Logic and its Applications. volume 75, pages 125–157. Springer, 2003.

[vdHWJ05] W. van der Hoek, M. Wooldridge, and W. Jamroga. A Logic for Strategic Reasoning. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 157–164. ACM, 2005.

[vdP01] J.C. van de Pol. A Prover for the μCRL-Toolset with Applications: Version 0.1. CWI Report SEN-R 0106, Centrum Wiskunde & Informatica (CWI) Amsterdam, 2001.

[VVK05] H. Völzer, D. Varacca, and E. Kindler. Defining fairness. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science*, pages 458–472. Springer, 2005.

[VW86] M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the Symposium on Logic in Computer Science (LICS'86)*, pages 332–345, Washington, D.C., USA, 1986. IEEE Computer Society Press.

[WBCG00] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *CAV '00*, pages 124–138, London, UK, 2000. Springer.

[WDR06] M. Wulf, L. Doyen, and J.F. Raskin. A Lattice Theory for Solving Games of Imperfect Information. In *Proceedings of HSCC 2006: Hybrid Systems—Computation and Control*, volume 3927 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2006.

[Weg00] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Monographs on Discrete Mathematics and Applications. SIAM, 2000.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Wol95] P. Wolper. An introduction to model checking. Position statement for panel discussion at the Software Quality workshop, 1995.

[Wou01] A.G. Wouters. Manual for the μCRL Tool Set (Version 2.8.2). CWI Report SEN-R 0130, Centrum Wiskunde & Informatica (CWI) Amsterdam, 2001.