Symbolic Controller Synthesis for LTL Specifications

Andreas Morgenstern

February 2010

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr. Ing.)

genehmigte Dissertation

Datum der wissenschaftlichen Aussprache:	12.02.2010
Erster Berichterstatter:	Prof. Dr. Klaus Schneider
Zweiter Berichterstatter:	Prof. Dr. Roderick Bloem

D 386

Abstract

It is an old dream in computer science to *automatically* generate a system from a formal specification or at least to *automatically check* whether a system is guaranteed to satisfy a specification.

The second problem is known as the verification problem and powerful tools exist that automatically check the correctness of a system with respect to a given declarative specification.

In this thesis we consider the first problem with respect to a given declarative specification in linear temporal logic (LTL). We refer to this problem as the controller synthesis problem why others prefer to use names like 'realizability' or 'supervisor synthesis'. The controller synthesis problem is to check whether an (incomplete) implementation of a system can be refined by a controller such that a given property holds, and if so, to automatically construct this controller. Although the idea to automatically synthesize an implementation from a formal declaration is nearly 50 years old, it has not yet made its way to practice.

A major breakthrough in verification has been achieved by considering symbolic representations of states and transitions by propositional formulas which lead to the invention of symbolic model checking. With the advent of succinct data structures and efficient decision procedures for propositional formulas, which are the heart of almost all approaches to hardware verification, it has become possible to verify complex systems.

The currently available procedures to the controller synthesis problem consist of two steps: Similar to verification, the first step consists of translating the LTL formula to an equivalent nondeterministic ω-automaton. While this automaton can be directly used for symbolic model checking, only deterministic automata (or pseudo-deterministic automata like the good-for-games automata) can be used for constructing a controller.

Hence, the second step involves a determinization of the obtained nondeterministic automaton, which is remarkably difficult for those kinds of ω-automata that are required in the general case. A major drawback of the currently known determinization procedures is their explicit representation of the automata. And since a translation from LTL to deterministic automata may lead to a state space of doubly exponential size in the length of the formula, these approaches are limited to very small LTL formulas with only few temporal operators.

Since symbolic methods have been the breakthrough in model checking, the research studied in this thesis focuses on approaches that can be implemented symbolically, e. g. by BDDs.

On the one hand, we concentrate on determinization procedures that are amenable to a symbolic implementation. We present two different determinization procedures. The first determinization construction determines the location of a LTL formula in the temporal logic hierarchy in order to avoid the considerably difficult determinization constructions needed in the general case. Our procedure uses instead a symbolical implementation of the Rabin-Scott subset construction or the Breakpoint construction to efficiently generate deterministic automata.

It is well-known that the automata that are generated from a LTL formula have special properties. Our second determinization construction makes use of such a special property in order to develop a symbolic determinization construction for full LTL.

Another important ingredient to bring controller synthesis from theory closer to practice is the use of minimization techniques. Here we concentrate on simulation based minimization techniques, since related techniques have been successfully used to minimize automata for the closely related model-checking problem. In particular, we consider the minimization problem of parity automata and give solutions based on direct, reverse and fair simulation.

Danksagung

Mein Dank gilt allen, die auf ihre Weise zum Gelingen meiner Promotion beigetragen haben. Im Besonderen möchte ich mich bei Prof. Dr. Klaus Schneider bedanken, der mir die Möglichkeit eröffnet hat, diese Arbeit zu erstellen. Ohne seine Ratschläge und seine Unterstützung wäre diese Arbeit niemals entstanden. Für die Zweitbegutachtung möchte ich mich bei Prof. Dr. Roderick Bloem bedanken, der durch hilfreiche Kommentare sehr zur Verbesserung der Arbeit beigetragen hat. Mein weiterer Dank geht an Prof. Dr. Markus Nebel, der sich als Vorsitzender der Promotionskommision zur Verfügung gestellt hat.

Des Weiteren möchte ich mich bei meinen Kollegen in unserer Arbeitsgruppe bedanken, die mir in vielen Gesprächen neue Denkanstöße geliefert haben.

Abschließend möchte ich mich bei meiner Familie für ihre langjährige Unterstützung danken. Ohne die Hilfe meiner Eltern, meiner Frau Corinna und unseres Sohnes Lukas hätte ich diese Arbeit wohl nie fertiggestellt.

Contents

List of Figures

1 Introduction

1.1 Motivation

In the last decades, the influence of computer systems on our everyday life has been constantly growing. As computer systems enter more and more safety critical areas their correctness is of urgent importance. Malfunctioning systems may lead to loss of life if those systems are used in transportation or military systems like the control of an aircraft or the controller of a cruise missile. Malfunctioning systems may also lead to loss of money. Consider for example the Ariane 5 incident caused by a computer error or the Pentium bug that made it necessary to exchange millions of malfunctioning processors. Another important area where computer systems play a vital role is the international financial market. Errors in computer systems may lead to the loss of billions of dollars.

Thus, one of the main challenges in computer science is the design of provably correct systems. Most of these safety critical computer systems and protocols are reactive in nature; i. e. systems of non-terminating processes that interact with each other over an infinite run. Parallelism and an incomplete view of the processes with respect to the environment make it difficult to analyze and design such systems correctly.

There are currently two main approaches to the design of systems that are guaranteed to satisfy a specification. The first approach is verification that checks that a manually written implementation satisfies a specification. The second approach that is followed by this thesis is to automatically synthesize a correct implementation from a given declarative specification. We refer to this problem as the controller synthesis problem why others prefer to use names like 'realizability' or 'supervisor synthesis'. In comparison to verification, the controller synthesis problem can be formulated as follows: The verification problem is to check for a given implementation $\mathcal{M}$ and a given LTL property φ, whether $\mathcal{M}$ satisfies φ in any environment, which is usually written as $\mathcal{M} \models \varphi$. The more general synthesis problem is to check whether an (incomplete) implementation[1] $\mathcal{M}$ can be restricted by a controller $\mathcal{C}$ such that a given property φ holds, i.e., whether there exists a system $\mathcal{C}$ such that $(\mathcal{M} \parallel \mathcal{C}) \models \varphi$ holds. This means that the combined behavior of $\mathcal{C}$ and $\mathcal{M}$ has to satisfy φ in *any* environment.

For the specification of the temporal behavior of reactive systems, different kinds

[1]$\mathcal{M}$ may also be only the model of the environment.

of specification logics including μ-calculus, monadic second order logics, ω-automata and temporal logics have been considered (see [91] for a recent overview) and powerful verification procedures are already available. In particular, model checking of linear time temporal logic LTL [80, 24] became one of the most comfortable standard verification techniques. To check whether a system $\mathcal{M}$ satisfies a LTL property φ, the negation $\neg\varphi$ is usually first translated to an equivalent *nondeterministic* ω-automaton $\mathfrak{A}_{\neg\varphi}$ so that the emptiness of the product $\mathcal{M} \times \mathfrak{A}_{\neg\varphi}$ can be checked in a second step. Algorithms that translate the LTL formulas to *symbolically*[2] *represented nondeterministic* ω-automata have been developed [106, 21, 53, 90, 91, 10] to benefit from symbolic set representations [13]. As the use of symbolic methods in verification was the major breakthrough to handle real-world problems, the computation of symbolic descriptions of the automata is, from a practical point of view, very important to deal with large state spaces.

In general, there are two approaches to the synthesis problem of LTL specifications: The first approach is based on a reduction to the emptiness problem of ω-tree automata [81], and the second approach is based on a reduction to the solution of ω-regular games [16, 37] . While in principle the two approaches are equivalent, there are subtle differences.

In this thesis, we follow the second approach: the controller synthesis game is played by two players, the controller $\mathcal{C}$ and its environment $\mathcal{E}$. While the environment $\mathcal{E}$ tries to violate the specification φ, the controller $\mathcal{C}$ tries to satisfy it. The controller $\mathcal{C}$ wins if it manages to determine the systems state such that irrespectively of the actions chosen by $\mathcal{E}$, the combined behavior $\mathcal{E} \parallel \mathcal{M} \parallel \mathcal{C}$ satisfies the specification φ.

In contrast to the verification problem, the solution of games with LTL winning conditions is so far not well supported by tools. The currently available procedures consist of two steps: Similar to verification, the first step consists of translating the LTL formula φ to an equivalent nondeterministic ω-automaton $\mathfrak{A}_{\varphi}$. While this automaton $\mathfrak{A}_{\varphi}$ can be directly used for symbolic model checking, only deterministic automata (or pseudo-deterministic automata like the good-for-games automata [46]) can be used for constructing a ω-regular game from the obtained automaton $\mathfrak{A}_{\varphi}$ and the incomplete implementation $\mathcal{M}$.

Hence, the second step involves a determinization of $\mathfrak{A}_{\varphi}$, which is remarkably difficult for those kinds of ω-automata [102, 91] that are required in the general case. In particular, Safra's construction [88] is often used which (in the worst case) generates for a Büchi automaton with n states an equivalent deterministic Rabin automaton with $12^n \cdot n^{2n}$ states and n acceptance pairs. However, Safra's construction is ex-

[2]For an LTL formula φ, these procedures compute in time $O(|\varphi|)$ a symbolic description of a nondeterministic ω-automaton $\mathfrak{A}_{\varphi}$. The symbolic description of the automaton $\mathfrak{A}_{\varphi}$ has size $O(|\varphi|)$ and encodes $O(2^{|\varphi|})$ states. Symbolically represented nondeterministic ω-automata are related to alternating ω-automata [105] (but are not the same).

tremely difficult to implement [54] and not amenable to a symbolic implementation and therefore, no powerful tools exist. In particular, a major drawback of the proposed implementations of Safra's construction is that they use an explicit representation of the automata, since the states of Safra's automaton consist of trees whose nodes are labeled with sets of states. As a consequence, they are limited to very small LTL formulas with only few temporal operators.

Since symbolic methods have been the breakthrough in model checking, the research studied in this thesis focuses on approaches that can be implemented symbolically, e. g. with BDDs. On the one hand, we concentrate on determinization procedures that are amenable to a symbolic implementation. Another important ingredient to bring controller synthesis from theory closer to practice is the use of minimization techniques. Here we concentrate on simulation based minimization techniques, since related techniques have been successfully used to minimize automata for the closely related model-checking problem.

1.2 Related Work

The history of controller synthesis started with a seminal paper of Church [20] in 1962 that considered the synthesis problem regarding specifications written in monadic second order theory of one successor (S1S). Given a relation $\mathcal{R} \subseteq (2^I)^\omega \times (2^O)^\omega$ that is represented by a formula in S1S, Church's problem is to find a strategy $\gamma : (2^I)^\star \to 2^O$ that generates for every finite sequence $i = i^{(0)}, i^{(1)} \cdots \in (2^I)^\star$ of input signals read so far a response in 2^O such that the specification is satisfied. Hence, the strategy generates a sequence of output signals $o = o^{(0)} o^{(1)} \cdots \in (2^O)^\omega$ with $o^{(t)} = \gamma(i^{(0)} i^{(1)} \dots i^{(t-1)})$ for every t that satisfies the specification, i. e. $(i, o) \in \mathcal{R}$ holds. Church's problem was solved independently by Rabin [82] using tree automata and from Büchi and Landweber [16] using infinite games. Since those seminal works, the close relation between finite automata over infinite objects and finite games of infinite length became apparent [39], and both areas have often inspired each other.

In the meantime, the verification community developed easy to grasp specification logics like the linear temporal logic LTL [80] and the branching time logic CTL [24]. It is a long standing debate which of the two to favor [107]. While the model-checking problem of CTL can be solved in linear time, its restricted syntax makes writing specifications in CTL rather complicated. Specifying is easier in linear temporal logic LTL with the drawback that the model-checking problem for LTL rises from linear time to PSPACE [96]. Despite this heavy theoretical upper bound, model checking of LTL has become more successful than CTL model checking.

Inspired by those developments of new specification languages, progress was also made in the context of synthesis. *Closed synthesis* is the case of a single process without any interaction with the environment. It was solved in the early 80's for CTL [25], LTL [112] and for the modal μ-calculus [56] by a reduction to the satisfiability problem of the corresponding logic. *Open synthesis* concerns systems that interact with their environment and is thus suitable to represent the synthesis of reactive systems. In the late 80's, Pnueli and Rosner [81] provided a solution to the open synthesis problem for linear temporal logic LTL by a reduction to the emptiness problem of Rabin tree automata. Furthermore, Rosner proved the problem to be 2EXPTIME-complete [86]. The first exponent is due to the translation of LTL to nondeterministic Büchi automata and the second due to the needed determinization step.

At the same time, Ramadge and Wonham [85] introduced the problem of *controller synthesis* which deals with the construction of a *controller* for a *plant*. They considered simple safety specifications of the form $\mathsf{G}\varphi$ and showed that the controller synthesis problem of this restricted class is solvable in linear time.

One possibility to overcome the complexity issues of LTL synthesis is to consider restricted classes of LTL. E.g. [63, 3] consider restricted classes of LTL to obtain deterministic automata with less than double-exponential size. Wallmeier et al. [110] developed a synthesis algorithm to synthesize request-response specifications which are of the form $\mathsf{G}(\varphi_i \rightarrow \mathsf{F}\psi_i)$ for multiple i which leads to a synthesis procedure with only exponential complexity. Piterman et. al proposed in [79] an approach to synthesize generalized Streett formulas with rank (1), i. e. formulas of the form $\left(\bigwedge_{i=0}^{N} \mathsf{GF}\varphi_i\right) \rightarrow \left(\bigwedge_{j=0}^{M} \mathsf{GF}\varphi_j\right)$. Their algorithm runs in time K^3 where K is the size of the state space of the design. If a collection Φ_i of LTL formulas representing assumptions on the environment and a collection Φ_j of formulas representing guarantees for the controller can all be represented using deterministic Büchi automata, this approach can be used to obtain a synthesis procedure for $\left(\bigwedge_{i=0}^{N} \Phi_i\right) \rightarrow \left(\bigwedge_{j=0}^{M} \Psi_j\right)$. In [11, 12, 50] Bloem et al developed a tool 'Anzu' for the synthesis of specifications of this form. To this end, a deterministic Büchi automaton (termed monitor there) has to be manually generated for each formula Φ_i and Ψ_j. This process of generating a deterministic automaton manually is considerably hard in general [59], since it also involves a determinization step. Another drawback is that the specification is now not an easy to grasp property in linear temporal logic but instead a deterministic monitor. This adds another potential source for errors since the equivalence between the original specification and this deterministic monitor is in most cases not obvious at all. Although most specifications can be brought into the proper form[3], it is not

[3]We want to emphasize that the set of specifications considered there is much smaller than the set of specifications considered by us in Chapter 4.

possible to obtain a synthesis procedure for full LTL from that approach.

Work concentrating on the synthesis of full LTL is rare. One reason for the high complexity of LTL synthesis is the determinization step. Emerson and Sistla developed in [27] a determinization procedure specifically for the Büchi automata that stem from LTL formula. They made the useful observation that those automata have a special structure which we term nonconfluent in Chapter 5. Using this property they obtained a determinization procedure that transforms nonconfluent automata to deterministic Rabin automata involving a (necessary [59]) exponential blowup. Since the states of this deterministic Rabin automaton are trees that encode different runs of the nondeterministic automaton, this procedure is not amenable to a symbolic implementation. In 1988, Safra [88] presented the first determinization procedure for arbitrary Büchi automata that was asymptotically optimal. The main disadvantage of Safra's solution is that the obtained deterministic Rabin automaton has as states trees of subsets of states of the original nondeterministic automaton which makes it hard for a symbolic implementation. Only recently, nearly 20 years after its publication, Safra's construction has been implemented [54]. However, since they used an explicit representation of the deterministic automaton, this tool is limited to small LTL formulas with only few temporal operators. The same disadvantage of not being amenable to a symbolic implementation holds also for other variants of Safra's construction that were later given e. g. by Muller and Schupp [75] and by Piterman [78].

Since the determinization step seems to be one major hurdle for the synthesis of full LTL, a recent research trend is to replace determinization by lightweight 'pseudo'-determinization procedures. In [60] Kupferman and Vardi present an approach that avoids Safra's determinization and goes through universal co-Büchi word and weak alternating tree automata instead. This approach has been refined in a couple of works [58] to allow compositional synthesis or to allow also controller synthesis [57][4].

Jobstmann and Bloem developed in [49] optimizations for this Safraless approach and developed the tool 'Lilly'. This tool was the first implementation that is able to synthesize designs that satisfy arbitrary LTL specifications. Although Kupferman and Vardi's approach is potentially amenable to a symbolic implementation, the tool 'Lilly' is implemented explicitly, thus limited also to small LTL formulas.

In [46], Henzinger and Piterman introduced nondeterministic automata that are *good for games* (GFG). These automata fairly simulate their deterministic equivalent and can be used to solve the game as a replacement for the deterministic automata. An algorithm is given in [46] that constructs GFG automata from nondeterministic Büchi

[4]The main disadvantage regarding controller synthesis is that the bound used in this so-called Safraless approach also depends on the number of states of the system (which is typically much larger than the specification). Hence we need a good approximation for the number of states of the system, otherwise the bound would become too large to be reasonable.

automata. Since this construction may be implemented symbolically, it is expected to perform better in practice. However, I am not aware of an implementation of this construction.

For simpler classes of ω-automata like safety or co-Büchi automata, simpler determinization procedures can be used. In particular, the Rabin-Scott subset construction can be used to determinize safety automata and Miyano and Hayashi's breakpoint constructions can be used to perform this task for co-Büchi automata. There are already symbolic versions of variants of the subset and breakpoint construction [5, 10]. In [10], procedures are described to compute a symbolically represented nondeterministic automaton from an alternating automaton, i.e., a nondeterminization procedure. Although there are some similarities to the procedure given in Chapter 4, non-determinization of alternating automata and determinization of nondeterministic automata is different for ω-automata [105]. Closer to our determinization procedure is [5] which generates a deterministic automaton for the safety fragment, and thus implements the subset construction. However, they also start with an alternating automaton which is then translated to an explicitly represented nondeterministic automaton. The nondeterministic automaton is generated on the fly, thus avoiding the construction of the whole explicit automaton. However, this step crucially relies on a translation from alternating automata to the corresponding nondeterministic automata while our procedure is independent of the previous translation from temporal logic to nondeterministic automata. In particular, it is not obvious how the work [5] could be generalized to more expressive classes like co-Büchi automata.

Typically, specifications are not given as one large formula, but instead they consist of several relatively small sub-formulas. In [98] an algorithm for LTL synthesis is presented that assumes that the overall specification is given as a conjunction of LTL formulas. Instead of performing determinization for the whole specification, the algorithm generates deterministic automata using the approach of [78] explicitly. Those explicitly represented automata are then encoded symbolically to obtain a generalized parity game which is then solved using the generalized parity algorithm given in [19]. Our algorithm assumes a similar setting. We also assume that the specification is a conjunction of LTL formulas. The determinization is also performed only on those small sub-formula to keep the automata size of the sub-automata small. The automaton for the overall specification is then obtained by combining the deterministic automata. However, the main difference is that we never represent the automata explicitly, so that we expect that our algorithm scales better. Unfortunately, the tool [98] is not publicly available so that we cannot make comparisons.

One of the most important applications of controller synthesis is error localization and error correction [51, 52]. Whenever it is detected that an (manually written) implementation is faulty by model checking, the designer is often left alone to actually locate the error. Although most model checkers generate a counterexample that shows why a specification is violated, it is still hard to detect where the fault is actually located. By iteratively replacing fault candidates with synthesized controllers (which are guaranteed to satisfy the specification by construction), it is possible to give a clue where the fault is actually located and even a hint how to fix the error. Although fault localization is an important application, we do not treat this application in this thesis and leave this for future work.

1.3 Structure of the Thesis

This section reflects the structure of this thesis. The core of this thesis is based on several papers we published in the recent years. Chapter 2 introduces the necessary concepts to understand this work. This thesis consists of three different main parts. Chapter 3 summarizes my work on the minimization of parity automata. Chapter 4 presents a determinization procedure based on the automata hierarchy [18] that has been partly published in [72]. In Chapter 5, a new determinization procedure that is amenable to a symbolic implementation is presented. To this end, we make use of the fact that the automata obtained from LTL formulas have a special structure. This Chapter is based on [71]. In Chapter 6 the presented approaches are summarized to obtain an algorithm to solve controller synthesis problems. Finally, in Chapter 7 experiments performed with the obtained controller synthesis algorithm are listed.

1.4 Contributions

In the following section, the three main parts of this thesis are summarized. First, we summarize the contributions regarding the minimization of parity automata. Then we discuss the work done to translate a large fragment of LTL to symbolically represented automata which is described in detail in Chapter 4. The third part of this thesis is denoted to a new determinization procedure for a special class of automata that are generated from LTL formulas described in Chapter 5.

Minimizing Parity Automata

In the worst case, the generation of a deterministic automaton equivalent to a given LTL formula may result in an automaton with $O(2^{2^{|\varphi|}})$ states. Minimizing automata is thus especially beneficial for the intermediate nondeterministic automaton. Since all our intermediate automata can be rather easily translated to parity automata, we study the minimization problem for parity automata.

In this work, we study simulation relations [69] as a tool for minimizing automata. Simulation relations capture the notion that the moves of one automaton can be mimicked by the moves of another automaton. Our approach to simulation relations is game-based and follows the ideas of [45, 28]. That is, we define simulation via a game of two players, the Spoiler and the Duplicator. We say that an automaton simulates another automaton if the Duplicator, controlling the nondeterministic choices of the first automaton, has a winning strategy against the Spoiler who controls the nondeterministic choices of the second automaton.

We consider three notions of simulation relations that correspond to the respective notions for Büchi automata. First direct simulation for parity automata is introduced. Although direct simulation is weaker than delayed simulation (it allows less minimization), it might be sometimes more appropriate, since it can be calculated in linear time whereas delayed simulation needs quadratic time. The second contribution of this Chapter is the presentation of reverse simulation for parity automata. First in the style presented in [99] of a direct simulation where a good transition in the simulated automaton must be directly matched by a good transition in the simulating automaton. Second also in the relaxed notation of delayed simulation, where the match is post-poned to the future of the game[5].

Finally, fair simulation for parity automata is considered. However, since we are particularly interested in algorithms that are well suited for a symbolic implementation, we do not consider a direct generalization of results obtained in [40] regarding the minimization of Büchi automata. The approach of [40] uses backtracking on individual states for the solution of the minimization problem which is not suitable for a symbolic implementation. Nevertheless in Section 3.7 an algorithm is presented that allows to efficiently perform fair simulation minimization with good minimization results. In particular all merging of states that are possible due to delayed simulation are also detected using the algorithm introduced in this chapter.

Generating Deterministic ω-Automata for most LTL Formulas by Considering the Automata Hierarchy

It has already been outlined that a main ingredient of nearly all controller synthesis algorithms are efficient determinization procedures. The first determinization procedure used in this thesis is based on the temporal logic hierarchy [65, 18, 90, 91].

It was well-known since [65] that for every formula from the temporal logic class a deterministic automaton from the corresponding class exists. However, it was *previously unnoticed* that by employing Boolean combinations of co-Büchi formula, we can avoid complex determinization procedures like the one invented by Safra. In order to develop a Safraless determinization construction for every formula of the temporal logic hierarchy, we proceed as follows:

[5]For Büchi automata, the two notations are equivalent.

We first determine the class of the given LTL formula in the temporal logic hierarchy. Every formula in this hierarchy turned out to be a boolean combination of formulas that can be translated to nondeterministic co-Büchi[6] automata. In [90, 91], algorithms are presented that run in linear time and produce linear sized symbolic descriptions of these co-Büchi automata. After this step, we can use the well-known breakpoint construction [70] of Miyano and Hayashi to determinize these automata and finally, we compute the boolean closure of the obtained deterministic automata, which is straightforward for deterministic automata.

Thus, we are able to translate every formula of the temporal logic hierarchy to an equivalent deterministic ω-automaton (either Rabin or Streett). Due to results of [65], we can moreover translate every LTL formula to a formula contained at least in the highest class of this hierarchy. However, such a translation requires a determinization step, and therefore, it is not useful for our purpose. Thus, we currently have the restriction that given LTL formulas must already syntactically belong to one of the classes of the hierarchy. In practice, we found that this is almost always the case, and in some other cases, it was not too difficult to rewrite the formula to achieve this membership (checking the equivalence of the rewritten LTL formulas is feasible by model checking).

A major ingredient to obtain a symbolic determinization construction is investigated in Chapter 4 where it is shown how the ordinary Rabin-Scott subset construction [84] can be implemented in a semi-symbolic manner. For a given symbolically represented nondeterministic automaton that can be represented by BDDs, we directly construct symbolic descriptions of the deterministic automata that are constructed by the subset construction. Although we can not avoid one exponential step (namely the enumeration of the reachable states of the nondeterministic automaton), we achieved that the symbolic description of the deterministic automaton can be obtained without building it explicitly. Thus, we avoid the enumeration of the exponentially larger state space of the *deterministic* automaton. All steps except for the enumeration of the reachable states of the nondeterministic automaton are symbolically implemented.

A new Determinization Procedure for Unambiguous Büchi Automata

Although nearly all formulas commonly used in practice belongs to one of the classes of the temporal logic hierarchy (or can be rather easily translated to a formula belonging to one of the classes), there may be still cases where such a manual rewriting step

[6]While Büchi automata demand that a run of a word must infinitely often visit a set of designated states, co-Büchi automata impose the stronger requirement that designated states are only finitely often visited.

is undesirable. In Chapter 5, therefore a new determinization procedure for Büchi automata that stem from the translation of arbitrary LTL formulas by the 'symbolic' translation of [21] is presented. It is well-known that the ω-automata that stem from LTL formulas are a special class that has several characterizations. Many translation procedures from LTL generate *unambiguous automata* [17] where every accepted word has a unique accepting run [91, 2] (although there may be additional non-accepting runs for the same word).

The above unambiguous property allows us to develop a determinization procedure that exploits symbolic set representations. In particular, it does not rely on Safra trees as used by Safra's original procedure [88] or by the improved version of Piterman [78]. The states of the deterministic automata obtained by these procedures are trees of subsets of states of the original automaton. In contrast, our procedure generates deterministic automata whose states consist of tuples of subsets of states, allowing a straight-forward symbolic implementation.

2 Preliminaries

In this chapter, the necessary theoretical background for this thesis is described. We give a short introduction to linear temporal logic, followed by an introduction to the theory of ω-automata. Another paragraph of this introductory chapter is concerned with the presentation of infinite games. Finally we give a short introduction to our language quartz which is used to develop the examples of the experimental section.

2.1 Linear Temporal Logic

Linear temporal logic (LTL) [80] is a popular language for the specification of temporal properties. In general, the formulas of a logic depend on the set of atomic formulas, i. e. available variables and constants, and on the set of available operators. LTL provides different kinds of operators that can be classified into the groups of boolean operators and future and past temporal operators. In the following, we assume that we are given a fixed set of variables $\mathcal{V}$ and provide the definitions with respect to this set of variables.

We use the following standard set of boolean operators with the usual semantics: $\mathsf{1}, \mathsf{0}$ denote the constants true and false while $\neg$, $\wedge$ and $\vee$ denote negation, conjunction and disjunction. Common abbreviations are $\varphi \to \psi := \neg\varphi \vee \psi$ and $\varphi \leftrightarrow \psi := (\varphi \to \psi) \wedge (\psi \to \varphi)$. We sometimes use $\overline{\varphi}$ for $\neg\varphi$ for a propositional formula φ. An *assignment* ϑ over $\mathcal{V}$ is a subset of $\mathcal{V}$. The set of propositional formulas over $\mathcal{V}$ is denoted with $\mathcal{L}_{\mathsf{Prop}}^{\mathcal{V}}$.

After having defined the propositional part of LTL, we can now formally define the syntax:

Definition 1 (Syntax of Linear Temporal Logic LTL). *The set of* LTL *formulas over a set of variables* $\mathcal{V}$ *is the smallest set with the following properties:*

- $\mathsf{1}, \mathsf{0} \in \mathsf{LTL}$
- $\mathcal{V} \subseteq \mathsf{LTL}$
- *boolean operators:* $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi \in \mathsf{LTL}$, *if* $\varphi, \psi \in \mathsf{LTL}$
- *future temporal operators:* $\mathsf{X}\varphi$, $[\varphi\ \underline{\mathsf{U}}\ \psi]$, $[\varphi\ \mathsf{B}\ \psi] \in \mathsf{LTL}$, *if* $\varphi, \psi \in \mathsf{LTL}$

- *past temporal operators:* $\overleftarrow{\mathsf{X}}\varphi$, $\overleftarrow{\underline{\mathsf{X}}}\varphi$, $[\varphi\ \overleftarrow{\underline{\mathsf{U}}}\ \psi]$,$[\varphi\ \overleftarrow{\mathsf{B}}\ \psi] \in \mathsf{LTL}$, *if* $\varphi, \psi \in \mathsf{LTL}$

The semantics of LTL is usually given with respect to a path through a structure (i. e. a Kripke structure or an Automaton).

Definition 2 (Path)**.** *Given a set of atomic propositions $\mathcal{V}$, an infinite path is a function $\pi : \mathbb{N} \to 2^{\mathcal{V}}$. For reasons of simplicity, $\pi(i)$ is often denoted by $\pi^{(i)}$ for $i \in \mathbb{N}$. Using this notation, paths are often given in the form $\pi^{(0)}\pi^{(1)}\dots$. The path starting at t is given as : $(\pi, t) := \pi^{(t)}\pi^{(t+1)}\dots$*

Notice that a path as defined above is nothing but a sequence of assignments over the variables $\mathcal{V}$. The semantics of LTL is informally given as follows (see e. g. [91] for a precise definition): $\mathsf{X}\varphi$ holds at a path π at position t_0 if φ holds at position $t_0 + 1$ on the path. $[\varphi\ \underline{\mathsf{U}}\ \psi]$ holds at t_0 iff ψ holds for some position $\delta \geq t_0$ and φ holds invariantly for every position t with $t_0 \leq t < \delta$ i. e. φ holds *until* ψ holds. The *weak before* operator $[\varphi\ \mathsf{B}\ \psi]$ holds at t_0 iff either φ holds before ψ becomes true for the first time after t_0 or ψ never holds after t_0.

In addition to the future time temporal operators, there are also the corresponding past time temporal operators. These are defined analogously with the only difference that the direction of the flow of time is reversed. For example, $[\varphi\ \overleftarrow{\underline{\mathsf{U}}}\ \psi]$ holds on a path at position t_0 iff there is a point of time δ with $\delta \leq t$ such that ψ holds on that path at position δ and φ holds for all positions t with $\delta < t \leq t_0$. The past time analogon of the next-time operator is called the previous operator: $\overleftarrow{\underline{\mathsf{X}}}\varphi$ holds on a path at position t_0 iff $t_0 > 0$ and φ holds at position $t_0 - 1$. Additionally, there is a weak variant, where $\overleftarrow{\mathsf{X}}\varphi$ holds on a path at position t_0 iff $t_0 = 0$ holds or φ holds at position $t_0 - 1$.

Other operators can be defined in terms of the above ones:

$$\begin{aligned}
\mathsf{G}\varphi &= [0\ \mathsf{B}\ \neg\varphi] & \overleftarrow{\mathsf{G}}\varphi &= [0\ \overleftarrow{\mathsf{B}}\ \neg\varphi] \\
\mathsf{F}\varphi &= [1\ \underline{\mathsf{U}}\ \varphi] & \overleftarrow{\mathsf{F}}\varphi &= [1\ \overleftarrow{\underline{\mathsf{U}}}\ \varphi] \\
[\varphi\ \mathsf{B}\ \psi] &= \neg\,[\neg\varphi\ \underline{\mathsf{U}}\ \psi] & [\varphi\ \overleftarrow{\mathsf{B}}\ \psi] &= \neg[\neg\varphi\ \overleftarrow{\underline{\mathsf{U}}}\ \psi] \\
[\varphi\ \mathsf{U}\ \psi] &= [\psi\ \mathsf{B}\ (\neg\varphi \wedge \neg\psi)] & [\varphi\ \overleftarrow{\mathsf{U}}\ \psi] &= [\psi\ \overleftarrow{\mathsf{B}}\ (\neg\varphi \wedge \neg\psi)] \\
[\varphi\ \underline{\mathsf{B}}\ \psi] &= [\neg\psi\ \underline{\mathsf{U}}\ (\varphi \wedge \neg\psi)] & [\varphi\ \overleftarrow{\underline{\mathsf{B}}}\ \psi] &= [\neg\psi\ \overleftarrow{\underline{\mathsf{U}}}\ (\varphi \wedge \neg\psi)] \\
[\varphi\ \mathsf{W}\ \psi] &= [(\varphi \wedge \psi)\ \mathsf{B}\ (\neg\varphi \wedge \psi)] & [\varphi\ \overleftarrow{\mathsf{W}}\ \psi] &= [(\varphi \wedge \psi)\ \overleftarrow{\mathsf{B}}\ (\neg\varphi \wedge \psi)] \\
[\varphi\ \underline{\mathsf{W}}\ \psi] &= [\neg\psi\ \underline{\mathsf{U}}\ (\varphi \wedge \psi)] & [\varphi\ \overleftarrow{\underline{\mathsf{W}}}\ \psi] &= [\neg\psi\ \overleftarrow{\underline{\mathsf{U}}}\ (\varphi \wedge \psi)]
\end{aligned}$$

For example, $[\varphi\ \mathsf{U}\ \psi]$ is the *weak until* operator that can be alternative defined as $[\varphi\ \mathsf{U}\ \psi] := [\varphi\ \underline{\mathsf{U}}\ \psi] \vee \mathsf{G}\varphi$, i. e. the event ψ that is awaited for need not hold in the future. To distinguish weak and strong operators, the strong variants of a temporal operator are underlined throughout this thesis (as shown above).

2.2 ω-Automata

In this section, finite automata over infinite words, called ω-automata [103], are introduced. To this end, words, semi-automata, runs and paths are introduced first. Then, these definitions are used to define ω-automata.

2.2.1 From Infinite Words to ω-Automata

Definition 3 (Words). *In the following, we fix a finite set of input variables $V_\Sigma = \{i_1, \dots i_m\}$ that define a finite set $\Sigma = 2^{V_\Sigma}$, called the* alphabet. *The elements of Σ are called* letters. *Obviously, each letter is an assignment over V_Σ. A* finite word *α over an alphabet Σ of length $|\alpha| = n + 1$ is a function $\alpha : \{0, \dots n\} \to \Sigma$. The finite word of length 0 is called the* empty word *(denoted by ϵ). An infinite word α over an alphabet Σ is a function $\alpha : \mathbb{N} \to \Sigma$. Its length is denoted by $|\alpha| = \infty$. Thus every (finite or infinite) word is a (finite or infinite) concatenation of letters, i.e. a concatenation of assignments over the variables in V_Σ.*

For reasons of simplicity, $\alpha(i)$ is often denoted by $\alpha^{(i)}$ for $i \in \mathbb{N}$. Using this notation, words are often given in the form $\alpha^{(0)}\alpha^{(1)} \dots \alpha^{(n)}$ or $\alpha^{(0)}\alpha^{(1)} \dots$. The set of all finite words over Σ is denoted by Σ^*, and the set of all infinite words over Σ is denoted by Σ^ω.
Counting of letters starts with zero, i.e. $\alpha^{(i-1)}$ refers to the i-th letter of α. Furthermore, $\alpha^{(i\dots)}$ denotes the suffix of α starting at position i, i.e. $\alpha^{(i\dots)} = \alpha^{(i)}\alpha^{(i+1)} \dots$. The finite word $\alpha^{(i)}\alpha^{(i+1)} \dots \alpha^{(j)}$ is denoted by $\alpha^{(i..j)}$. Notice that in case $j < i$ the expression $\alpha^{(i..j)}$ evaluates to the empty word ϵ. For two words α_1, α_2, we use $\alpha_1\alpha_2$ for the concatenation of α_1 and α_2. Finally, we write a^ω for the infinite word α with $\alpha^{(j)} = a$ for all j. We also write $\beta = (\alpha)^\omega$ for the infinite concatenation of a finite word α.

Unlike classical automata, ω-automata read a given infinite word. The acceptance of a given word is defined via an acceptance condition on the infinite run of the word. To this end, we may use linear temporal logic as a unified framework to reason about the infinite runs of the automata and thus need a labeling function that maps states to assignments, i. e. propositional variables that hold in that state.

Definition 4 (Semi-automaton). *Let Q be a finite set of state variables. Let V_Σ be a finite set of input variables disjoint from Q that defines an alphabet $\Sigma = 2^{V_\Sigma}$. Then, a* semi-automaton *$\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda)$ over the alphabet Σ is given by*

- *a finite set of states $\mathcal{S}$,*
- *a set of initial states $\mathcal{I} \subseteq \mathcal{S}$,*
- *a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ and*
- *a labeling function $\lambda : \mathcal{S} \to 2^Q$*

Using standard terminology, we say that $\mathfrak{A}$ is *deterministic*, if exactly one initial state exists and for each $s \in \mathcal{S}$ and each input $\sigma \in \Sigma$ there does exist exactly one $s' \in \mathcal{S}$ such that $(s, \sigma, s') \in \mathcal{R}$ holds. In this case, we also write the transition relation as a function $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$. Otherwise,we say that $\mathfrak{A}$ is *non-deterministic*. We will sometimes also need a semi-automaton that is obtained from $\mathfrak{A}$ by modifying the set of initial states. To this end, for $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda)$ and a set of states $\mathcal{I}' \subseteq \mathcal{S}$, define $\mathfrak{A}_{\mathcal{I}'} := (\mathcal{S}, \mathcal{I}', \mathcal{R}, \lambda)$. If $\mathcal{I}' = \{q\}$ is a singleton set, we omit the brackets and simply write $\mathfrak{A}_q := (\mathcal{Q}, \{q\}, \mathcal{R}, \lambda)$.

The acceptance of a word is defined with respect to the set of runs:

Definition 5 (Run of an Infinite Word)**.** *Given a semi-automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda)$ and an infinite word $\alpha : \mathbb{N} \to \Sigma$ over Σ. Then each infinite word $\beta : \mathbb{N} \to \mathcal{S}$ with $\beta^{(0)} \in \mathcal{I}$ and $\forall t. \left(\beta^{(t)}, \alpha^{(t)}, \beta^{(t+1)}\right) \in \mathcal{R}$ is called a run of α through $\mathfrak{A}$. The set of all runs of α through $\mathfrak{A}$ is hence defined as follows:*

$$\mathsf{RUN}_{\mathfrak{A}}(\alpha) := \left\{\beta : \mathbb{N} \to \mathcal{S} \mid \beta^{(0)} \in \mathcal{I} \wedge \forall t. \left(\beta^{(t)}, \alpha^{(t)}, \beta^{(t+1)}\right) \in \mathcal{R}\right\}$$

The infinity set of a run β is defined as :

$$\mathsf{INF}(\beta) := \{s \in \mathcal{S} \mid \beta^{(t)} = s \textit{ for infinitely many } t \in \mathbb{N}\}$$

The difference between classical finite automata and ω-automata is that ω-automata run over an infinite word whereas finite automata run over a finite word. Since there is no 'final' state in an infinite run, the acceptance of an infinite word must consider the (infinite) sequence of states that are visited by the ω-automaton. Notice that each run uniquely defines a path over the variables Q, since every state is uniquely equipped with an assignment by the labeling function. To this end, let $\beta = \beta^{(0)}\beta^{(1)}\ldots$ be a run of a semi-automaton. We extend the labeling function to runs by defining

$$\lambda(\beta) = \lambda(\beta^{(0)})\lambda(\beta^{(1)})\ldots$$

Notice that $\lambda(\beta)$ is a path over Q, called the *trace* of β. Since every trace of a semi-automaton $\mathfrak{A}$ over a word α is a path over Q, we can use LTL to specify the acceptance conditions for ω-automata[1].

[1] The acceptance of a word is then specified with the set of runs (thus the set of paths) over this word.

Since every state is uniquely determined by the labellng function, each propositional formula φ over Q together with the labeling function defines a set of states $\mathcal{F}_\varphi$ by $\mathcal{F}_\varphi := \{q \in 2^Q \mid \lambda(q) \models \varphi\}$. Having this view, the classical acceptance conditions can be formalized as a LTL-formula as shown in the following definition:

Definition 6 (Classical Acceptance Conditions [61, 109, 102, 65, 91])**.** *Let Φ_i and Ψ_i for all $i \in \{0, \dots f\}$ be propositional formulas over the state variables Q, then we define the following classes of acceptance conditions:*

- *Safety condition:* $\mathsf{G}\Phi_0$
- *Liveness condition :* $\mathsf{F}\Phi_0$
- *Büchi condition [14, 15]:* $\mathsf{GF}\Phi_0$
- *Persistence condition [66] :* $\mathsf{FG}\Phi_0$
- *Rabin condition[83] :* $\bigvee_{j=0}^{f} \mathsf{GF}\Phi_j \wedge \mathsf{FG}\Psi_j$
- *Streett condition[101] :* $\bigwedge_{j=0}^{f} \mathsf{GF}\Phi_j \vee \mathsf{FG}\Psi_j$
- *Prefix conditions[100] :* $\bigwedge_{j=0}^{f} \mathsf{G}\Phi_j \vee \mathsf{F}\Psi_j$

Hence, taking a closer look at the definition of LTL it is not hard to see that the following holds:

- A run is accepted by a safety condition $\mathsf{G}\varphi$ iff the run exclusively runs through the set $\mathcal{F}_\varphi$
- A run is accepted by a liveness condition $\mathsf{F}\varphi$ iff the run visits the set $\mathcal{F}_\varphi$ at least once.
- A run is accepted by a Büchi condition $\mathsf{GF}\varphi$ iff the run visits at least one state of the set $\mathcal{F}_\varphi$ infinitely often.
- A run is accepted by a persistence (or Co-Büchi) condition $\mathsf{FG}\varphi$ iff the run visits only states of the set $\mathcal{F}_\varphi$ infinitely often, i. e. each state from the complement set $\mathcal{F}_{\overline{\varphi}}$ is visited only finitely often.
- A run is accepted by a Rabin condition, if there is an index $j \in \{0, \dots f\}$ such that the trace infinitely often satisfies Φ_j and after some time always satisfies Ψ_j. In other words, for some j, the run must visit at least one state from $\mathcal{F}_{\Phi_j}$ infinitely often and may only finitely often visit a state from $\mathcal{F}_{\neg\Psi_j}$.

- The dual of a Rabin condition is a Streett condition that demands for every j that if some set $\mathcal{F}_{\neg\Phi_j}$ is visited infinitely often, then also $\mathcal{F}_{\Psi_j}$ is visited infinitely often.
- Finally, a prefix condition (or Staiger-Wagner condition[100] is satisfied by a run such that for all j either all positions of the trace satisfy Φ_j or at some position Ψ_j holds. That is, the run visits either only states from $\mathcal{F}_{\Phi_j}$ or at least once a state from $\mathcal{F}_{\Psi_j}$.

Another form of acceptance conditions is the parity condition [73] that has become popular during the last years. A parity condition is conveniently defined via a coloring function.

Definition 7 (Coloring Function)**.** *Let $f \in \mathbb{N}$ be a natural number. Then, any function $\Omega : \mathcal{S} \to \{0, \dots f\}$ is called a* coloring function*. The number $\Omega(s)$ for $s \in \mathcal{S}$ is called the color of state s.*

The acceptance of a parity condition is defined as follows:

Definition 8 (Parity Acceptance)**.** *Given a semi-automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda)$ and a coloring function Ω. Define the minimal color that is visited infinitely often during a run β by*

$$\mathsf{MIN}_\Omega(\beta) = \min\{\Omega(s) \mid s \in \mathsf{INF}(\beta)\}$$

Then the parity acceptance condition is defined as follows: A run β of $\mathfrak{A}$ satisfies the parity condition, iff $\mathsf{MIN}_\Omega(\beta)$ is even.

In the following, we identify a parity condition with its coloring function, and write e. g. $\beta \models \Omega$ whenever β satisfies the parity condition. An acceptance component is now defined as follows:

Definition 9 (Acceptance Component)**.** *Let $\mathfrak{A} = (Q, \mathcal{I}, \mathcal{R}, \lambda)$ be a semi-automaton. Then the set of all acceptance conditions $ac_\mathfrak{A}$ over $\mathfrak{A}$ is given by:*

- *If φ is an* LTL *formula over Q, then $\varphi \in ac_\mathfrak{A}$*
- *Any parity condition Ω over Q is in $ac_\mathfrak{A}$*
- *if $\varphi, \psi \in ac_\mathfrak{A}$, then $\phi \wedge \psi \in ac_\mathfrak{A}$ and $\phi \vee \psi \in ac_\mathfrak{A}$*
- *if $\varphi \in ac_\mathfrak{A}$, then $\neg\varphi \in ac_\mathfrak{A}$*

With the help of the preceding definitions, we can formally define ω-automata :

Definition 10 (ω-Automata)**.** *An existential [2] ω-automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda, \Phi)$ is a tuple where $(\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda)$ is a semi-automaton and Φ is an acceptance condition.*

We sometimes also specify safety, liveness, Büchi or co-Büchi automata by a set of states $\mathcal{F}_\varphi$ that define a propositional formula φ over Q and write $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F}_\varphi)$ i. e. we omit the labeling function. Similarly, we omit sometimes the labeling functions when dealing with parity automata and write $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$

Definition 11 (Acceptance of a Word by an ω-Automaton)**.** *Given an ω-automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda, \Phi)$ and a run β of $\mathfrak{A}$ over a word α. Then the following rules define the acceptance of β by $\mathfrak{A}$, $\beta \models \mathfrak{A}$:*

- *$\beta \models \varphi$ iff $\mathcal{L}(\beta) \models \varphi$ if $\phi \in$* LTL
- *$\beta \models \Omega$ iff $\beta \models \Omega$ for a parity condition Ω*
- *$\beta \not\models \varphi$ iff not $\beta \models \varphi$*
- *$\beta \models \varphi \wedge \vee \psi$ iff $\beta \models \varphi$ and $\beta \models \psi$*
- *$\beta \models \varphi \vee \psi$ iff $\beta \models \varphi$ or $\alpha \models \psi$*

A word α over Σ is accepted by $\mathfrak{A}$, $\alpha \models \mathfrak{A}$ iff there is a run of β of $\mathfrak{A}$ over α such that $\beta \models \mathfrak{A}$. The language *of $\mathfrak{A}$ is the set of accepted words, i. e.* $\mathsf{Lang}(\mathfrak{A}) := \{\alpha \in \Sigma \mid \alpha \models \mathfrak{A}\}$.

The above acceptance conditions define corresponding automaton classes. We denote the set of (non)deterministic safety, liveness, Büchi, persistence, Rabin, Streett and parity automata with $(\mathsf{N})\mathsf{Det}_\mathsf{G}$, $(\mathsf{N})\mathsf{Det}_\mathsf{F}$, $(\mathsf{N})\mathsf{Det}_\mathsf{GF}$, $(\mathsf{N})\mathsf{Det}_\mathsf{FG}$, $(\mathsf{N})\mathsf{Det}_\mathsf{prefix}$, $(\mathsf{N})\mathsf{Det}_\mathsf{Rabin}$, $(\mathsf{N})\mathsf{Det}_\mathsf{Streett}$ and $(\mathsf{N})\mathsf{Det}_\mathsf{Parity}$. Conjunctions of Büchi conditions are often called generalized Büchi conditions while generalized parity conditions are either conjunctions or disjunctions of parity conditions.

2.2.2 Symbolically Represented ω-Automata

In this section, we briefly describe how we represent ω-automata on infinite words using BDDs. We assume that the input alphabet is encoded using boolean variables V_Σ. To represent the state transition relation of the automata, we introduce two

[2] Some works related with ω-automata consider $\forall$-automata[64] that have an acceptance condition that must be fulfilled by *all* runs of a word. Usually, however, automata are considered that only require that a word has *at least* one run that fulfills the specified acceptance condition. In this thesis, we will only consider the second case, i.e. we restrict our attention to those automata that require that *at least* one accepting run exists.

state sets of propositional variables, one for the current and one for the next point of time. Using these propositional variables, we are able to encode the initial states, the transition relation and the acceptance condition using boolean functions which can be represented with BDDs or handled by SAT solvers.

Definition 12 (Automaton Formula)**.** *Given a finite set of variables Q with $Q \cap V_\Sigma = \{\}$, a propositional formula $\mathcal{I}$ over $Q \cup V_\Sigma$, a propositional formula $\mathcal{R}$ over $Q \cup V_\Sigma \cup \{v' \mid v \in Q \cup V_\Sigma\}$, and a LTL- formula $\mathcal{F}$ over $Q \cup V_\Sigma$, then $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ is an (existential) automaton formula.*

We will sometimes also need the evaluation of a propositional formula at the next point of time. Thus, for every propositional formula φ, we use $[\varphi]^{q_1,\ldots,q_m}_{q'_1,\ldots,q'_n}$ to mean the formula that is obtained from φ by replacing every state variable $q \in Q$ by q'.

2.3 From LTL to Nondeterministic ω-Automata

Symbolic methods have become state of the art in modern model checkers. Nearly all symbolic model checkers like NuSMV or Cadence-SMV use the algorithm of [21] to translate LTL formulas to symbolically represented generalized Büchi automata [87]. As explained in [90, 91], this 'standard' translation procedure from LTL to ω-automata traverses the syntax tree of the LTL formula in a bottom-up manner and abbreviates each sub-formula that starts with a temporal operator. For example, the sub-formula $[\varphi \ \underline{\mathsf{U}}\ \psi]$ is thereby abbreviated by a new state variable q such that the following holds:

$$q \leftrightarrow \psi \vee \varphi \wedge q'$$

The transition relation is then equivalent to

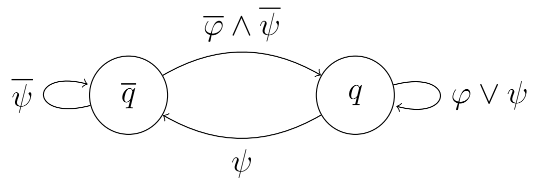
$$\mathcal{R} :\equiv q \leftrightarrow \psi \vee \varphi \wedge q'$$

so that we obtain the automaton shown in Figure 2.1[3] .

To define the acceptance condition, we have to add the fairness constraint $\mathcal{F}_i :\equiv (q \rightarrow \psi)$ as a new set of accepting states [4]. The sub-formula $[\varphi \ \mathsf{U}\ \psi]$ is also abbreviated by a new state variable q with the same update of the transition relation. However, we add the fairness constraint $\mathcal{F}_i :\equiv (\varphi \rightarrow q)$.

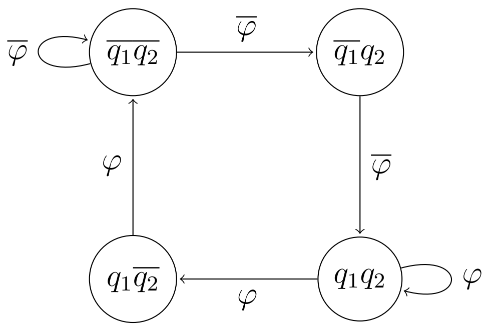
[3]To simplify the picture, we didn't used the set based notation, i.e. the set $\{\}$ is represented by $\overline{q}$ and similar in Figure 2.2 with $\overline{q_1 q_2}$.

[4]Notice that any fairness constraint ξ that is a formula over both state and input variables can be easily replaced by a formula that is defined over the state variables alone by adding a new state variable q' with the constraint $q' \leftrightarrow \xi$.



ψ	φ	$q \leftrightarrow \psi \vee \varphi \wedge q'$
0	0	$q \leftrightarrow 0$
0	1	$q \leftrightarrow q'$
1	0	$q \leftrightarrow 1$
1	1	$q \leftrightarrow 1$

Figure 2.1: Nondeterministic Automaton for $[\varphi\ \underline{\mathsf{U}}\ \psi]$



φ	$(q_1 \leftrightarrow \varphi) \wedge ({q_1}' \leftrightarrow q_2)$
0	$(q_1 \leftrightarrow 0) \wedge ({q_1}' \leftrightarrow q_2)$
1	$(q_1 \leftrightarrow 1) \wedge ({q_1}' \leftrightarrow q_2)$

Figure 2.2: Nondeterministic Automaton for $\mathsf{X}\varphi$

A sub-formula $\mathsf{X}\varphi$ introduces two new state variables q_1 and q_2 where the transition relation $\mathcal{R}$ is defined as:

$$\mathcal{R} :\equiv (q_1 \leftrightarrow \varphi) \wedge\ ({q_1}' \leftrightarrow q_2)$$

and no fairness constraint is generated so that we obtain the automaton in Figure 2.2.

In the general case, every elementary sub-formula is abbreviated in a bottom-up traversal through the syntax tree. During that traversal, for every elementary sub-formula a new state variable and a new fairness constrained is introduced as described above. Hence we obtain a so-called generalized Büchi automaton with multiple fairness constraints. However, it is well-known that multiple fairness constraints can be easily transformed to an automaton having a single acceptance condition. For more information regarding the translation procedure, see the detailed explanations in Chapter 5.4.1 of [91].

2.3.1 From LTL to Unambiguous ω-Automata

It is well-known that the ω-automata that stem from LTL formulas are a special class that has already found several characterizations. Due to results of [67], the automata can be characterized as *non-counting* automata, and in terms of alternating automata, the class of *linear weak* or *very weak* automata has been defined [68, 35, 77, 74]. Moreover, many translation procedures from LTL generate *unambiguous automata* [17] where every accepted word has a unique accepting run [91, 2] (although there may be additional non-accepting runs for the same word). In Chapter 5, a new determinization procedure is presented that makes use of the fact that the automata obtained by the translation procedure of the previous section yield unambiguous automata, which is shown in the following paragraph.

Definition 13 (Unambiguous Automata). *An ω-automaton $\mathfrak{A} = (\mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{L}, \mathcal{F})$ is called unambiguous if every word $\alpha \in \mathsf{Lang}(\mathfrak{A})$ labels exactly one accepting run*[5]*.*

To give an intuitive idea why the automata constructed from LTL formulas by the 'standard translation' procedure are unambiguous, reconsider the automaton of Figure 2.2 that is obtained by translating the formula $[\varphi \, \underline{\mathsf{U}} \, \psi]$ to a non-deterministic automaton.

As can be seen by Figure 2.1, the input $\varphi \wedge \neg\psi$ demands that the current state is maintained, but allows the automaton to be in any of the two states. The other three classes of inputs uniquely determine the current state, but leave the successor state completely unspecified. As a consequence, input words that infinitely often satisfy $\neg(\varphi \wedge \neg\psi)$, i.e., $\neg\varphi \vee \psi$, do only have one (infinite) run, while the remaining input words that satisfy $\varphi \wedge \neg\psi$ from a certain point of time on do have two runs that are of the form ξq^ω and $\xi \overline{q}^\omega$ with the same finite prefix ξ. However, the fairness constraint $\mathcal{F} = q \rightarrow \varphi$ is satisfied by only one of the two runs[6]. Hence, the automaton is unambiguous. A similar consideration shows that the automaton of Figure 2.2 is also unambiguous.

An example run tree (that encodes all the runs of a given word) is shown in Fig. 2.3. It can be seen that there is a uniquely determined run, since all other nondeterministic choices lead to finite paths. Another example run tree that contains two infinite runs is shown in Figure 2.4. Although this run tree has two infinite runs, the automaton is still unambiguous since the fairness constraint assures that only one of the two runs is accepting.

As every automaton $\mathfrak{A}_\varphi$ obtained by the translation of a LTL formula φ is a product of the unambiguous automata shown in Figures 2.1 and 2.2, and as the product automaton of two unambiguous automata is also unambiguous, it follows that the automata $\mathfrak{A}_\varphi$ obtained by the above 'standard' translation are unambiguous. The following proposition has been shown in [91] as Theorem 5.42:

[5]Notice that our definition of unambiguous automata slightly differs from the one given in [17]. An automaton is unambiguous according to [17] if every word has exactly one run starting in *any* state. We demand this only for initial runs.

[6]And this also holds when we abbreviate the fairness constraint with a new propositional variable $q' \leftrightarrow (q \rightarrow \varphi)$ which we neglect here to simplify the pictures.

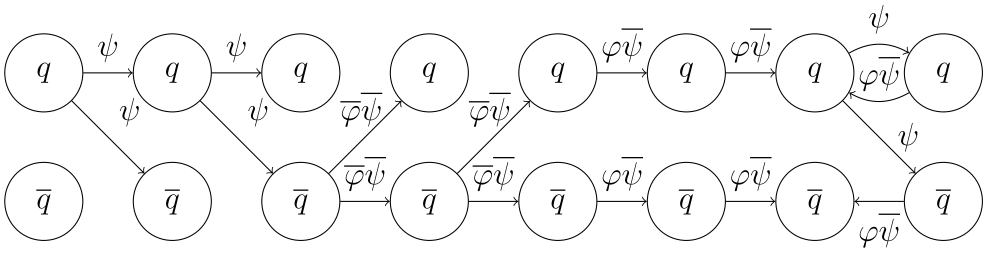


Figure 2.3: Run Tree with a Uniquely Determined Run of the Automaton of Fig. 2.1

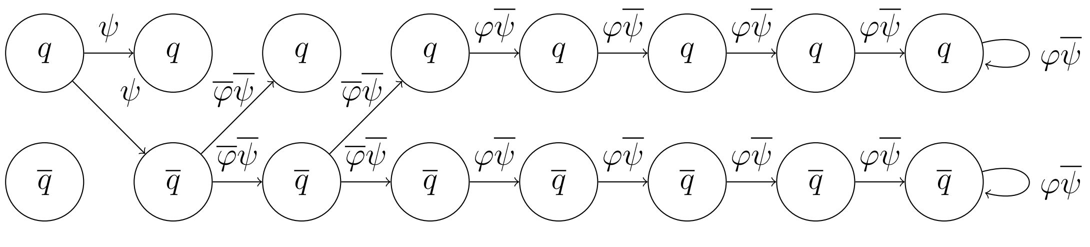


Figure 2.4: Run Tree with Two Runs of the Automaton of Fig. 2.1

Proposition 1. *Given a formula $\Phi \in$ LTL, we can construct a nondeterministic generalized Büchi automaton $\mathcal{A}_\exists(Q, \Phi_\mathcal{I}, \Phi_\mathcal{R}, \Phi_\mathcal{F})$ in time $O(|\Phi|)$ with $|Q| \leq |\Phi|$ state variables which is unambiguous. Therefore, $\mathcal{A}_\exists(Q, \Phi_\mathcal{I}, \Phi_\mathcal{R}, \Phi_\mathcal{F})$ is a symbolic representation of a nondeterministic automaton with $O(2^{|\Phi|})$ states.*

The resulting automaton of the above translation is a generalized Büchi automaton, since we obtain an accepting set of states (actually edges) for every occurrence of an until-operator. Thus, the acceptance condition is given as a formula $\mathcal{F} = \bigwedge_{i=1}^{n} \mathsf{GF}\Phi_i$. It is however straightforward to replace a conjunction of fairness constraints with a single fairness constraint [91]. To this end, we generate the deterministic automaton with $n+1$ states $q_1 \dots q_n, q_a$ shown in Figure 2.5.

Whenever the automaton reaches a state labeled with q_i, it waits until Φ_i is read and remains in q_i as long as this event does not occur. Hence, if q_a holds infinitely often, each of the events Φ_i must occur infinitely often. Since the automaton is deterministic, its runs are uniquely determined. This means that every run of the product of this automaton with the original unambiguous automaton is also uniquely determined, hence it is unambiguous [7].

[7] We use the usual automaton product, which can be alternatively seen as a conjunction of symbolically represented automata, i. e. the conjunctions of the formulas representing initial states and transition relation.

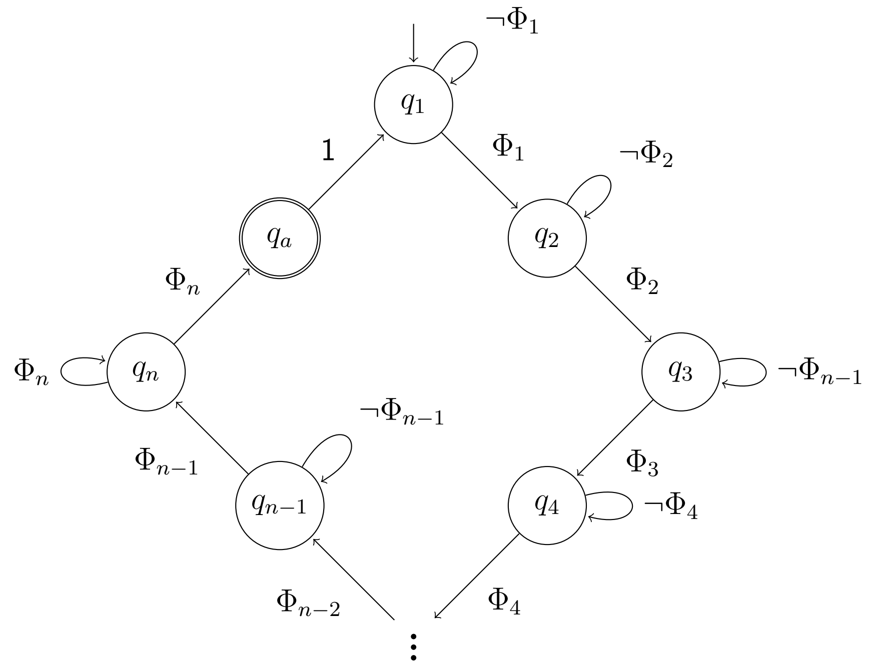


Figure 2.5: Automaton for the Conjunction of Finitely Many Fairness Constraints

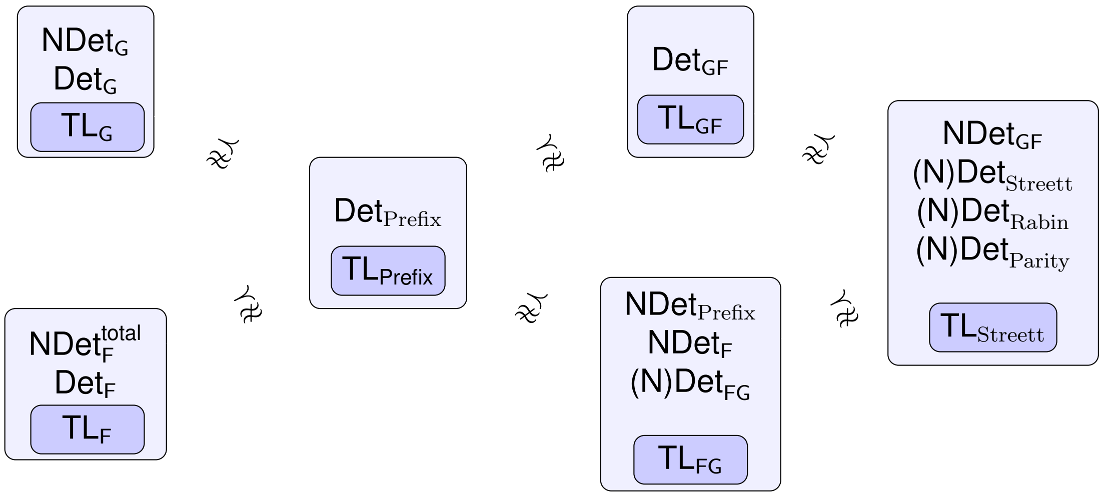
2.4 Relating the Automata Hierarchy and the Temporal Logic Hierarchy

It is well known that different acceptance conditions lead to different automaton classes [18, 91] with different power. The expressiveness of these classes is illustrated in Figure 2.6, where $\mathcal{C}_1 \precapprox \mathcal{C}_2$ means that for any automaton in $\mathcal{C}_1$, there is an equivalent one in $\mathcal{C}_2$. Moreover, we define $\mathcal{C}_1 \approx \mathcal{C}_2 := \mathcal{C}_1 \precapprox \mathcal{C}_2 \wedge \mathcal{C}_2 \precapprox \mathcal{C}_1$ and $\mathcal{C}_1 \precnapprox \mathcal{C}_2 := \mathcal{C}_1 \precapprox \mathcal{C}_2 \wedge \neg(\mathcal{C}_1 \approx \mathcal{C}_2)$. As can be seen, the hierarchy consists of six different classes, and each class has a deterministic representative.

In [90, 18], a corresponding hierarchy of temporal logics has been defined. It is shown now how this temporal logic hierarchy can be used to obtain an efficient determinization construction for a rich fragment of LTL. Hence, we want to complete the picture from Figure 2.6: we show that for every formula of a temporal logic class a corresponding deterministic automaton can be obtained.

Following [90], we define the hierarchy of temporal formulas by the grammar rules of Figure 2.7:

Definition 14 (Temporal Logic Classes)**.** *We define the logics* TL_κ *for* $\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{FG}, \mathsf{GF}, \mathsf{Streett}\}$ *by the grammar rules given in Figure 2.7, where* TL_κ *is the set*



$\mathsf{NDet}_{\mathsf{G}}$
$\mathsf{Det}_{\mathsf{G}}$
TL_{G}
$\mathsf{NDet}_{\mathsf{F}}^{\mathsf{total}}$
$\mathsf{Det}_{\mathsf{F}}$
TL_{F}
$\mathsf{Det}_{\mathrm{Prefix}}$
$\mathsf{TL}_{\mathsf{Prefix}}$
$\mathsf{Det}_{\mathsf{GF}}$
$\mathsf{TL}_{\mathsf{GF}}$
$\mathsf{NDet}_{\mathrm{Prefix}}$
$\mathsf{NDet}_{\mathsf{F}}$
$\mathsf{(N)Det}_{\mathsf{FG}}$
$\mathsf{TL}_{\mathsf{FG}}$
$\mathsf{NDet}_{\mathsf{GF}}$
$\mathsf{(N)Det}_{\mathrm{Streett}}$
$\mathsf{(N)Det}_{\mathrm{Rabin}}$
$\mathsf{(N)Det}_{\mathrm{Parity}}$
$\mathsf{TL}_{\mathrm{Streett}}$

Figure 2.6: Hierarchy of ω-automata and Temporal Logic Hierarchy

$P_{\mathsf{G}} ::= V_\Sigma \mid \neg P_{\mathsf{F}} \mid P_{\mathsf{G}} \wedge P_{\mathsf{G}} \mid P_{\mathsf{G}} \vee P_{\mathsf{G}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{G}}]$ $\mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{G}}]$ $\mid \mathsf{X} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \mathsf{U}\ P_{\mathsf{G}}]$	$P_{\mathsf{F}} ::= V_\Sigma \mid \neg P_{\mathsf{G}} \mid P_{\mathsf{F}} \wedge P_{\mathsf{F}} \mid P_{\mathsf{F}} \vee P_{\mathsf{F}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{F}}]$ $\mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{F}}]$ $\mid \mathsf{X} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \underline{\mathsf{U}}\ P_{\mathsf{F}}]$
$P_{\mathsf{Prefix}} ::= P_{\mathsf{G}} \mid P_{\mathsf{F}} \mid \neg P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \wedge P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \vee P_{\mathsf{Prefix}}$	
$P_{\mathsf{GF}} ::= P_{\mathsf{Prefix}}$ $\mid \neg P_{\mathsf{FG}} \mid P_{\mathsf{GF}} \wedge P_{\mathsf{GF}} \mid P_{\mathsf{GF}} \vee P_{\mathsf{GF}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{GF}} \mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{GF}} \mid \mathsf{X} P_{\mathsf{GF}}$ $\mid [P_{\mathsf{GF}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{GF}}] \mid [P_{\mathsf{GF}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{GF}}]$ $\mid [P_{\mathsf{GF}}\ \mathsf{U}\ P_{\mathsf{GF}}] \mid [P_{\mathsf{GF}}\ \underline{\mathsf{U}}\ P_{\mathsf{F}}]$	$P_{\mathsf{FG}} ::= P_{\mathsf{Prefix}}$ $\mid \neg P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \wedge P_{\mathsf{FG}} \mid P_{\mathsf{FG}} \vee P_{\mathsf{FG}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{FG}} \mid \mathsf{X} P_{\mathsf{FG}} \mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{FG}}$ $\mid [P_{\mathsf{FG}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{FG}}] \mid [P_{\mathsf{FG}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{FG}}]$ $\mid [P_{\mathsf{FG}}\ \underline{\mathsf{U}}\ P_{\mathsf{FG}}] \mid [P_{\mathsf{G}}\ \mathsf{U}\ P_{\mathsf{FG}}]$
$P_{\mathsf{Streett}} ::= P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \mid \neg P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \wedge P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \vee P_{\mathsf{Streett}}$	

Figure 2.7: Classes of Temporal Logic

of formulas that can be derived from the nonterminal P_κ (V_Σ represents any variable $v \in V_\Sigma$).

TL_{G} is the set of formulas where each occurrence of a weak/strong temporal future operator is positive/negative, and similarly, each occurrence of a weak/strong temporal future operator in TL_{F} is negative/positive. Hence, both logics are dual to each other, which means that one contains the negations of the other one. $\mathsf{TL}_{\mathsf{Prefix}}$ is the boolean closure of TL_{G} (and TL_{F}). The logics $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$ are constructed in the same way as TL_{G} and TL_{F}; however, there are two differences: (1) these logics allow occurrences of $\mathsf{TL}_{\mathsf{Prefix}}$ where otherwise variables would have been required in TL_{G}

and TL_{F}, and (2) there are additional 'asymmetric' grammar rules. It can be easily shown that $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$ are also dual to each other, and their intersection strictly contains $\mathsf{TL}_{\mathsf{Prefix}}$. Finally, $\mathsf{TL}_{\mathsf{Streett}}$ is the boolean closure of $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$.

It is well known that every formula of LTL can be rewritten to a formula from $\mathsf{TL}_{\mathsf{Streett}}$ [65]. However, known rewriting procedures from full LTL to $\mathsf{TL}_{\mathsf{Streett}}$ either use deterministic automata [91] as an intermediate step or use rewriting steps that are nonelementary [38]. Hence those known procedures do not describe a reasonable way to obtain a deterministic automaton for full LTL. It is however remarkable that almost all formulas that occur in practice belong to $\mathsf{TL}_{\mathsf{Streett}}$. If a given formula should not belong to $\mathsf{TL}_{\mathsf{Streett}}$, it is often straightforward to rewrite it to an equivalent $\mathsf{TL}_{\mathsf{Streett}}$ formula. For example, the formula $\mathsf{GF}\,[a \;\mathsf{U}\; b]$ that demands that infinitely often $[a \;\mathsf{U}\; b]$ holds is equivalent to the $\mathsf{TL}_{\mathsf{Streett}}$ formula $\mathsf{GF}\,[a \;\underline{\mathsf{U}}\; b] \vee \mathsf{FG}\,[a \;\mathsf{U}\; b]$. Clearly, there are many formulas outside $\mathsf{TL}_{\mathsf{Streett}}$, but we claim that these formulas seldom occur in practice. We substantiate our claim in Chapter 7 by showing a large diversity of specifications all belonging to $\mathsf{TL}_{\mathsf{Streett}}$.

In the following it is sketched how arbitrary formulas from TL_{κ} can be translated to automata from Det_{κ}. In [90, 91] several translation procedures are given to translate formulas from TL_{κ} to equivalent NDet_{κ} automata. In particular, translation procedures are given that translate TL_{G} and $\mathsf{TL}_{\mathsf{FG}}$ formulas to safety, i. e. $\mathsf{NDet}_{\mathsf{G}}$ or to co-Büchi, i. e. $\mathsf{NDet}_{\mathsf{FG}}$ automata. Both translation procedures are modifications of the translation procedure sketched in Section 2.3.

Theorem 1 ([91])**.** *Given a formula $\Phi \in \mathsf{TL}_{\kappa}$, with $\kappa \in \{\mathsf{G}, \mathsf{FG}\}$ we can construct a nondeterministic ω-automaton $\mathcal{A}_{\exists}\,(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ of the class NDet_{κ} in time $O(|\Phi|)$ with $|Q| \leq |\Phi|$ state variables. Therefore, $\mathcal{A}_{\exists}\,(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ is a symbolic representation of a nondeterministic automaton with $O(2^{|\Phi|})$ states.*

Moreover, it can be shown that nondeterministic safety automata of the class $\mathsf{NDet}_{\mathsf{G}}$ can be determinized using the subset construction [84] and that Miyano and Hayashi's breakpoint construction is sufficient to determinize $\mathsf{NDet}_{\mathsf{FG}}$ automata using time $O(2^n)$ where n is the number of states of the nondeterministic automaton.

Since $\mathsf{TL}_{\mathsf{Prefix}}$ and $\mathsf{TL}_{\mathsf{Streett}}$ are the boolean closures of $\mathsf{TL}_{\mathsf{G}}/\mathsf{TL}_{\mathsf{F}}$ and $\mathsf{TL}_{\mathsf{FG}}/\mathsf{TL}_{\mathsf{GF}}$, respectively, deterministic automata for $\mathsf{TL}_{\mathsf{Prefix}}$ and $\mathsf{TL}_{\mathsf{Streett}}$ can be obtained by boolean combinations of $\mathsf{Det}_{\mathsf{G}}/\mathsf{Det}_{\mathsf{F}}$ and $\mathsf{Det}_{\mathsf{FG}}/\mathsf{Det}_{\mathsf{GF}}$ automata.

To this end, it is shown in [91] how arbitrary boolean combinations of $\mathsf{G}\varphi$ and $\mathsf{F}\varphi$ with propositional formulas φ are translated to equivalent $\mathsf{Det}_{\mathsf{Prefix}}$ automata, and analogously, how arbitrary boolean combinations of $\mathsf{GF}\varphi$ and $\mathsf{FG}\varphi$ with propositional formulas φ are translated to equivalent $\mathsf{Det}_{\mathsf{Streett}}$ automata. Moreover, it is shown how acceptance conditions can be converted in more powerful acceptance conditions so that we obtain the following theorem:

Theorem 2 (Temporal Logic and Automaton Hierarchy). *Given a formula $\Phi \in \mathsf{TL}_\kappa$, we can construct a deterministic ω-automaton $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ of the class Det_κ in time $O(2^{|\Phi|})$ with $|Q| \leq O(3^{|\Phi|})$ state variables. Therefore, $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ is a symbolic representation of a deterministic automaton with $2^{O(3^{|\Phi|})}$ states.*

2.5 Infinite Games

In this thesis, we reduce different problems to the solution of infinite two player games. In particular, in Chapter 3, the minimization problem of parity automata is reduced to the solution of so-called simulation games and the controller synthesis problem can be regarded as a two-player game between a system and its environment (see Section 2.5.2). The games considered in this thesis have in common that they can be reduced to a special form of two-player infinite games, namely turn-based games, where both players move alternately.

2.5.1 Turn-Based Games

Definition 15 (Two-Player Turn-Based Games). *A* two-player turn-based game *is a tuple $\mathfrak{G} = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, \mathcal{R}, \mathcal{L}, \Phi)$ over a set of state variables Q where*

- $\mathcal{V} = \mathcal{V}_0 \cup \mathcal{V}_1$ *is a finite set of* locations *or* vertices with $\mathcal{V}_0 \cap \mathcal{V}_1 = \emptyset$,
- $\mathcal{V}_i$ are the locations of $\mathcal{V}$ that correspond with player i (where $i \in \{0, 1\}$),
- $\mathcal{R} \subseteq \mathcal{V} \times \mathcal{V}$ is the *transition relation*,
- $\mathcal{L} : \mathcal{V} \to 2^Q$ is the label function and
- Φ is an acceptance condition.

A *play* is a maximal sequence of locations $\pi = v^{(0)}v^{(1)}\ldots$ such that for all $i \geq 0$, we have $(v^{(i)}, v^{(i+1)}) \in \mathcal{R}$. A play π is *winning* for player 0 if $\mathcal{L}(\pi) \models \Phi$ or π is finite and π ends in a state of player 1. Otherwise, player 1 wins.

A *strategy* for player 0 is a partial function $\gamma : \mathcal{V}^*\mathcal{V}_0 \to 2^{\mathcal{V}}$ such that whenever $f(\pi v)$ is defined and $v' \in \gamma(\pi v)$, we have $(v, v') \in \mathcal{R}$. We extend the label function to plays in the usual way. The strategy γ is called *memoryless*, if the successor state chosen by player 0 depends only on the current location v, i.e., γ is a function $\gamma : \mathcal{V}_0 \to 2^{\mathcal{V}}$. A strategy is said to be *finite memory* or *forgetful*, if there exists a finite set M, an initial memory element $m_\mathcal{I} \in M$, and functions $\tau : V \times M \to M$ and $\xi : V \times M \to 2^V$ such that the following holds: If $\pi = v^{(0)}v^{(1)}\ldots v^{(j-1)}$ is a prefix of a play and the sequence $m_0, m_1, \dots m_j$ is determined by $m_0 = m_{\mathcal{I}}$ and $m_{i+1} = \tau(v^{(i)}, m_i)$, then $\gamma(\pi v_j) = \xi(v_j, m_j)$ holds.

A play $\pi = v^{(0)}v^{(1)}\dots$ is *γ-conform*, if whenever $v^{(i)} \in \mathcal{V}_0$ holds, then we have $v^{(i+1)} \in \gamma(v^{(0)} \dots v^{(i)})$. The strategy γ is *winning* from v, if every γ-conform play that starts in v is winning for player 0. We say that player 0 wins from v if she has a winning strategy from v. The *winning region* of player 0 is the set of locations from which player 0 wins. We denote the winning region of player 0 by $\mathcal{W}_0$. Strategy, winning and winning regions are defined analogously for player 1.

We *solve* a game by computing the winning regions $\mathcal{W}_0$ and $\mathcal{W}_1$. The winning regions of the games that we consider are disjoint, i.e. these games are *determined* [39, 26].

The winning condition Φ specifies the type of the game, e.g. a Büchi condition specifies a Büchi game. Many algorithms have been developed to solve two-player turn-based games [37, 108, 89]. We are particularly interested in Büchi games [104] and generalized parity games [19] since the simulation games of Section 3 and the controller synthesis problem can be reduced to them.

2.5.2 Controller Synthesis as a Two-Player Game

As already outlined in the introduction, the controller synthesis problem solves a more general problem than the model checking problem. Instead of giving a yes/no answer or a counterexample to show how a specification is violated, the controller synthesis problem asks how the system can be modified to fulfill the specification, i. e. to construct a controller $\mathcal{C}$ such that $\mathcal{M} \times \mathcal{C} \models \Phi$. To this end, we assume that our original semi-automaton has some input variables that are termed *controllable.* Those controllable input variables represent choices of the controller whereas the other input variables are chosen by the environment. In this thesis, the controller synthesis problem is reduced to the decision of the winner in an infinite two player game between the controller and the environment of the system. To this end, the so called *Moore* game is defined. While solving the game, i. e. deciding whether the controller or the environment wins, a strategy $\mathcal{C}$ in the form of a Moore machine for the controller can be synthesized such that $\mathcal{M} \times \mathcal{C} \models \Phi$.

Intuitively, a Moore game is an incompletely specified finite state machine together with a specification, thus an ω-automaton over the alphabet $\Sigma = 2^{V_u \cup V_c}$. The environment chooses the uncontrollable inputs whereas the controller chooses the controllable inputs. To ease notation, let $\Sigma_c = 2^{V_c}$ and $\Sigma_u = 2^{V_u}$ be the controllable and uncontrollable input alphabet. Formally, the Moore game is introduced in the following:

Definition 16 (Moore Game). *A* Moore game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_{\mathcal{I}}, \delta, \Lambda, \Phi)$ *over a set of state variables* V_S *consists of the following components:*

- *a finite set of uncontrollable input variables V_u that are determined by the environment*
- *a finite set of controllable input variables V_c that are determined by the Controller*
- *a finite set of states $\mathcal{S}$*
- *an initial state $s_{\mathcal{I}} \in \mathcal{S}$,*
- *a (deterministic) transition function $\delta : \mathcal{S} \times \Sigma_c \times \Sigma_u \to \mathcal{S}$*
- *a labeling function $\Lambda : \mathcal{S} \to 2^{V_S}$*
- *a winning condition Φ given by an acceptance component over V_S.*

The challenge of the controller is to find proper values for the input variables V_c such that the specification Φ is satisfied, while the environment tries to force a violation of Φ. We look at Moore games from the view of the controller, however, the definitions of plays and strategies can be dually given for the environment, too.

Given a game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_{\mathcal{I}}, \delta, \Lambda, \Phi)$, a (finite state) strategy is a tuple $\gamma = (\mathcal{S}, t_{\mathcal{I}}, \tau)$ where $\mathcal{Q}$ is a finite set of states, $q_{\mathcal{I}} \in \mathcal{Q}$ is the initial state and $\tau : \mathcal{S} \times \mathcal{Q} \to 2^{(\Sigma_c \times \mathcal{Q})}$ is the move function. Intuitively, a strategy automaton is a Moore machine that fixes a set of possible responses to an environment input, and its response may depend on a finite memory of the past. Note that strategies are nondeterministic, but we can easily extract a deterministic strategy by fixing the strategy to one of the possible responses. The strategy is nondeterministic because this simplifies the symbolic calculation [8].

A *play* on $\mathfrak{G}$ respecting γ is an infinite sequence $\pi = (s^{(0)}, q^{(0)}) \xrightarrow{c^{(0)}, u^{(0)}} (s^{(1)}, q^{(1)}) \xrightarrow{c^{(1)}, u^{(1)}} ..$ such that $s^{(0)} = s_{\mathcal{I}}$, $q^{(0)} = q_{\mathcal{I}}$, $(c^{(i)}, q^{(i+1)}) \in \tau(s^{(i)}, q^{(i)})$ and $s^{(i+1)} = \delta(s^{(i)}, c^{(i)}, u^{(i)})$. A play is winning if π is infinite and $s^{(0)}, s^{(1)} \ldots \models \Phi$. Notice that if $\tau(q^{(i)}, s^{(i)}) = \emptyset$ the strategy does not suggest a proper choice of the input variables and the game is lost. A strategy γ is winning if all plays according to γ are winning. The set of all winning states is the *winning region.* A game $\mathfrak{G}$ is winning (or won) if the initial state $s_{\mathcal{I}}$ is in the winning region. A memoryless strategy is a finite state strategy with only one state. A memoryless strategy will be written as a function $\gamma : \mathcal{S} \to 2^{\Sigma_c}$ and a play of a memoryless strategy as a sequence $\pi = s^{(0)} \xrightarrow{c^{(0)}, u^{(0)}} s^{(1)} \xrightarrow{c^{(1)}, u^{(1)}} \ldots$.

We say that $\mathfrak{G}$ is *controllable* if the controller has a winning strategy. Otherwise we say that $\mathfrak{G}$ is *uncontrollable.* The type of winning condition defines the name of the game. Thus we speak e. g. of a Büchi game when dealing with a Moore game whose acceptance condition is a Büchi condition.

[8]Although possible in theory, we do not consider Mealy games here, since the product of two Mealy automata may lead to causality problems.

Solving Simple Moore Games

There are already algorithms for the solution of simple Moore games like Büchi games. Whenever those algorithms are symbolically implemented, they rely on some form of predecessor function. For example, the algorithms presented in [49, 110] use a predecessor function $\boxed{\Diamond}_c\mathcal{F} := \{s \mid \exists c \in \Sigma_c \forall u \in \Sigma_u.\ \delta(s,c,u) \in \mathcal{F}\}$ to calculate the set of states from which the controller can force a visit to a target set $\mathcal{F}$ in one step.

This pre-operator gives correct results for simpler games like Büchi or co-Büchi games where the calculation of the winner of the game depends only on one player and where the winner is always decided on the whole game graph. However, in a parity game, the winner of the game is determined by successively calculating attractors to the minimal color that occurs in a *subgame* $\mathcal{S}'$. Thus, we need predecessor functions for both the controller and the environment. A possible definition that also considers the subgame would be:

$$\boxed{\Diamond}_{c'}\mathcal{F} := \{s \mid \exists c \in \Sigma_c.\forall u \in \Sigma_u.\, (\delta(q,c,u) \in \mathcal{S}' \rightarrow \delta(q,u,c) \in \mathcal{F})\} \tag{2.1}$$

whereas the corresponding function for the Environment would be:

$$\boxed{\Diamond}_{u'}\mathcal{F} := \big\{s \mid \forall c \in \Sigma_c.\ \ (\exists u \in \Sigma_u.\ \delta(s,c,u) \in \mathcal{S}') \rightarrow (\exists u' \in \Sigma_u.\ \delta(s,c,u') \in \mathcal{S}')\ \big\}$$

To see that these predecessor functions give not the correct results for every parity game, consider the parity game in Figure 2.8 together with the coloring function $\Omega(q_0) = 0$, $\Omega(q_1) = 1$ and $\Omega(q_2) = 2$. Clearly, the controller can win this game by always choosing c. However, the predecessor function $\boxed{\Diamond}_{c'}$ and $\boxed{\Diamond}_{u'}$ give the wrong result that q_1 is winning for the environment. The problem is that after q_0 has been removed by the Attractor to the states with the minimal color 0, the remaining subgame contains only the states $\{q_1\}$ and $\{q_2\}$. In the iterative step of the (generalized) parity games in which the subgame $\{q_1, q_2\}$ is considered, the information that choosing c in $\{q_1\}$ is winning is lost. And no matter how we define a predecessor function that is solely based on states (and not on tuples (q, c) that store the move chosen by controller), we can find a game that gives a wrong result.

Another alternative algorithm for the solution of Moore games is given in [22] where a game μ-calculus is introduced. This game μ-calculus is the ordinary μ-calculus where the predecessor functions are replaced by predecessor functions similar as in equation (2.1). Then it is shown that the same μ-calculus formula can be used for both model-checking and for game-solving provided some syntactic criterion on the μ-calculus formula is satisfied. It is moreover shown how parity conditions can be translated to the game μ-calculus so that a solution of parity games is possible. This game μ-calculus formula describes a symbolic algorithm for the solution of parity games that suffers not from the above sketched problem with subgames. However

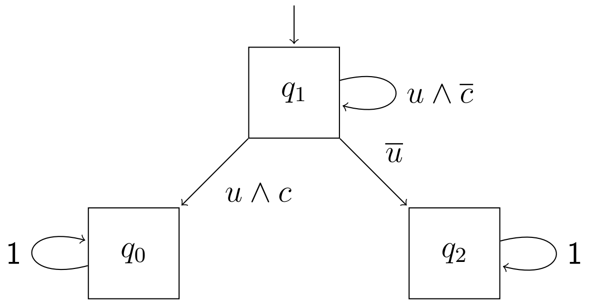


Figure 2.8: A Parity Game for which simple Pre-Operators fail

one particular drawback of this syntactic criterion is that arbitrary conjunctions are not possible. Therefore a straightforward solution of Streett games or generalized conjunctive parity games can not be obtained using the approach of [22].

Hence, instead of providing a specialized pre-operator, we use a reduction to turn-based games to solve Moore games. This reduction is straightforwardly obtained from related constructions given in [28].

Reducing Moore Games to Turn-Based Games

In the following, we show how every Moore game can be transformed to a turn-based game.

Definition 17 (Turn-Based Moore Game).
For every Moore game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_{\mathcal{I}}, \delta, \mathcal{L}, \Phi)$ we define the turn-based controller Game $\mathfrak{G}_t = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, \mathcal{R}, \mathcal{L}', \Phi)$ as follows:

- $\mathcal{V}_1 = 2^{V_S}$
- $\mathcal{V}_0 = \{(s,c) \mid s \in \mathcal{S} \wedge c \in \Sigma_c\}$
- $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_0$
- $\mathcal{R} = \left(\begin{array}{l} \{(s,(s,c))\} \cup \\ \{((s,c),s') \mid \exists c \in \Sigma_c . \delta(s,c,u) = s'\} \end{array} \right)$
- $\mathcal{L}'(s) = \mathcal{L}(s)$ *and* $\mathcal{L}'(s,c) = \mathcal{L}(q)$

Thus, the positions of Player 0, the controller are the states of the Moore game and environment starts from vertices (s,c) so that the controllable event chosen by the controller is memorized. The following is easily seen:

Proposition 2. *Given a Moore game and its corresponding turn-based game. Then Controller wins the Moore game from position $s_\mathcal{I}$ if and only if Player 0 wins from position $s_\mathcal{I}$ in the turn-based game.*

Thus we can use the large arsenal of algorithms for infinite turn-based games [37, 108, 89] also to solve Moore games[9].

2.5.3 LTL-Games

The scope of this thesis are specifications of the Moore game that are given as LTL formulas. To use the algorithms from the literature to solve infinite games, i. e. Büchi or (generalized) parity algorithms, we have to translate this LTL formulas to a deterministic automaton. The state space of this deterministic automaton is then combined with the original Moore game according to the following definition:

Definition 18 (Product Games). *Given a Moore game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_\mathcal{I}, \delta, \mathcal{L}, \Phi)$ over a set of variables V_S and a deterministic ω-automaton $\mathfrak{A} = (\mathcal{Q}, \{q_\mathcal{I}\}, \xi, \lambda, \varphi)$ whose input alphabet is V_S such that for every path π over V_S we have that $\pi \models \Phi$ iff $\pi \models \mathfrak{A}$, we define the product game*
$\mathfrak{G} \times \mathfrak{A} = (V_u, V_c, \mathcal{S} \times \mathcal{Q}, (s_\mathcal{I}, q_\mathcal{I}), \delta', \lambda, \varphi)$ with $\delta'((s,q),u,c) = (\delta(s,c,u), \xi(q,u,c))$

Notice that the product game is a Moore game as before and that the controller is winning from the initial state $s_\mathcal{I}$ if and only if it is winning from the initial state $(s_\mathcal{I}, q_\mathcal{I})$ of the product game. Thus, an efficient solution to the controller synthesis problem for LTL can be reduced to the task of finding deterministic ω automata that are equivalent to a given LTL formula. We thus consider efficient (symbolic) determinization procedure in in Chapters 4 and 5. However, our considerations for an efficient generation of deterministic ω-automata starts in Chapter 3 where we present techniques to efficiently minimize parity automata. Since all our intermediate automata of the determinization procedures can be easily translated to a parity automaton, this is an important step in our overall algorithm.

2.6 Essentials of the Synchronous Programming Language Quartz

Synchronous languages are becoming more and more attractive for the design and the verification of reactive real time systems. There are imperative languages like *Esterel*

[9]In a symbolic implementation adding the controllable variables as one part of the state space is not harmful, essentially the transition relations stay the same.

[9], data flow languages like *Lustre* [43], and graphical languages like some Statechart [44] variants as *SyncCharts* [4].

Synchronous languages are very appealing for the design of reactive systems since they provide language constructs for parallel execution of processes. Moreover, they are usually equipped with a formal semantics which allows both analysis with e. g. verification or controller synthesis but also makes it easy to automatically obtain hardware from a synchronous program.

For our experiments, we used the synchronous language *Quartz* [92] to develop the system for the controller synthesis game. From the quartz program we obtain a symbolic description of a Moore automaton that exactly describes the behavior of the Quartz program. Hence, although we use Quartz as a way to develop the systems, the algorithms developed during this thesis are general in the sense that they only assume a description of the system as a transition system. Nevertheless, for understanding some of the examples, a basic knowledge of the constructs of Quartz may be useful.

The common paradigm of synchronous languages is the perfect synchrony [42, 7] which means that the execution of programs is divided into macro steps that are usually interpreted as logical time. As this logical time is the same in all concurrent threads, these threads run in lockstep, which leads to a deterministic form of concurrency. Macro steps are divided into finitely many micro steps that are atomic actions of the programs. Moreover, variables change synchronously in macro steps, i.e., variables have unique values in each macro step.

In the following, we give a brief overview of the synchronous programming language Quartz. We do however not describe the entire language, and refer instead to [92]. Provided that S, S_1, and S_2 are statements, ℓ is a location variable, x is a variable, σ is a boolean expression, and α is a type, then the following are statements (keywords given in square brackets are optional):

- *nothing* (empty statement)
- $x = \tau$ and $next(x) = \tau$ (assignments)
- $assume(\varphi)$ (assumptions)
- $assert(\varphi)$ (assertions)
- ℓ : *pause* (start/end of macro step)
- *if* (σ) S_1 *else* S_2 (conditional)
- $S_1; S_2$ (sequential composition)
- *do* S *while*(σ) (iteration)

- $S_1 \parallel S_2$ (synchronous concurrency)
- *[weak] [immediate] abort* S *when* (σ) (abortion)
- *[weak] [immediate] suspend* S *when* (σ) (suspension)
- $\{\alpha\ x;\ S\}$ (local variable y with type α)

The *pause* statement defines a control flow location ℓ which is a boolean variable that is true iff the control flow is currently at the statement ℓ : *pause*. Since all other statements are executed in zero time, the control flow can only rest at these positions in the program, and therefore the possible control flow states are the subsets of the set of locations.

There are two variants of assignments; and both evaluate the right-hand side τ in the current macro step. While immediate assignments $x = \tau$ immediately transfer the value of τ to the left-hand side x, delayed assignments $\mathit{next}(x) = \tau$ transfer this value in the following step.

In case the value of a variable is not determined by an assignment, a default value is determined by the declaration of the variable. To this end, declarations provide a storage class in addition to the type of a variable. There are two storage classes, namely *mem* and *event* that choose the previous value or a default value (determined by the type), respectively, in case no assignment determines the value of a variable. Available types are booleans, signed and unsigned integers (both with limited and unlimited bounds), bitvectors, as well as arrays and tuples of types.

In addition to the control flow constructs that are well-known from other imperative languages like conditionals, sequences and loops, Quartz offers synchronous concurrency $S_1 \parallel S_2$ and sophisticated preemption and suspension statements, as well as further statements to allow comfortable descriptions of reactive systems (see [92] for the complete syntax and semantics). In $S_1 \parallel S_2$, both substatements S_1 and S_2 must run in lockstep as long as both are active, and the statement terminates when the last one of these substatements terminates. Preemption statements attach a guard to a statement and either suspend or abort it if that guard is true. The immediate forms do already check for preemption at starting time, while the default is to check the preemption only after starting time. The weak variants allow all actions of the data flow to take place even at the time of preemption, while the strong variant forbids them at the time of preemption.

As mentioned above, synchronous programs offer many advantages for system design, but they also challenge the compilers: In particular, schizophrenia and causality problems must be addressed. Schizophrenia problems occur if a statement is (re)-started at the time of its termination, so that the compiler has to keep track of different incarnations of the statement. In particular, local variable declarations have to be handled with some care to distinguish different incarnations of the local variables. Causality problems occur if an action modifies variables that are responsible for triggering the execution of the action. Hence, an action may disable its execution or it may justify its execution, which are both unwanted behaviors. The causality analysis has to determine that the actions of a synchronous program can be executed in a causal order where all trigger conditions are determined before the corresponding actions are executed.

3 Minimization of ω-Automata

The algorithms for the solution of LTL controller synthesis problems presented in this thesis make use of different intermediate ω-automata. The approach of Chapter 4 translates fragments of LTL to nondeterministic co-Büchi automata which are then determinized using the breakpoint construction [70] while our second determinization procedure from Chapter 5 translates nondeterministic Büchi automata to deterministic parity automata. Since both determinization procedures are in the worst case exponential, minimization procedures are especially beneficial for the nondeterministic automata. Although the size of symbolic representations like BDDs are not directly related to the number of states, the experiments in Chapter 7 confirm that controller synthesis of real life designs is nearly impossible using non-minimized automata whereas it becomes feasible when we use the minimization techniques presented in this chapter.

In contrast to deterministic finite automata on finite words, there is in general no unique minimal automaton for ω-automata. Moreover, even for those automata that belong to the class of deterministic weak automata and for which we know that a minimal automaton exists, the problem is known to be PSPACE-hard (this is even true for nondeterministic automata on finite words). Therefore, heuristics are used to minimize the state space of ω-automata. In this chapter, we study simulation relations [69] as a tool for minimizing automata. Simulation relations capture the notion that the moves of one automaton can be mimicked by the moves of another automaton. Thus simulation relations can be used to check language inclusion between automata [23] and can thus be used for model-checking where both the system and the specification are given as automata. We are interested in simulation relations because they can be used to efficiently minimize ω-automata.

There are quite many works related with the minimization problem of ω-automata. Somenzi and Bloem consider in [99] direct and reverse simulation for Büchi automata and use them to minimize nondeterministic automata obtained from LTL formulas. In [40], a relaxed notation, termed fair simulation, is considered. Since it is known that fair simulation is not language-preserving under quotient construction, the authors use additional checks to calculate a language equivalent minimized Büchi automaton. Etessami, Wilke and Schuller introduce in [28] an intermediate notation of simulation, called delayed simulation, that is finer than direct simulation but not as coarse as fair simulation. This intermediate simulation relation produces a language equivalent

Büchi automaton under quotient construction.

In [33, 30, 31] it is shown how the results regarding delayed simulation can be generalized for alternating Büchi automata. Finally, in [34, 32] delayed simulation is introduced for alternating parity automata. Since nondeterministic parity automata are special cases of alternating parity automata, the results of [34, 32] do also apply here.

In this chapter, the remaining cases for (nondeterministic) parity automata are considered. First, direct simulation is introduced for parity automata. Although direct simulation is weaker than delayed simulation (it allows less minimization), we use it as a preprocessing step, since it can be calculated in linear time whereas delayed simulation needs quadratic time. The second contribution of this chapter is the presentation of reverse simulation for parity automata: First, in the style presented in [99] of a direct simulation where a good transition in the simulated automaton must be directly matched by a good transition in the simulating automaton. Second, also in the relaxed notation of delayed simulation where this match of a good transitions is post-poned[1].

Since fair simulation has been shown to be the most efficient simulation relation for state space minimization of Büchi automata, the last paragraph of this chapter considers fair simulation minimization of parity automata. However, since we are particularly interested in algorithms that are well suited for a symbolic implementation, we do not consider a direct generalization of results obtained in [40] regarding the minimization of Büchi automata. The approach of [40] uses backtracking on individual states which is not amenable for a symbolic implementation. Instead, in Section 3.7 a surprisingly simple condition is introduced that can be used to iteratively minimize an automaton using fair simulation with intermediate checks that the result is correct.

The next section starts with the introduction of simulation relations as games, following related constructions given in [28].

3.1 Simulation Relations as a Two-Player Game

Following related work, we define simulation relations via infinite games. To handle the parity condition appropriately, we will first define an ordering on the natural numbers (also used e. g. in [32]) that takes the parity condition into account.

Definition 19 (Reward Order). *The reward order $\leq_\Omega \subset \mathbb{N} \times \mathbb{N}$ is defined as follows: $m \leq_\Omega n$ if and only iff*

- *m is even and n is odd, or*

[1]For Büchi automata, the two notations are equivalent.

- *m and n are even and $m \leq n$*
- *m and n are odd and $n \leq m$*

for all $m, n \in \mathbb{N}$.

That means, $0 <_\Omega 2 <_\Omega 4 <_\Omega \ldots <_\Omega 5 <_\Omega 3 <_\Omega 1$. Thus, every even number is better than every odd number and while the even numbers are ordered according to the standard order on $\mathbb{N}$, the order for the odd numbers is reversed. We will also phrase $n <_\Omega m$ as n is *better* than m, while terms like *minimum* and *smaller then* will always be used with respect to the standard order $\leq$.

For the remainder of this chapter, it is assumed that all dead-ends have been removed from the corresponding automata, i. e. given a parity automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ it must hold that for every state $s \in \mathcal{S}$ there must exist an input $a \in \Sigma$ and a successor state $s' \in \mathcal{S}$ so that $(s, a, s') \in \mathcal{R}$ holds.

3.1.1 The Basic Simulation Game

Following [28, 32], the simulation relations are defined via a game between two Players, the Spoiler and the Duplicator. Spoiler seeks to show that automaton $\mathfrak{A}$ is not simulated by $\mathfrak{A}'$. While performing this task, Spoiler controls the simultaneous inputs for both automata and the nondeterministic choices of $\mathfrak{A}$ while Duplicator tries to mimic each move of Spoiler through the nondeterministic choices that $\mathfrak{A}'$ allows. This intuitive idea is captured by the following definition:

Definition 20 (Basic Game). *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega), \mathfrak{A}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \Omega')$ be two parity automata over the alphabet Σ and $s^{(0)}$ and $s'^{(0)}$ be arbitrary states of $\mathfrak{A}$ and $\mathfrak{A}'$ respectively. The* basic game $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$ *is played by two players,* Spoiler *and* Duplicator*, in rounds, where, at the beginning of and at the end of each round, two pebbles,* Red *and* Blue*, are placed on two states. At the start round 0, Red and Blue are placed on $s^{(0)}$ and $s'^{(0)}$, respectively. Assume that at the beginning of round i, Red is on state $s^{(i)}$ and Blue is on $s'^{(i)}$. Then:*

1. *Spoiler chooses a letter $\alpha^{(i)} \in \Sigma$.*
2. *Spoiler chooses a transition $(s^{(i)}, \alpha^{(i)}, s^{(i+1)}) \in \mathcal{R}$ and moves Red to $s^{(i+1)}$.*
3. *Duplicator, responding, must choose a transition $(s'^{(i)}, \alpha^{(i)}, s'^{(i+1)}) \in \mathcal{R}'$ labeled by the same input letter and moves Blue to $s'^{(i+1)}$. If no $\alpha^{(i)}$-transition starting from $s'^{(i)}$ exists, then the game halts and Spoiler wins.*
4. *The starting position for the next round is $(s^{(i+1)}, s'^{(i+1)})$.*

Intuitively, Spoiler produces, letter by letter, an ω-word over Σ as simultaneous inputs for the two automata $\mathfrak{A}$ and $\mathfrak{A}'$. Spoiler controls the nondeterministic choices of $\mathfrak{A}$ while Duplicator controls the nondeterministic choices of $\mathfrak{A}'$. The first round begins with the pair $(s^{(0)}, s'^{(0)})$. If, at any point during the course of the game, Duplicator cannot proceed any more, she looses early. This corresponds to the case that from $s^{(0)}$ there exists a word α that labels a run starting in $s^{(0)}$, but no run of $\mathfrak{A}'$ labeled with α starts in $s'^{(0)}$. When the player proceed as above and Duplicator does not loose early, the players produce an infinite sequence $(s^{(0)}, \alpha^{(0)}, s'^{(0)})(s^{(1)}, \alpha^{(1)}, s'^{(1)})\ldots$ from $(\mathcal{S} \times \Sigma \times \mathcal{S}')^\omega$ that induce two runs $\pi = s^{(0)}s^{(1)}\ldots$ of $\mathfrak{A}$ and $\pi' = s'^{(0)}s'^{(1)}\ldots$ of $\mathfrak{A}'$, respectively. This sequence determines the winner, depending on the type of simulation we are interested in and is thus denoted with *outcome* of the game.

Definition 21 (Simulation Games)**.** *Let $\mathfrak{A}$ and $\mathfrak{A}'$ be parity automata, let $(s^{(0)}, s'^{(0)}) \in \mathcal{S} \times \mathcal{S}'$ and let $(s^{(0)}, \alpha^{(0)}, s'^{(0)})(s^{(1)}, \alpha^{(1)}, s'^{(1)})\ldots$ be an outcome of $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$. Let $\pi = s^{(0)}s^{(1)}\ldots$ and $\pi' = s'^{(0)}s'^{(1)}\ldots$.*

1. *The direct (strong) simulation game, denoted by $\mathfrak{G}_{di}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$, is the basic game $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$ extended by the rule that Duplicator is winning iff, for all i, $\Omega(s^{(i)}) \geq_\Omega \Omega'(s'^{(i)})$.*

2. *The left-hand delayed simulation game, denoted by $\mathfrak{G}_{\mathsf{de_l}}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$, is the basic game $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(i)}, s'^{(i)}\right)$ extended by the rule that the outcome is winning for Duplicator iff for all i, if $\Omega(s^{(i)}) <_\Omega \Omega(s'^{(i)})$, then for some $j > i$ we have*
   - *$\Omega(s^{(j)})$ is odd and*
   - *$\Omega(s^{(j)}) \leq \min\{\Omega(s^{(i)}), \Omega(s'^{(i)})\}$*

3. *The right-hand delayed simulation game, denoted by $\mathfrak{G}_{\mathsf{de_r}}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$, is the basic game $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(i)}, s'^{(i)}\right)$ extended by the rule that the outcome is winning for Duplicator iff for all i, if $\Omega(s^{(i)}) <_\Omega \Omega(s'^{(i)})$, then for some $j > i$ we have*
   - *$\Omega(s'^{(j)})$ is even and*
   - *$\Omega(s'^{(j)}) \leq \min\{\Omega(s^{(i)}), \Omega(s'^{(i)})\}$*

4. *The delayed simulation game,denoted by $\mathfrak{G}_{\mathsf{de}}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$, is the basic game $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(i)}, s'^{(i)}\right)$ extended by the rule that the outcome (π, π') is winning for Duplicator iff*

   for all i, if $\Omega(s^{(i)}) <_\Omega \Omega(s'^{(i)})$, then for some $j > i$ we have
   - *$\Omega(s^{(j)})$ is odd and $\Omega(s^{(j)}) \leq \min\{\Omega(s^{(i)}), \Omega(s'^{(i)})\}$ or*
   - *$\Omega(s'^{(j)})$ is even and $\Omega(s'^{(j)}) \leq \min\{\Omega(s^{(i)}), \Omega(s'^{(i)})\}$*

5. *The fair simulation game, denoted by* $\mathfrak{G}_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$*, is the basic game* $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(i)}, s'^{(i)}\right)$ *extended by the rule that Duplicator is winning iff if* $\mathsf{MIN}_\Omega(\pi)$ *is even, we have* $\mathsf{MIN}'_\Omega(\pi')$ *is even.*

Direct simulation is more or less the direct adaption of direct simulation from Büchi automata to parity automata. That is, whenever Spoiler visits a state, simultaneously Duplicator must visit a state that is better according to the ordering $<_\Omega$. In particular, whenever Spoiler visits a state s with even color, Duplicator must visit a state s' with an even color such that the color of s' is smaller than the color of s. If Spoiler visits a state with odd color, Duplicator must choose a state with even color or a state with a bigger color. This ensures that whenever $\mathfrak{A}$ accepts, then also $\mathfrak{A}'$ accepts.

Delayed simulation comes in three variants: Whenever we found that Spoiler visits a better state than Duplicator, i. e. that $\Omega(s^{(i)}) <_\Omega \Omega(s'^{(i)})$, this bad event must be recovered later on. Left-hand delayed simulation waits for a smaller odd color on the side of Spoiler (the left hand side) which ensures that the color of state $s^{(i)}$ has no effect. Similar, right-hand delayed simulation waits for a smaller even color on the right side so that the color $s'^{(i)}$ has no effect. Finally, (full) delayed simulation combines the two by allowing this recovering on both sides.

The most relaxed notation is fair simulation. Here a play is won for Duplicator iff either the minimal color visited infinitely often by Spoiler is odd (i. e. $\mathfrak{A}$ does not accept) or the minimal color visited infinitely often by Duplicator is also even (i. e. $\mathfrak{A}'$ does accept).

Having defined the games, we can in a straightforward way define strategies. The idea of strategies is sketched in Figure 3.1. In the first round, the strategy γ defines the (uniquely defined) successor state $s'^{(0)}$ based only on the actual state where Spoiler is in. In the second round, all information gained so far during the game, i. e. $s^{(0)}, s'^{(0)}, \alpha^{(0)}, s^{(1)}$ are used to define the successor state $s'^{(1)} = \gamma(s^{(0)}s'^{(0)}\alpha^{(0)}s^{(1)})$ and so on. Finally, the decision which successor to choose for Duplicator is based on all information gathered so far, i. e. in that case we have $s'^{(i+1)} = \gamma(s^{(0)}s'^{(0)}\alpha^{(0)}s^{(1)}s'^{(1)}, \alpha^{(1)} \dots \alpha^{(i)}s^{(i+1)})$ [2]. If we have a memoryless strategy, the decision is based simply on the actual state where Spoiler is in, the input Spoiler chooses and the state Duplicator is in, which means that in the figure the two edges from … to γ disappear.

This intuitive idea is formalized in the following definition:

Definition 22 (Strategy for Simulation Games). *Let* $\star$ *in* $\{\mathsf{di}, \mathsf{de}, \mathsf{de}_\mathsf{l}, \mathsf{de}_\mathsf{r}, \mathsf{f}\}$. *A* strategy *for Duplicator in Game* $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$ *is a partial function* $\gamma : \mathcal{S}(\mathcal{S}'\Sigma\mathcal{S})^* \to \mathcal{S}'$ *which, given the history of the game up to a certain point, determines the next move of Duplicator. Formally,* γ *is a strategy for Duplicator if* $\gamma(s^{(0)}) = s'^{(0)}$ *and*

[2]We did not draw every edge to the last γ to simplify the picture.

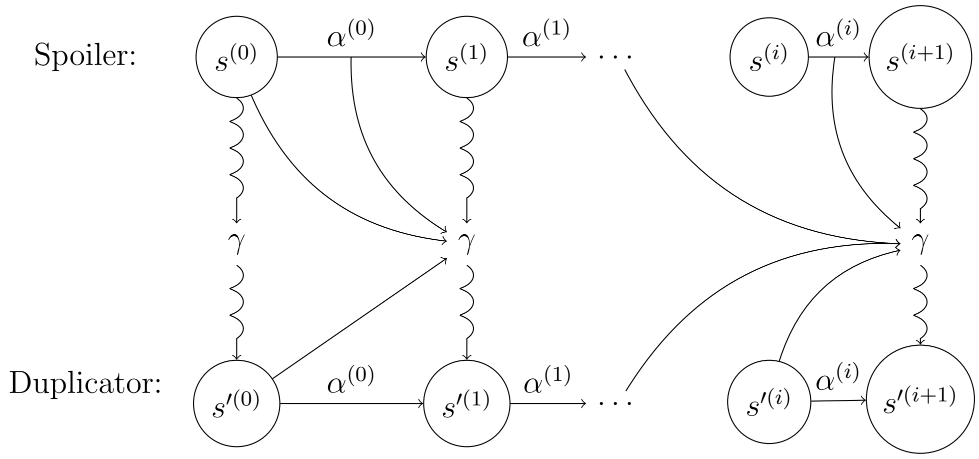


Figure 3.1: Defining Strategies for the Basic Simulation Game

$(s'^{(i)}, \alpha^{(i)}, s'^{(i)}) \in \mathcal{R}'$ holds for every sequence $s^{(0)}s'^{(0)}\alpha^{(0)}s^{(1)}s'^{(1)}\alpha^{(1)}, \dots \alpha^{(i-1)}s^{(i)}$ with $(s^{(j)}, \alpha^{(j)}, s^{(j+1)}) \in \mathcal{R}$ and $s^{(j)} = \gamma(s^{(0)}s'^{(0)}\alpha^{(0)}s^{(1)}s'^{(1)}\alpha^{(1)} \dots \alpha^{(j-1)}s^{(j)})$ for every $j \leq i$. A strategy is memoryless, if whenever $\eta, \eta' \in \mathcal{S}(\mathcal{S}'\Sigma\mathcal{S})^*$ and $\gamma(\eta) = \gamma(\eta') = s'$ it holds that for every $s \in \mathcal{S}$ and every $a \in \Sigma$ we obtain $\gamma(\eta s' sa) = \gamma(\eta' s' sa)$. i. e. Duplicator chooses the next state only based on the current state she is in and on the input and next state chosen by Spoiler. In this case, we usually write γ as a function $\gamma : \mathcal{S}' \times \Sigma \times \mathcal{S} \to \mathcal{S}'$. Strategies for Spoiler are defined analogously.

Observe that the existence of a strategy implies that Duplicator has a way of playing such that the game does not halt. A strategy γ for Duplicator is a *winning strategy* if, no matter how Spoiler plays, Duplicator always wins. Formally, a strategy γ for Duplicator is winning if for all $\alpha = \alpha^{(0)}\alpha^{(1)} \dots$ whenever $\pi = s^{(0)}s^{(1)} \dots$ is a run through $\mathfrak{A}$ and $\pi' = s'^{(0)}s'^{(1)}s'^{(2)} \dots$ is the run defined by

$$s'^{(i+1)} = \gamma(s^{(0)}s'^{(0)}\alpha^{(0)}s^{(1)}s'^{(1)}\alpha^{(1)} \dots \alpha^{(i)}, s^{(i+1)}) \tag{3.1}$$

then (π, π') is winning for Duplicator (as specified in Definition 21).

Intuitively, an automaton $\mathfrak{A}'$ simulates another automaton $\mathfrak{A}$, iff Duplicator has a winning strategy in the simulation game. That is, she can mimic every run generated by Spoiler over $\mathfrak{A}$ with a corresponding run of $\mathfrak{A}'$.
In the following, if not otherwise stated, $\star$ is always a element from $\{\mathsf{di}, \mathsf{de}, \mathsf{de_l}, \mathsf{de_r}, \mathsf{f}\}$.

Definition 23 (Simulation Relations)**.** *Let $\mathfrak{A}$, $\mathfrak{A}'$ be parity automata. A state s' of $\mathfrak{A}'$ $\star$ simulates a state s of $\mathfrak{A}$, iff there is a winning strategy for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A}')}(s, s')$. We denote such a relationship by $s \preceq_\star^{(\mathfrak{A},\mathfrak{A}')} s'$. If $\mathfrak{A}, \mathfrak{A}'$ are clear from the context, we sometimes omit them. Finally, we define $s \approx_\star^{(\mathfrak{A},\mathfrak{A}')} s'$ iff $s \preceq_\star^{(\mathfrak{A},\mathfrak{A}')} s'$ and $s' \preceq_\star^{(\mathfrak{A}',\mathfrak{A})} s$. In* *that case we say that s and s' are simulation equivalent. We will later show that this is an equivalence relation.*

An automaton $\mathfrak{A}$ is simulated by another automaton $\mathfrak{A}'$, if for every $s_i \in \mathcal{I}$ some $s_i' \in \mathcal{I}'$ exists such that $s_i \preceq_\star^{(\mathfrak{A},\mathfrak{A}')} s_i'$. We denote such relationships between automata with $\mathfrak{A} \preceq \mathfrak{A}'$. Moreover, we define $\mathfrak{A} \approx_\star \mathfrak{A}'$ if $\mathfrak{A} \preceq \mathfrak{A}'$ and $\mathfrak{A}' \preceq \mathfrak{A}$ holds.

Delayed simulation has been introduced in [32, 34] and has been used for the minimization of alternating parity automata. Since alternating automata are more general than nondeterministic automata, their results do also apply here. For this reason, delayed simulation will not be considered here in detail. However, since the proofs for delayed left-hand and reverse delayed left-hand simulation are completely analogous in some parts, we will also demonstrate some properties of delayed left-hand simulation. This also sheds a light on the difference and commons of the different forms of simulation relations.

We will start our considerations by a reduction of the simulation games to turn-based games. This enables us to use the well-known algorithms for turn-based games for the solution of the simulation games.

3.1.2 Simulation Games as Turn-Based Games

Although the simulation games introduced before have some similarity to the turn-based games we introduced in Section 2.5.1 they differ in the way how the winner of a game is defined. Thus, in the following, a reduction of simulation games to the infinite games introduced in previous sections is presented. The basic games can be translated to a turn-based game with the techniques presented in [28]:

Definition 24 (Turn-Based Game for the Basic Simulation Game)**.**
Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$, $\mathfrak{A}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \Omega')$ be two parity automata over the alphabet Σ. We define a turn-based game $\mathfrak{G} = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, E, \Phi)$ to simulate the basic simulation game as follows:

$$\begin{aligned}
\mathcal{V}_0 &= \{v_{(s,s',a)} \mid s \in \mathcal{S} \wedge s' \in \mathcal{S}' \wedge \exists r.\ (r,a,s) \in \mathcal{R}\} \\
\mathcal{V}_1 &= \{v_{(s,s')} \mid s \in \mathcal{S} \wedge s' \in \mathcal{S}'\} \\
E &= \{(v_{(s_1,s_1',a)}, v_{(s_1,s_2')}) \mid (s_1', a, s_2') \in \mathcal{R}'\} \\
&\qquad \cup \{(v_{(s_1,s_1')}, v_{(s_2,s_1',a)}) \mid (s_1, a, s_2) \in \mathcal{R}\}
\end{aligned}$$

Intuitively, Duplicators role is taken by player 0 in the turn-based game. Consider a simulation game that has reached position (r, r'). Simultaneously, the turn-based game has reached position $v_{(r,r')}$ which is a player 1 (Spoilers) node. Now Spoiler chooses a transition $(r, a, s) \in \mathcal{R}$ and moves the red token to s. In the turn-based game, player 1 goes to node $v_{(s,r',a)}$ which is a player 1 node. From that node, Duplicator (player 1) chooses a transition $(r', a, s') \in \mathcal{R}'$ and moves to node $v_{(s,s')}$.

So far we have said nothing about the acceptance condition. This acceptance condition is defined to respect the acceptance of the respective simulation game. Clearly, a direct simulation game is a safety game with the safety condition that in each round i, we have $\Omega(s^{(i)}) \geq_\Omega \Omega'(s'^{(i)})$. Moreover, it is not hard to see that the fair simulation game is a disjunctive generalized parity game with the disjunctive parity condition $[\overline{\Omega}, \Omega']$, i. e. Duplicator wins whenever either the parity condition Ω of $\mathfrak{A}$ is not satisfied or the parity condition Ω' of $\mathfrak{A}'$ is satisfied.

A slight modification of Lemma 4 of [28] shows that the number of edges of the turn-based game is $O(mn)$ where $n = \max\{|\mathcal{S}|, |\mathcal{S}'|\}$, $m = \max\{|\mathcal{R}|, |\mathcal{R}'|\}$. Thus the following proposition follows directly from the corresponding complexity bounds of safety and generalized parity games given e. g. in [28, 19].

Proposition 3. *Let $\mathfrak{A}, \mathfrak{A}'$ as before. Let $n = \max\{|\mathcal{S}|, |\mathcal{S}'|\}$, $m = \max\{|\mathcal{R}|, |\mathcal{R}'|\}$, $d = \max\{|\Omega|, |\Omega'|\}$. Then the greatest direct simulation relation $\preceq_{\mathsf{di}}^{(\mathfrak{A},\mathfrak{A}')}$ can be calculated symbolically in time $O(nm)$ and the greatest fair simulation relation $\preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')}$ can be calculated symbolically in time $O((nm)^{4d+1}) \cdot \frac{2d!}{d!^2}$.*

3.2 Basic Properties of Simulation Relations

In the following, some basic properties of simulation relations are proved, in particular that the simulation equivalence is really an equivalence relation. After that, three different quotient constructions are given. The following lemma is a generalization of Proposition 3 from [28]:

Lemma 1. *Let $\mathfrak{A}$ be a parity automaton.*

1. *For $\star$ in $\{\mathsf{di}, \mathsf{de_l}, \mathsf{de_r}, \mathsf{de}, \mathsf{f}\}$, $\preceq_\star^{(\mathfrak{A},\mathfrak{A})}$ is a reflexive, transitive relation.*
2. *The simulation relations are ordered by containment : $\preceq_{\mathsf{di}} \subseteq \preceq_{\mathsf{de_r}} \subseteq \preceq_{\mathsf{de}} \subseteq \preceq_{\mathsf{f}}$ and $\preceq_{\mathsf{di}} \subseteq \preceq_{\mathsf{de_l}} \subseteq \preceq_{\mathsf{de}} \subseteq \preceq_{\mathsf{f}}$*
3. *For $\star \in \{\mathsf{di}, \mathsf{de_l}, \mathsf{de_r}, \mathsf{de}, \mathsf{f}\}$, if $q \preceq_\star s'$, then $\mathsf{Lang}(\mathfrak{A}_s) \subseteq \mathsf{Lang}(\mathfrak{A}_{s'})$*
4. *For $\star$ in $\{\mathsf{di}, \mathsf{de_r}, \mathsf{de_l}, \mathsf{de}, \mathsf{f}\}$, $\approx_\star^{(\mathfrak{A},\mathfrak{A})}$ is an equivalence relation.*

Proof. To ease notation, we omit the $(\mathfrak{A}, \mathfrak{A}')$ and denote the simulation relation with $\preceq_\star$

1. Reflexivity is obvious. To show transitivity, suppose that $s^{(i)} \preceq_\star s'^{(i)} \preceq_\star s''^{(i)}$. for some $\star \in \{\mathsf{di}, \mathsf{de_l}, \mathsf{de_r}, \mathsf{de}, \mathsf{f}\}$. Then, by definition, Duplicator has winning strategies in the games $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s,s')$ and $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s',s'')$, say γ and γ'. We combine these to get a winning strategy γ'' for duplicator in the game $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s,s'')$ as follows: if
$$\gamma(s^{(0)}s'^{(0)}\alpha^{(0)}s^{(1)}s'^{(1)}\alpha^{(1)}\dots s'^{(i)}\alpha^{(i)}s^{(i+1)}) = s'^{(i+1)} \tag{3.2}$$
and
$$\gamma'(s'^{(0)}s''^{(0)}\alpha^{(0)}s'^{(1)}s''^{(1)}\alpha^{(1)}\dots s''^{(i)}\alpha^{(i)}s'^{(i+1)}) = s''^{(i+1)}$$
then we define
$$\gamma''(s^{(0)}s''^{(0)}\alpha^{(0)}s^{(1)}s''^{(1)}\alpha^{(1)}\dots s''^{(i)}\alpha^{(i)}s^{(i+1)}) = s''^{(i+1)}. \tag{3.3}$$
It is easy to see that this defines a strategy for Duplicator. To see that γ'' is in fact winning, let $\pi = s^{(0)}s^{(1)}\dots$ be a run through $\mathfrak{A}$ over $\alpha^{(0)}\alpha^{(1)}\dots$ and let $\pi'' = s''^{(i)}s''^{(1)}\dots$ be the run defined by equation (3.3). We need to argue that (π,π'') is winning for Duplicator. By induction, one easily proves that if $\pi' = s'^{(i)}s'^{(1)}\dots$ is defined by (3.1), then $s''^{(i+1)}$ is totally defined by γ', i. e.
$$s''^{(i+1)} = \gamma'(s'^{(0)}s''^{(0)}\alpha^{(0)}s'^{(1)}s''^{(1)}\alpha^{(1)}\dots s''^{(i)}\alpha^{(i)}s'^{(i+1)}).$$
This means that (π,π') is winning for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}\left(s^{(i)},s'^{(i)}\right)$ and (π',π'') is winning for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}\left(s'^{(i)},s'^{(i)}\right)$. For instance, when $\star = f$, this implies the following: if π is an accepting run of $\mathfrak{A}_{s^{(i)}}$, i. e. $\mathsf{MIN}_\Omega(\pi)$ is even, then necessarily $\mathsf{MIN}_\Omega(\pi')$ is even since Duplicator wins in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}\left(s^{(i)},s'^{(i)}\right)$. This implies that also $\mathsf{MIN}_\Omega(\pi'')$ must be even, since otherwise γ' would be no winning strategy in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}\left(s'^{(i)},s'^{(i)}\right)$. A similar argument shows the case for direct and delayed simulation.

2. The cases for inclusion of right-hand and left-hand delayed simulation in delayed simulation has been shown in [34]. The rest of the inclusions follows from the definition of the winning conditions since those are also ordered by containment. That is, whenever Duplicator wins the direct (delayed) simulation game, he also wins the delayed (and fair simulation) game.

3. To prove part 3, assume $\star \in \{\mathsf{di}, \mathsf{de_r}, \mathsf{de_l}, \mathsf{de}, \mathsf{f}\}$, $s^{(i)} \preceq_\star s'^{(i)}$ and $\alpha \in \mathsf{Lang}(\mathfrak{A}_s)$ with $\alpha = \alpha^{(0)}\alpha^{(1)}\dots$. Then, there exists a winning strategy γ for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s,s')$ and an accepting run $\pi = s^{(0)}s^{(1)}\dots$ of $\mathfrak{A}$ starting with $s^{(0)} = q$. Assume Spoiler plays in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s,s')$ and Duplicator replies this according to γ. Then a run $\pi' = s'^{(i)}s'^{(1)}\dots$ of $\mathfrak{A}'$ is built up according to (3.1). Since π is accepting and γ is winning, π' will also be accepting.

4. $\approx_\star$ is an equivalence relation since it is reflexive, transitive and symmetric.

□

3.3 Quotient Constructions

Having defined an equivalence relation enables us to define the usual quotient construction, termed the naive quotient in [31].

Definition 25 (Quotient Structure for Parity Automata)**.** *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton. Let ζ be an equivalence relation on $\mathcal{S}$. We denote the equivalence class of $s \in \mathcal{S}$ with $[s]$, i. e. $[s] = \{s' \in \mathcal{S} \mid (s, s') \in \zeta\}$. The quotient structure $\mathfrak{A}_{/\zeta}$ is defined as:*

- $\mathcal{S}_{/\zeta} = \{[s] \mid s \in \mathcal{S}\}$
- $\mathcal{I}_{/\zeta} = \{[s] \mid s \in \mathcal{I}\}$
- $\mathcal{R}_{/\zeta} = \{[s_1], a, [s_2]) \mid \exists s_1' \in [s_1], \exists s_2' \in [s_2]. (s_1', a, s_2') \in \mathcal{R}\}$
- $\Omega_{/\zeta}([s]) = \min\{\Omega(s') \mid s' \in [s]\}$

In a symbolic implementation, it is easier to represent an equivalence class with one representative instead of a set. This leads to the definition of a canonical selection function and a canonical quotient construction (see e. g. [91]).

Definition 26 (Canonical Selection Function and Canonical Quotient Structure)**.** *Given a parity automaton $\mathfrak{A} = (\mathcal{Q}, \mathcal{I}, \mathcal{R}, \Omega)$ and an equivalence relation ζ. Each function $\varrho : \mathcal{Q} \to \mathcal{Q}$ is called a canonical selection function for ζ iff the following conditions are met:*

- $\forall s \in \mathcal{Q}.(s, \varrho(s)) \in \zeta$
- $\forall s_1, s_2 \in \mathcal{Q}.(s_1, s_2) \in \zeta \to (\varrho(s_1) = \varrho(s_2))$

For a canonical selection function $\varrho : \mathcal{Q} \to \mathcal{Q}$, we define the canonical quotient automaton $\mathfrak{A}_{/\varrho}$ as follows:

- $\mathcal{Q}_{/\varrho} = \{\varrho(s) \mid s \in \mathcal{Q}\}$
- $\mathcal{I}_{/\varrho} = \{\varrho(s) \mid s \in \mathcal{I}\}$
- $\mathcal{R}_{/\varrho} = \{(\varrho(s_1), a, \varrho(s_2)) \mid (s_1, a, s_2) \in \mathcal{R}\}$
- $\Omega_{/\varrho}(s) = \Omega(s)$

If the canonical selection function always selects a representative with minimal color, the two definitions are obviously equivalent (see also [91]).

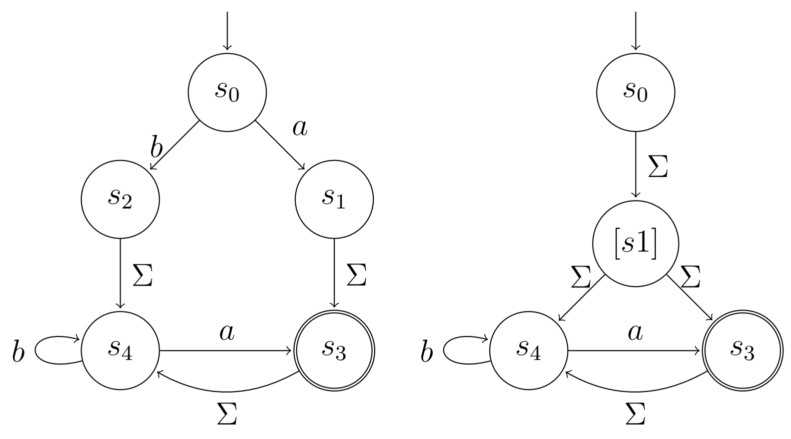


Figure 3.2: Merging of Fair Simulation Equivalent States may Destroy Determinism

3.3.1 Merging of States may Destroy Determinism

It is well known that the (canonical) quotient construction returns a deterministic automaton in case of the greatest simulation relation equivalence with respect to direct, delayed or fair simulation if it is used for the minimization of a deterministic automaton. This is due to the fact that if two states are simulation equivalent in the greatest simulation relation, then also all successors must be simulation equivalent and are thus merged to one (equivalence class) state. However, this need not hold if we do not use the *largest* simulation relation.

Even correct merges of fair simulation equivalent states may destroy determinism which is demonstrated by the left automaton in Figure 3.2. Here we have a Büchi automaton where the accept states are marked by a double circle. Clearly, all states are fair simulation equivalent, however, the automaton that is obtained by merging all states accepts all words and thus is not correct. Hence, a possible approach regarding fair simulation, followed also by [40], is to use smaller simulation relations. In that case nondeterminism may occur. When we merge the states s_2 and s_1 to one state, the automaton shown in the right of Figure 3.2 is obtained. This automaton is correct but nondeterministic.

3.3.2 A Quotient Construction that Preserves Determinism

In order to preserve determinism, we develop in the following a new quotient construction. The principle behind this construction is that instead of merging states, we replace ingoing transitions. To give an intuitive idea for this construction, consider again the automaton in Figure 3.2. By first replacing any ingoing edge to s_2 by an

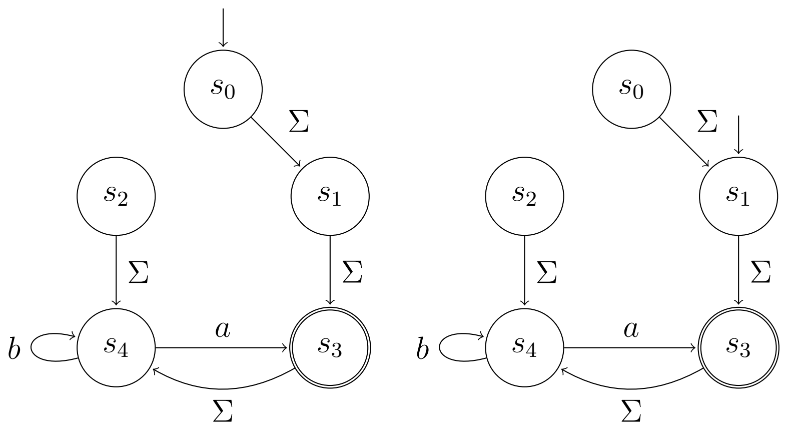


Figure 3.3: Determinism is Preserved by Replacing Edges

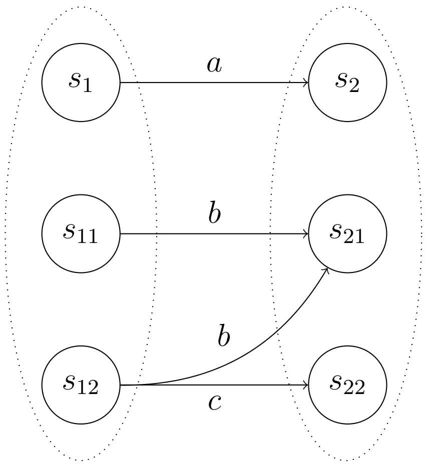
edge to s_1, the automaton in the left of Figure 3.3 is obtained. Any initial state can be interpreted as having an ingoing edge. This ingoing edge may be replaced by an edge to s_1 so that we obtain the automaton on the right of Figure 3.3. At the end, the states s_0 and s_2 are no longer reachable and can be removed.

This intuitive idea of replacing ingoing edges is used to obtain the following definition which is a modification of the canonical quotient construction:

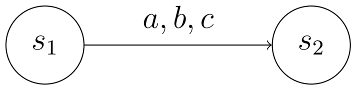
Definition 27 (The Successor Quotient for Parity Automata)**.** *Given a parity automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ and an equivalence relation ζ together with a canonical selection function $\varrho : \mathcal{S} \to \mathcal{S}$. The successor quotient $\mathfrak{A}{\div_\varrho}$ is defined as:*

- $\mathcal{S}{\div_\varrho} = \{\varrho(s) \mid s \in \mathcal{S}\}$
- $\mathcal{I}{\div_\varrho} = \{\varrho(s) \mid s \in \mathcal{I}\}$
- $\mathcal{R}{\div_\varrho} = \{(s_1, a, \varrho(s_2)) \mid (s_1, a, s_2) \in \mathcal{R} \wedge s_1 = \varrho(s_1)\}$
- $\Omega{\div_\varrho}(s) = \Omega(s)$

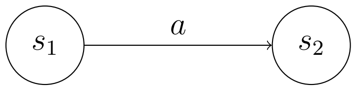
The difference between the canonical quotient structure and the successor quotient structure are not obvious, and best explained with some diagrams. First examine Figure 3.4. Here, some part of an automaton is drawn. We have two equivalence classes $\tilde{s_1} = \{s_1, s_{11}, s_{12}\}$ and $\tilde{s_2} = \{s_2, s_{21}, s_{22}\}$. As the canonical selection function, we take $\rho_1(s_1) = \rho_1(s_{11}) = \rho_1(s_{12}) = s_1$ and $\rho_1(s_2) = \rho(s_{21}) = \rho(s_{22}) = s_2$. The automata obtained from the canonical and the successor quotient construction are drawn in the top-right of this Figure. As another example, if we choose $\rho_2(s_1) = \rho_2(s_{11}) = \rho_2(s_{12}) = s_{12}$ and $\rho_2(s_2) = \rho_2(s_{21}) = \rho_2(s_{22}) = s_{21}$, we would obtain the two



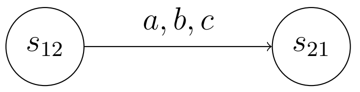
(a) Original automaton



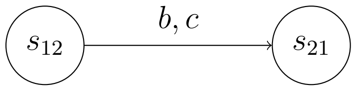
(b) Automaton obtained by Definition 26 and ρ_1



(c) Automaton obtained by Definition 27 and ρ_1



(d) Automaton obtained by Definition 26 and ρ_2



(e) Automaton obtained by Definition 27 and ρ_2

Figure 3.4: The Difference between Canonical Quotient and Successor Quotient

automata shown in the bottom-right of this figure, since every outgoing edge of s_{12} to a successor state in $\tilde{s_2}$ is preserved.

Although we do not add all transitions compared to the canonical quotient construction, the following theorem states that with respect to the largest simulation equivalence relation, the automata are direct simulation equivalent.

Theorem 3. *Let $\mathfrak{A}$ be a parity automaton, $\star \in \{\mathsf{di}, \mathsf{de}, \mathsf{de_l}, \mathsf{de_r}, \mathsf{f}\}$. Let ϱ be a canonical selection function for $\approx_\star$ that selects for any equivalence class a minimal representative, i. e. a state with minimal color in the equivalence class. Then,any state s_i in $\mathfrak{A}_{/\varrho}$ is direct simulation equivalent to the state of the same name in $\mathfrak{A}{\div}_\varrho$.*

Proof. Obviously, we have that any state in $\mathfrak{A}_{/\varrho}$ direct simulates the state of the same name in $\mathfrak{A}{\div}_\varrho$. For the opposite direction, we define the following (memoryless) strategy for Duplicator in $\mathfrak{G}^{(\mathfrak{A}_{/\varrho}, \mathfrak{A}\div_\varrho)}\left([s_i^{(0)}], s_i^{(0)}\right)$: $\gamma(s, a, [s']) = \varrho(s')$. Any infinite play in which Duplicator plays according to this strategy is obviously winning for her, since the colors seen on both sides are the same. Thus assume that π is a finite prefix of a game in which Duplicator plays according to this strategy. Let $([s], s)$ be the last position of this prefix where Spoiler chooses a transition $([s], a, [s']) \in \mathcal{R}_{/\varrho}$ but there is no transition $(s, a, \varrho(s')) \in \mathcal{R}\div_{\varrho}$. According to the definition of $\mathcal{R}_{/\varrho}$, there must exist $\hat{s}, \hat{s'} \in \mathcal{S}_{/\varrho}$ such that (i) $\hat{s} \approx_{\star} s$ and $(\hat{s}, a, \hat{s'}) \in \mathcal{R}$. Since the equivalence relation is maximal, this implies that for some state s' with $\hat{s'} \approx_{\star} s'$ we have that $(s, a, s') \in \mathcal{R}$. According to the definition of $\mathfrak{A}\div_{\varrho}$, this implies that there must also exist a transition from s to s', since s' is the lowest representative of the equivalence class, a contradiction to our assumption. □

An important consequence of this theorem is that the only modification we need to consider is the replacement of an ingoing transition from one state s to a state t while still being sure that any minimization that would be possible according to one of the largest simulation equivalence relations is detected in that way. Although the successor quotient construction is not really needed to show direct and left-hand delayed simulation, only considering the replacement of ingoing edges simplifies some of the proofs in the following, so that we have introduced it already here.

Before we start with the main parts, we define also a simplified successor quotient, in which exactly one state s is made superfluous by replacing each ingoing edge to s by a transition to t.

Definition 28. *For a parity automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ and two states $s, t \in \mathcal{S}$, we define $\mathfrak{A}_{s\to t} = (\mathcal{S}_{s\to t}, \mathcal{I}_{s\to t}, \mathcal{R}_{s\to t}, \Omega_{s\to t})$ by*

- $\mathcal{S}_{s\to t} = \mathcal{S} \setminus \{s\}$
- $\mathcal{I}_{s\to t} = \begin{cases} \mathcal{I} \setminus \{s\} & \textit{if } t \in \mathcal{I} \\ (\mathcal{I} \setminus \{s\}) \cup \{t\} & \textit{if } s \in \mathcal{I} \wedge t \notin \mathcal{I} \\ \mathcal{I} & \textit{else} \end{cases}$
- $\mathcal{R}_{s\to t} \Leftrightarrow (\mathcal{R}(p, a, q) \wedge q \neq s) \vee (q = t \wedge \mathcal{R}(p, a, s))$
- $\Omega(q)_{s\to t} = \Omega(q)$ *if* $q \neq s$

3.4 Direct and Left-Hand Delayed Simulation

We will start our considerations by the following lemma that shows that whenever we have a direct or left-hand delayed simulation strategy to show that $s \preceq_{\star} t$ we also have a strategy that can avoid a transition (r, a, s) provided there is a transition (r, a, t).

Lemma 2. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton. For $p, q, r, s, t \in \mathcal{S}$ and some $\star \in \{\mathsf{di}, \mathsf{de_l}\}$, assume that $p \preceq_{\star} q$, $s \preceq_{\star} t$ and $\{(r, a, s), (r, a, t)\} \subseteq \mathcal{R}'$. Then, there exists a winning strategy γ' in $\mathfrak{G}_{\star}^{(\mathfrak{A},\mathfrak{A})}(p, q)$ such that $\gamma'(\eta r a u) \neq s$ for every $u \in \mathcal{S}$ and every $\eta \in S(S'\Sigma S)^*$, i. e. Duplicator never chooses transition (r, a, s).*

Proof. Assume that γ is a memoryless winning strategy for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s,t)$ which must exist since $s \preceq_\star t$ [3]. Let γ_0 be a winning strategy for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(p,q)$. We construct a winning strategy for Duplicator in $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(p,q)$ which never chooses transition (r,a,s). To give an intuitive idea for the construction, consider Figure 3.5.

The leftmost column gives the position of Spoiler in the original game, while the second column give the position of Duplicator that would be obtained if she would play according to the original strategy γ_0. When in this game transition (r,a,s) would be chosen for the first time, we start a puppeteer game where Spoiler starts from position s and Duplicator from position t (point k in the figure). We term the two players in that game Spoiler puppet and Duplicator puppet. In the puppeteer game, the positions of Spoiler puppet equals the positions $q_0^{(i)}$ of Duplicator in the second row and the positions $q_1^{(i)}$ are obtained from $q_0^{(i)}$ by an application of γ. Hence, the second and the third row can be seen as a γ-consistent play of the simulation game $\mathfrak{G}_\star^{(\mathfrak{A},\mathfrak{A})}(s,t)$ which gives a justification for the term puppeteer game. In the original simulation game, the transitions chosen by Duplicator have to be chosen to mimic the one chosen by Duplicator puppet in the rightmost puppeteer game. This is true unless in the puppeteer game, the transition (r,a,s) would be chosen. In that case, a new puppeteer game is started (see again Figure 3.5, here position l) and the decision in the original game is now based on this new puppeteer game. Hence, in Figure 3.5 solid lines indicate transitions in the automaton, other lines indicate strategy usage. Accordingly, the transitions of Spoiler in the simulation game are given by the leftmost solid lines and the transitions of Duplicator by the rightmost solid lines.

Notice that all puppeteer games can be recalculated in each step solely based on the history of the play. Nevertheless, in order to check how many puppet games are used, Duplicator may use a memory variable n which simplifies some of the definitions.

Formally, we use the following to define our strategy γ': We denote with p the positions of Spoiler in the simulation game between p and q over a word α. We denote with q_0 the positions of Duplicator that would be obtained if played according to the original strategy γ_0, i. e. we set $p^{(0)} = p$. We define $q_0^{(0)} = q$ and for every $i > 0$ set

$$q_0^{(i)} = \gamma_0(p^{(0)} q_0^{(0)} \alpha^{(0)} p^{(1)} q_0^{(1)} \alpha^{(1)} \dots q_0^{(i-1)} \alpha^{(i-1)} p^{(i)})$$

Finally, we initialize the variable to count the number of puppet games by $n^{(0)} = 0$ and update n by the following rule for $i > 0$:

$$n^{(i)} = \begin{cases} n^{(i-1)} + 1 & \text{if } q_{n^{(i-1)}}^{(i)} = r \wedge \alpha^{(i-1)} = a \wedge q_{n^{(i-1)}}^{(i)} = s \\ q^{(i)} & \text{else} \end{cases}$$

[3]It is not really necessary that the strategy is memoryless, but simplifies some of the considerations.

$$\begin{tikzcd}
\text{p}=p^{(0)} \arrow[d, "\alpha^{(0)}"] & \text{q}=q_0^{(0)} \arrow[d, "\alpha^{(0)}"] & & \\
p^{(1)} \arrow[r, rightsquigarrow, "\gamma_0"] \arrow[d, "w"] & q_0^{(1)} \arrow[d, "w"] & & \\
\dots \arrow[d, "\alpha^{(k-2)}"] & \dots \arrow[d, "\alpha^{(k-2)}"] & & \\
p^{(k-1)} \arrow[r, rightsquigarrow, "\gamma_0"] \arrow[d, "a"] & q_0^{(k-1)} = r \arrow[d, "a"] \arrow[dr, "a"] & & \\
p^{(k)} \arrow[r, rightsquigarrow, "\gamma_0"] \arrow[d, "\alpha^{(k)}"] & q_0^{(k)} = s \arrow[r, rightsquigarrow, "\gamma"] \arrow[d, "\alpha^{(k)}"] & q_1^{(k)} = t \arrow[d, "\alpha^{(k)}"] & \\
p^{(k+1)} \arrow[r, rightsquigarrow, "\gamma_0"] \arrow[d, "w"] & q_0^{(k+1)} \arrow[r, rightsquigarrow, "\gamma"] \arrow[d, "w"] & q_1^{(k+1)} \arrow[d, "w"] & \\
\dots \arrow[d, "\alpha^{(l-2)}"] & \dots \arrow[d, "\alpha^{(l-2)}"] & \dots \arrow[d, "\alpha^{(l-2)}"] & \\
p^{(l-1)} \arrow[r, rightsquigarrow, "\gamma_0"] \arrow[d, "a"] & q_0^{(l-1)} \arrow[r, rightsquigarrow, "\gamma"] \arrow[d, "a"] & q_1^{(l-1)} = r \arrow[d, "a"] \arrow[dr, "a"] & \\
p^{(l)} \arrow[r, rightsquigarrow, "\gamma_0"] & q_0^{(l)} \arrow[r, rightsquigarrow, "\gamma"] & q_1^{(l)} = s \arrow[r, rightsquigarrow, "\gamma"] & q_2^{(l)} = t
\end{tikzcd}$$

Figure 3.5: Defining Strategies using the Puppeteer Metaphor.

In puppeteer game j, the positions of Spoiler puppet are given by q_{j-1} and the positions of Duplicator puppet are chosen according to the memoryless strategy γ unless transition (r, a, s) is chosen. Formally, we define for every $0 < j < n^{(i-1)}$:

$$q_j^{(i)} = \begin{cases} t & \text{if } n^{(i-1)} < j \\ \gamma(q_j^{(i-1)}, \alpha^{(i-1)}, q_{j-1}^{(i)}) & \text{else} \end{cases}$$

that means, unless the puppet game starts, i. e. $j \geq n^{(i-1)}$, the position of Duplicator is set arbitrarily to t which is the starting point of each puppet game. Finally, the strategy of Duplicator is determined in each step by $q_{n^{(i)}}^{(i)}$, i. e. the position of Duplicator puppet in the last puppet game, i. e. we set

$$\gamma' \left(p^{(0)} q_{n^{(0)}}^{(0)} \alpha^{(0)} p^{(1)} q_{n^{(1)}}^{(1)} \alpha^{(1)} \dots p^{(i)} \right) = q_{n^{(i)}}^{(i)}$$

Notice that for every puppet game j and every position i, we have $(q_j^{(i)}, \alpha^{(i)}, q_j^{(i+1)}) \in \mathcal{R}$: for $j = 0$ this follows from the fact that γ_0 is a strategy and for $j > 0$ this follows from the validity of γ. Moreover, whenever a new puppet game starts, in the previousl last puppet game, transition (r, a, s) is taken, whereas in the overall simulation game, transition (r, a, t) is taken instead. Hence, γ' defines a valid strategy for the simulation game.

We now show that γ' is winning and consider the case of left-hand delayed simulation (the case for direct simulation is shown analogously). Obviously, if no puppet game is started, simply strategy γ_0 is played and Duplicator is winning without using transition (r, a, s). Otherwise, according to the definition of left-hand delayed simulation, whenever for some puppet game i we have that $\Omega(q_i^{(j)}) <_\Omega \Omega(q_{i+1}^{(j)})$, there must exist some position k such that $\Omega(q_i^{(k)})$ is odd and $\Omega(q_i^{(k)}) \leq \min\{\Omega(q_i^{(j)}), \Omega(t_{i+1}^{(j)})\}$, since γ is a winning strategy. Assume now that $\Omega(p^{(i)}) <_\Omega \Omega(q_{g^{(i)}}^{(i)})$, i. e. in the overall simulation game, duplicator visits a state worse than the state visited by spoiler. Let o be the lowest odd color in any puppeteer game and j be the minimal position of o. Obviously we have that $o \leq \Omega \left(q_{g^{(i)}}^{(i)} \right)$. According to the previous remark we must have that for some position $i' > i$ we have that $\Omega(q_{j-1}^{(i')})$ is odd and $\Omega(q_{i-1}^{(i')}) < o$. Using the same argument at most i times, we finally reach a position $i'' > i$ such that $\Omega(p^{(i'')}$ is odd and $\Omega(p^{(i'')}) < o \leq \Omega \left(q_{g^{(i)}}^{(i)} \right)$. □

Notice that a similar result does not hold for the other simulation relations. Direct and left-hand delayed simulation are restricted notions in the sense that whenever something bad happens, i. e. whenever we have that Spoiler visits a better state than Duplicator in one round, the only chance for Duplicator to win is that Spoiler visits a bad state later. This is in contrast to the other simulation relations where Duplicator can assure winning by her own will (as long as she follows the same word).

This lemma allows us to formulate the following theorem which shows that we can remove a redundant transition without affecting the simulation relations:

Theorem 4. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton. For $r, s, t \in \mathcal{S}$ and some $\star \in \{\mathsf{di}, \mathsf{de_l}\}$, assume that $s \preceq_\star t$ and $(r, a, t) \in \mathcal{R}$. Let $\mathcal{R}'$ be obtained from $\mathcal{R}$ by removing all transitions (r, a, s) except for transition (r, a, t). Let $\mathcal{I}' = \mathcal{I} \setminus \{s\}$ if $t \in \mathcal{I}$ and $\mathcal{I}' = \mathcal{I}$ else. Let $\mathfrak{A}' = (\mathcal{S}, \mathcal{I}', \mathcal{R}', \mathcal{F})$. Then, the following holds:*

1. *$p \preceq_\star^{(\mathfrak{A},\mathfrak{A})} q$ if and only if $p \preceq_\star^{(\mathfrak{A}',\mathfrak{A}')} q$*
2. *Any state in $\mathfrak{A}'$ is $\star$-equivalent to the state of the same name in $\mathfrak{A}$.*

Proof. The only if direction of the first part is trivial. The other direction can be seen by Lemma 2 since a strategy to show $p \preceq_\star^{(\mathfrak{A},\mathfrak{A})} q$ that never uses transition (r, a, s) can also be used to show $p \preceq_\star^{(\mathfrak{A}',\mathfrak{A}')} q$. Finally, in a same manner we can use Lemma 2 to generate a strategy to show both $p \preceq_\star^{(\mathfrak{A},\mathfrak{A}')} p$ and $p \preceq_\star^{(\mathfrak{A}',\mathfrak{A})} p$ by using the identity function as γ_0. $\square$

The following corollary holds because simulation implies language equivalence.

Corollary 1. *Let $\mathfrak{A}$ and $\mathfrak{A}'$ as in Theorem 4. Then, $\mathsf{Lang}(\mathfrak{A}) = \mathsf{Lang}(\mathfrak{A}')$.*

Theorem 4 obviously works in both directions. Hence, instead of removing a transition from r to s whenever $s \preceq_\star t$ and $\mathcal{R}(r, a, t)$, we can also add a transition from r to t when $\mathcal{R}(r, a, s)$ and $s \preceq_\star t$ holds. Repeated application of this transformation gives us the following corollary, which implies that we can remove any of two simulation-equivalent states.

Corollary 2. *Let $\mathfrak{A}$ be a parity automaton, $\star \in \{\mathsf{di}, \mathsf{de_l}\}$. Let $s, t \in \mathcal{S}$ such that $s \neq t$ and $s \approx_\star^{(\mathfrak{A},\mathfrak{A})} t$. Let $\mathfrak{A}_{s\to t}$ be the automaton given in Definition 28 where any transition to s has been replaced by a transition to t.*

- *any state in $\mathfrak{A}_{s\to t}$ is $\star$-equivalent to the state of the same name in $\mathfrak{A}$.*
- $\mathsf{Lang}(\mathfrak{A}) = \mathsf{Lang}(\mathfrak{A}_{s\to t})$

When we perform the calculation of the greatest direct (left-hand delayed) simulation equivalence, the repeated application of this corollary allows us to use the successor quotient construction for minimization with respect to direct and delayed simulation. So, we obtain:

Theorem 5. *Let $\mathfrak{A}$ be a parity automaton, $\star \in \{\mathsf{di}, \mathsf{de_l}\}$. Let $\star \in \{\mathsf{di}, \mathsf{de_r}\}$. Let ζ be the largest simulation relation with respect to $\star$ and ρ be a canonical selection function for ζ. Then, any state in $\mathfrak{A}{\div_\zeta}$ is is $\star$-equivalent to the state of the same name in $\mathfrak{A}$.*

Thus there are at least two mechanisms to minimize parity automata using direct simulation: first two $\mathsf{di}, \mathsf{de_l}$ simulation-equivalent states can be merged (respectively, one can choose one representative for every simulation-equivalence class) and second, every transition to s can be removed provided there are simultaneous transitions to t and $s \preceq_\star t$ for $\star \in \{\mathsf{di}, \mathsf{de_l}\}$.

3.5 Reverse Simulation

In this paragraph, reverse simulation for parity automata is considered. Reverse simulation has been introduced in [99] for Büchi automata. Instead of calculating the simulation relations in the order of the transition relation, reverse simulation is calculated backwards, i. e. the simulation games are defined in a way that Spoiler, when in state s instead of choosing a successor t in some transition $(s, a, t) \in \mathcal{R}$, chooses some predecessor r from some transition $(r, a, s) \in \mathcal{R}$ and Duplicator must also respond with a predecessor state. In this section, three variants of reverse simulation will be introduced, namely the counterparts of direct simulation and left- (right-) hand delayed simulation. While reverse direct and left-hand delayed simulation can be used for state space minimization, a counterexample will show that this is not true for right-hand delayed simulation. Similar as in the preceding paragraph, we restrict our attention to removing and replacing of edges.

The following definition introduces reverse simulation games.

Definition 29 (Reverse Simulation Games).
Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega), \mathfrak{A}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \Omega')$ be two parity automata over the alphabet Σ and $s^{(0)}$ and $s'^{(0)}$ be arbitrary states of $\mathfrak{A}$ and $\mathfrak{A}'$ respectively. The reverse basic game $\overleftarrow{\mathfrak{G}}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s'^{(0)}\right)$ *is played by two players,* Spoiler *and* Duplicator*, in rounds, where, at the beginning and at the end of each round, two pebbles,* Red *and* Blue*, are placed on two states. At the start, round 0, Red and Blue are placed on $s^{(0)}$ and $s'^{(0)}$, respectively. Assume that, at the beginning of round i, Red is on state $s^{(i)}$ and Blue is on $s'^{(i)}$. Then:*

1. *Spoiler chooses a letter $\alpha^{(i)} \in \Sigma$.*
2. *Spoiler chooses a transition $(s^{(i+1)}, \alpha^{(i)}, s^{(i)}) \in \mathcal{R}$ and moves Red to $s^{(i+1)}$.*
3. *Duplicator, responding, must choose a transition $(s'^{(i+1)}, \alpha^{(i)}, s'^{(i)}) \in \mathcal{R}'$ such that the following holds: if $s^{(i+1)} \in \mathcal{I}$, then also $s'^{(i+1)} \in \mathcal{I}'$. Duplicator moves*

Blue to $s'^{(i+1)}$. If no such $\alpha^{(i)}$-transition going to $s'^{(i+1)}$ exists, then the game halts and Spoiler wins.

4. *The starting position for the next round is $(s^{(i+1)}, s'^{(i+1)})$.*

For $\overleftarrow{\star} \in \{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_l}, \overleftarrow{\mathsf{de}_r}, \overleftarrow{\mathsf{de}}, \overleftarrow{\mathsf{f}}\}$ the corresponding reverse simulation games $\overleftarrow{\mathfrak{G}}^{(\mathfrak{A},\mathfrak{A}')}_{\overleftarrow{\star}}\left(s^{(0)}s'^{(0)}\right)$ are defined by replacing $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}_{\star}\left(s^{(0)}s'^{(0)}\right)$ with $\overleftarrow{\mathfrak{G}}^{(\mathfrak{A},\mathfrak{A}')}_{\overleftarrow{\star}}\left(s^{(0)}s'^{(0)}\right)$ in Definition 21. Additionally, strategies can be defined analogously as for the forward simulation relations.

Finally, the reverse simulation relations are obtained by the following definition:

Definition 30 (Reverse Simulation Relations)**.** *Let $\mathfrak{A}$, $\mathfrak{A}'$ be parity automata. A state s' of $\mathfrak{A}'$ reverse direct, reverse left-hand, reverse right-hand simulates, reverse delayed , reverse fair simulates a state s of $\mathfrak{A}$, iff there is a winning strategy for Duplicator in $\overleftarrow{\mathfrak{G}}^{(\mathfrak{A},\mathfrak{A}')}_{\star}(s, s')$ where $\star \in \{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_l}, \overleftarrow{\mathsf{de}_r}, \overleftarrow{\mathsf{de}}, \overleftarrow{\mathsf{f}}\}$. We denote such a relationship by $s \preceq^{(\mathfrak{A},\mathfrak{A}')}_{\overleftarrow{\star}} s'$. If $\mathfrak{A}, \mathfrak{A}'$ are clear from the context, we sometimes omit them. The reverse simulation relation between automata and the reverse simulation equivalence is defined analogously to Definition 23.*

As before, Spoiler produces, letter by letter, an ω-word over Σ as simultaneous inputs for the two automata $\mathfrak{A}$ and $\mathfrak{A}'$. However, instead of producing two ordinary runs over $\mathfrak{A}, \mathfrak{A}'$ in forward order, two runs $(\overleftarrow{\pi}, \overleftarrow{\pi'})$ in backward order are produced.

Notice that in contrast to the forward simulation relations, reverse simulation does not imply language containment. It only implies finitary language containment. This is the reason why fair simulation is not considered in a reverse fashion. Fair simulation is only useful if it is possible to check whether the two automata are still language equivalent. Since this is not possible using reverse fair simulation, fair simulation will not be considered here. Moreover, we will show that even reverse right hand simulation relations can not be used for state space minimization by a simple quotient construction as will be shortly shown, so even the full reverse delayed simulation relation is useless.

3.5.1 Basic Properties of Reverse Simulation Relations

However, the following proposition that is shown in analogy to Lemma 1 states that reverse simulation relations in two directions are in fact equivalence relations.

Proposition 4. *Let $\mathfrak{A}$ be a parity automaton.*

1. *For $\star$ in $\{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_r}, \overleftarrow{\mathsf{de}_l}, \overleftarrow{\mathsf{de}}, \overleftarrow{\mathsf{f}}\}$, $\approx^{(\mathfrak{A},\mathfrak{A})}_{\star}$ is an equivalence relation.*

2. *The simulation relations are ordered by containment : $\preceq_{\overleftarrow{\mathsf{di}}} \subseteq \preceq_{\overleftarrow{\mathsf{de}_r}} \subseteq \preceq_{\overleftarrow{\mathsf{de}}} \subseteq \preceq_{\overleftarrow{\mathsf{f}}}$ and $\preceq_{\overleftarrow{\mathsf{di}}} \subseteq \preceq_{\overleftarrow{\mathsf{de}}} \subseteq \preceq_{\overleftarrow{\mathsf{de}}} \subseteq \preceq_{\overleftarrow{\mathsf{f}}}$*

3.5.2 Reducing Reverse Simulation Games to Turn-Based Games

In a similar manner as for the forward simulation games, we can reduce the reverse simulation game to a turn-based game:

Definition 31 (Turn-Based Game for the Reverse Basic Game).
Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega), \mathfrak{A}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \Omega')$ be two parity automata over the alphabet Σ. We define a turn-based game $\mathfrak{G} = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, E, \Phi)$ to simulate the basic simulation game as follows:

$$\begin{aligned}
\mathcal{V}_0 &= \{v_{(s,s',a)} \mid s \in \mathcal{S} \wedge s' \in \mathcal{S}' \wedge \exists t.\ (s,a,t) \in \mathcal{R}\} \\
\mathcal{V}_1 &= \{v_{(s,s')} \mid s \in \mathcal{S} \wedge s' \in \mathcal{S}' \wedge s \in \mathcal{I} \rightarrow s' \in \mathcal{I}\} \\
E &= \{(v_{(s_2,s'_1,a)}, v_{(s_2,s'_2)}) \mid (s'_2, a, s'_1) \in \mathcal{R}' \wedge s_2 \in \mathcal{I} \rightarrow s'_2 \in \mathcal{I}\} \\
&\qquad \cup \{(v_{(s_1,s'_1)}, v_{(s_2,s'_1,a)}) \mid (s_2, a, s_1) \in \mathcal{R}\}
\end{aligned}$$

Thus, the first difference is that the two players choose transitions in reverse order and the second difference is that Duplicator may only choose an initial state as successor whenever Spoiler has managed the game on his side to an initial state as well.

The acceptance conditions for the direct simulation game is again the same safety condition as for the forward direct simulation, namely that $\Omega(s) \leq_\Omega \Omega(s')$ for every $v_{(s,s')}$.

In order to define an acceptance condition for the reverse left hand simulation game, we use the memory technique introduced in [32] for delayed reverse simulation games, i. e. we add a memory variable k that is used to remember when a position $v_{(s,s')}$ has been reached such that $\Omega(s) <_\Omega \Omega(s')$, i. e. a bad event according to the reverse left-hand simulation has occurred. To this end, the smaller of the two values are stored in the memory variable k. If at some point during the play, a position $v_{s,s'}$ is reached such that $\Omega(s) < k$ and $\Omega(s)$ is even, the memory variable is cleared, indicated by $\surd$. If the value $\surd$ is read infinitely often, the game is won. Thus, we obtain the following modified game graph:

Definition 32 (Turn-Based Game for the Reverse Left-Hand Game).
Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega), \mathfrak{A}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \Omega')$ be two parity automata over the alphabet Σ with $d = \max\{|\Omega|, |\Omega'|\}$. We define a turn-based game $\mathfrak{G} = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, E, \Phi)$ to simulate the basic simulation game as follows:

$$\begin{aligned}
\mathcal{V}_0 &= \{v_{(s,s',a,k)} \mid s \in \mathcal{S} \wedge s' \in \mathcal{S}' \wedge \exists t.\ (t,a,s) \in \mathcal{R}\} \\
\mathcal{V}_1 &= \{v_{(s,s',k)} \mid s \in \mathcal{S} \wedge s' \in \mathcal{S}' \wedge s \in \mathcal{I} \rightarrow s' \in \mathcal{I}\} \\
E &= \{(v_{(s_2,s'_1,a,k)}, v_{(s_2,s'_2,k')}) \mid (s'_2, a, s'_1) \in \mathcal{R}' \wedge s_2 \in \mathcal{I} \rightarrow s'_2 \in \mathcal{I}\} \\
&\qquad \cup \{(v_{(s_1,s'_1,k)}, v_{(s_2,s'_1,a,k')}) \mid (s_2, a, s_1) \in \mathcal{R}\}
\end{aligned}$$

such that k, k' satisfies the following constraints:

- $k, k' \in \{0, \dots d\} \cup \{\surd\}$
- *if we reach a position $v_{(s_1, s'_1, k)}$, the next value k' is defined by:*
  - $k' = \surd$ *if* $k = \surd$ *and* $\Omega(s_1) \geq_\Omega \Omega(s'_1)$
  - $k' = \min(\Omega(s_1), \Omega(s'_1))$ *if* $\Omega(s_1) <_\Omega \Omega(s'_1)$
  - $k' = \surd$ *if* $k \neq \surd$, $\Omega(s_1) \geq_\Omega \Omega(s'_1)$ *and* $\Omega(s_1) < k$ *is odd*
- $k' = k$ *if we reach a position* $v_{(s,s',a,k)}$.

The acceptance condition is a Büchi condition stating that $\surd$ is visited infinitely often during a play.

Proposition 5. *Let $\mathfrak{A}, \mathfrak{A}'$ as before. Let $n = \max\{|\mathcal{S}|, |\mathcal{S}'|\}$, $m = \max\{|\mathcal{R}|, |\mathcal{R}'|\}$, $d = \max\{|\Omega|, |\Omega'|\}$. Then the greatest reverse direct simulation relation $\preceq^{(\mathfrak{A},\mathfrak{A}')}_{\overleftarrow{\mathsf{di}}}$ can be calculated symbolically in time $O(nm)$ and the greatest reverse left hand simulation relation $\preceq^{(\mathfrak{A},\mathfrak{A}')}_{\overleftarrow{\mathsf{de}_\mathsf{l}}}$ can be calculated symbolically in time $O((mnd)^2)$.*

Proof. The turn-based game associated with reverse direct simulation relation is a safety game with at most nm states. The corresponding game can be solved in time $O(mn)$. The turn-based game associated with reverse left hand simulation is a Büchi game with $O(mnd)$ states and $O(mnd)$ transitions. Thus the corresponding game can be solved in time $O((mnd)^2)$. $\square$

In Section 3.5.4 it is shown that reverse right hand delayed simulation can not be used for state space reduction, so that we do not consider a reduction to turn-based games here.

3.5.3 Minimizing ω-Automata with Reverse Direct and Left-Hand Simulation

Similar to the forward simulation, if $s \preceq^{(\mathfrak{A},\mathfrak{A})}_{\overleftarrow{\star}} t$ for $\star \in \{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_\mathsf{l}}\}$ and we have simultaneous transitions from both s and t to some state u, the (back)-transition (s, a, u) need not be taken by the strategy showing that $s \preceq^{(\mathfrak{A},\mathfrak{A})}_{\overleftarrow{\star}} t$ holds.

Proposition 6. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton. For $p, q, s, t, u \in \mathcal{S}$ and some $\overleftarrow{\star} \in \{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_\mathsf{l}}\}$, assume that $p \preceq_{\overleftarrow{\star}} q$, $s \preceq_{\overleftarrow{\star}} t$ and $\{(s, a, u), (t, a, u)\} \subseteq \mathcal{R}'$. Then, there exists a winning strategy γ' in $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A})}_{\star}(p, q)$ such that $\gamma'(\eta u a r) \neq s$ for every $r \in \mathcal{S}$ and every $\eta \in S(S'\Sigma S)^*$, i. e. transition (s, a, u) is never chosen in the play.*

Proof sketch. The proof of Lemma 2 remains valid if one replaces every ingoing edge with an outgoing edge. □

As it is the case for the forward simulation relations, the following theorem is the counterpart of Theorem 4.

Theorem 6. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton. For $s, t, u \in \mathcal{S}$ and some $\overleftarrow{\star} \in \{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_\mathsf{l}}\}$, assume that $s \preceq_{\overleftarrow{\star}} t$. Let $\mathcal{R}'$ be obtained from $\mathcal{R}$ by removing all transitions (s, a, u) provided there is a transition (t, a, u). Let $\mathfrak{A}' = (\mathcal{S}, \mathcal{I}, \mathcal{R}', \mathcal{F})$. Then, the following holds:*

1. *If $p \preceq_{\overleftarrow{\star}}^{(\mathfrak{A},\mathfrak{A})} q$ then $p \preceq_{\overleftarrow{\star}}^{(\mathfrak{A}',\mathfrak{A}')} q$.*

2. *Any state in $\mathfrak{A}'$ is $\overleftarrow{\star}$-equivalent to the state of the same name in $\mathfrak{A}$.*

3. *We have $\mathsf{Lang}(\mathfrak{A}) = \mathsf{Lang}(\mathfrak{A}')$.*

Proof. The first two parts of the theorem are shown analogously to the proof of Theorem 4. The only difference is that the runs are generated in reverse order and we have to additionally keep track of the initial states. We will thus concentrate on language equivalence. $\mathsf{Lang}(\mathfrak{A}') \subseteq \mathsf{Lang}(\mathfrak{A})$ is clear since we only remove transitions. For the opposite direction, let $\alpha \in \mathsf{Lang}(\mathfrak{A})$ and $\pi = p^{(0)} p^{(1)} \ldots$ be the run of $\mathfrak{A}$ over α. For the first case, assume that transition (s, a, u) is only taken finitely often in π and let i be the last position where this happens. Then $p^{(i+1)} \ldots$ is a run of $\mathfrak{A}'$ as well. Moreover, since u is reverse simulation equivalent in $\mathfrak{A}$ and $\mathfrak{A}'$, there must exist a (finite) run π' of $\mathfrak{A}'$ labeled with $\alpha^{(0)}, \ldots \alpha^{(i)}$ that leads to u. So $\pi' p^{(i+1)} p^{(i+2)} \ldots$ is a run of $\mathfrak{A}'$ over α that is accepting. At the second case, the transition (s, a, u) is taken infinitely often in π. Since π is an accepted run, there must exist a position i_0 such that the minimal color seen after i_0 is even, say e. Let $I = \{i_1, i_2, \ldots\}$ be the set of positions where transition (s, a, u) is taken and between two consecutive visits to (s, a, u) the color e is seen. Notice that since u is reverse-simulation equivalent in $\mathfrak{A}$ and $\mathfrak{A}'$, for every $i \in I$, there must exist a run π_i of $\mathfrak{A}'$ ending in u, labeled with $\alpha^{(0)} \alpha^{(1)} \ldots \alpha^{(i)}$ that describes a winning strategy in the reverse simulation game. Assume now that there does exist positions $i, j \in I$ such that $i_0 < i < j$ and for some position $i \leq t \leq j$ we have that $\Omega(\pi^{(t)}) = e$, $\Omega(\pi_j^{(t)})$ is odd and $\Omega(\pi_j^{(t)}) < e$. Notice that since e is the minimal color seen after i_0, we have that $\Omega(\pi^{(t')}) > e$ for every $t' \in \{i, i+1, \ldots, j\}$. That means that the infinite run $\pi^{(j)} \pi^{(j-1)} \ldots \pi^{(t)} \pi^{(t-1)} \ldots \pi^{(i)} \left(\pi^{(j-1)} \pi^{(j-2)} \ldots \pi^{(j)} \right)^{\omega}$ describes a winning strategy for Spoiler in the reverse simulation game starting in $u = \pi^{(j)}$, since the bad event at position t is not recovered later on, a contradiction to π_j describing a winning strategy.

That means that at those positions where π visits the minimal color e, every π_j visits an even smaller color e'. Moreover, assuming that at some other position an odd color smaller than e is visited on π_j (in the interval of interest) leads to the same contradiction as before. That means that there does exist a position j'_0 such that the minimal color seen on every π_j after position j' is even. We now construct from the infinitely many finite runs π'_i for every initial state q_0 an infinite tree as follows: Let $Q^{(0)} = \{(q_0, 0)\}$ and $Q^{(i)} = \left\{(q, i) \mid q = \pi_j^{(i)} \text{ for some } j \in \mathbb{N}\right\}$ and $i > 0$. We connect (q, i) and $(q', i+1)$ whenever there is some path π_j such that $q = \pi^{(i)}$ and $q' = \pi^{(i+1)}$. Clearly, for at least one initial state, the constructed tree is infinite and finite branching, since $\mathfrak{A}'$ is finite and we have infinitely many indices in I. Thus, according to Königs Lemma, this tree must contain an infinite path π'. Since we constructed those infinite path from the finite paths π_j, the minimal color seen on π' must be even after position j'_0 as well. □

In analogy to forward simulation, we only need to retain one state in every reverse-similar equivalence class:

Corollary 3. *Let* $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ *be a parity automaton and* $s, t \in \mathcal{S}$ *such that* $s \neq t$ *and* $s \approx_{\overleftarrow{\star}}^{(\mathfrak{A},\mathfrak{A})} t$ *for* $\overleftarrow{\star} \in \{\overleftarrow{\mathsf{di}}, \overleftarrow{\mathsf{de}_\mathsf{l}}\}$. *Let* $\mathfrak{A}' = (\mathcal{S}, \mathcal{I}, \mathcal{R}', \Omega)$ *be defined by*

$$\mathcal{R}'(q, a, q') \Leftrightarrow (\mathcal{R}(q, a, q') \wedge q \neq s) \vee (q = t \wedge \mathcal{R}(s, a, q'))$$

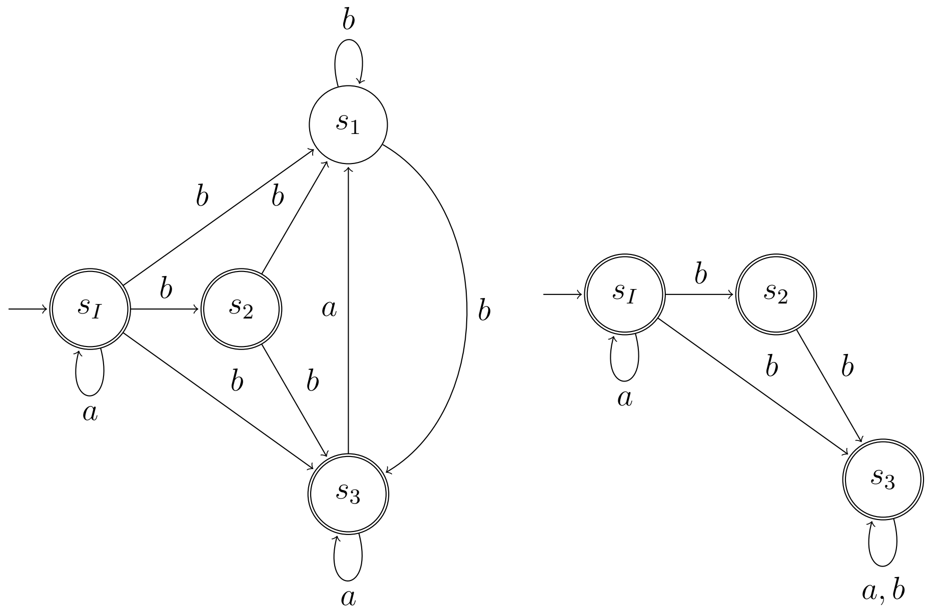
i. e. we remove any transition starting from s *and add those transition to the set of transitions starting in* t. *Then*

- *any state in* $\mathfrak{A}'$ *is* $\star$*-equivalent to the state of the same name in* $\mathfrak{A}$ *and*
- $\mathsf{Lang}(\mathfrak{A}) = \mathsf{Lang}(\mathfrak{A}')$.

3.5.4 Reverse Right-Hand Delayed Simulation

To see that reverse right-hand simulation is useless for state space reduction in any sense, consider the Büchi automaton shown in the left of Figure 3.6 where we have drawn accepting states with a double circle. Clearly, this automaton accepts only words that end with infinitely many a. The problem is that s_3 and s_1 are reverse right-hand simulation equivalent. To see that the two states are reverse right hand equivalent, a winning strategy of Duplicator has to choose the same successor state that has been chosen by Spoiler. More specifically, we have the rules defined in Figure 3.7 for the first step of every play. Thus, after the first step, the states of Spoiler and Duplicator coincide, so winning is obvious. But merging the two states will lead to the automaton shown on the right of the figure[4] which also accepts e. g. b^ω, so that we see that the automata are no longer equivalent.

[4] Remember that marked states of Büchi automata have color 0, so the merged state inherits the marking from state s_3.



(a) Non-Minimized Automaton (b) Minimized Automaton

Figure 3.6: A Büchi automaton that is not safe under minimization chosen with reverse left-hand delayed simulation

3.6 Fair Simulation for Co-Büchi Automata

The following sections considers minimization of parity automata with respect to fair simulation. To this end, it is first investigated in this section whether fair simulation can be directly used for merging fair-simulation equivalent states for co-Büchi automata. We give a negative answer to this question which gives the counterpart of the well-known fact that fair-simulation is not safe for quotients with respect to Büchi automata [28].

A co-Büchi condition is given with respect to a marked state set $\mathcal{F}$ such that no non-marked state is visited infinitely often. Co-Büchi automata can be regarded as parity automata where the color 2 is assigned to marked states while color 1 is assigned to non-marked states. Thus fair simulation is well-defined for co-Büchi automata. We will nevertheless redefine it in the following, to simplify the following considerations:

Definition 33 (Fair Simulation Game for Co-Büchi Automata)**.** *The fair simulation game for co-Büchi automata, or the persistence simulation game, denoted by* $\mathfrak{G}_{pe}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s^{(0')}\right)$*, is the basic game* $\mathfrak{G}^{(\mathfrak{A},\mathfrak{A}')}\left(s^{(0)}, s^{(0')}\right)$ *extended by the rule that the outcome* (π, π') *is winning for Duplicator iff whenever there exists an* i *such that* $s^{(j)} \in \mathcal{F}$ *for every* $j > i$*, there exists an* i' *such that* $s'^{(j')} \in \mathcal{F}'$ *for every* $j' > i'$

To show that $s_3 \preceq_{\overleftarrow{\mathsf{de}_\mathsf{r}}} s_1$	
transition chosen by Spoiler	transition chosen by Duplicator
(s_3, b, s_I)	(s_1, b, s_I)
(s_3, b, s_2)	(s_1, b, s_2)
(s_3, a, s_3)	(s_1, a, s_3)

To show that $s_1 \preceq_{\overleftarrow{\mathsf{de}_\mathsf{r}}} s_3$	
transition chosen by Spoiler	transition chosen by Duplicator
(s_1, b, s_I)	(s_3, b, s_I)
(s_1, b, s_2)	(s_3, b, s_2)
(s_1, b, s_1)	(s_3, b, s_2)
(s_1, a, s_3)	(s_3, a, s_3)

Figure 3.7: Strategy to show Right-Hand Simulation Equivalence of s_1 and s_3 of the Automaton of Figure 3.6

When considering Büchi automata, there is also the known definition of delayed simulation given in [28] that coincides with our definition of (left-hand) delayed simulation. Delayed simulation for Büchi automata is intuitively defined as: whenever Spoiler chooses a transition to a marked state, then at some point later, Duplicator must also reach a marked state.
Since this definition has some similarity to the definition of persistence simulation for co-Büchi automata, it might be possible that fair simulation can be used to merge two persistence-simulation equivalent states of a co-Büchi automaton. The following proposition shows that this is not the case:

Theorem 7. *For $n \geq 2$, there is a co-Büchi automaton $\mathfrak{A}_n$ with n states, each of which persistence simulates every other state, but no co-Büchi automaton with fewer than n states accepts $\mathsf{Lang}(\mathfrak{A}_n)$.*

Proof. Consider the automaton $\mathfrak{A}_n$ shown in Figure 3.8. It has n states and an alphabet $\Sigma = \{a_1, \dots a_{n-1}\}$. To see that every state of $\mathfrak{A}_n$ persistence simulates any other state, first note that because the automaton is deterministic Duplicator has no choice in her strategy. Consider an accepting run chosen by Spoiler. It is not hard to see that in this case Spoiler must choose some a_i such that the word labeling the run is of the form $w = \Sigma^* a_i^\omega$. Thus after finitely many rounds, the red pebble of Spoiler ends up in state s_i. Then, after at most $n-1$ steps, the blue pebble will also reach state s_i (irrespectively where it is when the red pebble enters state s_i). Thus, irrespectively at which position Duplicator starts, she has a winning strategy.

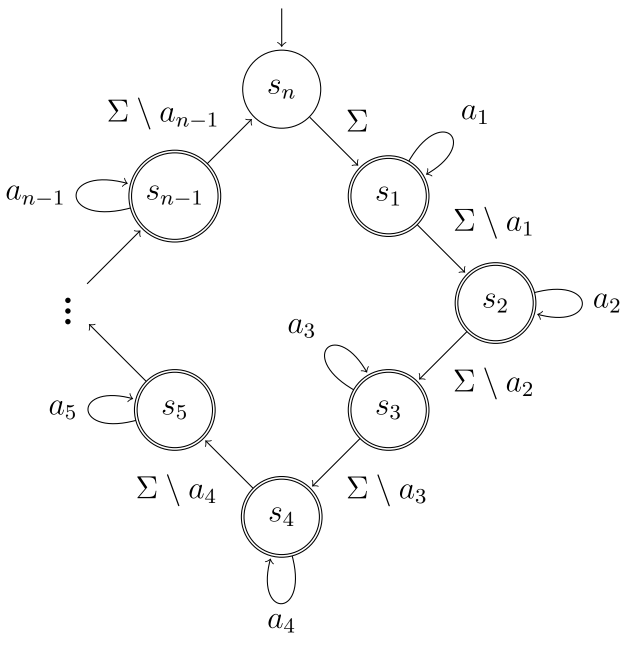


Figure 3.8: Problematic Family of Co-Büchi Automata for Fair-Simulation

Now notice that the Language $\mathsf{Lang}(\mathfrak{A}_n) = \bigcup_{i=1}^{n-1} \Sigma^* a_i^\omega$. It is not too hard to see that there is no co-Büchi automaton recognizing $\mathsf{Lang}(\mathfrak{A}_n)$ with fewer than n states. $\square$

This proposition shows that, in general, we can not hope to minimize a (co)-Büchi automaton and thus also not a parity automaton with fair simulation and quotienting alone[5].

3.7 Fair Simulation for Parity Automata

It is known that fair simulation (for Büchi automata) can be used for state space reductions provided some post-processing is done that ensures that the fair simulation equivalence is preserved [40]. The algorithm of [40] incrementally calculates the fair simulation relation. Whenever two states are found to be fair simulation equivalent, it is checked by adding and removing transitions whether fair simulation between the modified automaton and the original automaton is still preserved. This addition and removal of transitions is done in a manner such that the parity progress measure of

[5]One might get confused that the same might also hold for (left-hand) delayed simulation. This is not true. Whenever $\Omega(p^{(i)}) <_\Omega \Omega(s^{(i)})$ for co-Büchi automata and $\Omega(p^{(i)})$ is even, this implies that a matching good state for Spoiler must have priority 0 which is obviously not possible.

the underlying parity game is preserved provided the merge is correct. This allows to iteratively calculate the whole fair simulation relation with intermediate checks in a factor of k of the time complexity for calculating the fair simulation alone where k is the number of failed modifications. If a merge is later found to be wrong, i. e. language equivalence is not preserved, backtracking is used to return to a valid configuration. This backtracking on individual states is not well suited for a symbolic implementation. Hence we develop here an alternative that is not based on individual states but instead on the set of all states reachable from a merge candidate.

3.7.1 Merging States using Fair Simulation

We start our considerations with the following proposition that easily follows from previous results, since fair simulation implies language inclusion :

Proposition 7. *Let* $\mathfrak{A}$, $\mathfrak{A}'$. *If* $\mathfrak{A} \preceq_f \mathfrak{A}'$ *then* $\mathsf{Lang}(\mathfrak{A}) \subseteq \mathsf{Lang}(\mathfrak{A}')$

This proposition allows us to check validity of a whole set of proposed changes to an automaton at once. To this end we could check whether the quotient of $\mathfrak{A}$ with respect to fair simulation is equivalent to the original automaton. Whenever the check succeeds we now that all merges are correct. However, if it fails we would know nothing. We will generalize this proposition in the following to obtain a more refined check.

The first lemma that we need is a generalization of the transitivity of fair simulation:

Lemma 3. *Let* $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$, $\mathfrak{A}' = (\mathcal{S}', \mathcal{I}', \mathcal{R}', \Omega')$ *be parity automata. If* $s \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A})} s'$ *and* $s' \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')} s''$, *then* $s \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')} s''$.

Proof sketch. The proof of Lemma 1 remains unchanged for this case.

Now we can formulate the following theorem that states that we can safely replace any transition from s to a transition to t for fair-simulation equivalent states t if state t in the modified automaton still fair simulates his counterpart in the unmodified automaton.

Theorem 8. *Let* $\mathfrak{A}$ *be a parity automaton, let* $s, t \in \mathcal{S}$ *such that* $s \neq t$, $s \approx_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A})} t$. *If* $t \approx_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}_{s\to t})} t$ *then any state in* $\mathfrak{A}_{s\to t}$ *is fair simulation equivalent to the state of the same name in* $\mathfrak{A}$.

Proof. We denote with $t' \in \mathcal{S}_{s\to t}$ the state corresponding to $t \in \mathcal{S}$. We first show one direction, the other direction can be shown analogously. More specifically, let $t' \preceq_{\mathsf{f}}^{(\mathfrak{A}_{s\to t},\mathfrak{A})} t$. We show that this is enough to guarantee that any state in $\mathfrak{A}_{s\to t}$ is fair simulated by the state of the same name in $\mathfrak{A}$. To do this, let $p'^{(0)} \in \mathfrak{A}_{s\to t}$ be any state in $\mathfrak{A}_{s\to t}$ and $p^{(0)} \in \mathcal{S}$ be the corresponding state. We want to show that Duplicator has a winning strategy in $\mathfrak{G}_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}_{s\to t})}\left(p'^{(0)},p^{(0)}\right)$. The critical points in this case is that t' is visited in the simulation game, since transitions may not be part of the original automaton.

Since s and t are fair simulation equivalent in the original automaton, we have $t\preceq_{\mathsf{f}}^{(\mathcal{A},\mathcal{A})}s$. Thus, by transitivity, we have $t'\preceq_{\mathsf{f}}^{(\mathfrak{A}_{s\to t},\mathfrak{A})}s$. Let γ_s be the strategy to show $t'\preceq_{\mathsf{f}}^{(\mathfrak{A}_{s\to t},\mathfrak{A})}s$, and γ_t be the strategy to show $t'\preceq_{\mathsf{f}}^{(\mathfrak{A}_{s\to t},\mathfrak{A})}t$. Intuitively, as long as t' is not visited by Spoiler in $\mathfrak{G}_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}_{s\to t}}\left(t'^{(0)},p^{(0)}\right)$, Duplicator simply takes as successor the corresponding state from the original automaton. However, if at some point this is not the case, according to the definition of $\mathcal{R}'$, the original automaton has a transition that ends in s or ends in t. But then, we can continue the play with one of the strategies γ_s or γ_t. To do this, Duplicator uses a memory variable τ. At the beginning, we set $\tau = \surd$. Now assume, we are in state $(p'^{(i)},p^{(i)})$ and Spoiler chooses transition $(p'^{(i)},\alpha^{(i)},p'^{(i+1)})$. The next value of τ, denoted withτ' is determined by:

$$\tau' = \begin{cases} \surd & \text{if } \tau = \surd \wedge p'^{(i+1)} \neq t' \\ s & \text{if } \tau = s \vee p'^{(i+1)} = t' \wedge (p^{(i)},\alpha^{(i)},s) \in \mathcal{R} \\ t & \text{if } \tau = t \vee p'^{(i+1)} = t' \wedge (p^{(i)},\alpha^{(i)},t) \in \mathcal{R} \end{cases}$$

The strategy is now defined as follows:

$$\gamma(p^{(i)}a^{(i)}p'^{(i+1)}) = \begin{cases} p^{(i+1)} & \text{if } \tau' = \surd \\ \gamma_{\tau'}(p^{(i)},a^{(i)},p'^{(i+1)}) & \text{else} \end{cases}$$

Notice that γ' is well defined, because when $\tau = \surd \wedge \tau' \neq \surd$ for the first time i, the run $\pi' = p'^{(0)}\dots p'^{(i)}$ that is induced by Spoiler during the play must have visited t', i. e. we have $p'^{(i)} = t'$. This implies that either $(p^{(i-1)},\alpha^{(i-1)},s) \in \mathcal{R}'$ or $(p^{(i-1)},\alpha^{(i-1)},t) \in \mathcal{R}'$. After that point, the strategy simply mimics one of the strategies γ_s or γ_t. Thus γ is winning since both strategies are winning and the winning condition of fair simulation does not depend on a finite prefix.
The other direction is shown analogously by assuming that $t\preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}_{s\to t})}t'$ and showing that then any state in the original automaton is fair simulated by the state of the same name in the minimized automaton. □

This theorem seems to be very powerful. But it has one drawback that is discussed in the following: Let ϱ be a selection function that selects t as a representative for an equivalence class. Assume that in $\mathfrak{A}\div_{\varrho}$ some state r is reachable from t that is wrongly merged with some state q. This means that the language accepted from r has changed and might potentially change the language accepted from t on. In some (although very unlikely but still possible) cases it might now happen that when we redo the merge of r and q, we might destroy the fair simulation equivalence of t with its counterpart in the original automaton.

So at first sight it seems that we need to check each merge individually. However, this is not the case. If from t only states are reachable that pass the check of Theorem 8, we can be sure that the subautomaton that is reachable from $\varrho(t)$ will never again change and thus this merge is indeed correct. This is captured in the following corollary.

Corollary 4. *Let $\approx \subseteq \approx_f^{(\mathfrak{A},\mathfrak{A})}$, i. e. it holds that if $s \approx t$, then $s \approx_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A})}$. Let ϱ be a canonical selection function for $\approx$. Let*

$$\zeta(s) = \begin{cases} \varrho(s) & \text{if } \forall \alpha \in \Sigma^* \forall q \in \mathcal{S}.(\varrho(s), \alpha, \varrho(q)) \in \mathcal{R}\div_{\varrho} \rightarrow \varrho(q) \approx_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}\div_{\varrho})} \varrho(q) \\ s & \text{else} \end{cases}$$

Then, any state in $\mathfrak{A}\div_{\zeta}$ is fair simulation equivalent to the state of the same name in $\mathfrak{A}$.

Notice that this check can be done by a simple reachability analysis of the state space, so that we obtain an efficient procedure to check multiple changes at one.

3.7.2 Removing Edges using Fair Simulation

Similar to merging states using fair simulation, we can also check removing edges. This is the topic of this subsection.

Theorem 9. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton. Let $r, s, t \in \mathcal{S}$ such that $s \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A})} t$. Let $\mathcal{R}'$ be obtained from $\mathcal{R}$ by removing one transition (r, a, s) provided there is a transition (r, a, t). Let $\mathfrak{A}' = (\mathcal{S}, \mathcal{I}, \mathcal{R}', \Omega)$. Denote with q the states from $\mathfrak{A}$ and with q' the corresponding state in $\mathfrak{A}'$. Then, the following holds: If $r \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')} r'$ then $q \approx_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')} q'$ for any state $q' \in \mathcal{S}'$.*

Proof. Obviously, since we remove an edge in $\mathfrak{A}'$, it is clear that $q' \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')} q$ for any state $q' \in \mathcal{S}'$. The other direction is shown as in the proof of Theorem 8, i. e. as long as Duplicator needs not to take some transition (r, a, s), Duplicator responds with the same transition, otherwise she continues playing according to the winning strategy that shows that $r \preceq_{\mathsf{f}} r'$. □

Since removing more edges makes it harder to show fair simulation, when we found that a state in the more minimized automaton still simulates the original state, this implies that we can safely remove the edge also in a less minimized automaton. This is stated in the following corollary.

Corollary 5. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \Omega)$ be a parity automaton, and $E = \{s, t \in \mathcal{S} \mid s \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A})} t\} t$. Let $\mathcal{R}'$ be obtained from $\mathcal{R}$ by removing one or more transitions (r, a, s)*

provided there is a transition (r, a, t) for a pair $(s, t) \in E$. Let $\mathfrak{A}' = (\mathcal{S}, \mathcal{I}, \mathcal{R}', \Omega)$. Denote with q the states from $\mathfrak{A}$, with q' (q'') the corresponding state in $\mathfrak{A}'$ ($\mathfrak{A}''$). Then, the following holds: If $r \preceq_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A}')} r'$ then any removal of an edge (r, a, s) preserves fair simulation equivalence, i. e. any state in $\mathfrak{A}''$ that is obtained by removing a transition (r, a, s) as in $\mathfrak{A}'$ is fair-simulation equivalent to the state of the same name in $\mathfrak{A}$.

3.7.3 A Fair Minimization algorithm

In this section we describe a method to minimize a parity automaton using the techniques presented before. To perform fair simulation minimization, the fair simulation relation of the automaton $\mathfrak{A}$ is calculated which gives us the fair simulation equivalent states. We perform the successor quotient construction where we replace any state in an equivalence class by a state with minimal color from that class. Afterwards we check which states are still fair simulation equivalent in both automata. The states that have only good successors according to this check are merged according to Corollary 4. Notice that after this step, every pair of states that are left-hand (or right-hand) delayed equivalent are merged, since in that case all successors must be also left-hand (right-hand) equivalent. Afterwards, we pick one equivalence class, merge all states to a minimal representative and check whether the merge succeeded. This is done until we find an equivalence class where this check fails. For this equivalence class, we can still perform minimization, by performing the check according to Theorem 8 for every pair (s, t) we want to merge. When we find a merge that is incorrect, we can update the equivalence relation and restart the process. After having performed merging of states, we perform removal of edges according to Corollary 5. The overall algorithm is sketched in Algorithm 1.

In practice, the chances that merges succeed are quite high. Nevertheless, assume that the number of possible merges of state pairs are k. In the worst case, we have to check all equivalence classes but one before we find the equivalence class that is responsible for a non-successful merge. Clearly, the number of nontrivial equivalence classes is smaller than k. Thus, the overall running time of the algorithm from step 1 until 6 is reached is in a factor of $O(k)$ times in the time of performing the fair simulation equivalence calculation alone. Clearly, $k \in O(|\mathcal{R}|)$. Since we can not remove more than $|\mathcal{S}|$ states or $|\mathcal{R}|$ transitions, the overall run time of this algorithm is in a factor of $O(|\mathcal{R}|^2)$ of performing the fair simulation calculation on $\mathfrak{A}$ alone. Since the calculation of the fair simulation relation is exponential in the number of colors, this quadratic factor does not dominate the overall running time of the algorithm.

Theorem 10. *Let $\mathfrak{A}$ be a parity automaton. Let $n = |\mathcal{S}|$, $m = |\mathcal{R}|$, $d = |\Omega|$. Then the algorithm from Figure 1 has a worst case running time of $O((nm)^{4d} \cdot m^4) \cdot \frac{2d!}{d!^2}$.*

Algorithm 1 A fair minimization algorithm

Input: A parity automaton $\mathfrak{A}$

Output: a minimized parity automaton equivalent to $\mathfrak{A}$

```
1. calculate a canonical selection function $\varrho$ for $\approx_{\mathsf{f}}^{(\mathfrak{A},\mathfrak{A})}$
2. if $\mathfrak{A}\div_{\varrho}\approx_f\mathfrak{A}$ return $\mathfrak{A}\div_{\varrho}$
3. calculate the refinement $\zeta$ for $\varrho$ according to Corollary 4, i. e. merge the states
   with fair simulation equivalent successors
4. let $\mathfrak{A}' = \mathfrak{A}\div_{\zeta}$
5. For every $r \in \mathcal{S}$ such that $\varrho(r) = s$ for some $s \neq r$ (s represents a nontrivial
   equivalence class ) do
   5.1. let $\zeta(s') = \begin{cases} s & \text{if } \varrho(s') = r \\ s' & \text{else} \end{cases}$
   5.2. if $\mathfrak{A}'\div_{\zeta}\approx_f\mathfrak{A}'$ then
      5.2.1. let $\mathfrak{A}' = \mathfrak{A}'\div_{\zeta}$
      5.2.2. restrict $\varrho$ to the states in $\mathfrak{A}'\div_{\zeta}$
      5.2.3. goto 5
   5.3. else
      5.3.1. for every pair $(s,t)$ such that $s\approx_{\mathsf{f}}t$
             • if $\mathfrak{A}'_{s\to t}\approx_f\mathfrak{A}'$ then let $\mathfrak{A}' = \mathfrak{A}'_{s\to t}$
             • else
               – remove $(s,t)$ from the equivalence class, i. e. set $\varrho(s) = s$
               – goto 2
6. Remove edges from $\mathfrak{A}'$ according to Corollary 5
7. if $\mathfrak{A}'$ contains less states or less transitions than $\mathfrak{A}$, continue with step 1 with $\mathfrak{A}'$
   as $\mathfrak{A}$, otherwise return $\mathfrak{A}'$ as the simplified automaton.
```

4 Symbolic Determinization via the Automaton Hierarchy

We have already outlined in the introduction that all algorithms for the solution of the controller synthesis problem rely on some form of (pseudo)-determinization. However, the determinization of ω-automata is considerably more difficult than the determinization of automata on finite words, which is usually done by the well-known Rabin-Scott [84] subset construction. In order to translate nondeterministic Büchi automata to deterministic ω-automata, more difficult algorithms like Safra's algorithm [88] have to be used. However, these algorithms are difficult to implement [41, 110, 54] due to the used complex data structures, which do not allow the use of symbolic set representations like BDDs. As a consequence, the related tools are limited to very small LTL formulas which limits their application in practice.

Due to the lack of efficient determinization algorithms, a recent research trend is to avoid determinization [60, 46] and to integrate the applications with lightweight parts of the determinization procedures. However, at least for model-checking, experimental results [95] indicate that the more deterministic an automaton is, the more efficiently it can be handled. In [5] it is shown that for SAT-based model checking of safety properties, the SAT-Algorithm terminates much faster if one uses deterministic automata instead of nondeterministic automata, although the deterministic automata may be exponentially larger than the nondeterministic ones.

A different approach to deal with the determinization problem is to consider restricted classes of LTL properties [91, 3], so that simpler kinds of ω-automata with simpler determinization procedures can be used. It is well-known that simple modifications of the Rabin-Scott subset construction (including the breakpoint construction of Miyano and Hayashi [70]) can be used for the determinization of some of these classes. In order to apply these determinization procedures to translate LTL formulas to deterministic ω-automata, we exploit the temporal logic hierarchy of [65, 18, 90, 91] as described in Section 2.4. This hierarchy is based on the definition of six classes of temporal logics TL_κ such that all formulas of TL_κ can be efficiently translated to the corresponding automaton class Det_κ. Due to results of [65], it moreover follows that if a temporal logic formula can be translated to Det_κ, then it is equivalent to a formula in TL_κ.

Thus we can compute for every formula $\varphi \in \mathsf{TL}_\kappa$ an equivalent deterministic ω-

automaton $\mathfrak{A}_\varphi \in \mathsf{Det}_\kappa$. Moreover, it is already known that the subset and the breakpoint constructions are sufficient for this purpose, so that there is no need to implement Safra's considerably more difficult determinization procedure for this purpose. However, a critical issue of the known translations from the classes TL_κ to Det_κ is still their complexity: It is well-known [3] that there exists formulas $\Phi \in \mathsf{TL}_\kappa$ such that all equivalent deterministic automata have at least $2^{2^{\Omega|\Phi|}}$ states. Symbolic translation procedures from TL_κ to NDet_κ elegantly circumvent a first bottleneck. However, all determinization procedures including the subset and the breakpoint construction are given for explicitly represented automata, so that this advantage can no longer be exploited. Even worse, the resulting deterministic automata are also given in an explicit representation, so that naive implementations really suffer from the double exponential complexity. For this reason, a major improvement is obtained by the results of this chapter that show how the subset and the breakpoint construction can be implemented in a semi-symbolic way: For a given symbolically represented nondeterministic automaton $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with reachable states $\{\vartheta_1, \ldots, \vartheta_n\}$, we directly construct symbolic descriptions of the deterministic automata that are constructed by the subset and the breakpoint constructions. Although we can not avoid one exponential step (namely the enumeration of the reachable states[1]), we achieved that the symbolic description of the deterministic automaton can be obtained without enumerating its states. All steps except for the enumeration of the reachable states of the nondeterministic automaton are symbolically implemented. As a result, we obtain highly efficient determinization procedures that allow us to translate large LTL formulas to equivalent deterministic automata. In the following, we describe these algorithms in detail, but we assume that the reader is familiar with the subset and the breakpoint construction.

This chapter is divided in three parts: the first part describes the semi-symbolic subset construction. Based on the subset construction, a semi-symbolic variant of the breakpoint construction is obtained. Finally, we describe how by a dependent variable analysis we can reduce the number of propositional variables needed to encode the state transition diagrams of the deterministic automaton.

4.1 A Semi-Symbolic Subset Construction

In Section 2.4, we have sketched by results from [90, 91] that we can compute for every formula $\varphi \in \mathsf{TL}_\kappa$ an equivalent deterministic ω-automaton $\mathfrak{A}_\varphi \in \mathsf{Det}_\kappa$. Besides the translation procedures already given in [90, 91], the main ingredient is the use of the subset and the breakpoint constructions for ω-automata [84, 70, 91]. The remaining

[1]Notice that together with the constructions given in Chapter 3 we only have to enumerate the simulation equivalent states.

problem is that these procedures are only described in the literature for explicitly given automata.

In this section, we therefore explain how these algorithms can be implemented in a semi-symbolic manner[2]. Our algorithms are symbolic in the sense that they can make effective use of symbolic set representations as given by BDDs. However, as we have to explicitly enumerate the reachable states at some point, not all parts of the algorithm can be implemented with symbolic methods. Hence, we obtain *semi-symbolic* algorithms.

As already explained, our algorithms expect a *symbolic* representation of a nondeterministic automaton, i.e. a formula $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$, where $\mathcal{I}$ is a propositional formula over the state variables Q, and where the acceptance condition $\mathcal{F}$ is based on a propositional formula $\mathcal{F}_\varphi$ over the state variables Q that is used to construct a safety, liveness, fairness or co-Büchi property $\mathcal{F}$. In the following, we assume that $Q = \{q_1, \ldots, q_m\}$ and that $V_\Sigma = \{x_1, \ldots, x_k\}$ holds and for notational reasons, we identify $\mathcal{F}$ with $\mathcal{F}_\varphi$.

The first step of our algorithms consists of *computing the reachable states* of the automaton. To this end, we first eliminate all variables that are not state variables, i.e., we define $\mathcal{R}_\exists :\equiv \exists x_1 \ldots x_k.\ \mathcal{R}$, and compute then the reachable states. The reachable states can be computed as the fixpoint of the μ-calculus formula $\mu x.\ \mathcal{I} \vee \overleftarrow{\Diamond}\, x$[3], but since we additionally eliminate deadend states, we compute $\mathcal{S}_{\mathsf{reach}} :\equiv (\nu y. \Diamond y) \wedge (\mu x.\ \mathcal{I} \vee \overleftarrow{\Diamond}\, x)$ instead. Using symbolic methods, the result $\mathcal{S}_{\mathsf{reach}}$ is a propositional formula over the state variables Q.

Having computed the reachable states $\mathcal{S}_{\mathsf{reach}}$ of the automaton, we now perform a onehot encoding of the original nondeterministic automaton $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$. To this end, we have to explicitly enumerate the reachable states which are the variable assignments of the state variables Q that satisfy the formula $\mathcal{S}_{\mathsf{reach}}$. We identify a variable assignment with a set $\vartheta \subseteq Q$ such that exactly the variables contained in ϑ are true. Thus, assume the reachable states are $\{\vartheta_1, \ldots, \vartheta_n\}$. We now introduce new state variables $Q_{\mathsf{oh}} = \{p_1, \ldots, p_n\}$ such that each p_i is identified with a single reachable state, i. e. we use a one-hot encoding. To generate an encoding with the new state variables, we use the *minterms* of the states $\vartheta \subseteq Q$ from the original variables to map old states to new onehot states:

$$\mathsf{mt}_Q(\vartheta) :\equiv \left(\bigwedge_{x \in \vartheta} x\right) \wedge \left(\bigwedge_{x \in Q \setminus \vartheta} \neg x\right)$$

Based on these definitions, the re-encoding of $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ as a onehot automaton

[2]We assume that the reader is familiar with the subset and the breakpoint construction.

[3]For a set of states x, $\overleftarrow{\Diamond}\, x$ denotes the existential successor states of x.

can be easily done as follows:

Definition 34 (Symbolic Onehot Encoding).
Given an automaton $\mathfrak{A} = \mathcal{A}_{\exists}(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with the reachable states $\{\vartheta_1, \ldots, \vartheta_n\}$ and new state variables $Q_{\mathsf{oh}} = \{p_1, \ldots, p_n\}$. Then, we define the following automaton $\mathfrak{A}_{\mathsf{oh}} = \mathcal{A}_{\exists}(Q_{\mathsf{oh}}, \mathcal{I}_{\mathsf{oh}}, \mathcal{R}_{\mathsf{oh}}, \mathcal{F}_{\mathsf{oh}})$:

- $\mathsf{OneHot}(Q_{\mathsf{oh}}) :\equiv \bigwedge_{i=1}^{n} \left(p_i \rightarrow \bigwedge_{j=1, j\neq i}^{n} \neg p_j \right)$
- $\mathcal{H} :\equiv \bigvee_{j=1}^{n} p_j \wedge \mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_j)$
- $\mathcal{I}_{\mathsf{oh}} :\equiv \mathsf{OneHot}(Q_{\mathsf{oh}}) \wedge \exists q_1 \ldots q_m.\ \mathcal{H} \wedge \mathcal{I}$
- $\mathcal{F}_{\mathsf{oh}} :\equiv \exists q_1 \ldots q_m.\ \mathcal{H} \wedge \mathcal{F}$
- $\mathcal{R}_{\mathsf{oh}} :\equiv [(\mathsf{OneHot}(Q_{\mathsf{oh}}))]_{q_1', \ldots, q_n'}^{q_1, \ldots, q_m} \wedge \bigvee_{i=1}^{n} {p_i}' \wedge \eta_i$, *where*
  $\eta_i :\equiv \exists q_1 \ldots q_m {q_1}' \ldots {q_m}'.\ \mathcal{H} \wedge \mathcal{R} \wedge [(\mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_i))]_{q_1', \ldots, q_n'}^{q_1, \ldots, q_m}$

We will state below that $\mathfrak{A}_{\mathsf{oh}}$ encodes the same state transition system as $\mathfrak{A}$, but its encoding with the variables Q_{oh} is a onehot encoding, i.e., each reachable state of $\mathfrak{A}_{\mathsf{oh}}$ corresponds with one of its state variables p_i. To see this, consider first the acceptance condition $\mathcal{F}_{\mathsf{oh}}$: Note that $\mathcal{F}_{\mathsf{oh}}$ can be rewritten as $\bigvee_{j=1}^{n} p_j \wedge \exists q_1 \ldots q_m.\ \mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_j) \wedge \mathcal{F}$. The subformula $\mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_i) \wedge \mathcal{F}$ is false in case that ϑ_i is a variable assignment that does not satisfy $\mathcal{F}$. Otherwise, $\mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_i) \wedge \mathcal{F}$ is equivalent to $\mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_i)$. Thus, the existential quantification used in the definition of $\mathcal{F}_{\mathsf{oh}}$ yields $\mathsf{1}$ if ϑ_i belongs to $\mathcal{F}$ (since every minterm $\mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_i)$ is satisfiable), and yields $\mathsf{0}$ if ϑ_i does not belong to $\mathcal{F}$. Consequently, $\mathcal{F}_{\mathsf{oh}}$ is equivalent to the disjunction of those p_j that correspond to states ϑ_j of $\mathcal{F}$, i.e. $\mathcal{F}_{\mathsf{oh}} \Leftrightarrow \bigvee_{\vartheta_i \in \mathcal{F}} p_i$.

The construction of $\mathcal{I}_{\mathsf{oh}}$ follows the same pattern, and only adds the constraint that at most one of the state variables p_i may be active [4].

Finally, for the correctness of $\mathcal{R}_{\mathsf{oh}}$, note that η_i is equivalent to

$$\bigvee_{j=1}^{n} p_j \wedge \underbrace{\exists q_1 \ldots q_m q_1' \ldots q_m'.\ \mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_j) \wedge \mathcal{R} \wedge [(\mathsf{mt}_{Q_{\mathsf{oh}}}(\vartheta_i)]_{q_1', \ldots, q_n'}^{q_1, \ldots, q_m}}_{\tau_{j,i})}$$

$\tau_{j,i}$ is thereby the condition on the input variables $V_\Sigma = \{x_1, \ldots, x_k\}$ that must hold to enable the transition from state p_j to state p_i. Hence, η_i lists all possibilities to

[4] We have to establish $\mathsf{OneHot}(Q_{\mathsf{oh}})$ as an invariant, and therefore add this constraint to the initial states and the (next) states reachable from other states in $\mathcal{R}_{\mathsf{OH}}$.

reach state p_i from any other reachable state. It is easily seen that $\mathcal{R}_{\mathsf{oh}}$ maintains the invariant that at most one of the state variables p_i can be active due to the conjunct $[\mathsf{OneHot}(Q_{\mathsf{oh}})]^{q_1,\ldots,q_m}_{q_1',\ldots,q_n'}$ which is obtained from $\mathsf{OneHot}(Q_{\mathsf{oh}})$ by replacing each current state variable by the corresponding next state variable.

Theorem 11. (Symbolic Onehot Encoding) *For every automaton $\mathfrak{A}$, the automaton $\mathfrak{A}_{\mathsf{oh}}$ as constructed in Definition 34 is isomorphic to $\mathfrak{A}$. Moreover, its encoding is a onehot encoding, i.e., each state is encoded by a singleton set of the state variables.*

The well-known subset construction that is usually used for the determinization of automata can be symbolically done for automata that are given by a onehot encoding. As we know by Definition 34 how arbitrary automata can be re-encoded with a onehot encoding, we are able to obtain the following symbolic subset construction:

Definition 35 (Symbolic Subset Construction).
Given an automaton $\mathfrak{A} = \mathcal{A}_{\exists}(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with the reachable states $\{\vartheta_1, \ldots, \vartheta_n\}$ and new state variables $Q_{\mathsf{det}} = \{p_1, \ldots, p_n\}$, we define the following automaton $\mathfrak{A}_{\mathsf{det}} = \mathcal{A}_{\exists}(Q_{\mathsf{det}}, \mathcal{I}_{\mathsf{det}}, \mathcal{R}_{\mathsf{det}}, \mathcal{F}_{\mathsf{det}})$:

- $\mathcal{H} :\equiv \displaystyle\bigvee_{j=1}^{n} p_j \wedge \mathsf{mt}_{Q_{\mathsf{det}}}(\vartheta_j)$
- $\mathcal{I}_{\mathsf{det}} :\equiv \displaystyle\bigwedge_{i=1}^{n} p_i \leftrightarrow \exists q_1 \ldots q_m.\ \mathsf{mt}_{Q_{\mathsf{det}}}(\vartheta_i) \wedge \mathcal{I}$
- $\mathcal{F}_{\mathsf{det}} :\equiv \exists q_1 \ldots q_m.\ \mathcal{H} \wedge \mathcal{F}$
- $\mathcal{R}_{\mathsf{det}} :\equiv \displaystyle\bigwedge_{i=1}^{n} {p_i}' \leftrightarrow \eta_i$, *with*
  $\eta_i :\equiv \exists q_1 \ldots q_m {q_1}' \ldots {q_m}'.\ \mathcal{H} \wedge \mathcal{R} \wedge [\mathsf{mt}_{Q_{\mathsf{det}}}(\vartheta_i)]^{q_1,\ldots,q_m}_{q_1',\ldots,q_n'}$

As can be seen, the initial condition and the transition relation of $\mathfrak{A}_{\mathsf{det}}$ are given as equation systems, which is beneficial for many applications. The ideas of the construction are very similar to those used for Definition 34. To explain them, we consider $\mathfrak{A}$ in the re-encoded form $\mathfrak{A}_{\mathsf{oh}}$, so that we can identify the states ϑ_i with the state variables p_i.

The initial superstate of the usual subset construction is the set of initial states $\mathcal{I}$, i.e., the initial condition is $\bigwedge_{i=1}^{n} p_i \leftrightarrow \alpha_i$, where $\alpha_i \in \{\mathsf{1}, \mathsf{0}\}$, so that $\alpha_i = \mathsf{1}$ iff $\vartheta_i \in \mathcal{I}$. By the explanations we gave after Definition 34, it is easily seen that our above construction is therefore correct. The correctness of $\mathcal{F}_{\mathsf{det}}$ is immediately clear.

The transition relation of the usual subset construction is determined as follows: The successor states of a superstate $\Theta \subseteq Q_{\mathsf{det}}$ under the input condition $\tau_{j,i}$ are given

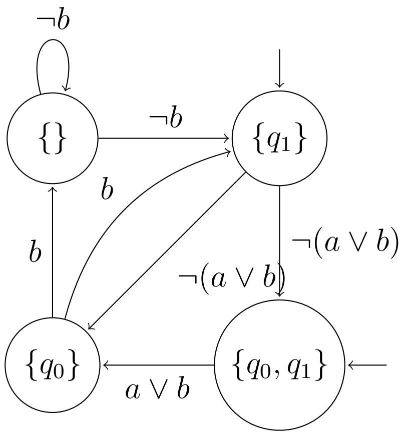


Figure 4.1: Nondeterministic ω-Automaton obtained from $\varphi :\equiv \mathsf{X}\,[a\ \mathsf{U}\ b]$

as the set of states p_j that have a transition under the input condition $\tau_{j,i}$ to a state $p_j \in \Theta$. Our definition of $\mathcal{R}_{\mathsf{det}}$ directly implements this: in the next step, all p_i are true (thus belong to the superstate Θ) where η_i holds. As already explained before, η_i is equivalent to $\bigvee_{j=1}^{n} p_j \wedge \tau_{j,i}$.

Therefore, the above definition implements the subset construction in a symbolic way. We therefore can now state the following theorem:

Theorem 12 (Symbolic Subset Construction)**.** *For every automaton $\mathfrak{A}$, the automaton $\mathfrak{A}_{\mathsf{det}}$ as constructed in Definition 35 is deterministic and is a symbolic description of the automaton obtained by the well-known subset construction [84, 91].*

Since the subset construction can not only be used to determinize automata on finite words, but also ω-automata of the classes $\mathsf{NDet}_{\mathsf{G}}$, $\mathsf{NDet}_{\mathsf{F}}$, and $\mathsf{NDet}_{\mathsf{Prefix}}$ [91], we can already handle these classes with the construction given in Definition 35.

As an example, consider the LTL formula $\varphi :\equiv \mathsf{X}\,[a\ \mathsf{U}\ b]$. Using the translation given in [91], we obtain the following equivalent nondeterministic safety automaton

$$\mathfrak{A}_\varphi = \mathcal{A}_\exists\left(\{q_0, q_1\}, q_1, (q_0 \leftrightarrow b \vee a \wedge {q_0}') \wedge (q_1 \leftrightarrow {q_0}'), 1\right).$$

Its state transition diagram is given in Figure 4.1, and its acceptance condition simply demands that there must be an infinite run. Using the above algorithm, we obtain the following equation systems for $\mathfrak{A}_{\mathsf{det}}$ (where we encoded $p_0 \sim \vartheta_0 = \{\}$, $p_1 \sim \vartheta_1 = \{q_1\}$, $p_2 \sim \vartheta_2 = \{q_0\}$, and $p_3 \sim \vartheta_3 = \{q_0, q_1\}$):

$$\mathcal{I}_{\mathsf{det}} = \begin{cases} p_0 \leftrightarrow 0 \\ p_1 \leftrightarrow 1 \\ p_2 \leftrightarrow 0 \\ p_3 \leftrightarrow 1 \end{cases} \qquad \mathcal{R}_{\mathsf{det}} = \begin{cases} {p_0}' \leftrightarrow p_0 \wedge \neg b \vee p_2 \wedge b \\ {p_1}' \leftrightarrow p_0 \wedge \neg b \vee p_2 \wedge b \\ {p_2}' \leftrightarrow p_1 \wedge \neg(a \vee b) \vee p_3 \wedge (a \vee b) \\ {p_3}' \leftrightarrow p_1 \wedge \neg(a \vee b) \vee p_3 \wedge (a \vee b) \end{cases}$$

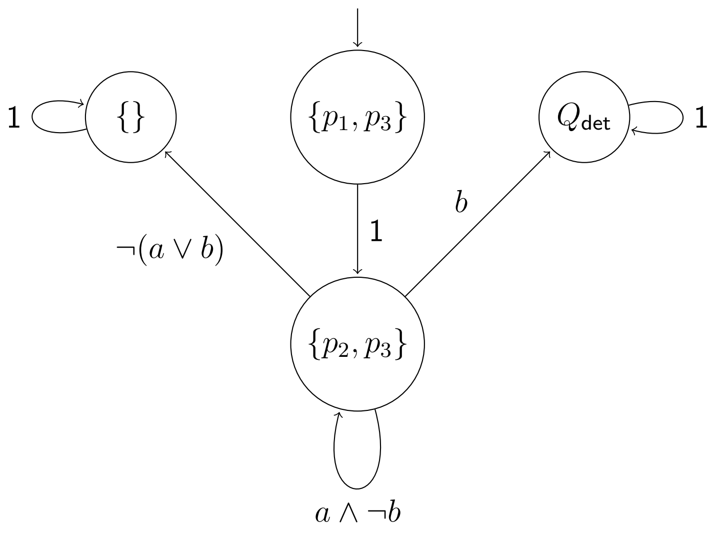


Figure 4.2: Deterministic ω-Automaton for $\varphi :\equiv \mathsf{X}\,[a\ \mathsf{U}\ b]$ obtained from the subset construction with the acceptance condition $\mathsf{G}(p_0 \vee p_1 \vee p_2 \vee p_3)$ and $Q_{\mathsf{det}} := \{p_0, p_1, p_2, p_3\}$.

The state transition diagram of this deterministic automaton is shown in Figure 4.2.

4.2 A Semi-Symbolic Breakpoint Construction

In order to handle further classes of the temporal logic hierarchy, we show in the next definition how the breakpoint construction for the determinization of $\mathsf{Det}_{\mathsf{FG}}$ can be implemented in a symbolic manner. Again, by using dualities between the classes, we are then able to handle the classes $\mathsf{TL}_{\mathsf{GF}}, \mathsf{TL}_{\mathsf{FG}}$ and $\mathsf{TL}_{\mathsf{Streett}}$.

Definition 36 (Symbolic Breakpoint Construction). *Given $\mathfrak{A} = \mathcal{A}_\exists\,(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with the reachable states $\{\vartheta_1, \ldots, \vartheta_n\}$ so that $\mathcal{F}$ is the set $\{\vartheta_{n+1-\ell}, \ldots, \vartheta_n\}$. Using new state variables $Q_{\mathsf{bpt}} = \{p_1, \ldots, p_n, b_1, \ldots, b_\ell\}$ and the definitions of $\mathcal{I}_{\mathsf{det}}$ and $\mathcal{R}_{\mathsf{det}}$ of Definition 35, we define $\mathfrak{A}_{\mathsf{bpt}} = \mathcal{A}_\exists\,(Q_{\mathsf{bpt}}, \mathcal{I}_{\mathsf{det}} \wedge \mathcal{I}_{\mathsf{bpt}}, \mathcal{R}_{\mathsf{det}} \wedge \mathcal{R}_{\mathsf{bpt}}, \mathcal{F}_{\mathsf{bpt}})$ as follows:*

- $\mathcal{H} :\equiv \bigvee_{j=1}^{n} p_j \wedge \mathsf{mt}_{Q_{\mathsf{det}}}(\vartheta_j)$
- $\mathcal{I}_{\mathsf{bpt}} :\equiv \bigwedge_{i=1}^{\ell} b_i \leftrightarrow 0$

- $\mathcal{F}_{\mathsf{bpt}} :\equiv \neg \bigvee_{i=1}^{\ell} b_i$
- $\mathcal{R}_{\mathsf{bpt}} :\equiv \bigwedge_{i=1}^{\ell} b_i{}' \leftrightarrow \left(\mathcal{F}_{\mathsf{bpt}} \wedge \eta_i \vee \neg\mathcal{F}_{\mathsf{bpt}} \wedge [\eta_i]_{\varrho}\right)$, *with*
  $\eta_i :\equiv \exists q_1 \dots q_m q_1{}' \dots q_m{}'.\ \mathcal{H} \wedge \mathcal{R} \wedge [(\mathsf{mt}_{Q_{\mathsf{det}}}(\vartheta_i))]_{q_1',\dots,q_n'}^{q_1,\dots,q_m}$
  and ϱ *is the substitution that maps each* $p_1, \dots, p_{n-\ell}$ *to* $\mathsf{0}$ *and* $p_{n+1-\ell}, \dots, p_n$ *to* $b_1, \dots, b_\ell$, *respectively*

The idea behind the breakpoint construction is to maintain pairs of sets of states, where the first component is computed by the subset construction. The second component is the set of states that have never left the set of designated states since the last breakpoint, where a breakpoint is a state whose second component is empty.

States of $\mathfrak{A}_{\mathsf{bpt}}$ correspond with subsets of Q_{bpt} which may be considered as pairs of subsets of $\{p_1, \dots, p_n\}$ and $\{b_1, \dots, b_\ell\}$. A breakpoint is a pair (S_1, S_2) where S_2 represents an empty state set, i.e., all of the variables b_j are false and $\mathcal{F}_{\mathsf{bpt}}$ evaluates to true. Whenever a breakpoint is reached, the second set is filled with the designated successors of the first set. In this case, we evaluate the status of the b_j according to the variables p_i, thus all we have to do is to copy the formula representing the transition relation of the subset construction. Otherwise, the usual subset construction is performed on the second step in the explicit breakpoint construction. To calculate the transition relation in our symbolic setting, it is sufficient to eliminate transitions from non-accepting states (which is done by setting $p_i = 0$) and then replacing each occurrence of p_i for $\vartheta_i \in \mathcal{F}$ by the corresponding b_i.

Hence, also the breakpoint construction can be implemented in a symbolic manner. It is well-known that $\mathfrak{A}_{\mathsf{bpt}}$ may have at most $O(3^n)$ reachable states, while the automaton $\mathfrak{A}_{\mathsf{det}}$ obtained from the subset construction may have at most $O(2^n)$ reachable states.

Theorem 13 (Symbolic Breakpoint Construction)**.** *For every automaton* $\mathfrak{A}$*, the automaton* $\mathfrak{A}_{\mathsf{bpt}}$ *as constructed in Definition 36 is deterministic and is a symbolic description of the automaton obtained by the well-known breakpoint construction [70, 91].*

As an example for the application of the symbolic breakpoint construction, consider the LTL formula $\varphi := \mathsf{G}\,(a \rightarrow \mathsf{F}b)$ from $\mathsf{TL}_{\mathsf{GF}}$. Since this formula corresponds to a deterministic Büchi automaton, the first step consists of negating the formula. We obtain $\overline{\varphi} = \mathsf{F}\,(a \wedge \mathsf{G}\neg b)$. Using the translation given in [91], we obtain the equivalent nondeterministic co-Büchi automaton $\mathfrak{A}_{\overline{\varphi}} = \mathcal{A}_{\exists}\left(\{q_0, q_1, q_2\}, \neg q_1 \wedge \neg q_2, \Phi_{\mathcal{R}}, \mathsf{FG}q_2\right)$ with

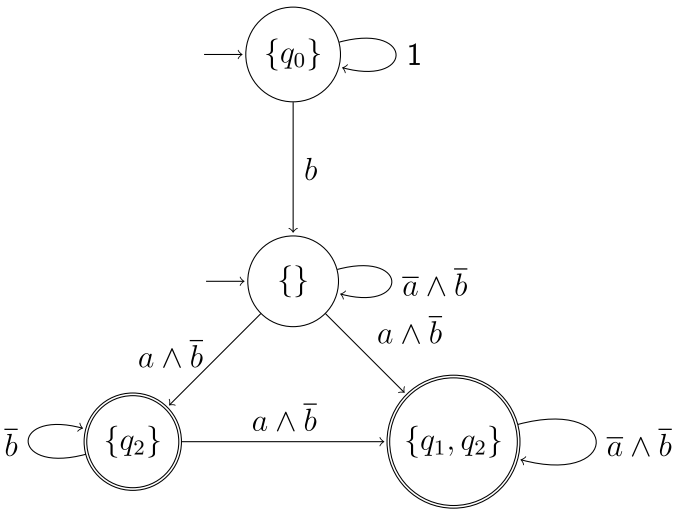


Figure 4.3: Nondeterministic Co-Büchi automaton obtained from $\varphi :\equiv \mathsf{F}\,(a \wedge \mathsf{G}\neg b)$

$$\begin{aligned}\Phi_{\mathcal{R}} = {} & (q_0 \leftrightarrow b \vee q_0{}')\wedge \\ & (q_1 \leftrightarrow (q_0 \vee \neg a) \wedge q_1{}')\wedge \\ & (q_2{}' \leftrightarrow (q_2 \vee ((q_0 \vee \neg a) \rightarrow q_1))\end{aligned}$$

Its state transition diagram is given in Figure 4.3. We encode this automaton using onehot variables $p_0 \sim \vartheta_0 = \{q_0\}$, $p_1 \sim \vartheta_1 = \{\}$, $p_2 \sim \vartheta_2 = \{q_2\}$ and $p_3 \sim \vartheta_3 = \{q_1, q_2\}$. Additionally we introduce the Breakpoint variables b_2 and b_3 for the accepting states ϑ_2 and ϑ_3. Using the symbolic breakpoint construction of Definition 36, we obtain the following equation system for $\mathfrak{A}_{\mathsf{bpt}}$:

$$\mathcal{I}_{\mathsf{bpt}} = \begin{cases} p_0 \leftrightarrow 1 \\ p_1 \leftrightarrow 1 \\ p_2 \leftrightarrow 0 \\ p_3 \leftrightarrow 0 \\ b_2 \leftrightarrow 0 \\ b_3 \leftrightarrow 0 \end{cases}$$

$$\mathcal{R}_{\mathsf{bpt}} = \begin{cases} p_0{}' \leftrightarrow p_0 \\ p_1{}' \leftrightarrow p_0 \wedge b \vee p_1 \wedge \neg a \wedge \neg b \\ p_2{}' \leftrightarrow p_1 \wedge (a \wedge \neg b) \vee p_2 \wedge \neg b \\ p_3{}' \leftrightarrow p_1 \wedge (a \wedge \neg b) \vee p_2 \wedge (a \wedge \neg b) \vee p_3 \wedge \neg a \wedge \neg b \\ b_2{}' \leftrightarrow \neg(b_2 \vee b_3) \wedge p_1 \wedge (a \wedge \neg b) \vee p_2 \wedge \neg b \ \vee (b_2 \vee b_3) \wedge b_2 \wedge \neg b \\ b_3{}' \leftrightarrow \neg(b_2 \vee b_3) \wedge (p_1 \wedge (a \wedge \neg b) \vee p_2 \wedge (a \wedge \neg b) \vee p_3 \wedge \neg a \wedge \neg b) \\ \qquad \vee(b_2 \vee b_3) \wedge (b_2 \wedge \neg b \vee b_3 \wedge \neg a \wedge \neg b) \end{cases}$$

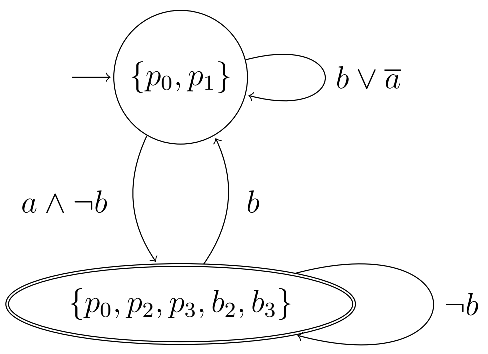


Figure 4.4: Deterministic Co-Büchi automaton obtained from $\varphi :\equiv \mathsf{F}\,(a \wedge \mathsf{G}\neg b)$

The acceptance condition is given by $\mathcal{F}_{\mathsf{bpt}} = b_2 \vee b_3$ so that we obtain the automaton given in Figure 4.4. To obtain an automaton for our original formula φ, all we have to do is to dualize the acceptance condition. Thus $\mathfrak{A}_\varphi = \mathcal{A}_\exists\,(Q_{\mathsf{bpt}}, \mathcal{I}_{\mathsf{bpt}}, \mathcal{R}_{\mathsf{bpt}}, \mathsf{GF}\neg\mathcal{F}_{\mathsf{bpt}})$ is equivalent to φ.

Although the automaton is minimal in the sense that it has a minimal number of states, it is not minimal regarding the number of state variables. Therefore, we present in the following a way to minimize the number of state variables by a dependent variable analysis.

4.3 Removing Dependent Variables

After computing the determinized automaton, we perform *dependent variable analysis* [47] on the set of reachable states to simplify the representation of the propositional formula that are used to encode the transition relation.

Definition 37 ([47])**.** *Given a Boolean function f over $x_0, x_1, \ldots, x_n$, a variable x_i is functionally dependent in f iff $\forall x_i.\ f = 0$.*

Notice that if x_i is functionally dependent, it is uniquely determined by the remaining variables of f and can be replaced by a suitable function $g(x_0, x_1, \ldots x_{i-1}, x_{i+1}, \ldots x_n)$. It has been shown in [47] that removing dependent variables can dramatically speed up the overall running time of BDD based model checking, so that we perform it here on the small subautomata that are obtained after minimization.

Example 1. *A dependent variable analysis on the automaton from Figure 4.4 is quite efficient and yields a deterministic automaton formula*

$$\mathcal{A}_\exists\,(\{p\}, p, p' \leftrightarrow b \vee p \wedge \neg a, \mathsf{GF}p)$$

5 Symbolic Determinization via Unambiguous Büchi Automata

In Chapter 4 a determinization procedure is presented that syntactically localizes a LTL formula in the temporal logic hierarchy and based on this knowledge chooses an appropriate determinization procedure, i. e. either the subset or the breakpoint construction. Although nearly all formulas commonly used in practice belongs to one of the classes of the temporal logic hierarchy (or can be rather easily translated to a formula belonging to one of the classes), there may be still cases where such a manual rewriting step is undesirable.

In this chapter, we therefore present a new determinization procedure for Büchi automata that stem from the translation of arbitrary LTL formulas by the 'standard' translation of [21] that is sketched in Section 2.3.1. It is well-known that the ω-automata that stem from LTL formulas are a special class that has already found several characterizations. Due to results of [67], the automata can be characterized as *non-counting* automata, and in terms of alternating automata, the class of *linear weak* or *very weak* automata has been defined [68, 35, 77, 74]. Moreover, many translation procedures from LTL generate *unambiguous automata* [17] where every accepted word has a unique accepting run [91, 2] (although there may be additional non-accepting runs for the same word). The determinization procedure presented in this chapter makes use of the fact that the automata generated from LTL are unambiguous. Without useless states, the transition relation of an unambiguous automaton has a certain form that we call *non-confluence* (see Definition 38 for a precise definition):

> *An automaton is non-confluent if whenever two runs of the same infinite word meet at a state q, then they must share the entire finite prefix up to state q.*

The above non-confluence property allows us to develop a determinization procedure that exploits symbolic set representations. In particular, it does not rely on Safra trees as used by Safra's original procedure [88] or by the improved version of Piterman [78]. The states of the deterministic automata obtained by these procedures are trees of subsets of states of the original automaton. In contrast, our procedure generates deterministic automata whose states consist of n-tuples of subsets of states, where n is the number of states of the nondeterministic automaton.

The non-confluence property has already been used in [27] to obtain a deterministic (Rabin) automaton from a nondeterministic Büchi automaton. However, the algorithm of [27] still uses a tree structure and is therefore not well suited for a symbolic implementation. In contrast, our automata are amenable to a symbolic implementation and are additionally defined with the simpler parity acceptance condition which further reduces the complexities for game solving and emptiness checks.

The outline of this chapter is as follows: In the next section, the relationship between unambiguous automata and non-confluence is discussed. Here, we show first that unambiguity implies non-confluence (but not vice versa) and that the minimization techniques presented in Chapter 3 preserves the unambiguous property. The core of this chapter is the determinization procedure described in Section 5.2 that is a specialization of Safra's procedure for non-confluent automata. In Section 5.3, we discuss a symbolic implementation of this algorithm.

5.1 Properties of Unambiguous Automata

The unambiguous automata have some special features that makes them very appealing. One is that every finite prefix of a run is uniquely determined by the last visited state. This is captured in the following definition:

Definition 38 (Non-Confluent Automata).
An ω-automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ is called non-confluent if for every word α the following holds: if ξ_1 and ξ_2 are two runs of $\mathfrak{A}$ on α that intersect at a position t_0 (i.e. $\xi_1^{(t_0)} = \xi_2^{(t_0)}$ holds), then we have $\xi_1^{(t)} = \xi_2^{(t)}$ for every $t \leq t_0$.

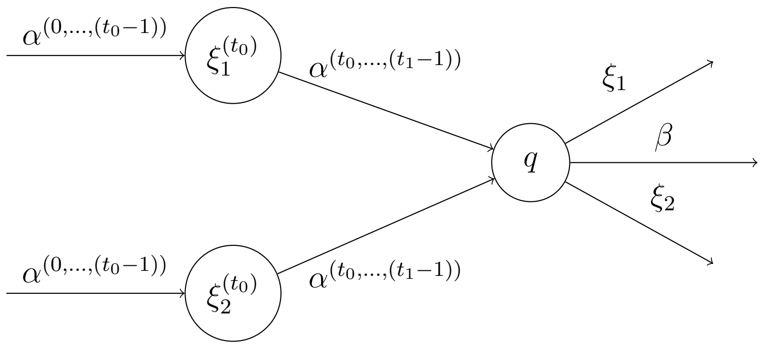
To simplify some of the considerations, it is assumed that the Büchi automata that are considered in the following do not have useless states. That means that for any state $q \in \mathcal{S}$ the following holds:

- q is reachable from at least one initial state
- $\mathsf{Lang}(\mathfrak{A}_q) \neq \emptyset$

Without useless states, unambiguity implies non-confluence:

Lemma 4. *Any unambiguous automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ without useless states is non-confluent.*

Proof. Assume for contradiction that $\mathfrak{A}$ is unambiguous, has no useless state, but is not non-confluent. Because of the latter, there is a word α with two runs ξ_1 and ξ_2, a position $t_1 \in \mathbb{N}$ with $\xi_1^{(t_1)} = \xi_2^{(t_1)} = q$, but there is also a $t_0 < t_1$ with $\xi_1^{(t_0)} \neq \xi_2^{(t_0)}$ so that we obtain the situation shown in Figure 5.1. Since $\xi_1^{(t_1)} = \xi_2^{(t_1)} = q$ is not useless,



$\alpha^{(0,\dots,(t_0-1))}$
$\xi_1^{(t_0)}$
$\alpha^{(t_0,\dots,(t_1-1))}$
q
ξ_1
β
ξ_2
$\alpha^{(0,\dots,(t_0-1))}$
$\xi_2^{(t_0)}$
$\alpha^{(t_0,\dots,(t_1-1))}$

Figure 5.1: Situation in the proof of Lemma 4

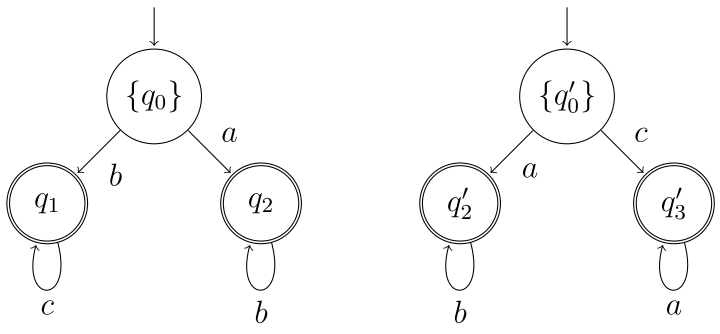


Figure 5.2: A Nonconfluent Automaton that is Ambiguous

there must exist a word β that is accepted from q. Since the prefixes $\xi_1^{(0,\dots,(t_0-1))}$ and $\xi_2^{(0,\dots,(t_0-1))}$ for reading $\alpha^{(0,\dots,(t_0-1))}$ start in initial states and end in q, we can construct two different runs for $\alpha^{(0,\dots,(t_0-1))}\beta$. Thus, $\mathfrak{A}$ cannot be unambiguous. □

Clearly, there exists unambiguous automata that are not non-confluent. We can make an unambiguous and non-confluent automaton confluent by adding a self-loop to a non-accepting state. Moreover, non-confluence does not imply unambiguity as Figure 5.2 shows. This automaton is non-confluent but ambiguous, since two runs exist for ab^ω.

As already outlined, minimizing automata is crucial to obtain manageable deterministic automata. The following theorem shows that the unambiguity property is stable under minimization according to the forward simulation relations.

Theorem 14 (Minimizing Unambiguous Automata)**.** *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ be an unambiguous automaton with $(p, q) \in \mathcal{S}$ such that any state in $\mathfrak{A}_{p\to q}$ is fair-simulation* *equivalent to the state of the same name in $\mathfrak{A}$. Then $\mathfrak{A}_{p\to q}$ is unambiguous.*

Proof. Assume by contradiction that there does exist two different runs π_1, π_2 of $\mathfrak{A}_{p\to q}$ labeled with the same accepted word α. We consider first the case that at the initial position the two runs differ. Since every state in $\mathfrak{A}_{p\to q}$ is fair simulation equivalent to the state of the same name in $\mathfrak{A}$, $\mathfrak{A}$ must accept α from both states $\pi_1^{(0)}$ and $\pi_2^{(0)}$, a contradiction to $\mathfrak{A}$ being unambiguous.

Otherwise, there must exist a minimal position t_0 such that $\pi_1^{(t_0)} \neq \pi_2^{(t_0)}$ and for every $n < n_0$ we have $\pi_1^{(t)} = \pi_2^{(t)}$. That implies that from both states $\pi_1^{(t_0)}$ and $\pi_2^{(t_0)}$, the same word $\alpha^{(t_0\cdots)}$ is accepted by $\mathfrak{A}$ since the states are fair simulation equivalent in both automata. If both runs $\pi_1^{(0\ldots t_0)}$ and $\pi_2^{(0\ldots t_0)}$ are runs of $\mathfrak{A}$ we get a contradiction to the unambiguity of $\mathfrak{A}$. Otherwise there must exist a maximal position $t < t_0$ such that $\pi_1^{(t)} = \pi_2^{(t)} = q$, and for some state $u \in \mathcal{S}$ we have $\pi_1^{(t+1)} = \pi_1^{(t+1)} = u$ and $(q, \alpha^{(t)}, u) \notin \mathcal{R}$. But then we must have that $(p, \alpha^{(t)}, u) \in \mathcal{R}$. Let α_0 be any word leading to $p \in \mathfrak{A}$ and π_0 be the corresponding run. Since we have chosen t maximal, the word $\alpha_0\alpha^{(t\cdots)}$ is accepted by $\mathfrak{A}$ with the two different runs $\pi_0 p \pi_1^{(t+1\ldots)}$ and $\pi_0 p \pi_2^{(t+1\ldots)}$, a contradiction to $\mathfrak{A}$ being unambiguous. □

Since any simplification of an automaton is either the merge of two states p, q as done in $\mathfrak{A}_{p\to q}$ that preserves fair-simulation equivalence of any state or the removal of transitions (that can clearly not destroy unambiguity), this theorem implies that the forward simulation minimizations do not destroy the unambiguity of an automaton. Notice that the previous theorem does not hold for non-confluent automata: fair minimizing a non-confluent automaton might destroy the non-confluence property. To see this, consider again the non-confluent, but ambiguous automaton given in Figure 5.2. Since q_2 and q_2' are direct-simulation equivalent, we can merge the two states. It is however straightforward to see that the obtained automaton is no longer non-confluent.

The following theorem shows that minimization with respect to reverse simulation preserves non-confluence[1].

Theorem 15. *Let $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ be a non-confluent Büchi automaton and $s, t \in \mathcal{S}$ such that $s \neq t$ and $s \approx^{(\mathfrak{A},\mathfrak{A})}_{\overleftarrow{\mathsf{de}}_{\mathsf{I}}} t$. Let $\mathfrak{A}' = (\mathcal{S} \setminus \{s\}, \mathcal{I}, \mathcal{R}', \mathcal{F})$ be defined by*

$$\mathcal{R}'(q, a, q') :\Leftrightarrow (\mathcal{R}(q, a, q') \wedge q \neq s) \vee (q = t \wedge \mathcal{R}(s, a, q'))$$

i. e. we remove any transition starting from s and add those transition to the set of transitions starting in t. Additionally, we remove any dead-end from $\mathfrak{A}'$. Then $\mathfrak{A}'$ is non-confluent.

[1]We do not yet know whether unambiguity is also preserved. However, non-confluence is everything that we need for our determinization construction.

Proof. Notice that since s and t are reverse left-hand simulation equivalent, every state in $\mathfrak{A}'$ is reverse left-hand simulation equivalent to the state of the same name in $\mathfrak{A}$ according to Theorem 6. Assume by contradiction that $\mathfrak{A}'$ is confluent and let π, τ be two different runs of $\mathfrak{A}'$ over the same finite word α such that for some $t_0 \in \mathbb{N}$ we have $\pi^{(t_0)} \neq \tau^{(t_0)}$ and $\pi^{(t_0+1)} = \tau^{(t_0+1)}$. Since states of the same name in $\mathfrak{A}$ and $\mathfrak{A}'$ are reverse left-hand simulation equivalent, there must exist two different runs over α in $\mathfrak{A}$ that lead to $\pi^{(t_0)}$ and $\tau^{(t_0)}$. If we have $(\pi^{(t_0)}, \alpha^{(t_0)}, \pi^{(t_0+1)}) \in \mathcal{R}$ and $(\tau^{(t_0+1)}, \alpha^{(t_0)}, \tau^{(t_0+1)}) \in \mathcal{R}$, we have constructed two different runs over the same word $\alpha^{(0 \dots t_0)}$ leading to $\pi^{(t_0+1)} = \tau^{(t_0+1)}$. Otherwise, either $\pi^{(t_0)} = t$ or $\tau^{(t_0)} = t$. Without loss of generality, assume that $\pi^{(t_0)} = t$, the other case is handled analogously. Notice that according to the definition of $\mathfrak{A}'$, there must exist a transition $(s, \alpha^{(t_0)}, \pi^{(t_0+1)}) \in \mathcal{R}$. Notice that t of $\mathfrak{A}'$ is reverse-simulation equivalent to the state of the same name in $\mathfrak{A}$ and s and t are reverse-simulation equivalent to t in $\mathfrak{A}$. Hence $s \approx^{(\mathfrak{A},\mathfrak{A}')}_{\overleftarrow{\mathsf{de}_l}} t$. This means that there must also exist a path in $\mathfrak{A}$ labeled with the same word α that leads to s in $\mathfrak{A}$. Since we removed any dead-end from $\mathfrak{A}'$, s is removed from $\mathfrak{A}'$ which gives us that $\tau^{(t_0)} \neq s$. But since $\tau^{(t_0)}$ is reverse simulation equivalent in $\mathfrak{A}$ and $\mathfrak{A}'$, there must also exist a path in $\mathfrak{A}$ that is labeled with $\alpha^{(0 \dots t_0-1)}$ and leads to $\tau^{(t_0)}$. That means that we have constructed two different runs of $\mathfrak{A}$ labeled with the same word that lead to $\pi^{(t_0+1)} = \tau^{(t_0+1)}$, a contradiction to $\mathfrak{A}$ being non-confluent. $\square$

Since the Büchi automata that are obtained from a LTL formula are unambiguous, we can first minimize the automata with respect to some forward simulation relation (which preserves unambiguity) and later with reverse simulation (which at least preserves non-confluence). This is important in the next paragraph where the knowledge that an automaton is non-confluent is used to develop a determinization procedure[2].

5.2 Determinizing Nonconfluent Automata

The well-known subset construction [84] collects the sets of states that the nondeterministic automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ can reach after having read a finite input word from one of its initial states $\mathcal{I}$. Thus, every state $\tilde{s} \subseteq \mathcal{S}$ of the deterministic automaton $\tilde{\mathfrak{A}}$ is a set of states of $\mathcal{S}$. The final states $\tilde{\mathcal{F}}$ are those states $\tilde{s} \subseteq \mathcal{S}$ that contain an accepting state of $\mathfrak{A}$, i.e. where $\tilde{s} \cap \mathcal{F} \neq \emptyset$ holds.

The acceptance condition of a Büchi automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ is also specified with a set of accepting states $\mathcal{F}$. However, the subset construction is not sufficient to handle the Büchi acceptance, so that the more complex construction of Safra [88] is often used instead. The idea of Safra's construction is to define so-called breakpoints

[2]Thus non-confluence can be seen as a technical vehicle to state that each finite run of an unambiguous automaton is uniquely determined by the last visited state.

on the path $\tilde{\xi}$ of a word α through $\tilde{\mathfrak{A}}$ so that all paths ξ contained in $\tilde{\xi}$ must visit at least once the accepting states $\mathcal{F}$ in between two subsequent breakpoints. A path $\tilde{\xi}$ is then accepting iff it visits infinitely often breakpoints with non-empty state sets. To this end, the states of the automaton obtained by Safra's construction are trees of subsets of states.

Our determinization procedure is a specialization of Safra's procedure for non-confluent automata. Given a nondeterministic Büchi automaton with n states, the states of the constructed automaton are n-tuples of pairs with (S_i, f_i, e_i) where $S_i \subseteq \mathcal{S}$ and $f_i, e_i \in \{0, 1\}$ flag the visit of accepting states or the clearance of a set. We start with the initial state $((\mathcal{I}, 0, 1), (\emptyset, 0, 1), \dots, (\emptyset, 0, 1)$. To compute the successor of a state $((S_0, e_0, f_0), \dots, (S_{n-1}, e_{n-1}, f_{n-1})$ under input σ, we first compute the existential successors $S'_i := \mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_i)$ for the subsets of states which is the set

$$\mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_i) = \{q' \in \mathcal{S} \mid \exists q \in \mathcal{S}.(q, \sigma, q') \in \mathcal{R}\}. \tag{5.1}$$

Hence, the first state set S_0 of a tuple state is the result of the subset construction, i.e., it contains the sets of states that $\mathfrak{A}$ can reach after having read a finite input word from one of its initial states $\mathcal{I}$. The other sets S_i with $i > 1$ are subsets of S_0 that are generated as follows: whenever the set of successors of S_i is empty, i. e. whenever $\mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_i) = \emptyset$, we flag e'_j for $j \geq i$. Now, whenever $e'_i \neq 0$, the ordinary subset construction is applied, i. e. we set $S'_i = \mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_i)$. Otherwise, all runs of some set on the left (or the runs of the set S_i itself) reached a dead-end. In this case simply the subset construction of the direct right neighbor is taken over, i. e. we set $S'_i = \mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_{i+1})$. In case we are on the rightmost set S_{n-1}, we take over the marked states of S_0 so that this marking gets noticed on a state set on the right. We can however cleanup the states $S_0, \dots S_{n-1}$, so that not all combinations of state sets can occur: As $\mathfrak{A}$ is non-confluent, we know that a finite run is uniquely characterized by its last visited state. Hence, if a state occurs in a set S_i and parallel in some S_j , then we know that both sets follow the same run. Hence whenever we find that a state set S_i contains only states that also occur in a state set S_j for $j > i$, we know that all runs in S_i has visited a marked state recently. Accordingly we mark S_i as accepting by setting its mark $f'_i := 1$ and remove the states S_i from $S_{j'}$ for every $j' > i$. As a consequence each S_i must contain at least one $q \in S_i$ that is not contained in any S_j where $i < j$. This shows that n state sets are sufficient[3].

If an entry S_i never becomes empty after a certain position on a path $\tilde{\xi}$ and is

[3]In [71] we presented a determinization procedure based on the same idea that did not use the empty-flag. The construction given there has a error in that it needs $n + 1$ state sets and not n state sets. This is due to the reason that whenever all state sets are full (but one of them will dead-end), the construction of [71] will not notice a new marking, since in [71] marked states are only introduced in empty sets. Having the empty flag avoids this additional set.

marked infinitely often, then we know that $\tilde{\xi}$ is introduced infinitely often in M and hence $\tilde{\xi}$ contains an accepting run of $\mathfrak{A}$.

This intuitive idea if formalized in the following definition:

Definition 39 (Determinization of Non-Confluent Automata). *Given a nondeterministic Büchi automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with $|\mathcal{S}| = n$, we construct a deterministic parity automaton $\mathfrak{P} = (\mathcal{S}', s_\mathcal{I}, \Delta, \Omega)$ as follows:*

- *The states of the parity automaton are n-tuples of subsets of $\mathcal{S}$ augmented with two boolean flags:*
  $\mathcal{S}' = \{((S_0, e_0, f_0), \dots, (S_{n-1}, e_{n-1}, f_{n-1}) \mid S_i \subseteq \mathcal{S} \wedge f_i, e_i \in \{0, 1\}\}$.
- *The initial state is $s_\mathcal{I} = ((\mathcal{I}, 1, 0), (\emptyset, 1, 0), \dots, (\emptyset, 1, 0))$.*
- *The transition function Δ is determined by the following rules: Given a state $s = ((S_0, e_0, f_0), \dots, (S_{n-1}, e_{n-1}, f_{n-1}))$, the successor state*
  $s' = ((S'_0, e'_0, f'_0), \dots, (S'_{n-1}, e'_{n-1} f'_{n-1}))$ of automaton $\mathfrak{P}$ when reading input σ is determined by the following rules for $i = 0 \dots n-1$

$$e'_i := \left(\mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_i) = \emptyset\right) \vee e'_{i-1}$$

$$f'_i := \begin{cases} \left(\mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_i) \setminus \mathcal{F}\right) \subseteq \left(\bigcup_{j=i+1}^{n} \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_j)\right) & \text{if } e'_i = 0 \\ 0 & \text{else} \end{cases}$$

$$S'_0 = \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_i)$$

$$S'_{(i, i \notin \{0,n\})} = \left(\begin{cases} \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_i) & \text{if } e'_i = 0 \\ \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_{i+1}) & \text{else} \end{cases}\right) \setminus \bigcup_{j=0, f'_j=1}^{i-1} S'_j$$

$$S'_{n-1} = \left(\begin{cases} \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_{n-1}) & \text{if } e'_{n-1} = 0 \\ \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_0) \cap \mathcal{F} & \text{else} \end{cases}\right) \setminus \bigcup_{j=0, f'_j=1}^{n-2} S'_j$$

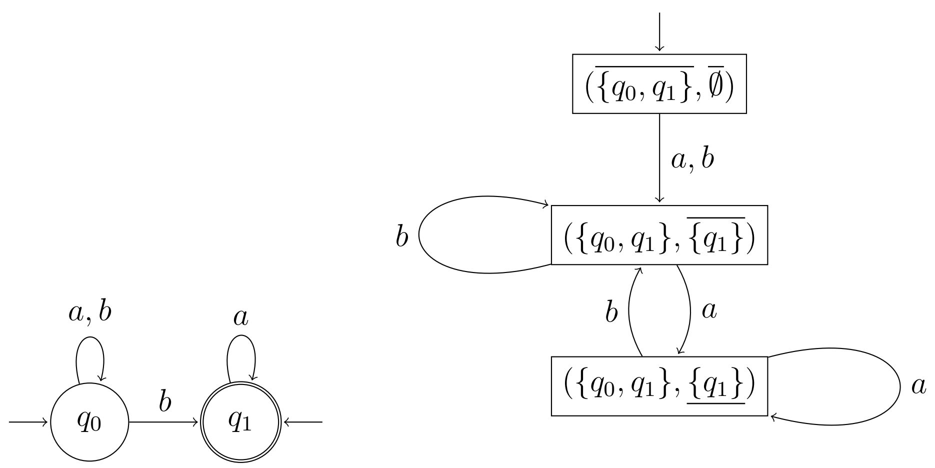
- *To determine the color of a state $s = ((S_0, e_0, f_0), \dots, (S_{n-1}, e_{n-1}, f_{n-1})$ set $e = \min\{i \mid e_i = 1\}$ and $f = \min\{i \mid f_i = 1\}$. We define:*

$$\Omega(s) := \begin{cases} 1 & \text{if } e = 0 \\ 2 * i & \text{if } f < e \wedge f = i \\ 2 * i - 1 & \text{if } e \leq f \wedge e = i \end{cases}$$

We have already given some explanations regarding the transition relation. However, to give a connection to the formal definition, notice that $\bigcup_{j=0,f'_j=1}^{i-1} S'_j$ is for every set S_i the set of states on the left that have currently set the flag signal f, i. e. for which it is currently detected that every surviving run has visited a marked state recently. According to the previous explanation, those states need only be considered once in the tuple, so that we remove them from S'_i.

For the definition of the coloring function, notice that the clearance of a state set S_i noticed by the flag e_i is a bad event that unwinds a previously marking f_i. Moreover, notice that if $S_0 = \emptyset$, the state s is a rejecting sink state, since from that point on we have $S'_0 = \emptyset$ for every successor state.This state corresponds to a situation in the nondeterministic automaton where all runs lead to a dead end.

As a first example for the construction, consider the Büchi automaton together with its deterministic automaton given in Figure 5.3. To improve readability, we omitted the boolean flags, and instead overlined those state sets S_i where e_i holds and underlined those state sets S_i where f_i holds. This example gives a justification why we need the flags e_i: Without the flag e_0, both states at the bottom of the deterministic automaton would merge to one single state which would make it impossible to determine whether the run should be accepted or not. Having the flag, it is clear that whenever the run enters state $(\{q_0, q_1\}, \overline{\{q_1\}})$, the run previously ending in q_1 has reached a dead-end and thus the run is not accepting.


a, b
a
b
q_0
q_1
$(\overline{\{q_0, q_1\}}, \overline{\emptyset})$
a, b
b
$(\{q_0, q_1\}, \overline{\{q_1\}})$
b
a
$(\{q_0, q_1\}, \underline{\{q_1\}})$
a

(a) Nondeterministic Automaton

(b) Corresponding Deterministic Automaton

Figure 5.3: Determinizing Non-Confluent Automata

As a second example, consider the automaton in Figure 5.4. This automaton accepts every word that either ends with suffix ab^ω or cb^ω or that contains infinitely many occurrences of a. On the right of this figure an example run with intermediate steps (in gray) is shown. The first interesting transition if the first occurrence of c. Here the run that was previously in q_1 dead-ends and hence in an intermediate step, the rightmost subset becomes empty. At the end of this step, this set is filled with the marked successors of S_0 under c which gives us the state drawn in the middle of the run. The next transition, labeled with b is even more interesting: as an intermediate step, state set S_1 gets filled with the successors of S_2, which gives us the intermediate state $(\{q_0, q_1, q_2\}, \underline{\overline{\{q_2\}}}, \overline{\emptyset})$. As the final step of this transition, the marked successors of set S_0 under b, namely q_1 determine the new entry of S_2. Finally, the last drawn transition is obtained due to the following rules: q_2 has no a-successor, hence S_1 is cleared. The successors of q_1 are $\{q_0, q_1, q_2\}$ so in an intermediate step we obtain the state $(\{q_0, q_1, q_2\}, \underline{\overline{\emptyset}}, \overline{\{q_1, q_2, q_3\}})$. Now we check that every (non-marked) state of S_0 occurs also on the right, which leads to the marking of S_1 and finally to the state in the bottom. The complete deterministic automaton is shown in Figure 5.5.

After having defined our determinization construction formally and give some explanatory examples,the following Lemma states some important properties of the construction:

Lemma 5. *Given a non-confluent Büchi automaton $\mathfrak{A} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{F})$, and the corresponding deterministic automaton $\mathfrak{P} = (\mathcal{S}, s_\mathcal{I}, \Delta, \Omega)$ as given in Definition 39. Then, for every infinite word $\alpha : \mathbb{N} \to \Sigma$, and the corresponding run $\pi = s^{(0)}s^{(1)}\ldots$ of $\mathfrak{P}$ over α with $s^{(t)} = ((S_0^{(t)}, e_0^{(t)}, f_0^{(t)}), \ldots (S_{n-1}^{(t)}, e_{n-1}^{(t)}, f_{n-1}^{(t)}))$ and for every $t \in \mathbb{N}$, the following holds:*

1. *For all $i > 0$ and $t \in \mathbb{N}$, we have $S_i^{(t)} \subseteq S_0^{(t)}$.*

2. *For all i and $t \in \mathbb{N}$ with $S_i^{(t)} \neq \emptyset$, there exists a $q \in S_i^{(t)}$ such that $q \notin S_j^{(t)}$ for all $i < j < n$.*

3. *For every $t_0 \in \mathbb{N}$ and for every $0 \leq i < n$, we have:*

$$q \in S_i^{(t_0)} \Rightarrow \left(\begin{array}{ll} \exists \xi : \mathbb{N} \to \mathcal{S}. & [\xi^{(0)} \in \mathcal{I}] \wedge [\xi^{(t_0)} = q] \wedge \\ & [\forall t < t_0 . \xi^{(t+1)} \in \mathcal{R}(\xi^{(t)}, \alpha^{(t)})] \end{array} \right)$$

4. *Let $t_0 < t_1$ be positions such that*
   - *$S_i^{(t_0)}$ and $S_i^{(t_1)}$ are marked, i.e. $f_i^{(t_0)} = 1$ and $f_i^{(t_1)} = 1$*
   - *$e_i^{(t)} = 0$ for $t_0 \leq t \leq t_1$*
   - *$f_j^{(t)} = 0$ for $j < i$ and $t_0 \leq t \leq t_1$*

Then, each finite run ξ of $\mathfrak{A}$ with $\xi^{(t_0)} \in S_i^{(t_0)}$ and $\xi^{(t_1)} \in S_i^{(t_1)}$ must have visited $\mathcal{F}$ at least once between t_0 and t_1.

5. *For all $t \in \mathbb{N}$ and $0 \le j < n$ we have: If $q \in S_j^{(t)} \cap \mathcal{F}$, then either $q \in S_{n-1}^{(t)}$ or there exists some $0 \le i < n$ such that $f_i = 1$ and $q \in S_i^{(t)}$.*

Proof. Properties 1 and 2 follow directly from the definition of the transition function of $\mathfrak{P}$. Property 3 holds trivially for S_0: as long as the run continues (i.e. it does not end in a deadend state), we have $S_0 \neq \emptyset$ according to the definition of Δ. Thus $S_0^{(t+1)} = \mathsf{suc}_\exists^{\delta,\alpha^{(t)}}(S_0^{(t)})$ for every $t \in \mathbb{N}$. For $i > 0$ the result follows from Property 1.

To prove Property 4, consider a run ξ of $\mathfrak{A}$ with $\xi^{(t_0)} \in S_i^{(t_0)}$ and $\xi^{(t_1)} \in S_i^{(t_1)}$. For every position t between t_0 and t_1 we have (1) $e_i^{(t)} = 0$ and (2) $f_j^{(t)} = 0$ for every $j < i$. We thus have $S_i^{(t+1)} = \mathsf{suc}_\exists^{\mathcal{R},\alpha^{(t)}}(S_i^{(t)})$ for every $t \in \{t_0, \dots, t_1 - 1\}$. In this case, we have $\xi^{(t)} \in S_i^{(t)}$ for every $t \in \{t_0, \dots, t_1\}$. Since $f_i^{(t_0)} = 1$, we have $\xi^{(t_0)} \notin S_j^{(t_0)}$ for every $j > i$ according to the definition of f_i. Let $t' > t_0$ be the first position after t_0 where $\xi^{(t'+1)} \in \bigcup_{j=i+1}^{n} S_j^{(t')}$. Such a position t' must exist in the run of $\mathfrak{P}$, since S_i is marked at position t_1. There must either exist an index $j > i$ such that $\xi^{(t'+1)} \in \mathcal{R}(S_j^{(t')}, \alpha^{(t')})$ or $\xi^{(t'+1)} \in \mathcal{F} \cap \mathsf{suc}_\exists^{\mathcal{R},\alpha^{(t')}}(S_0^{(t')})$. We will now show that the first case is impossible, leading to our desired result that ξ visits $\mathcal{F}$ at least once between t_0 and t_1. Assume by contradiction that $\xi^{(t'+1)} \in \mathsf{suc}_\exists^{\mathcal{R},\alpha^{(t')}}(S_j^{(t')})$. Then there must exist a state $q \in S_j^{(t')}$ such that $\xi^{(t'+1)} \in \mathcal{R}(q, \alpha^{(t')})$. Thus, according to Property 3 there does exist a run ξ' that leads to q, i.e. $\xi'^{(t')} = q$. However, continuing this run with $\mathcal{R}(q, \alpha^{(t')})$ leads to $\xi'^{(t'+1)}$. Either ξ' and ξ coincide which leads to a contradiction to $t' + 1$ being the first position where $\xi^{(t'+1)}$ is introduced in some $S_j, j > i$ or they do not coincide which is a contradiction to $\mathfrak{A}$ being non-confluent.

To prove Property 5, we distinguish two cases: First case is that $e^{(n-1)} = 1$. Now according to property 1 we have $S_j^{(t)} \subseteq S_0^{(t)}$ and moreover, we have $S_0^{(t)} = \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_0^{(t-1)})$ for every $t > 0$. Thus, we have $S_{n-1}^{(t)} = \mathsf{suc}_\exists^{\mathcal{R},\sigma}(S_0) \cap \mathcal{F}$ which gives us $q \in S_{n-1}^{(t)}$. For the other case, notice that $e^{(n-1)} = 0$ implies that $e^{(i)} = 0$ for every $0 \le i < n$. This means that $S_i^{(t)} \neq \emptyset$ for every $0 \le i < n$. According to property 2 every subset must contain one unique state. This implies that either $S_i^{(t)} = \{q\}$ in which case we have $f_i^{(t)} = 1$. Otherwise we have $\left| \bigcup_{i=1}^{n-1} S_i^{(t)} \right| \ge n - 1$ and q is the unique state of $S_0^{(t)}$. But then any state except q is contained in some set $S_i^{(t)}$ which means that $f_0^{(t)} = 1$. □

With the help of this lemma, we are prepared to show the correctness of our determinization procedure:

Theorem 16. *The deterministic parity automaton $\mathfrak{P}$ constructed for an arbitrary non-confluent Büchi automaton $\mathfrak{A}$ as described in Definition 39 is equivalent to $\mathfrak{A}$.*

Proof.

$\mathsf{Lang}(\mathfrak{P}) \subseteq \mathsf{Lang}(\mathfrak{A})$**:** Let $\alpha \in \mathsf{Lang}(\mathfrak{P})$ and $\pi = s^{(0)}s^{(1)}\ldots$ bet the corresponding run of $\mathfrak{P}$ over α with $s^{(t)} = ((S_0^{(t)}, e_0^{(t)}, f_0^{(t)}), \ldots (S_{n-1}^{(t)}, e_{n-1}^{(t)}, f_{n-1}^{(t)}))$ for every $t \in \mathbb{N}$. Since α is accepted, there does exist t_0 and a minimal color such that $\Omega(s^{(t)}) = 2i$ for infinitely many t and $\Omega(s^{(t)}) \leq 2i$ for every $t > t_0$. Thus, we have that $S_j^{(t)} \neq \emptyset$ and $e_j^{(t)} = 0$ for every $j < i$ and every $t > t_0$ and moreover, we have that $f_i^{(t)} = 1$ for infinitely many t. Let $t_1 < t_2 < \cdots$ be the infinitely many i-breakpoints, i.e. positions where $f_i^{(t)}$ is marked. Define $Q^{(0)} = \mathcal{I} = S_0^{(0)}$ and for each $t > 1$ define $Q^{(j)} = S_i^{(t_j)}$. For each initial state, we construct a tree as follows: the vertices are taken from the set $\{(q,t) \mid q \in Q^{(t)}\}$. As the parent of $(q, t+1)$ (with $q \in Q^{(t+1)}$) we pick one of the pairs (p,t) such that $p \in Q^{(t)}$ holds and that there is a run ξ of $\mathfrak{A}$ on $\alpha[t, t+1]$ between p and q according to Property 3 of Lemma 5. Clearly, these trees are finitely branching, since each S_i is finite. Moreover, for at least one initial state q_0 the corresponding tree must have an infinite number of vertices, because we have an infinite sequence of breakpoints and $S_j^{(t)} \neq \emptyset$ for every $j \leq i$. Therefore, we conclude by Königs lemma that there is an initial state such that there is an infinite path $(q_0, 0), (q_1, 1), \cdots$ through the tree we have constructed for q_0. This infinite path corresponds to an infinite run of $\mathfrak{A}$ on α. Recall now that according to the construction of the tree the finite pieces of the run that connect q_i to q_{i+1} while consuming the word $\alpha^{(t_i)}, \cdots \alpha^{(t_{i+1}-1)}$ visit, at least once, the set of accepting states between t_i and t_{i+1} due to Property 4. Since we have an infinite number of breakpoints, we have constructed an accepting run of $\mathfrak{A}$.

$\mathsf{Lang}(\mathfrak{A}) \subseteq \mathsf{Lang}(\mathfrak{P})$**:** Given a word $\alpha \in \mathsf{Lang}(\mathfrak{A})$ with an accepting run ξ. Since ξ is an accepting run of α we have $S_0^{(t)} \neq \emptyset$ for every t. If S_0 is marked infinitely often, we are done. Otherwise, let t_0 be the first visit of ξ of an accepting state after the last time where S_0 is marked. According to property 5 of Lemma 5 there must exist some $i > 1$ such that $\xi^{(t)} \in S_i^{(t)}$. Let j be the minimal index such that $\xi^{(t_0)} \in S_j^{(t_0)}$. According to the definition of the subset construction, since ξ is an infinite run, either S_j contains the run ξ from t_0 on, i. e. $\xi^{(t)} \in S_j^{(t)}$ for every $t \geq t_0$ or there must exist some index $0 < i < j$ and a position t_0' such that $e_i^{(t_0')} = 1$ or $f_i^{(t_0')} = 1$. We can not have $f_i^{(t_0')} = 1$ since in that case there must have been two different runs that lead to $\xi^{(t_0')}$ namely the run followed by S_j and the run followed by S_i, a contradiction to the non-confluence property or to the minimality of j. Thus the only chance that $\xi^{(t_0)} \notin S_j^{(t_0)}$ is that the run ξ has moved to the left neighbor $S_{j-1}^{(t_0)}$. However, this move to the left can happen at most $j-1$ times, since $S_0^{(t)} \neq \emptyset$. Thus, we see that there must exist a minimal index $i_1 > 0$ such that $\xi^{(t)} \in S_i^{(t)}$ for every $t > t_0$ and for every $0 < i < i_0$ we must have $e_i^{(t)} = \mathsf{0}$ for every $t > t_0$. We again distinguish two cases: either $f_{i_0}^{(t)} = \mathsf{1}$ infinitely often, in case $\mathfrak{P}$ accepts or there must exist a position $t_1 > t_0$ such that $f_{i_0}^{(t_1)} = \mathsf{0}$ but $\xi^{(t_1)} \in \mathcal{F}$. Again, $\xi^{(t_1)}$ must be introduced in some $S_{j_1}^{(t_1)}$ for some $j_1 > i_0$ according to property (5) of Lemma 5 and we are in a completely analogous situation as before. Repeating this argumentation $n = |\mathcal{S}|$ times means that all S_i, $i \in \{0, \ldots, n\}$ follow run ξ and never get empty according to our assumption. According to Lemma 5 every state set contains at least one unique state. Thus either some $f_i^{(t)} = \mathsf{1}$ at some position t for some $i < n-1$, a contradiction to our assumption, or $S_{n-1}^{(t)}$ can not contain more than one state in every position $t > t_n$. But then $S_{n-1}^{(t)}$ is marked in those positions where this uniquely defined state $q \in S_{n-1}^{(t)}$ is a marked state of the nondeterministic automaton, so that we see that the automaton accepts in that case as well.

□

Concerning the complexity, we have the following result:

Theorem 17. *Given a non-confluent Büchi automaton with n states, the construction given in Definition 39 yields a deterministic parity automaton with at most $n \cdot 2^{(n+n^2)}$ states and $2n+1$ colors.*

Proof. There are 2^n possibilities for the marking variables f_i. Moreover, in every state the e_i are uniquely determined by the minimal index j such that $e_i = \mathsf{1}$. So the empty flags give a multiplicator of n. Finally, the membership of a state q in one of the n state sets can be represented with n boolean variables, which requires n^2 variables for the n states. All in all, this yields the upper bound $n \cdot 2^{(n+n^2)}$ for the possible states of $\mathfrak{P}$. □

5.3 Symbolic Implementation

For the symbolic implementation, we introduce for every state $\vartheta^k \in \mathcal{S}$ exactly n state variables q_i^k that represent that $\vartheta^k \in S_i$. Additionally we need the corresponding next state variables $\vartheta_i^{k\prime}$. We moreover introduce state variables e_i and f_i for $0 \leq i < n$ that represent the flags. Assume we have calculated formulas for the subset construction according to Chapter 4, i. e. for every state $\vartheta^k \in \mathcal{S}$ we have calculated a formula φ^k such that the transition function δ of the subset construction is given by a function of the input and state variables of the first set by

$$\delta \equiv \bigwedge_{\vartheta^k \in \mathcal{S}} {\vartheta_0^k}' \leftrightarrow \varphi_0^k$$

For every set i, let φ_i be the corresponding formula of φ for set i. The state variables of our determinization construction can then be determined by the following equations:

The first set is determined by the following equations:

$$\begin{aligned}
{e_0}' &\leftrightarrow \bigwedge_{\vartheta^k \in \mathcal{S}} \neg\varphi_0^k \\
{f_0}' &\leftrightarrow \neg e_0 \wedge \bigwedge_{\vartheta^k \in \mathcal{S}\setminus\mathcal{F}} \left(\varphi_0^k \rightarrow \bigvee_{j>0}^{n-1} \varphi_j^k\right) \\
{q_0^k}' &\leftrightarrow \varphi_0^k
\end{aligned}$$

For every $0 < i < n$, the flag variables are determined by :

$$\begin{aligned}
{e_i}' &\leftrightarrow {e_{i-1}}' \vee \bigwedge_{\vartheta^k \in \mathcal{S}} \neg\varphi_i^k \\
{f_i}' &\leftrightarrow \neg e_i \wedge \bigwedge_{\vartheta^k \in \mathcal{S}\setminus\mathcal{F}} \left(\varphi_i^k \rightarrow \bigvee_{j>i}^{n-1} \varphi_j^k\right)
\end{aligned}$$

For every $0 < i < (n-1)$, the state variables are determined by :

$${q_i^k}' \leftrightarrow \left(\neg {e_i}' \wedge \varphi_i^k \vee {e_i}' \wedge \varphi_{((i+1) \mod n)}^k\right) \wedge \bigwedge_{j<i} \neg(f_j' \wedge \varphi_j^k)$$

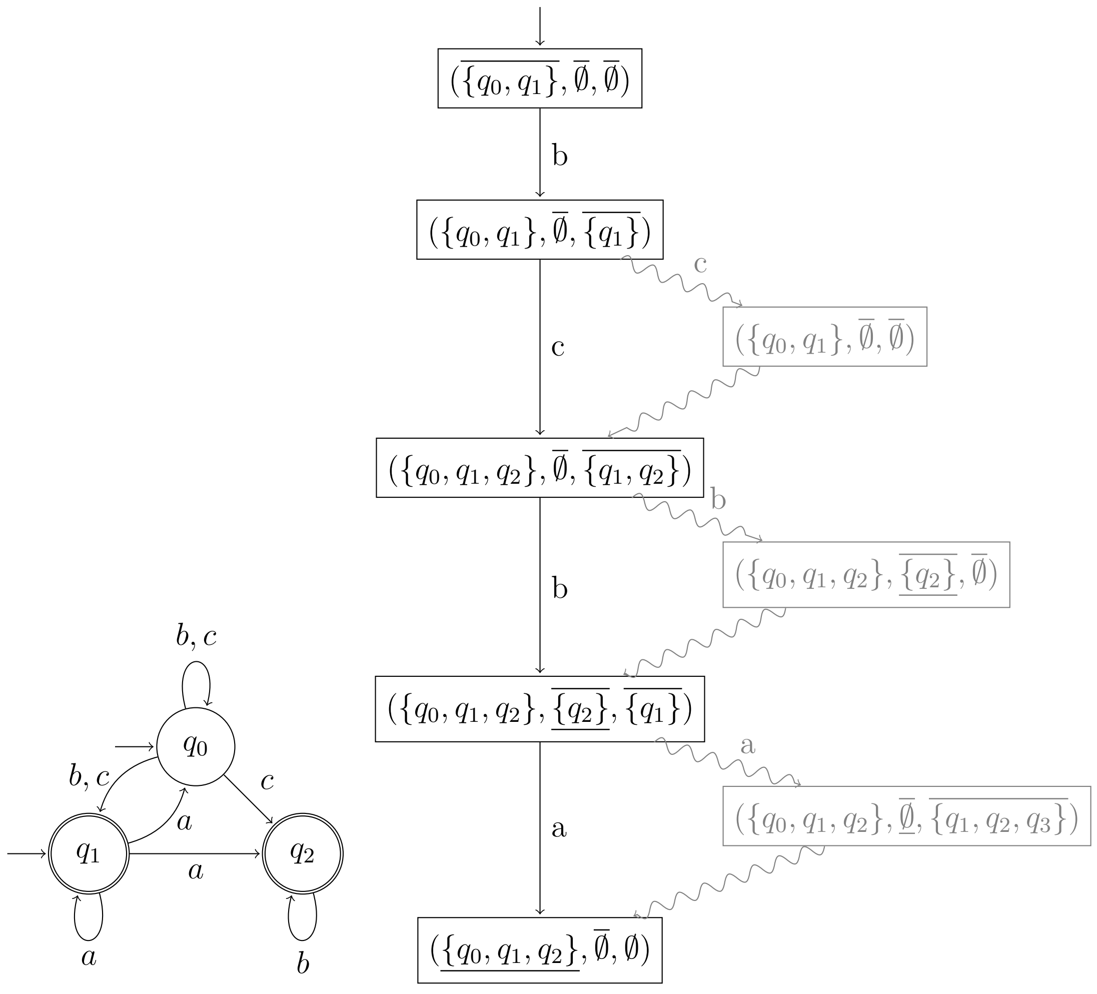
For the last state variables, we obtain for the marked variables $\varphi_i \in \mathcal{F}$:

$${q_{n-1}^k}' \leftrightarrow \left(\neg {e_{n-1}}' \wedge \varphi_{n-1}^k \vee {e_{n-1}}' \wedge \varphi_0^k\right) \wedge \bigwedge_{j<(n-1)} \neg(f_j' \wedge \varphi_j^k)$$

For the last state variables, we obtain for the unmarked variables:

$${q_{n-1}^k}' \leftrightarrow \left(\neg {e_{n-1}}' \wedge \varphi_{n-1}^k\right) \wedge \bigwedge_{j<(n-1)} \neg(f_j' \wedge \varphi_j^k)$$

Notice that since $\mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_i)$ are determined by the ordinary subset construction, the different φ_i determine exactly $\mathsf{suc}_{\exists}^{\mathcal{R},\sigma}(S_i)$, thus the conjunction of the equations given above is equivalent to the definition of the deterministic parity automaton in Definition 39. Finally, the coloring function Ω can be represented using $2n+1$ boolean formulas over the flag variables by setting $\Phi_0 = \neg e_0 \wedge f_0$, $\Phi_1 = e_0 \vee e_1$, $\Phi_2 = \neg e_0 \wedge \neg e_1 \wedge f_1$ and $\Phi_{2i} = \bigwedge_{j=0}^{i}(\neg e_j \wedge \neg f_j) \wedge f_i$ and $\Phi_{2i-1} = \bigwedge_{j=0}^{i-1}(\neg e_j \wedge \neg f_j \wedge e_i)$ for $2 \leq i < n$.


b, c
q_0
b, c
c
a
q_1
a
q_2
a
b
$(\overline{\{q_0, q_1\}}, \overline{\emptyset}, \overline{\emptyset})$
b
$(\{q_0, q_1\}, \overline{\emptyset}, \overline{\{q_1\}})$
c
c
$(\{q_0, q_1\}, \overline{\emptyset}, \overline{\emptyset})$
$(\{q_0, q_1, q_2\}, \overline{\emptyset}, \overline{\{q_1, q_2\}})$
b
b
$(\{q_0, q_1, q_2\}, \underline{\overline{\{q_2\}}}, \overline{\emptyset})$
$(\{q_0, q_1, q_2\}, \underline{\overline{\{q_2\}}}, \overline{\{q_1\}})$
a
a
$(\{q_0, q_1, q_2\}, \underline{\overline{\emptyset}}, \overline{\{q_1, q_2, q_3\}})$
$(\underline{\{q_0, q_1, q_2\}}, \overline{\emptyset}, \emptyset)$

(a) A Non-Confluent Automaton (b) Intermediate Steps of the Determinization Construction

Figure 5.4: Another Example for the Determinization of Non-Confluent Automata

$$\begin{tikzcd}
 & & {} \arrow[d] \\
 & & \boxed{(\overline{\{q_0, q_1\}}, \overline{\emptyset}, \overline{\emptyset})} \arrow[dl, "{a, c}"'] \arrow[d, "b"] \\
\boxed{(\{q_0, q_1, q_2\}, \underline{\overline{\{q_2\}}}, \overline{\{q_1\}})} \arrow[r, bend left, "c"] \arrow[d, "b"] \arrow[dr, "a"] & \boxed{(\{q_0, q_1, q_2\}, \overline{\emptyset}, \overline{\{q_1, q_2\}})} \arrow[l, "b"'] \arrow[d, bend left, "a"] \arrow[loop above, "c"] & \boxed{(\{q_0, q_1\}, \overline{\emptyset}, \overline{\{q_1\}})} \arrow[l, "c"'] \arrow[dl, "a"] \arrow[loop above, "b"] \\
\boxed{(\{q_0, q_1, q_2\}, \underline{\{q_2\}}, \overline{\{q_1\}})} \arrow[ur, "c"] \arrow[r, "a"] \arrow[loop below, "b"] & \boxed{(\underline{\{q_0, q_1, q_2\}}, \overline{\emptyset}, \emptyset)} \arrow[u, "{b, c}"] \arrow[loop below, "a"] &
\end{tikzcd}$$

Figure 5.5: Deterministic Automaton for the Automaton of Figure 5.4

6 Symbolic Controller Synthesis

This chapter is intended to summarize the presented approaches in one single Algorithm. First we briefly sketch how the developed tool is embedded in the Averest framework developed by the Embedded Systems group of the University of Kaiserslautern. Then, it is shown why it is infeasible to generate deterministic automata from one monolithic automaton. Thus, we focus on a modular approach to the determinization problem, so that we obtain a deterministic automaton that is composed of many small deterministic automata. The last two paragraphs are denoted to the solution of the obtained games. To this end, first it is shortly sketched how the solution of subgames that are obtained by subformula of the overall specification can be used to restrict the game graph. The obtained strategy is nondeterministic in the sense that multiple (equally good) controllable events may be enabled in the same state. We shortly sketch how to obtain a deterministic strategy using an approach presented in [12] that can then be used to obtain a description in e. g. Verilog or VHDL.

6.1 The Averest Framework

Averest[36] is a set of tools for the specification, verification, and implementation of reactive systems. It includes a compiler and a simulator for synchronous programs, a symbolic model checker and a tool for hardware-software synthesis. Averest can be used for modeling and verifying finite as well as infinite state systems at various levels of abstraction. In particular, Averest is not only well-suited for hardware design, but also for modeling communication protocols, concurrent programs, software in embedded systems, etc. The design flow using Averest consists of the following steps: First, the system is described as a program in our synchronous language Quartz, a descendant of Esterel [92]. Then, the program is translated to a transition system in the Averest Interchange Format (AIF) using the Quartz compiler Ruby. This intermediate description can be directly used for verification with the symbolic model checker Beryl to check whether the system satisfies its specifications. Afterwards, the tool Topaz can be used to generate an implementation in hardware or software with output formats VHDL, Verilog or ISO-C. The compiler Ruby does not only compile a Quartz program to a transition system, but also provides procedures to translate LTL specifications to symbolically represented ω-automata as described in sections

2.3.1 and 2.4. The tool developed during this thesis is called Opal and is an extension of Ruby that uses the basic functionality of Ruby to generate nondeterministic automata. In the following, the different steps of Opal are described in more detail. The implementation and experiments reported in this thesis have been made on top of version 1.9 of Averest.

6.2 Why monolithic approaches fail

Typically, the input for a controller synthesis problem consists of assumptions Φ_e the environment is supposed to fulfill and guarantees Φ_c that the controller must enforce provided the assumptions hold. While in theory one could join all specifications and then determinize them, the high complexity of determinization makes this approach infeasible. For example consider the following conjunction of LTL properties that belong to the guarantees of the controller:

$$\begin{aligned}
&\mathsf{G}\,(s_1 \rightarrow (\mathsf{X}r_1 \vee \mathsf{XX}r_1 \vee \mathsf{XXX}r_1)) \wedge\\
&\mathsf{G}\,(s_2 \rightarrow (\mathsf{X}r_2 \vee \mathsf{XX}r_2 \vee \mathsf{XXX}r_2)) \wedge\\
&\mathsf{G}\,(s_3 \rightarrow (\mathsf{X}r_3 \vee \mathsf{XX}r_3 \vee \mathsf{XXX}r_3))
\end{aligned}$$

Each conjunct leads to an automaton with four states, however the conjunction of the three formulas leads to an automaton with 64 states. And none of the states simulates another one (intuitively speaking, each state waits for exactly one combination of the variables s_i, r_i, so that we really obtain 64 states and thus at least 64 state variables for the deterministic automaton. Thus, determinizing the whole automaton is in general infeasible. Hence, we generate the deterministic automaton for each part separately which results in 12 state variables which are easily manageable using modern BDD packages.

6.3 A Modular Approach to Determinization

Specifications are often made up of several relatively simple components- for instance, a collection of LTL properties whose conjunction should be satisfied. Thus, given a Moore game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_{\mathcal{I}}, \delta, \Phi)$ over a set of state variables Q, we consider specifications

$$\Phi = \bigwedge_{j=0}^{N} \Phi_j$$

Instead of translating the whole specification at once, we generate separate deterministic automata for every part. Clearly, since we allow any LTL property, we may have to perform the determinization procedure outlined in Chapter 5 to translate Φ_j. This is the case if Φ_j is a single formula φ where the top-level operator is a temporal operator and φ belongs not to one of the lower Borel classes $\mathsf{TL}_{\mathsf{GF}}$ or $\mathsf{TL}_{\mathsf{GF}}$. In practice, this is nearly never the case. Instead, mostly also the subformula are a boolean combination of smaller formulas. Instead of handling them all at once, we break also them into smaller parts so that we obtain for every Φ_j a collection $\varphi_1 \dots \varphi_k$ of LTL properties that all start with a temporal operator, and that either belong to one of the classes TL_{G}, TL_{F}, $\varphi_i \in \mathsf{TL}_{\mathsf{GF}}$, $\mathsf{TL}_{\mathsf{FG}}$ or to none of these classes.

6.3.1 Handling Safety and Liveness formulas

In the general case, we can translate safety formulas using the approach described in Chapter 4. Thus, we first translate φ_i to a nondeterministic safety automaton. Although safety automata can not be transformed to a parity automaton, it is possible to minimize them using direct simulation[1]. After minimization, we perform the ordinary subset construction and afterwards minimize the automaton again using the direct simulation relation. However, one important subclass of properties does not scale well using this approach. Since many specifications are of the form if something now happens, in the next step something else happens, we treat this subclass separately. This subclass can be formally described by boolean combinations of formula of the form ψ, $\mathsf{X}\xi$ where both ψ and ϕ are boolean formula over the input variables. In that case, every X operator doubles the state space of the non-deterministic automaton and thus leads to a Blowup in the number of BDD-variables of the deterministic automaton. Even worse, the simulation relations can neither minimize the nondeterministic nor the deterministic automaton since the two states that occur because of a $\mathsf{X}a$ can not be equivalent since one of the two will lead to a non-satisfying loop. Thus, the basic translation procedure really suffers from a double-exponential blowup. Instead, we translate those formulas by abbreviating each variable a that is not under the scope of a X variable by a previous variable a_p such that the following holds:

$$a_p{}' \leftrightarrow a$$

and replace any subformula $\mathsf{X}a$ with a. We moreover introduce a new fresh variable p that is true as long as ϕ holds and in case ϕ is violated, remains false forever.

Proposition 8. *Given a formula $\mathsf{G}\varphi$ where φ is a boolean combination of formulas $v \in V_a$ and $\mathsf{X}v \in V$. Then $\mathsf{G}\varphi$ is initially equivalent to the symbolically represented*

[1]Indeed, the nondeterministic automata generated by the translation procedures given in [91] translates the safety fragment to an automaton with the trivial acceptance condition $\mathsf{G}1$, so that even the ordinary simulation relation as described in [28] could be applied.

deterministic automaton $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \Psi)$ where $Q = \{a_p \mid a \in V\} \cup \{p\}$, $\mathcal{I} = p \wedge \bigwedge_{a \in V} \neg a_p$, $\Psi = \mathsf{G}(p)$ and the transition relation is defined by

$$\mathcal{R} = \ p' \leftrightarrow \varphi' \wedge \left(\bigwedge_{a \in V} {a_p}' \leftrightarrow a \right)$$

Here φ' is obtained from φ by replacing any occurrence of a' with a and any occurrence of a formula $a \in V$ with a_p

Obviously, this leads to an automaton that has at most $|a| + 1$ state variables, thus the automaton is only exponential in the size of the specification. Nevertheless, if many subformula a exist but little subformula $\mathsf{X}a$, the ordinary translation may give better results, so that this translation is optional in our algorithm [2].

For liveness formulas φ_i, we translate $\neg\varphi_i$ and dualize the corresponding deterministic safety automaton to obtain a deterministic liveness automaton.

6.3.2 Handling Co-Büchi and Büchi specifications

For co-Büchi specification, we use the translation from $\mathsf{TL}_{\mathsf{FG}}$ to nondeterministic co-Büchi automata and minimize this automaton using the minimization techniques of Chapter 3. The minimized automaton is then determinized using the breakpoint construction and again minimized. Büchi specifications are translated using the dual deterministic automaton of the formula obtained from negating the formula.

6.3.3 Handling LTL Formulas that do not belong to a lower Borel Class

If a subformula Φ_i does not belong to one of the lower borel classes, we have to resort on the determinization procedure from Chapter 5. Thus we first translate Φ_i to a nonconfluent Büchi automaton and minimize it. This nondeterministic automaton is determinized using the procedure described in Chapter 5 . However, we do not need to construct the whole automaton at once. Instead, we construct the automaton for a fixed bound k and check whether every marking of a state has been noticed by a state set. If so, we return this automaton, otherwise we do the same with an increased bound. Afterwards, we use the minimization techniques of Chapter 3 to minimize the obtained parity automaton.

[2] We could have also defined a similar class for liveness, however, since this type nearly never occurs in practice, we neglect this.

Although this procedure gives back a parity automaton that is from a theoretical point of view more efficient than a Streett automaton, the heavy complexity of determinization makes even this approach infeasible in practice. Thus we break up also the formulas Φ_i into smaller parts until every subformula starts with a temporal operator. Those subformula are then translated as explained before. It is well known that every parity automaton can also be interpreted as a Streett or a Rabin automaton. We thus interpreted the obtained automaton as a Streett automaton and combine the deterministic Streett automata to obtain a Streett automaton for Φ_i that is afterwards translated to a generalized parity automaton.

6.4 Solving Generalized Parity Games

Instead of solving the whole generalized parity game at once using the approach described in Section 2.5.3, we first solve the subgames that are obtained by constructing the game for the subformula Φ_j. For safety games, we know from [8] that any safety strategy is most permissive, which means that the strategy allows as many moves as possible. Thus after pruning the Game with the safety strategy, we know that the remaining game satisfies the safety specification while we have not lost any possibility to ensure the overall strategy. This is captured in the following proposition[3] :

Proposition 9. *Given a Moore game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_{\mathcal{I}}, \delta, \Phi)$ where $\Phi = \mathsf{G}\varphi \wedge \Psi$ with a propositional formula φ. Let $\mathfrak{G}' = (V_u, V_c, \mathcal{S}, s'_{\mathcal{I}}, \delta', \Psi)$ be obtained by pruning the state set according to the winning strategy of $\mathsf{G}\varphi$. Then, any winning strategy of $\mathfrak{G}'$ is also a winning strategy for $\mathfrak{G}$.*

After having performed this reduction for the safety specifications, we also solve the corresponding subgames for each separate subformula. The set of states that are loosing for the controller need not be considered in the overall game. Obviously, also the states from which the environment can force a visit to those loosing states need also not considered, thus we also remove the attractor set to those states. However, unlike in the safety case, the sub-specification still needs to be considered in the overall game. Afterwards, we solve the reduced overall game using the generalized parity game algorithm of [19] with a slight but important exception. Whenever we have a parity condition where the lowest odd color labels no state, we can reduce the number of colors according to the following proposition.

Proposition 10. *Given a Moore game $\mathfrak{G} = (V_c, V_u, \mathcal{S}, s_{\mathcal{I}}, \delta, \Phi)$ where $\Phi = \Omega_0 \wedge \Phi'$ is a generalized parity condition with a parity condition Ω_0 where $\{s \in \mathcal{S} \mid \Omega(s) = 1\} = \emptyset$.*

[3]In [97], a similar observation is used to prune the state space. Their work has been published during the evaluation of this thesis.

Then Φ *is equivalent to* $\Omega_0' \wedge \Phi'$ *where*

$$\Omega_0'(s) = \begin{cases} 0 & \textit{if } \Omega_0(s) = 2 \textit{ or } \Omega_0(s) = 0 \\ \Omega_0(s) - 2 & \textit{else} \end{cases}$$

In the experiments section, we will see e. g. by the AMBA example that this modification enables us to solve much larger examples than without.

6.5 Generating Circuits from BDDs

The output of the generalized parity algorithm is a BDD over the (current state) variables $V_u, V_c, \mathcal{S}$ and over newly introduced state variables V_M to encode counter variables that are used to switch between the sub-strategies calculated by the generalized parity algorithm. A slight modification of the algorithm given in Figures 2 and 3 of [12] allows us to generate for every controllable input variable c a BDD φ_c with the meaning that c should hold whenever φ_c holds. We then write those BDDs to a file in our Averest interchange format [36]. The tool Topaz in our averest tool set can be used to obtain either Verilog, VHDL or C code from the generated file.

7 Experimental Results

This chapter describes the experiments performed using the controller synthesis algorithm that was developed during this thesis. All experiments have been performed on a 3.0 GHz Quad Core Pentium Duo with 16 GB of RAM.

The first part investigates the effect of using the different determinization constructions presented in this thesis. To this end, the 23 specifications that come with the Lily tool [49] are used as a benchmark set. Those specifications are used for pure LTL synthesis, i. e. there is no system interacting with the environment, only the controller and the environment. In this section, we give also a comparison to the tool Lily.

In a similar manner, we compare the minimization techniques in Section 7.2 on this Benchmark set and additionally use some of the case studies developed later to evaluate the minimization power.

The rest of this chapter are five case studies. The task for the first case study is to synthesize a controller for the AMBA AHB-case study from [12], which is also an example of pure synthesis. Finally, we present four examples for controller synthesis that come from different application domains. The first controller synthesis problem is the well-known dining philosopher problem. The second problem we consider is constructing a winning strategy for the NIM-Game. From a practical point of view the most interesting problem is the Island Traffic Control problem. Finally, we consider the synthesis of a sorting network using controller synthesis.

7.1 The Effect of Different Determinization Constructions

As a first benchmark set, the 23 examples included with the LTL synthesis tool Lily [49] has been used to evaluate the performance of the determinization procedures presented in this thesis. All Lily examples are contained in the temporal logic hierarchy, i. e. are boolean combinations of co-Büchi specifications. Those 23 handwritten formulas are mostly traffic light examples or arbiters.

Example 2. *Example 10 of the formula included in Lily is the following* LTL *specification:*

$$(\mathsf{GF}req \vee \mathsf{F}cancel) \rightarrow (\mathsf{GF}grant \vee \mathsf{G}ack)$$

□

Example 3. *Example 23 of the formula included in Lily is the following* LTL *specification*

$$\begin{pmatrix} \mathsf{G}(\neg r1 \vee \neg r2)\wedge \\ \mathsf{G}(\neg r1 \vee \neg r3)\wedge \\ \mathsf{G}(\neg r1 \vee \neg r4)\wedge \\ \mathsf{G}(\neg r2 \vee \neg r3)\wedge \\ \mathsf{G}(\neg r2 \vee \neg r4)\wedge \\ \mathsf{G}(\neg r3 \vee \neg r4) \end{pmatrix} \rightarrow \begin{pmatrix} \mathsf{G}(r1 \rightarrow (\mathsf{X}(g1) \vee \mathsf{X}(\mathsf{X}(g1)) \vee \mathsf{X}(\mathsf{X}(\mathsf{X}(g1))))) \wedge \\ \mathsf{G}(r2 \rightarrow (\mathsf{X}(g2) \vee \mathsf{X}(\mathsf{X}(g2)) \vee \mathsf{X}(\mathsf{X}(\mathsf{X}(g2))))) \wedge \\ \mathsf{G}(r3 \rightarrow (\mathsf{X}(g3) \vee \mathsf{X}(\mathsf{X}(g3)) \vee \mathsf{X}(\mathsf{X}(\mathsf{X}(g3))))) \wedge \\ \mathsf{G}(r4 \rightarrow (\mathsf{X}(g4) \vee \mathsf{X}(\mathsf{X}(g4)) \vee \mathsf{X}(\mathsf{X}(\mathsf{X}(g4))))) \wedge \\ \mathsf{G}(\neg g1 \vee \neg g2)\wedge \\ \mathsf{G}(\neg g1 \vee \neg g3)\wedge \\ \mathsf{G}(\neg g1 \vee \neg g4)\wedge \\ \mathsf{G}(\neg g2 \vee \neg g3)\wedge \\ \mathsf{G}(\neg g2 \vee \neg g4)\wedge \\ \mathsf{G}(\neg g3 \vee \neg g4) \end{pmatrix}$$

□

To analyze the effect of using the nonconfluent determinization instead of the much simpler breakpoint or subset construction, we performed three different experiments on each of the formulas where the runtimes are summarized in Figure 7.1. The first column gives the number of the example. The second column gives some information about the formula in form temporal operators, boolean operators and number of input variables. The third column gives the number of X operators inside the formula. The fourth column gives the number of subformulas. The conjunction of those subformula form the overall specification. Notice that the number of temporal operators includes the number of X operators while the number of boolean operators do not include the conjuncts stemming from the top-level conjuncts. The next column indicates how many subformulas are not safety formula followed by a column indicating whether the example consists solely of prefix formula[1]. The next two columns give the time for determinization followed by the overall time needed to perform the task. When a column contains no entry, this means that the runtime took more than 1 hour.

We ran our algorithm with different determinization constructions. Algorithm Nonc uses the determinization construction given in Chapter 5. Algorithm Nonc2 additionally turned on the splitting of boolean optimizations. Finally, algorithm Opal uses the minimal possible determinization construction on every subformula. For all experiments the special treatment of safety formula is turned off and we used fair simulation minimization.

In all experiments we performed, using the lowest possible determinization construction is beneficial and significantly outperforms the approaches that use the nonconfluent determinization construction. The effect significantly grows with the size of

[1]Recall that for prefix formula, only the ordinary subset-construction is used.

No.	T,B,AP	X	subf	NoS	p	NoncDetT	Nonc2DetT	OpalDetT	NoncT	Nonc2T	OpalT	LilyT
L1	12,5,4	8	3	3	√	0.48	0.46	0.1	0.5	0.49	0.12	0.13
L2	12,5,4	8	3	3	√	0.5	0.5	0.09	0.53	0.52	0.11	0.12
L3	12,5,4	9	1	1			0.52	0.09		0.57	0.13	0.65
L4	16,10,4	12	1	1			0.47	0.1		0.52	0.13	1.31
L5	20,11,4	11	1	1	√		0.59	0.09		0.67	0.16	0.98
L6	18,14,4	14	1	1			0.58	0.11		0.72	0.19	1.98
L7	16,15,4	11	1	1			0.52	0.09		0.6	0.14	0.87
L8	4,1,2	0	1	1		0.32	0.01	0.01	0.33	0.02	0.02	0.07
L9	8,6,2	0	1	1		11.98	0.12	0.02	12.02	0.15	0.06	0.22
L10	6,3,2	0	1	1		50.33	0	0	50.38	0.01	0.02	0.39
L11	4,3,2	0	1	1		4.91	0.14	0.02	4.94	0.15	0.03	0.66
L12	5,4,2	0	1	1		186.87	0.15	0.03	186.9	0.16	0.03	0.19
L13	4,3,2	0	1	2		10.72	0.02	0.01	10.73	0.03	0.03	0.03
L14	9,3,4	0	3	3		0.57	0.01	0	0.59	0.03	0.03	0.26
L15	9,5,4	0	5	5		0.27	0.16	0.03	0.29	0.19	0.08	0.18
L16	15,9,6	0	9	7		0.37	0.22	0.02	0.43	0.29	0.1	1
L17	13,5,5	0	6	3		0.62	0	0.01	0.65	0.03	0.05	0.37
L18	21,10,7	0	10	4		1	0	0	1.05	0.04	0.06	1.67
L19	10,7,4	0	1	1			0.17	0.03		0.26	0.1	2.46
L20	17,20,5	6	1	1			0.12	0.03		0.18	0.12	4.38
L21	40,38,8	24	1	1	√		0.2	0.06		1.2	0.29	8.11
L22	22,18,4	12	1	1			0.24	0.04		0.37	0.14	9.98
L23	8,5,2	4	1	1		34.75	0.08	0.01	34.78	0.11	0.05	0.28

Figure 7.1: Experiments performed with different Determinization Constructions

the formula. When all optimizations are turned on, our tool is significantly faster compared to the explicitly implemented tool Lily. The difference grows again significantly with the size of the formula which is best seen by the bigger formula 19-22. When the splitting of boolean formula is turned on, also the nonconfluent determinization performs well compared to the explicit tool Lily. Only for the smaller examples L1-L3 the tool Lily is faster than our tool based on the nonconfluent determinization.

The picture changes when we turn off the boolean splitting of formula. In that case, our tool was not able to solve 9 Problems within one hour, indeed the construction of the deterministic automaton was not possible in either case. Since this is a quite surprising result, we try to analyze this behavior in the following. In table 7.2 we give the results of the experiments we performed with nonconfluent determinization when both the nondeterministic automata and the deterministic parity automata are minimized using fair simulation. Here, the columns have the following meaning: NDet, NMin, Det and DetMin gives states and transitions of the nondeterministic, nondeterministic minimized, deterministic and deterministic minimized automaton The corresponding runtimes needed to perform those procedures are termed NDetT, NMinT, DetT and and DetMinT respectively. Additionally, we give the time to construct the deterministic automaton (DetT), the time to solve the game (Solve) and the overall time.

There are nine formulas, for which the determinization construction can not be finished. Included are the examples 20 and 21 where even the generation of the nondeterminisitc automaton was not possible in a reasonable amount of time.We believe that this inability to construct the minimized nondeterministic automaton has to do with the state-based acceptance that we need. In our current implementation, every acceptance condition of the generalized Büchi automaton as well as the initial condition needs to be abbreviated using new propositional variables. All failed instances contain rather many X operators which we believe is the main source for our problems: in contrast to model-checking where only one state variable is introduced for each X operator, we have to use 2 new state variables for each X which seems to blow-up the BDDs.

If the sizes of the automata are rather small, we are able to generate the deterministic automata in a reasonable amount of time. However, as the sizes of the automata grows, the nonconfluent symbolic determinization procedure from Chapter 5 comes to its limit [2] which is reached when the nondeterministic automaton has more than approximately 15 states.

We believe that this unfavorable behavior has two reasons: the first reason is that the state variables of the deterministic automaton are highly interconnected with each

[2]The nondeterministic states given in table NMin do not necessarily mean that an automaton of this size is determinized, since e. g. example 1 consists of three sub-automata.

No.	T,B,AP	X	subf	NoS	p	NDet	NMin	Det	DetMin	MinDT	DetT	MinDT	Solve	Overall
L1	12,5,4	8	3	3	√	26.0(100.0)	21.0(42.0)	19.0(132.0)	13.0(84.0)	0.3	0.48	0.3	0.01	0.5
L2	12,5,4	8	3	3	√	26.0(124.0)	21.0(46.0)	19.0(132.0)	13.0(84.0)	0.35	0.5	0.35	0.02	0.53
L3	12,5,4	9	1	1		1036.0(980.0)	38.0(109.0)	Det failed						
L4	16,10,4	12	1	1		120.0(1036.0)	47.0(105.0)	Det failed						
L5	20,11,4	11	1	1	√	1048.0(2820.0)	98.0(469.0)	Det failed						
L6	18,14,4	14	1	1		1072.0(2916.0)	114.0(477.0)	Det failed						
L7	16,15,4	11	1	1		1048.0(1492.0)	76.0(153.0)	Det failed						
L8	4,1,2	0	1	1		9.0(9.0)	4.0(8.0)	13.0(52.0)	6.0(24.0)	0.28	0.32	0.28	0.01	0.33
L9	8,6,2	0	1	1		1049.0(44.0)	13.0(41.0)	28.0(112.0)	12.0(48.0)	9.02	11.98	9.02	0.04	12.02
L10	6,3,2	0	1	1		12.0(11.0)	5.0(9.0)	70.0(1120.0)	14.0(224.0)	48.52	50.33	48.52	0.05	50.38
L11	4,3,2	0	1	1		18.0(238.0)	8.0(133.0)	32.0(512.0)	12.0(192.0)	2.87	4.91	2.87	0.03	4.94
L12	5,4,2	0	1	1		11.0(41.0)	7.0(27.0)	194.0(3104.0)	12.0(192.0)	170	186.87	170	0.03	186.9
L13	4,3,2	0	1	2		12.0(28.0)	12.0(28.0)	156.0(624.0)	8.0(32.0)	5.2	10.72	5.2	0.01	10.73
L14	9,3,4	0	3	3		18.0(18.0)	8.0(16.0)	26.0(104.0)	12.0(48.0)	0.51	0.57	0.51	0.02	0.59
L15	9,5,4	0	5	5		18.0(68.0)	14.0(54.0)	44.0(176.0)	20.0(80.0)	0.17	0.27	0.17	0.02	0.29
L16	15,9,6	0	9	7		27.0(102.0)	21.0(81.0)	66.0(264.0)	30.0(120.0)	0.22	0.37	0.22	0.06	0.43
L17	13,5,5	0	6	3		18.0(18.0)	8.0(16.0)	26.0(104.0)	12.0(48.0)	0.56	0.62	0.56	0.03	0.65
L18	21,10,7	0	10	4		27.0(27.0)	12.0(24.0)	39.0(156.0)	18.0(72.0)	0.92	1	0.92	0.04	1.05
L19	10,7,4	0	1	1		145.0(536.0)	18.0(155.0)	Det failed						
L20	17,20,5	6	1	1		NDet failed								
L21	40,38,8	24	1	1	√	NDet failed								
L22	22,18,4	12	1	1		198.0(605.0)	86.0(141.0)	Det failed						
L23	8,5,2	4	1	1		69.0(205.0)	42.0(30.0)	60.0(240.0)	14.0(56.0)	30.49	34.75	30.49	0.03	34.78

Figure 7.2: Experiments performed with nonconfluent determinization where boolean splitting is turned off

other: First, whenever a state set on the left is marked, we have to remove the corresponding state from a set. On the other hand, they are also coupled to the variables for the right neighbor sets, in case a dead-end set takes over its neighbor. Another reason for the bad behavior is that we can not minimize the obtained automata on-the-fly. Given a nondeterministic automaton with n states, we need nearly never construct a deterministic automaton with n subsets of states. Instead we generate subautomata with $k < n$ state sets as described in Chapter 6. However, we can not build up the overall automaton from minimized sub-automata so that we first have to construct the overall automaton which is afterwards minimized.

Nevertheless, we believe that the nonconfluent determinization in connection with the splitting of boolean formula is able to perform quite well in practice: It has been already noticed by others [6] that all commonly used formulas in practice nearly never contain more than 3 temporal operators and instead are composed of Boolean combinations of smaller sub-formula. Thus, by enabling the splitting of Boolean combinations of formulas, we are able to solve most of the problems occurring in practice which is demonstrated by the experiments we performed on the Lily examples.

7.2 The Effect of Minimization

This section is intended to experimentally evaluate the performance of the various minimization procedures developed in Chapter 3. First, it is investigated how these algorithms perform on co-Büchi automata, followed by a section denoted to the minimization of parity automata.

7.2.1 Minimizing Co-Büchi Automata

The determinization construction from Chapter 4 uses Boolean combinations of co-Büchi automata to generate deterministic automata for formulas of the temporal logic hierarchy. Since this determinization construction is the most important from a practical point of view, optimizations for this construction are essential for a good overall performance. Accordingly, we investigate in this section the effect of the minimization techniques of Chapter 3 on co-Büchi automata.

Comparing Forward Simulation Relations

As a first experiment to evaluate the effect of minimization, we used the different minimization procedures on the 23 formulas that have been given before. Those ex-

No.	NDet	DiN	DiD	DiDiD	DiNT	DeNT	FaNT	DiDT	DeDT	FaDT	DiS	DeS	FaS
L1	28(696)	20(68)	11(68)	9(52)	0	0	0	0	0	0.01	0.02	0.02	0.02
L2	28(720)	20(92)	11(68)	9(52)	0	0.01	0	0	0.01	0.02	0.02	0.02	0.02
L3	32(732)	24(104)	14(80)	12(64)	0	0	0	0	0	0.01	0.04	0.04	0.04
L4	28(716)	20(88)	14(80)	12(64)	0.01	0.01	0.01	0.01	0.01	0.02	0.03	0.03	0.03
L5	36(808)	28(180)	15(104)	13(88)	0.01	0	0.01	0.01	0	0.03	0.07	0.08	0.07
L6	44(840)	36(212)	16(108)	14(92)	0	0	0	0	0.01	0.02	0.08	0.08	0.08
L7	36(814)	28(186)	15(104)	13(88)	0	0.01	0	0	0.01	0.01	0.05	0.05	0.05
L9	13(170)	7(26)	6(24)	5(20)	0	0	0	0	0	0.01	0.04	0.04	0.04
L11	12(166)	6(22)	4(16)	4(16)	0	0	0	0	0	0	0.01	0.02	0.02
L12	12(166)	6(22)	4(16)	4(16)	0	0	0	0	0.01	0	0.01	0.01	0.01
L15	16(182)	10(38)	10(40)	10(40)	0	0.01	0.01	0	0.02	0.01	0.04	0.04	0.04
L16	24(273)	15(57)	15(60)	15(60)	0.01	0	0	0.01	0.01	0	0.1	0.09	0.08
L19	20(257)	11(41)	10(40)	8(32)	0	0	0	0	0.01	0	0.08	0.07	0.08
L20	14(103)	11(31)	9(28)	9(28)	0	0.01	0.01	0	0.02	0.02	0.09	0.08	0.08
L21	64(240)	64(240)	20(80)	20(80)	0	0.02	0.01	0	0.02	0.01	0.21	0.21	0.21
L22	53(256)	46(112)	18(60)	18(60)	0	0	0.03	0	0	0.03	0.09	0.1	0.1
L23	32(56)	20(56)	7(28)	7(28)	0	0	0	0	0	0	0.02	0.02	0.02
A2	273(7099)	33(1260)	24(1036)	14(716)	0.05	0.04	0.05	0.05	0.04	0.05	0.71	0.69	0.67
A3	275(24110)	35(3271)	27(2584)	17(2264)	0.04	0.03	0.04	0.05	0.04	0.05	6	5.91	5.74
A4	277(46789)	37(5950)	30(4644)	20(4324)	0.08	0.06	0.05	0.09	0.07	0.06	10.52	11.03	10.25
A5	279(182828)	39(21989)	33(16944)	23(16624)	0.28	0.29	0.27	0.33	0.35	0.33	41.79	42.11	39.9
A6	281(364211)	41(43372)	36(33340)	26(33020)	0.48	0.5	0.49	0.61	0.63	0.61	59.86	221.88	219.04
A7	335(844577)	95(203738)	86(332288)	76(331968)	73.93	76.56	75.43	74.29	76.86	75.74	401.05	382.34	378.81
P3	24(546)	15(114)	15(120)	15(120)	0.01	0	0.01	0.01	0	0.01	0.06	0.06	0.05
P4	32(728)	20(152)	20(160)	20(160)	0.01	0.01	0	0.01	0.01	0	0.13	0.13	0.11
P5	40(910)	25(190)	25(200)	25(200)	0	0.01	0	0	0.02	0	0.34	0.32	0.31

Figure 7.3: Experiments performed with forward simulation relations on Co-Büchi Automata

periments are summarized in the table shown in Figure 7.3 [3]. Moreover, we performed several experiments on the AMBA case study and the Dining Philosophers case study that are also given in the same table.. The headings of the columns represent the following measurements, where x stands for either direct (Di), delayed (De) and fair (fa) simulation.

No. - Gives the experiment where Lx means Lily experiment x, Ax the AMBA case study with x masters and Px Philosophers case study with x philosophers.

NDet - Gives the total number of nondeterministic states and transitions

xN - the number of states / transitions of the minimized nondeterministic automaton.

xD - gives the number of states and transitions of the non-minimized deterministic automaton from the x-minimized nondeterministic one

xxD - gives the x-minimized deterministic automaton from the x-minimized nondeterministic one

xNT - gives the time to perform nondeterministic minimization

xDT - gives the time to perform deterministic minimization

xS - gives the time for game solving

For every automaton, we used the lowest possible determinization construction, i. e. for safety and liveness automata we used the subset construction and for co-Büchi automata, we used the breakpoint construction. We evaluate here the difference when the minimization used is direct, delayed or fair simulation.

Surprisingly, the three minimization procedures performed equally well. They generated the same number of states and transitions for every automaton we considered[4]. We did not check for every example whether the generated automata are the same for every minimization procedure, but randomly picked some of them and checked that they are indeed the same. Surprisingly, the minimized deterministic automata (shown in column DiDiD) had sometimes even less states and transitions than the nondeterministic automaton (shown in column DiN). The runtimes of the three minimization procedures do also not differ that much. When they differ, it seems that

[3]L8,L10, L13,L14, L17 and L18 contained only formulas that can be directly translated to a formula of the form $\mathsf{GF}\varphi$ for a propositional φ or directly to a transition system as described by Proposition 8 so that we left them out here.

[4]In the table, we show only the numbers for direct simulation, but the numbers are the same for delayed and fair simulation

this is more an error in measurement then a real difference. Since all minimization procedures generate the same automata, the solution time is also always the same.

We performed the same experiments also on the AMBA and dining philosophers case study with the same result. Neither do the automata differ nor the runtime.

The effect of Reverse Simulation

We performed the same experiments with reverse simulation turned on. Not too surprising, the three different forward simulation relations performed also equally well when used together with reverse simulation. Accordingly, we show only the number of states and transitions for direct simulation so that we only give the results for direct simulation in connection with reverse simulation and neglect the other simulation relation.

Comparing the automata sizes and runtimes when reverse simulation is turned on with the sizes and times where it is turned of gives some interesting results. For the Lily examples (shown in Figure 7.4), no surprise happened. Reverse simulation only minimized the number of transitions of the nondeterministic automaton. This intermediate minimization had no effect on the size of the deterministic automaton (see column DiDiD without and DiRDiD with reverse simulation) and accordingly also not on the runtime for synthesis.

We won't go into detail for those experiments and instead take a look at the AMBA case study that are also given in that table. Here, reverse simulation can both reduce the number of states and transitions of the nondeterministic automaton. However, this intermediate reduction does not lead to a reduction in the number of states and transitions of the deterministic automaton. Indeed the deterministic automata are exactly the same whether or not reverse simulation is enabled or not[5].

One notices that, although the automata generated with or without reverse simulation are equivalent, the runtimes for game solving differs. But this has to do with a different encoding of the automata that lead to different variable orders in the BDD and has nothing to do with reverse simulation. This is best seen by the AMBA example. Here we generate for n masters the same automata as for $n + 1$ masters, but every sub-automaton is multiplicated $n + 1$ times instead of n times. Nevertheless, for 5 masters, the runtimes with reverse simulation is slower than without, while the situation is reversed for other number of masters.

[5] We believe that this has to do with the special nonconfluent structure of the automata that we use, or equivalently, the reverse determinism of those automata. In [111] Watson mentions that in order to perform minimization of finite automata (which our safety automata essentially are), one can use the following steps: reverse the automaton (i. e. every ingoing edge is now an outgoing edge), do the subset construction, reverse the automaton and afterwards do again a subset construction. It is not hard to see that the automaton that is obtained before the last subset construction is reverse deterministic. So applying a subset construction (or the closely related breakpoint construction) on a reverse deterministic automaton seems to be indeed a minimization step. However, this observation is only a conjecture that should be closer examined outside of this thesis.

No.	NDet	DiN	DiRN	DiD	DiRD	DiDiD	DiRDiD	DiRNT	DiNT	DiRNT	DiDT	DiRDT	DiS	DiRS
L1	28(696)	20(68)	20(34)	11(68)	11(68)	9(52)	9(52)	0.01	0	0.01	0	0.01	0.02	0.02
L2	28(720)	20(92)	20(38)	11(68)	11(68)	9(52)	9(52)	0.03	0	0.03	0	0.03	0.02	0.02
L3	32(732)	24(104)	24(44)	14(80)	14(80)	12(64)	12(64)	0	0	0	0	0	0.04	0.05
L4	28(716)	20(88)	20(44)	14(80)	14(80)	12(64)	12(64)	0.01	0.01	0.01	0.01	0.01	0.03	0.03
L5	36(808)	28(180)	28(62)	15(104)	15(104)	13(88)	13(88)	0	0.01	0	0.01	0	0.07	0.07
L6	44(840)	36(212)	36(66)	16(108)	16(108)	14(92)	14(92)	0.02	0	0.02	0	0.02	0.08	0.07
L7	36(814)	28(186)	28(65)	15(104)	15(104)	13(88)	13(88)	0	0	0	0	0	0.05	0.06
L9	13(170)	7(26)	7(18)	6(24)	6(24)	5(20)	5(20)	0	0	0	0	0	0.04	0.03
L11	12(166)	6(22)	6(14)	4(16)	4(16)	4(16)	4(16)	0	0	0	0	0	0.01	0.01
L12	12(166)	6(22)	6(14)	4(16)	4(16)	4(16)	4(16)	0	0	0	0	0.01	0.01	0.01
L15	16(182)	10(38)	10(30)	10(40)	10(40)	10(40)	10(40)	0	0	0	0	0	0.04	0.04
L16	24(273)	15(57)	15(45)	15(60)	15(60)	15(60)	15(60)	0.01	0.01	0.01	0.01	0.01	0.1	0.09
L19	20(257)	11(41)	11(29)	10(40)	10(40)	8(32)	8(32)	0	0	0	0	0	0.08	0.09
L20	14(103)	11(31)	11(21)	9(28)	9(28)	9(28)	9(28)	0.01	0	0.01	0	0.01	0.09	0.08
L21	64(240)	64(240)	64(56)	20(80)	20(80)	20(80)	20(80)	0.03	0	0.03	0	0.03	0.21	0.23
L22	53(256)	46(112)	46(39)	18(60)	18(60)	18(60)	18(60)	0.03	0	0.03	0	0.03	0.09	0.09
L23	32(56)	20(56)	20(14)	7(28)	7(28)	7(28)	7(28)	0.01	0	0.01	0	0.01	0.02	0.02
A2	273(7099)	33(1260)	18(580)	24(1036)	26(1100)	14(716)	14(716)	0.04	0.05	0.04	0.05	0.04	0.66	0.64
A3	275(24110)	35(3271)	20(1655)	27(2584)	29(2648)	17(2264)	17(2264)	0.03	0.03	0.03	0.03	0.03	5.51	3.91
A4	277(46789)	37(5950)	22(3086)	30(4644)	32(4708)	20(4324)	20(4324)	0.06	0.05	0.06	0.07	0.09	12.83	10.87
A5	279(182828)	39(21989)	24(11637)	33(16944)	35(17008)	23(16624)	23(16624)	0.24	0.27	0.24	0.33	0.29	50.44	128.87
A6	281(364211)	41(43372)	26(23036)	36(33340)	38(33404)	26(33020)	26(33020)	0.44	0.47	0.44	0.57	0.54	54.09	49.66
A7	335(844577)	95(203738)	80(163434)	86(332288)	88(332352)	76(331968)	76(331968)	1,083.09	68.58	1,083.09	68.89	1,083.43	528.12	409.37
P3	24(546)	15(114)	15(90)	15(120)	15(120)	15(120)	15(120)	0.01	0.01	0.01	0.01	0.01	0.06	0.06
P4	32(728)	20(152)	20(120)	20(160)	20(160)	20(160)	20(160)	0.02	0.01	0.02	0.01	0.02	0.13	0.08
P5	40(910)	25(190)	25(150)	25(200)	25(200)	25(200)	25(200)	0.02	0	0.02	0	0.02	0.34	0.28

Figure 7.4: The effect of reverse simulation

7.2.2 Minimizing Parity Automata

The situation totally changes when we consider parity automata. In Figure 7.5 the experimental results are given when parity automata are generated using the nonconfluent determinization where the splitting of boolean formula is turned on. We generated the nondeterministic automaton using fair simulation and reverse simulation. Accordingly, we obtain for every minimization construction the same (non-minimized) deterministic parity automaton.

Direct and delayed simulation is able to reduce the size of the automata modestly, where delayed simulation performs slightly better on the AMBA case study. However, fair simulation is able to minimize the automata much better than the other simulation relations. But this reduction comes at a price. The computation time for fair simulation is in most cases much higher than the computation time for the other two simulation relations. However, this higher computation time for minimization is well invested when it comes to controller synthesis. Here the fair simulation minimized games are much faster solved than the other games. Given that for big examples like the AMBA case study or the Philosophers case study, the overall computation time is dominated by the solution of games and not by the minimization, we conclude that fair simulation is a valuable tool to reduce the time needed to synthesize a controller.

After having evaluated the different determinization constructions and the different minimization constructions, the last parts of this chapter are devoted to different case studies that demonstrate the feasibility of controller synthesis. For those case studies, we used fair simulation in connection with the determinization construction based on the hierarchy. Again, each of the formula already belonged to the temporal logic hierarchy so that no manual rewriting was necessary.

No.	NDet	FaN	FaD	FaDiD	FaDeD	FaFaD	DiDT	DeDT	FaDT	DiS	DeS	FaS
L1	24(84)	21(72)	26(188)	23(164)	23(164)	13(84)	0.01	0.18	0.34	0.03	0.03	0.02
L2	24(108)	21(96)	26(188)	23(164)	23(164)	13(84)	0.12	0.27	0.42	0.04	0.04	0.01
L3	24(108)	21(96)	31(208)	28(184)	28(184)	18(104)	0.06	0.38	0.5	0.16	0.13	0.08
L4	24(104)	21(92)	37(232)	34(208)	34(208)	24(128)	0.11	0.29	0.49	0.11	0.13	0.05
L5	32(196)	29(184)	39(280)	36(256)	36(256)	26(176)	0.14	0.31	0.5	0.17	0.19	0.12
L6	40(228)	37(216)	41(288)	38(264)	38(264)	28(184)	0.04	0.38	0.41	0.27	0.2	0.19
L7	32(202)	29(190)	39(280)	36(256)	36(256)	26(176)	0.02	0.32	0.5	0.19	0.19	0.07
L9	8(28)	6(25)	20(80)	18(72)	18(72)	10(40)	0.03	0.06	0.08	0.14	0.12	0.04
L10	1(1)	1(1)	2(4)	2(4)	2(4)	2(4)	0	0	0.01	0.01	0.01	0
L11	8(28)	6(26)	18(72)	16(64)	16(64)	8(32)	0	0.07	0.1	0.1	0.09	0.01
L12	8(28)	6(26)	18(72)	16(64)	16(64)	8(32)	0.01	0.07	0.1	0.09	0.08	0.01
L13	2(2)	2(2)	4(8)	4(8)	4(8)	4(8)	0	0	0.01	0.02	0.02	0.03
L15	12(44)	10(42)	24(96)	22(88)	22(88)	14(56)	0.01	0.07	0.13	0.39	0.41	0.08
L16	18(66)	15(63)	36(144)	33(132)	33(132)	21(84)	0.02	0.07	0.16	4.16	4.29	0.08
L19	12(42)	9(37)	31(124)	28(112)	28(112)	16(64)	0.01	0.11	0.15	0.2	0.14	0.08
L20	8(22)	7(21)	17(52)	16(48)	16(48)	12(32)	0.01	0.06	0.08	0.2	0.25	0.09
L21	64(240)	64(240)	40(160)	40(160)	40(160)	40(160)	0.04	0.18	0.2	1.19	0.69	1.19
L22	44(101)	41(99)	36(120)	34(112)	34(112)	26(80)	0.02	0.16	0.18	0.67	0.59	0.11
L23	16(8)	6(8)	11(44)	10(40)	10(40)	8(32)	0.01	0.06	0.07	0.02	0.04	0.02
A2	270(2295)	19(784)	38(1868)	26(1484)	24(1356)	16(844)	0.08	0.1	0.16	1.79	1.74	0.65
A3	272(4894)	21(2471)	41(5720)	29(5336)	27(4824)	19(2776)	0.09	0.14	0.23	40.81	12.88	3.89
A4	274(8357)	23(4718)	44(10852)	32(10468)	30(9444)	22(5348)	0.16	0.22	0.3	49.39	50.3	14.15
A5	276(29100)	25(18165)	47(41584)	35(41200)	33(37104)	25(20720)	0.89	0.91	0.81	735.38	695.52	77.84
P3	24(156)	15(108)	45(360)	42(336)	42(336)	30(240)	0.02	0.14	0.26	35.64	29.98	0.2

Figure 7.5: Minimizing parity automata

7.3 AMBA AHB Case Study

ARM's Advanced Micro-controller Bus Architecture (AMBA) [62] defines the *Advanced High performance Bus (AHB)*, an on-chip communication standard that connects devices like processor cores, caches and DMA Arbiters. It is possible to connect up to 16 Masters and up to 16 slaves to the bus. While the masters initiate communication with a slave of their choice, the slaves are passive and respond only to a request. AHB is a pipelined bus, that means that multiple masters can access the bus while one of the masters is transferring data and yet another master transfers address information. A bus access can be a single transfer or a burst which consists of a specified or unspecified number of transfers.

The controller we want to synthesize here needs to implement an arbiter that handles the different requests coming from the masters. This case study has been chosen as an example of a specification that has industrial size in order to evaluate whether synthesis of real world examples is possible. It has been already used as a case study in [12, 48] from where the formal specification is taken. We will give a short summary and refer the interested reader to [48] which gives a detailed description of the problem. In the following we give a description of the signals used in the specification. The notation $S[n:0]$ denotes an (n+1)-bit signal.

- $hbusreq_i$ - a request from Master i to access the bus (driven by the masters)
- $hlock_i$ - a request to receive a locked (uninterruptible) access to the bus (driven by the masters)
- *hmaster[3:0]* - the master that currently owns the bus (driven by the arbiter)
- *hready* - high if the slave has finished processing the current data (driven by the arbiter)
- $hgrant_i$ - signals that if *hready* is high, *hmaster=i* will hold in the next tick (driven by the arbiter)
- *hmastlock* - indicates that the current master is performing a locked access (driven by the arbiter)
- *hburst[1:0]* - one of SINGLE (a single transfer), BURST4 (a four-transfer burst access) or INCR (unspecified length burst) (driven by the arbiter)

To formalize the specification, additional signals driven by the arbiter has been introduced in [12, 48].

- *start* indicates the start of an access

- *locked* indicates that the bus will be locked at the next start of an access
- *decide* indicates the time slot in which the arbiter decides who the next arbiter will be.

Since some guarantees the controller has to satisfy may only be satisfiable in case some assumptions on the environment holds, we start in the following by the assumptions on the environment, followed by the guarantees for the controller:

A1: During a locked unspecified length burst, leaving busreq high locks the bus. This is forbidden by the standard.

$$\mathsf{G}\,((\mathit{hmastlock} \wedge \mathit{hburst}{=}{=}\mathit{incr}) \rightarrow \mathsf{XF}\neg \mathit{busreq})$$

A2: Leaving *hready* low is forbidden by the standard.

$$\mathsf{GF}\mathit{hready}$$

A3: The lock signal is asserted by a master at the same time as the bus request.

$$\bigwedge_{i=0}^{N} \mathsf{G}\,(\mathit{hlock[i]} \rightarrow \mathit{hbusreq[i]})$$

A4: All inputs are initially low.

$$\bigwedge_{i=0}^{N} (\neg \mathit{hbusreq[i]} \wedge \neg \mathit{hlock[i]} \wedge \neg \mathit{hready})$$

G1: A new access can only start when *hready* is high.

$$\mathsf{G}\,(\neg \mathit{hready} \;\rightarrow \neg\; \mathit{start})$$

G2: When a locked unspecified length burst starts, a new access can only start after the current master (*hmaster*) releases the bus by lowering *hbusreq[hmaster]*.

$$\mathsf{G}\,((\mathit{hmastlock} \wedge \mathit{hburst}{=}{=}\mathit{INCR} \wedge \mathit{start}) \rightarrow \mathsf{X}\,[\neg \mathit{hbusreq}\;\mathsf{B}\;\mathit{start}])$$

G3: When a length-four locked burst starts, no other access starts until the end of the burst. The burst ends after the fourth occurrence of *hready*. [6]

$$\mathsf{G}\left(\begin{array}{l} \mathit{hmastlock} \wedge \mathit{hburst} == \mathit{BURST4} \wedge \mathit{start} \rightarrow \\ \mathit{hready} \wedge \mathsf{X}\left[\neg \mathit{start}\;\mathsf{U}_{[3]}\;\mathit{hready} \wedge \neg \mathit{start}\right] \vee \\ \mathsf{X}\left[\neg \mathit{start}\;\mathsf{U}_{[4]}\;\mathit{hready} \wedge \neg \mathit{start}\right] \end{array}\right)$$

[6] $[\varphi\;\mathsf{U}_{[n]}\;\psi]$ means that φ holds until n occurrences of ψ occurred. This means e. g. that $[\varphi\;\mathsf{U}_{[3]}\;\psi]$ is rewritten to $[\varphi\;\mathsf{U}\;\mathsf{X}\,[\varphi\;\mathsf{U}\;\mathsf{X}([\varphi\;\mathsf{U}\;\psi])]]$.

G4: When *hready* is high, *hmaster* is set to the master that is currently granted the bus.

$$\bigwedge_{i=0}^{N} (\mathit{hready} \rightarrow (\mathit{hgrant[i]} \leftrightarrow \mathsf{X}(\mathit{hmaster} == i))$$

G5: Whenever *hready* is high, the signal *hmastlock* copies the signal *locked* to ensure that an uninterrubtable access to the bus is possible.

$$\mathsf{G}\,(\mathit{hready} \rightarrow (\mathit{locked} \leftrightarrow \mathsf{X}(\mathit{hmastlock})))$$

G6: If the arbiter does not start a new access, *hmaster* and *hmastlock* do not change.

$$\bigwedge_{i=0}^{N} \mathsf{G}\left(\mathsf{X}\left(\neg \mathit{start} \rightarrow \begin{pmatrix} \mathsf{X}(\mathit{hmaster} == i) \leftrightarrow (\mathit{hmaster} == i) \wedge \\ \mathsf{X}(\mathit{hmastlock}) \leftrightarrow \mathit{hmastlock} \end{pmatrix}\right)\right)$$

G7: When the arbiter decides to grant the bus, it uses *locked* to remember whether a locked access was requested.

$$\bigwedge_{i=0}^{N} \mathsf{G}\,((\mathit{decide} \wedge \mathsf{X}(\mathit{hgrant[i]})) \rightarrow (\mathit{hlock[i]} \leftrightarrow \mathsf{X}(\mathit{locked})))$$

G8: The *grant* or *locked* signals change only if *decide* is high.

$$\bigwedge_{i=0}^{N} \mathsf{G}\left((\neg \mathit{decide}) \rightarrow \begin{pmatrix} \mathsf{X}(\mathit{hgrant[i]}) \leftrightarrow \mathit{hgrant[i]} \\ \mathsf{X}(\mathit{locked}) \leftrightarrow \mathit{locked} \end{pmatrix}\right)$$

G9: The bus is fair, i. e. every request that is active and not lowered is eventually served [7].

$$\bigwedge_{i=0}^{N} \mathsf{GF}\,(\neg \mathit{hbusreq[i]} \vee \mathit{hgrant[i]})$$

G10: The bus is not granted without a request, except to master 0. If there is no request, the bus is given to master 0.

$$\bigwedge_{i=1}^{N} \mathsf{G}\,(\neg \mathit{hgrant[i]} \rightarrow [\neg \mathit{hgrant[i]}\ \mathsf{U}\ \mathit{hbusreq[i]}]) \wedge$$
$$\mathsf{G}\left(\mathit{decide} \wedge \bigwedge_{i=0}^{N} (\neg \mathit{hbusreq[i]} \rightarrow \mathsf{X}(\mathit{hgrant[0]}))\right)$$

[7] We have slightly rewritten the equivalent specification given in [48] to improve the synthesis problem. Actually the input file for Anzu [48] used also this rewritten specification.

G11: An access by master 0 starts in the first clock tick which means that a decision is taken. All other signals are initially low.

$$\mathit{decide} \wedge \mathit{start} \wedge \mathit{hgrant[0]} \wedge \neg \mathit{hmastlock} \wedge \bigwedge_{i=1}^{N} \neg \mathit{hgrant[i]}$$

It actually turned out that not all assumptions are necessary to satisfy the guarantees. Assumptions 3 and 4 are superfluous so that we left them out in the experiments. However, Assumptions 1 and 2 are necessary to satisfy Guarantee 9 so that we change Guarantee 9 to :

G 9':

$$\bigwedge_{i=0}^{N} \left(\begin{array}{l} (\mathsf{GF}\mathit{hready} \wedge \mathsf{G}\,((\mathit{hmastlock} \wedge \mathit{hburst}{=}{=}\mathit{incr}) \rightarrow \mathsf{XF}\neg \mathit{busreq})) \rightarrow \\ (\mathsf{GF}\,(\neg \mathit{hbusreq[i]} \vee \mathit{hgrant[i]})) \end{array} \right)$$

The overall specification is now given as:

$$\bigwedge_{i=1}^{8} \mathrm{G}i \wedge \mathrm{G}9'$$

We ran our algorithm three times to evaluate the performance of the optimizations given in Chapter 6. The first optimization we consider is the removal of colors whenever the minimal color of the environment does not label a state of the subgame according to Proposition 10. The second optimization refers to Proposition 9 and means that we first perform the calculation of the winning set of the safety-fragment before solving the overall game. Figure 7.6[8] shows the time needed to synthesize an arbiter for those specifications. In the first column, the safety optimization is turned on, but not the removal of colors. In the second column, the removal of colors is enabled but not the safety handling. In the third column, both optimizations are turned on. Finally the last column displays the runtime of Anzu [12]. Remember that Anzu is a tool that is also symbolically implemented and uses the specialized generalized Streett(1) approach of Piterman et. al [79]. But instead to our tool, the deterministic automata have to be generated manually.

Without the color-optimization, our algorithm is not able to perform the task for more than 5 masters. The problem is that when the color-optimization is turned off, the generalized parity algorithm does much unnecessary work and indeed generates

[8]Here TO means that the task could not performed within 20 hours, whereas MO means that more than 2 GB of memory has been used which also terminated the process.

No	color-unoptimized	safety-unoptimized	all-optimized	anzu
AMBA2	2.6	2.09	1.36	2.36
AMBA3	9.77	26.23	6.47	17.14
AMBA4	62.34	38,98	13.03	52.83
AMBA5	2881	335.30	67.77	131.71
AMBA6	TO	581.09	81.61	716.88
AMBA7	TO	3949.44	823.82	1338.13
AMBA8	TO	42757.21	5594.70	4542.37
AMBA9	TO	TO	2749.62	3363.56
AMBA10	TO	TO	678.46	6163.66
AMBA11	TO	TO	982.48	9475.98
AMBA12	TO	TO	7544.34	MO
AMBA13	TO	TO	2161.24	16683.55
AMBA14	TO	TO	2293.57	46514.46
AMBA15	TO	TO	2022.44	MO

Figure 7.6: Time Spent During Synthesis on the AMBA Case Study

for k masters $k!$ sub-strategies. Joining those $k!$ sub-strategies to an overall strategy is impossible for larger numbers of k. Although this blowup is in general unavoidable, since the memory needed to win Streett games is factorial in the number of Streett games, the color-optimization manages to avoid the introduction of unnecessary sub-strategies so that for k masters, k sub-strategies suffice. This has to do with the special structure of Guarantee 9[9] that has the form $\bigwedge_{i=0}^{n} \Phi_i \rightarrow \Psi_i$. In the special case of the AMBA example, all Φ_i are the same. So indeed, the Streett conditions of the AMBA example have a restricted form that can be used efficiently: states that do not satisfy Φ_i need only be considered once in the generalized parity algorithm and afterwards can be removed according to Property 10. We believe that this is not only a special case of the AMBA example, but often occurs when one considers specifications with assumptions on the environment that will always be the same for all Streett conditions.

The safety-optimization is not as important as the color-optimization but nevertheless it is impossible to generate an arbiter for more than 8 masters without it. Hence, turning both optimizations on leads to an enormous performance boost. Not only are we able to solve the AMBA problem for up to 15 masters, which is the maximal number of masters the specification allows, but we are also faster than the tool Anzu which is specifically tailored for those form of specifications and where the

[9]All other guarantees are purely safety guarantees

deterministic automata are generated manually. As can be seen, this superiority of our algorithm comes mainly from the ability to solve safety games separately and from the color-optimization.

7.4 Dining Philosophers

Although the dining philosophers problem is more whimsical than practical, it is similar to realistic problems in which different processes requires simultaneous access to more than one resource. Consequently, the problem is often used to illustrate and compare different synchronization mechanisms. We use it here as a tutorial example since it is rather small and shows well how the environment can be modeled using a Quartz program. Moreover, in contrast to the other examples, *multiple* resources must be arbitrated by the controller. The dining philosopher problem is formulated as follows:

> Five philosophers sit around a circular table. Each philosopher spends his life either thinking or eating. In the center of the table is a plate of spaghetti. Each philosopher must use two forks to be able to eat the spaghetti. Unfortunately, only one fork is available for each philosopher, each placed in between two pairs of philosophers. The problem is to write a controller that assures that every philosopher that is hungry must eventually be able to eat, i. e. he may acquire both forks on the left and right of him. Thus we must avoid the unfortunate situation that the philosophers are all hungry, but none is able to eat because every one tries to pick up a fork which prevents his neighbor from eating.

We model this environment using the Quartz program given in Listing 7.1. Everything that is done here is to simply model the assumption that whenever one philosopher n wants to require the forks, he can eat in the next step whenever the controller grants fork n and fork $n+1$, thus the forks directly on his left and on his right which is done in parallel. As the formal specification we want to assure that every philosopher who wants to require the forks is able to eat which is formalized as follows:

$$\Phi = \bigwedge_{i=0}^{N} \mathsf{G}\,(\mathit{req[i]} \rightarrow \mathsf{F}(\mathit{eat[i]}))$$

However, this specification alone is not controllable. We grant the forks only whenever at that point of time the philosopher still leaves req high. Hence, we have to add the

```
module Philosopher (bool ? grant [N], req[N] , & eat [N])
{
    loop{
        parallel (nat [sizeOf(N-1)] i=0 .. (N-1))
        {
            next (eat[i])=req[i] & grant[i]
                        & (!grant[(i+1)%N ]);
        }
        pause;
    }
}
```

Listing 7.1: Dining Philosophers

additional constraint that the request is not lowered until the respective philosopher is allowed to eat.

$$\Phi = \bigwedge_{i=0}^{N} (\mathsf{G}\,(req[i] \rightarrow [req[i]\ \mathsf{U}\ eat[i]])) \rightarrow (\mathsf{G}\,(reg[i] \rightarrow \mathsf{F}(eat[i])))$$

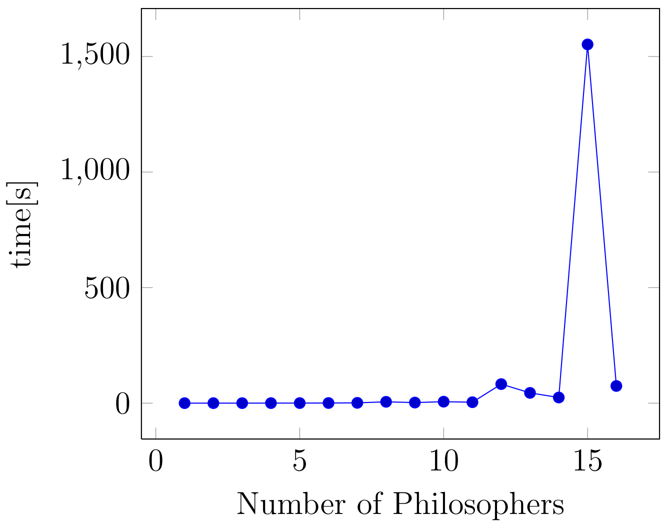


Figure 7.7: Experimental Result for Dining Philosophers

The different runtimes for the experiments performed on the dining philosophers case study are shown in Figure 7.7. Quite surprisingly, the time needed to solve the problem remains nearly constant with the difference of 15 philosophers where a large peak can be seen. For 16 philosophers the synthesis time is again modestly. However, for larger number of Philosophers than 16 we were not able to finish the synthesis task within one hour.

7.5 NIM Game

The roots of the NIM game are unclear. Some variants of NIM have been played since ancient times. It is especially popular in mathematical game theory because it is a game that is fairly easy to describe and thus as early as 1901 the complete theory for the game has been developed.

> NIM is a two-player turn-based game in which players alternately remove objects from heaps. On each turn, one of the two players chooses a heap and a non-zero number of objects that he removes from this heap. The player who removes the last object wins the game.

The key to the theory of the game is based on an exclusive or (xor) calculation on the heap sizes which is often called the NIM-sum. The NIM-sum is calculated by the digit-wise calculation of the exclusive or over the digits of the heap sizes, e. g. if the heap sizes are (in binary form) 011,100,101 we obtain 010. The winning strategy (for any of the two players) is to finish every move with a NIM-sum of 0. Whenever the NIM-sum of the game is 0 at the beginning, the player that starts has a winning strategy. So e. g. when heap i contains $2i + 1$ objects at the beginning, the start player has a winning strategy for 3 heaps, no winning strategy for 4 heaps but again a winning strategy for 5 heaps. We model this game with the Quartz program given in Listing 7.2 where the Controller takes the role of player A. During the initialization phase, the heap on position i is filled with $2i + 1$ objects. The two player alternate in turns. From outside, we trigger the Boolean variable first that determines whether the controller chooses its move first. Whenever one of the player moves, it is checked whether he chooses more than 0 objects and whether the heap on position i contains more objects than the player wants to remove. If so, it is a valid move and the next player takes over, otherwise the player chooses again. The game ends, if *EndOfGame* is true, which stands for

$$\mathit{EndOfGame} := \bigwedge_{i=0}^{\mathit{rows}} \mathit{Board[i]{=}{=}0}$$

```
module  NIM_Game(nat[sizeOf(rows)]  ?rowA,rowB,
           nat[sizeOf(size)]  ?numA,numB,
           nat[sizeOf(size)]  &Board[rows], & NIM_Sum,
           bool &turnA)
{
    // Initialization
    turnA = first;
    parallel(nat[sizeOf(rows)]  i=0 .. rows)
        Board[i] = 2*i+1;
    // now play the game
    while(!EndOfGame) {
            if(turnA) {
                if( (numA<=Board[rowA]) & (numA>0)) {
                        next(Board[rowA]) = Board[rowA]-numA;
                        next(turnA) = false;
                    }
                else
                    next(turnA) = true;
            }
            else {
                if((numB<=Board[rowB]) & (numB>0)) {
                        next(Board[rowB]) = Board[rowB]-numB;
                            next(turnA) = true;
                    }
                else
                    next(turnA)=false;
            }
            pause;
    }
}
```

Listing 7.2: Quartz Implementation of the NIM Game

Since moves of a player take effect at the beginning of the next round (every assignment is a *next*-assignment), the specification to fulfill is

$$\mathsf{G}(\mathit{EndOfGame} \rightarrow \neg \mathit{turnA})$$

However, this specification alone is not sufficient to generate a valid controller. Each player can stop the game from making progress by choosing to remove either zero objects from a heap or more objects than present on a heap. Since this is the fastest way to win, the controller that is constructed from our algorithm for this specification chooses always zero objects. However, since the opponent player can do the same, replacing the G operator with an F operator generates an uncontrollable specification. Instead, we modify the specification to:

$$\mathsf{G}(((\mathit{turnA}- > (\mathit{numA} \leq \mathit{Board[rowA]} \wedge \mathit{numA} > 0))) \wedge \mathsf{G}(\mathit{EndOfGame} \rightarrow \neg\ \mathit{turnA})$$

This case study is interesting, since we can in an elegantly easy way turn a controllable problem into an uncontrollable problem without changing the transition relation of the game by letting either the controller or the environment choose first (Remember that the start player has a winning strategy if and only if the NIM-sum at the beginning is 0).

As our experiment (shown in Figure 7.8) indicates, the total runtimes do not differ whether the environment or controller chooses first. The only difference is due to the strategy construction in case the problem instance is controllable, however, this difference is in the order of milliseconds for the small examples and is not a large factor in the overall solution time.

7.6 Island Traffic Control Problem

The island traffic controller problem goes back to [29] and has been considered several times as an example of a model checking problem [93, 94, 114, 113]. Like any traffic light control problem, it is a metaphor for coordinating access to a restricted resource, namely a one-lane tunnel between an island and a mainland. Hence, the controller of the tunnel must either grant access to the tunnel for the cars that start from the island or for those that start from the mainland, but never for both directions at the same time. At each end of the tunnel, there is a traffic light which is controlled by signals *ml_red_light*, *ml_green_light* for the mainland side and *il_red_light*, *il_green_light* on the island side. Moreover, on both sides of the tunnel, there are sensors that detect whether a car is in *front* of the traffic light and wants to enter the tunnel (signals *il_enter* and *ml_enter*), and whether a car is leaving the tunnel at this side (signals *il_leave* and *ml_leave*). We assume that the sensors are fast enough to detect different

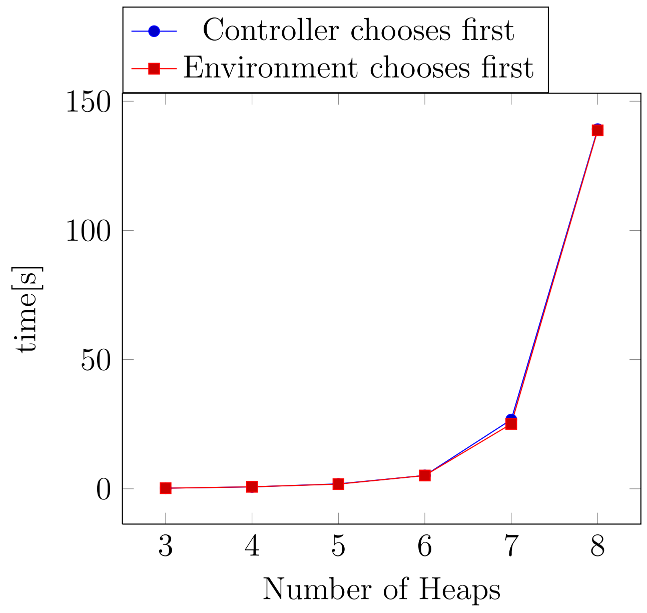


Figure 7.8: Experimental Result for the NIM Game

cars, e. g. whenever a car wants to enter the tunnel from the island, it will drive over the sensor, signal *il_enter* will become true and remain true until the car leaves the sensor when it becomes false until the next car arrives on the sensor. The task of the island traffic controller problem is to implement a control system that satisfies certain safety conditions, namely the island and the mainland side should not have green lights at the same point of time, and a change from red to green light should only be possible when the tunnel is empty. Moreover, there is a constraint on the maximal number of cars on the island that should never exceed a threshold N. Clearly, we want the system to be fair, i. e. no traffic light should be indefinitely long red on any side.

Our Averest tool set [92] contains a solution of the island traffic control problem implemented in Quartz. In this solution the control problem is divided into three processes, one for managing exclusive access to the tunnel, and two for handling the sensor signals at the island/ mainland side i. e. the traffic light controllers. Additionally, two counters IC and TC are present for counting the number of cars on the island and inside the tunnel respectively. We remove the manager for tunnel access but keep the counters and the handling of the sensors inside the Quartz program. We start our considerations by introducing the signals used in the implementation. Since many signals are present on both sides, we will only introduce them once,

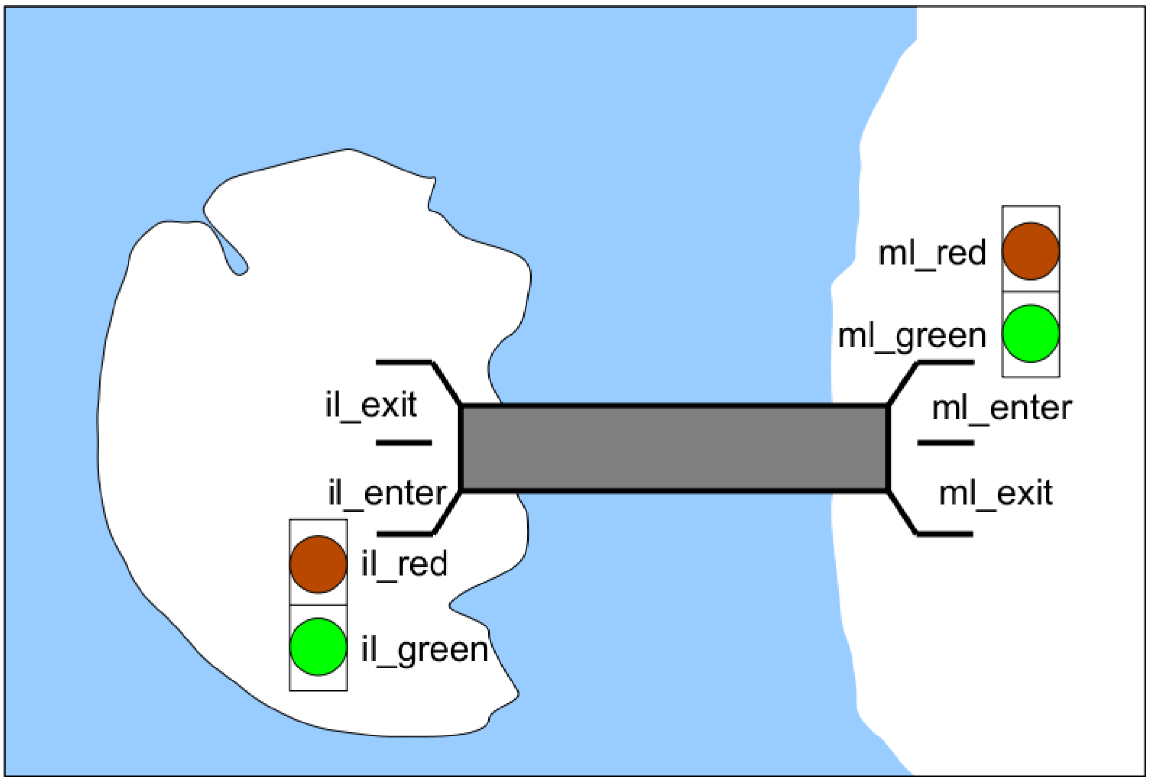


Figure 7.9: Island Traffic Control Problem

e. g. *X_red_light* means either red light signal on the island (*il_red_light*) or on the mainland (*ml_red_light*) side. We start by the sensor signals:

- *X_enter* - car wants to enter the tunnel from X side
- *X_leave* - car leaves the tunnel to X side

The following signals are outputs of the traffic light controller:

- *X_red_light* - used to switch on the red traffic lights
- *X_green_light* - used to switch on the green traffic lights
- *X_use* - tunnel is in use from X side, i. e. still some car is inside
- *X_req* - request to access the tunnel
- *tc_dec,tc_inc* - decrement/ increment tunnel counter

```
module Counter(event inc,dec,nat[sizeOf(MaxCars+1)] &v) {
    loop{
        if(inc & !dec)
            next(v) = v+1;
        else if(!inc & dec)
        next(v) = v-1;
        pause;
    }
}
```

Listing 7.3: Implementation of the Counters

- *ic_dec,ic_inc* - decrement/ increment island counter

Finally, the following signals are used by the arbiter to satisfy his guarantees:

- *X_grant* - grant access to the tunnel for the X side
- *X_release* - X side should release access to the tunnel

We use the same implementation for the counters (Listing 7.3) and for the traffic lights (Listing 7.4) on both sides of the tunnel. The implementation of the counters is straightforward, however, the implementation of the traffic light controllers needs some more attention. We have two loops that are working in parallel. The first loop handles the correct behavior of the *use* variable such that whenever a car is entering the tunnel (in the second loop) this flag remains true until all cars have left the tunnel. The second loop is divided in two phases: the green and the red light phase. While the controller is in the red light phase, cars can only leave the tunnel. We wait until signal *leave* is true, then we decrement the tunnel counter (signal *tc_dec*) and remain waiting until the car leaves the sensor (signal *leave* becomes false). A similar behavior can be observed in the green light phase with the difference that now also the island counter needs to be changed. To handle this, we introduce a signal *ic_change*. In case of the mainland controller, this signal is mapped to *ic_inc* and for the island controller to *ic_dec*. This has the effect that whenever a car enters the tunnel in island direction, the island counter is incremented and in the other direction decremented. Thus the island counter does not only count the number of cars on the island, but also the number of cars traveling to the island. This allows us to use the same traffic light controller for both sides, in opposition to e. g. [114] where two different controllers where given[10].

[10]Nevertheless, in a hardware implementation, both controllers need to be present, and thus, there are also variables in the BDD representing the implementation.

```
module TrafficLightController
(event grant, release, enter, leave, &green_light, &red_light,
    &use,&req, &tc_inc,&tc_dec,&ic_change){
    loop{
        if (use & tc>0) emit next (use);     pause;
    } ||
    loop{
    // Red light phase: cars can only leave the tunnel.
        weak abort loop{
            while(!leave){
                pause;
                emit red_light;
                if(enter) emit req;
            }
            emit tc_dec;
            while(leave){
                pause;
                emit red_light;
                if(enter) emit req;
                    }
        }
        when (!leave & grant);
    // Green light phase: cars can only enter the tunnel.
        weak abort loop{
            while(!enter){
                pause;
                emit green_light;emit use;
            }
            emit ic_change;emit tc_inc;
            while(enter) {
                pause;
                emit green_light;emit use;
            }
        }
        when( release);
    }
}
```

Listing 7.4: Implementation of the Island and Mainland Controller

In the following, a specification for the tunnel access controller is developed. To be able to synthesize a controller, we need to impose some assumptions on the environment. Those assumptions correspond to the behavior of the sensors in the real physical world. For example, since *ic_leave* and *ml_leave* are input signals, they might potentially also hold in case the tunnel is empty, but this can never be the case in reality. This gives us the first assumption on the environment, the others are formulated using a similar pattern:

EmptyTunnel: No car can leave the tunnel when the tunnel is empty:

$$\mathsf{G}\,((tc == 0) \rightarrow (\neg il\_leave \wedge \neg ml\_leave))$$

EmptyIsland: No car can leave the island when the island is empty:

$$\mathsf{G}\,(ic == 0) \rightarrow \neg il\_enter)$$

FiniteTunnelLeaving: No car remains indefinitely long in the tunnel:

$$\mathsf{G}\,((il\_red\_light \wedge ml\_red\_light) \rightarrow \mathsf{F}(tc == 0))$$

LeaveIsland No car remains indefinitely long on the island:

$$\mathsf{G}\,((ic > 0) \rightarrow \mathsf{F}(il\_leave))$$

Notice that the last assumption prevents the system from blocking because otherwise a full island would mean that no new car would be allowed to enter the island.

After having defined the assumptions, the following specifications describe the conditions that should be guaranteed by the tunnel access controller :

Lightsignal : The traffic lights should not be green on both sides :

$$\mathsf{G}\,(\neg(il\_green\_light \wedge ml\_green\_light))$$

ChangeLight: The traffic light should change from red to green only when the tunnel is empty:

$$\mathsf{G}\begin{pmatrix} ((ml\_red\_light \wedge \mathsf{X}ml\_green\_light) \rightarrow (tc == 0)) \\ \wedge\,((il\_red\_light \wedge \mathsf{X}il\_green\_light) \rightarrow (tc == 0)) \end{pmatrix}$$

IslandNotOverCrowded: The maximal number of cars on the island is N:

$$\mathsf{G}\,(ic \leq N)$$

IslandFair: No car should wait infinitely long on the island side :

$$\mathsf{G}\,(\mathit{il\_enter} \rightarrow \mathsf{F}(\mathit{il\_green\_light}))$$

MainlandFair: No car should wait infinitely long on the mainland :

$$\mathsf{G}\,(\mathit{ml\_enter} \rightarrow \mathsf{F}(\mathit{ml\_green\_light}))$$

The overall specification is the conjunction of the following formula:

- **Lightsignal**
- **ChangeLight**
- **EmptyTunnel** $\rightarrow$ **IslandNotOverCrowded**
- (**EmptyTunnel** $\wedge$ **FiniteTunnelLeaving**) $\rightarrow$ **IslandFair**
- (**EmptyTunnel** $\wedge$ **FiniteTunnelLeaving** $\wedge$ **LeaveIsland**) $\rightarrow$ **MainlandFair**

The specification given above is nearly independent on the number of cars that are handled by the arbiter. In particular, the only thing that changes for different numbers of cars are the binary encoding of the counter variables. Thus, the number of state variables obtained from the specifications stays the same for every amount of cars. However, the number of state variables from the Quartz program linearly grows with the number of cars, respecting the formula $k = 19 + 2 \cdot \log_2 n$ when n is the number of cars and k is the number of state variables. In Figure 7.10 we have shown the runtime of our synthesis algorithm on the island traffic control problem. This example showed some unexpected behavior. As expected, the runtime grows with the number of cars to be handled. However, there are two sinks at 7 cars and at 13 cars that we can not satisfactorily explain. We believe that this has to do with the underlying BDD package to handle the propositional formula that symbolically encode the transition system. Since the counters are binary encoded, different car sizes may lead to totally different BDDs where the sizes need not directly depend on the number of cars[11]. This behavior is however not special to controller synthesis, but is also commonly seen in symbolic model checking.

[11] To e. g. represent 4 cars , we need to represent the values $\{0, \ldots 4\}$ and accordingly need 3 BDD variables which gives us the don't care values $\{5, 6, 7\}$. Those don't care values can be used by the BDD package to generate more or less efficient variable orderings.

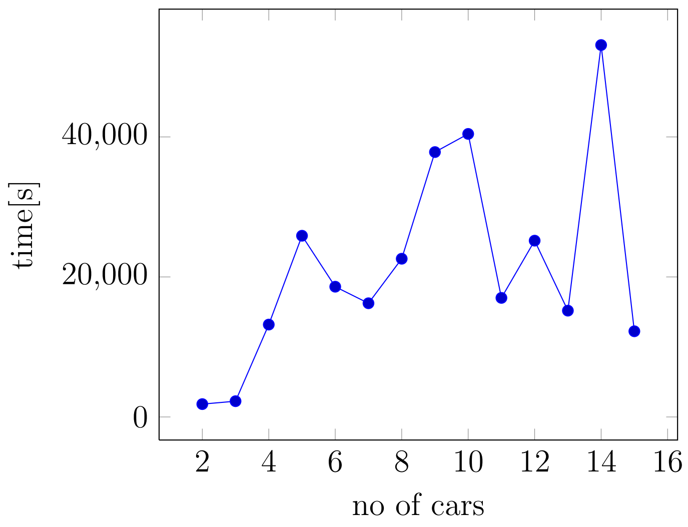


Figure 7.10: Running Time on the ITC Case Study

7.7 Synthesis of a Sorting Network

The problems considered so far are taken from two domains: either a strategy for a game is sought or an arbiter needs to be implemented. When really a piece of code is needed that satisfies a certain requirement, arbitration is a typical application domain of controller synthesis. However, controller synthesis is not limited to this application domain. In principle, arbitrary algorithms can be generated provided we can specify the correct behavior in LTL. This section is intended to investigate whether it is possible to generate a sorting algorithm using controller synthesis. This example shows that although controller synthesis is able to synthesize a valid controller, there is still much manual work to do in order to provide a skeleton so that the controller synthesis algorithm provides a meaningful result.

In order to apply controller synthesis to the problem of sorting an array, a skeleton is needed that defines the available choices of the controller. An interesting skeleton for this task is a sorting network [55] that can implement parallel sorting algorithms.

A sorting network is a network of wires and comparator modules to sort a sequence of numbers. The importance of sorting networks is that they allow a parallel implementation of sorting algorithms in hardware or on parallel computers. Any sorting network is build up from comparator nodes as shown in Figure 7.11 and wires that connect these comparator nodes.

A comparator node has inputs x_0, x_1 and outputs y_0, y_1. The maximum of the values x_0, x_1 is forwarded to y_0 and the minimum to y_1. In other words, the minimum value *sinks* to y_0 which is indicated by an arrow between the two connecting lines

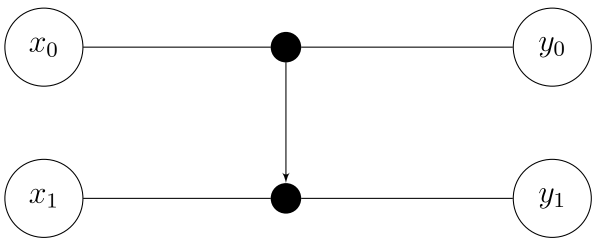


Figure 7.11: A Comparator Node

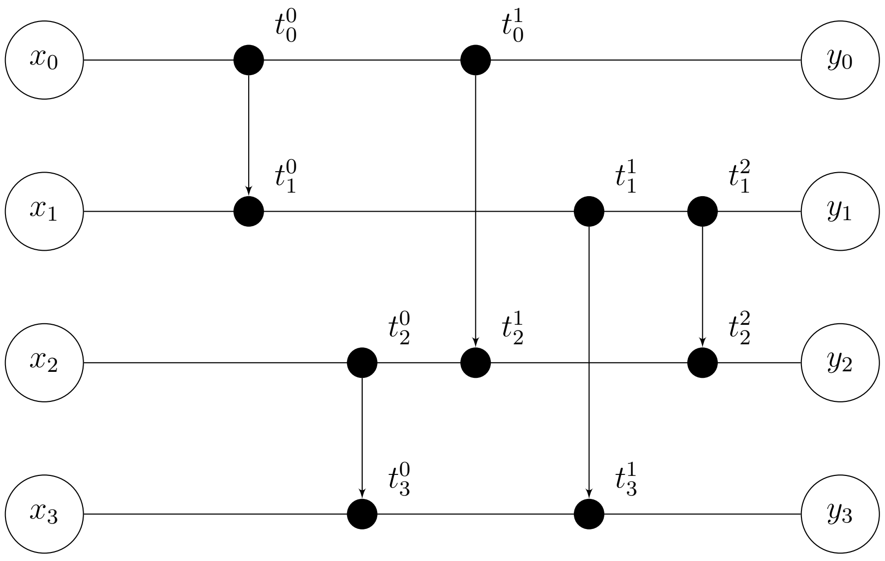


Figure 7.12: A Sorting Network

between inputs and outputs.

An example of a sorting network is given in Figure 7.12. First, the values x_0 and x_1 are compared which gives us the intermediate results t_0^0 and t_1^0. In a second comparator node, the intermediate results t_2^0 and t_3^0 are obtained from x_2 and x_3. In a second step, the intermediate results t_0^1 and t_2^1 are calculated from t_0^0 and t_2^0. The rest of the intermediate results are obtained analogously until the sorted array finally arrives at the outputs $y_0, \dots y_3$.

In realizations of sorting networks, and on parallel computers, it is possible to do non-overlapping comparisons at the same time; thus it is natural to try to minimize the delay time. The delay time of a sorting network depends on the maximum number of comparators in contact with any path through the network, where a path is any route from the inputs to the outputs that possibly switches wires at the comparators. Consider again the network from Figure 7.12. Since the inputs and outputs of the

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Upper	0	1	3	3	5	5	6	6	7	7	7	8	9	9	9	9
Lower	0	1	3	3	5	5	6	6	7	7	7	7	7	7	7	7

Figure 7.13: Upper and Lower bounds for optimal depth sorting networks

comparator nodes that calculate t_0, t_1 and the comparator node that calculate t_2, t_3 are independent, those two sorting operations can be performed in parallel without any additional delay time.

The maximum number of comparators in a row is often denoted the *depth* of a network, whereas the size denotes the total number of comparators used. The efficiency of a sorting network can be measured by its total size or by its depth. In Figure 7.12, the maximal depth is 3 while its size is 5.

The asymptotically best known sorting network, the AKS network [1] achieves depth $O(\log n)$ and size $O(n \log n)$ for n inputs, which is asymptotically optimal. Although theoretically optimal, the AKS network has little practical application because of a big constant hidden in the O-notation.

Finding sorting networks with little size and depth remains a fundamental open problem. For up to 10 inputs, optimal sorting networks are known. Figure 7.13 gives an overview of upper and lower bounds for up to 16 inputs [76].

The scope of this section is to investigate whether it is possible to automatically synthesize a sorting network using controller synthesis. In particular, we would like to investigate whether it is possible to synthesize a network with a maximal given depth. By iteratively increasing this bound on the depth we seek to find minimal depths for sorting networks.

7.7.1 Zero-one Principle

Proving the correctness of sorting networks is a hard task due to the intrinsic space explosion of a sorting networks. There are n! permutations of numbers in an n-wire network and testing them all is nearly infeasible with any tool that can be used for the analysis of systems. Hence, an important abstraction that we use also here is the zero-one principle:

Theorem 18 ([55]). *If a sorting network sorts every sequence of 0's and 1's, then it sorts every arbitrary sequence of values.*

Thus, instead of sorting arbitrary arrays, we concentrate on arrays with boolean values. Thanks to the zero-one principle, this is no restriction. The sorting networks for boolean inputs that we automatically obtain from our controller synthesis procedure are thus equally well suited for inputs with arbitrary values.

7.7.2 Problems to Define Skeletons for Sorting Networks

As a first try to synthesize a sorting network, we used the Quartz program given in Listing 7.5. It consists of three different modules, *CMP* represents a comparator node that are used in the main module sorter. Moreover, it is simplified in the sense that in every step, at most one comparison may take place.

In that module, first the input given in the array a is stored in an array b. This ensures that the environment can not falsify the specification by choosing another input than in the first step. This input is also copied to the array s that represents the wires (respectively the results carried by the wires in different stages) of our sorting network. As controllable inputs we have indices *i0, i1, j0 and j1* that represent the input (*i0, i1*) respectively the output (*j0, j1*) wires of the comparator node at the currently active stage of the sorting network. Moreover, we have the controllable event *ready* that signals that the controller has finished its operation. As additional output signals we have *error* and *counter*.

The *error* signal is needed to prevent the controller from doing illegal wire connections. First it prevents write conflicts [92]. Write conflicts occur in a Quartz program when two different values are assigned to the same variable. This corresponds to the case in the physical world that two different output signals are connected to the same wire. Clearly, this makes no sense in the physical world and neither in the transition relation that is obtained from the Quartz program. Having a write conflict makes the result of the controller synthesis procedure unpredictable since in that case the transition relation obtained from the Quartz program is not properly defined [92]. Additionally, it is checked by *error* that the values chosen by the controller are proper indices to access the array

Finally, the *counter* signal is used to count the number of different stages in the sorting network, i. e. the number of comparisons done.

We used the following specifications:

Guarantee 1: The algorithm terminates.

$$\mathsf{F}(\mathit{ready})$$

Guarantee 2: When *ready* is emitted, array s is sorted.

$$\mathsf{G}\left(\mathit{ready} \rightarrow \bigwedge_{i=0}^{\mathit{SIZE}-1} (s[i] \leq s[i+1])\right)$$

```
module Sorter(bool a[SIZE], & b[SIZE], & s[SIZE], →
    nat[sizeOf(SIZE)] ?i0, ?i1, ?j0,?j1, event ?ready,event & →
    correct,event & error,nat[sizeOf(DEPTH)+1] &counter){
    // initialization
    parallel(nat[sizeOf(SIZE-1)] i=0 .. (SIZE-1))
        {
            next(s[i]) = a[i];
            next(b[i]) = a[i];
        }
    pause;
    while (!ready) {
         if ((j0==j1) | (j0>SIZE) | (j1>SIZE)) emit error;
         else
          CMP(b,s,i0,i1,j0,j1);
         next(counter)=counter+1;
         pause;
    }
    Monitor(b,s,correct);
}

module CMP(nat[1] b[SIZE], & s[SIZE], nat[sizeOf(SIZE)] i0, →
    i1,j0,j1){
       if (!error)
          if (b[i0] > b[i1])
              next(s[j1]) = b[i0];
              next(s[j0]) = b[i1];
           }
           else{
              next(s[j1]) = b[i1];
              next(s[j0]) = b[i0];
           }

}
```

Listing 7.5: A wrongly Implemented Skeleton for Sorting Networks

Guarantee 3: The number of 0's in s and b are equivalent in every step. [12]

$$\mathsf{G}\left(\left(\sum_{i=0}^{SIZE-1} s[i]{=}0\right) = \left(\sum_{i=0}^{SIZE} b[i]{=}0\right)\right)$$

Guarantee 4: The controller chooses only proper values

$$\mathsf{G}(\neg error)$$

Guarantee 5: The algorithm terminates before *BOUND* is reached, i. e. the maximal steps performed is lower than *BOUND*

$$\mathsf{G}(counter < BOUND)$$

We run the controller synthesis algorithm for array sizes of up to 5 which is finished in a couple of seconds. The generated controller is also correct, as we have checked with SMV. However, it produces a controller that is useless at all. Instead of using the CMP nodes as comparator nodes, the controller uses them simply to permute the array and checks the correctness of the permutation directly on the array. For example, if the array is sorted from the beginning, the controller does nothing at all. This behavior can again be explained by the way the controller is constructed. Always the fastest way to achieve a goal is chosen. If the array is sorted, nothing needs to be done. However, obviously, this was not our intention. The problem is that in writing specification Guarantee 2, we encoded any sorted array of a given length in the transition relation. All the controller has to do is to modify s in a way that it is equivalent to this part of the transition relation. Generating a sorting algorithm in that way means that we have to encode all possible sorted arrays in the transition relation, and thus we implemented a multiplexer for all possible inputs. Clearly, this is infeasible for larger input sizes.

7.7.3 A Skeleton for Sorting Networks

Thus, another way to synthesize a sorting network is described in the following. In the previous implementation, the decision which wires are connected to a comparator node is done in multiple steps. Instead, now the connection between all the wires is done in the first step using a permutation matrix and afterwards it is checked that

[12]The expression $\mathsf{G}\left(\left(\sum_{i=0}^{SIZE-1} s[i]{=}0\right) = \left(\sum_{i=0}^{SIZE-1} b[i]{=}0\right)\right)$ can be rewritten to a boolean expression by enumerating every possible permutation of a boolean array with length *SIZE-1*.

i	$P_{0,i}$	$P_{1,i}$	$P_{2,i}$
0	1	2	0
1	0	3	1
2	3	0	2
3	2	1	3

Figure 7.14: Permutation matrix for the sorting network of Figure 7.12

those connections are valid. A permutation matrix is a two dimensional array P that describes the positions of the comparator nodes. Whenever in depth k there is a comparator node that connects i and j, set $P_{k,i} = j$ and $P_{k,j} = i$. Thus the sorting network of Figure 7.12 would be represented by the permutation matrix given in Figure 7.14, when we assume that the first two comparisons are done in parallel.

Using a permutation matrix we obtain the skeleton in Listing 7.6. As uncontrollable inputs we have the input array. As controllable inputs we have a permutation matrix *P[DEPTH][SIZE]*. In the first parallel statement, the inputs are compared according to the permutation matrix *P[0]*, so that the result of the comparator nodes are assigned to the Wires[0], which is the first level of the sorting network. The *if* statement ensures that *P[0]* represents a valid permutation of $\{0, \ldots SIZE - 1\}$. This ensures that no write conflict occurs. The rest of the levels *W[1]* ... *W[SIZE-1]* of the sorting network are calculated in a similar manner.

Since everything is calculated in the first step, the specification to synthesize a sorting network is:

Guarantee 1: The result at the end of the network is sorted:

$$\bigwedge_{i=0}^{SIZE-2} (Wire[DEPTH-1][i] \leq Wire[DEPTH][i+1])$$

Guarantee 2: The result is a permutation of the inputs:

$$\left(\left(\sum_{i=0}^{SIZE-1} \textit{input[i]=0}\right) = \left(\sum_{i=0}^{SIZE-1} \textit{Wire[DEPTH][i]=0}\right)\right)$$

7.7.4 Experimental Result for Sorting Networks

Having changed the skeleton in the described way, we performed several experiments to evaluate the time needed to synthesize a sorting network. The runtime of our algorithm is given in Figure 7.15 for different sizes and different depths. For size 5

```
module SortingNetwork (nat[1] input[SIZE], nat[sizeOf(SIZE)] ? →
   P[DEPTH][SIZE],nat[1] & Wire[DEPTH][SIZE]) implements →
   SortSpec(input,Wire){

    parallel (nat[sizeOf(SIZE-1)] i=0 .. (SIZE-1))
        {    if (P[0][P[0][i]]==i)
                {
                    if (P[0][i]==i) Wire[0][i]=input[i];
                    else if (i<P[0][i])
                        if (input[i]<input[P[0][i]]) {
                            Wire[0][i]=input[i];
                            Wire[0][P[0][i]]=input[P[0][i]];
                        }
                        else{
                            Wire[0][i]=input[P[0][i]];
                            Wire[0][P[0][i]]=input[i];
                        }
                }
        }
        parallel (nat[sizeOf(DEPTH-1)] k=1 .. (DEPTH-1))
        {
            parallel (nat[sizeOf(SIZE-1)] i=0 .. (SIZE-1))
            if (P[k][P[k][i]]==i)
                {
                    if (P[k][i]==i) Wire[k][i]=Wire[k-1][i];
                    else if (i<P[k][i])
                        if (Wire[k-1][i]<Wire[k-1][P[k][i]]) {
                            Wire[k][i]=Wire[k-1][i];
                            Wire[k][P[k][i]]=Wire[k-1][P[k][i]];
                        }
                        else{
                            Wire[k][i]=Wire[k-1][P[k][i]];
                            Wire[k][P[k][i]]=Wire[k-1][i];
                        }
                }
        }
}
```

Listing 7.6: A Skeleton for Sorting Networks

Size	Depth	Time[seconds]
2	2	0
2	3	0
2	4	0
2	5	0
3	2	0
3	3	0
3	4	1
3	5	4
4	2	4
4	3	225
4	4	7295
5	2	24
5	3	2269
5	4	terminated

Figure 7.15: Time to synthesize a sorting network

and depth 4 our algorithm did not terminate within 24 hours while consuming 4 GB of memory. Therefore we didn't try our algorithm on larger sizes.

In the following, we take a closer look at the sorting networks generated for size 3 and 4, since they show some unexpected behavior.

Our algorithm correctly identifies that for array sizes of 3 and 4 it is impossible to generate a sorting network with depth less than 3. For size 3 and depth 3, the sorting network of Figure 7.16 is obtained.

We also tried our algorithm with size 3 and depth 4. In that case, a non-optimal sorting network with respect to the delay time should be obtained. However, instead of simply copying all entries in one step, as we expected, our algorithm generated the sorting network of Figure 7.17. We believe that this has to do with the way a deterministic solution is obtained after the strategy has been calculated. So, although a solution that copies every entry once is a valid solution and thus also calculated in the strategy, the function to select exactly one solution need not pick it.

For size 4 and depth 3, the sorting network from Figure 7.18 is obtained.

The result for size 4 and depth 4 is shown in Figure 7.19. The sorting network is the same as for depth 3 regarding the first 3 levels of the sorting network. For the last level (that is superfluous), the permutation matrix simply copies the last step. Thus, the last two comparisons are done twice, so that they have no effect at all.

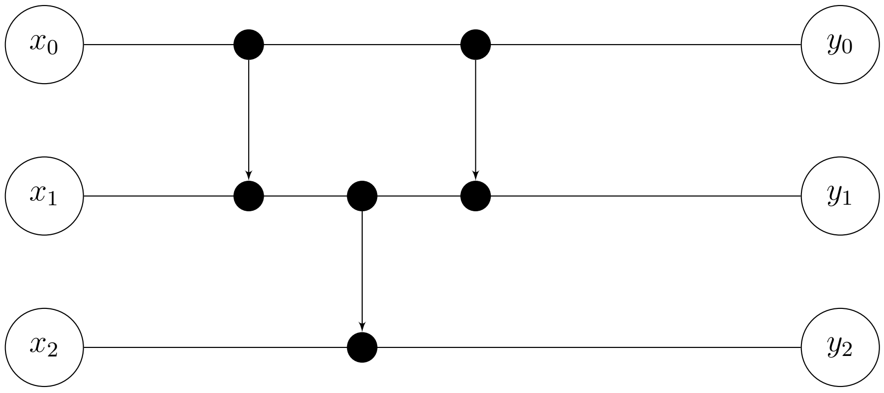


Figure 7.16: The Synthesized Sorting Network for Array Size 3 and Depth 3

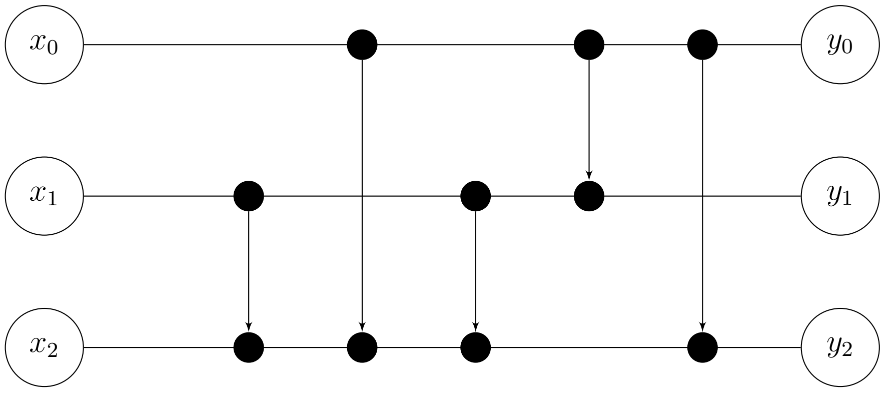


Figure 7.17: The Synthesized Sorting Network for Array Size 3 and Depth 4

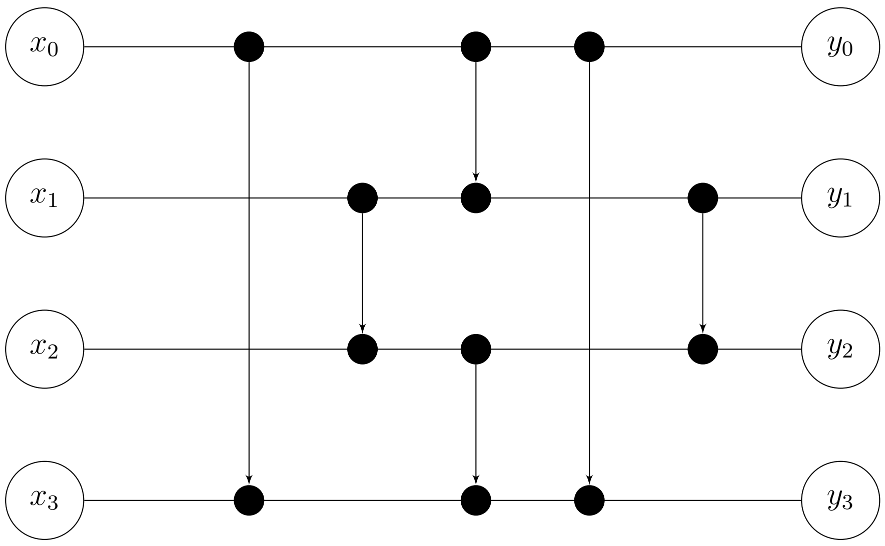


Figure 7.18: The Synthesized Sorting Network for Array Size 4 and Depth 3

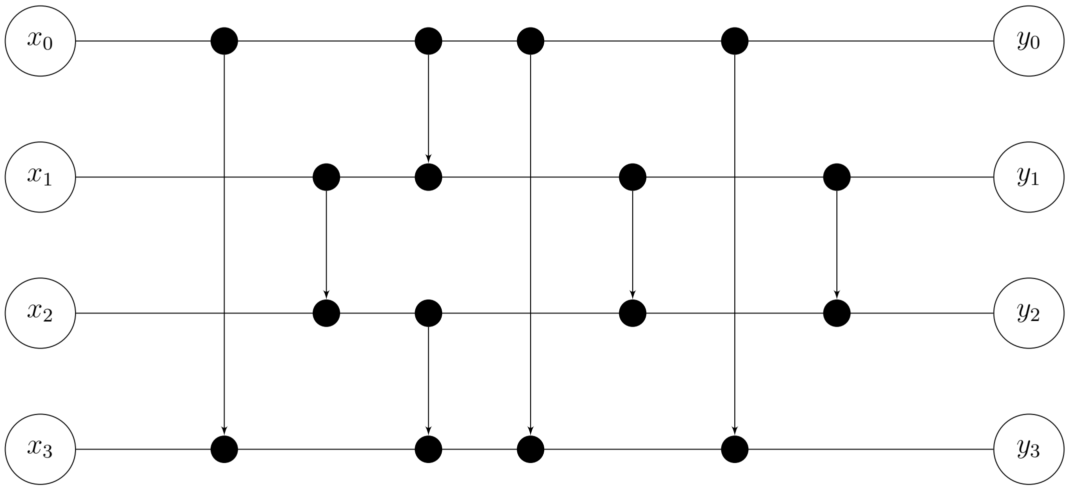


Figure 7.19: The Synthesized Sorting Network for Array Size 4 and Depth 4

8 Conclusion and Outlook

Automatic Synthesis of reactive systems from (temporal) logic specifications has ever been a dream in computer science. However, due to the high complexity of the problem, the synthesis problem was deemed to be infeasible in practice. Only recently, researches has been attracted to the problem again and progress has been made to make synthesis practical.

Inspired by [49] that investigated the minimization of tree automata in the context of synthesis, the first topic considered in this thesis was minimization of parity automata. We developed the theory of fair, reverse and direct simulation relations for parity automata and demonstrated their effectiveness in Chapter 7. The examples considered indicate that for co-Büchi automata, direct simulation minimization is as good as fair simulation minimization or as the previously published delayed simulation. For reverse simulation, we must conclude that their usefulness for controller synthesis is very limited. Although reverse simulation can be used to minimize the nondeterministic automaton, this minimization does not have an effect for the deterministic automaton that stays the same whether reverse simulation is enabled or not. Finally, we considered the fair simulation minimization of parity automata and demonstrated that it is a valuable tool to minimize parity automata. Although the time needed to perform fair simulation is much larger than the time needed to perform delayed simulation, fair simulation minimization is beneficial for larger examples where the overall runtime is dominated by the synthesis procedure and not the minimization procedure.

Inspired by some of the works targeted at an efficient determinization procedure, we presented in this thesis two different determinization constructions. The first one is based on the temporal logic hierarchy and has been demonstrated to be a valuable tool to obtain a deterministic automaton for the formulas of the hierarchy. We have shown how the subset and the breakpoint construction can be implemented efficiently using symbolic methods and based on that procedure, our tool was able to clearly outperform the explicitly implemented tool Lily, the first implementation of a synthesis procedure that could handle full LTL. Whenever a formula is not part of the temporal logic hierarchy, more powerful determinization constructions need to be employed. To this end, we developed in Chapter 5 a new determinization construction that makes use of the fact that the nondeterministic Büchi automata obtained by the 'standard' translation yields unambiguous automata. This determinization construction has also been implemented. However, unlike the determinization construction based on the temporal logic hierarchy, this determinization construction did not scale that well. As one problem we identified that boolean combinations of temporal logic formulas lead to an enormous blowup of the nondeterministic automata and thus makes the determinization construction infeasible. By breaking up boolean combinations, we were nevertheless able to synthesize controllers for most of the specifications that we considered and also in that case, the Lily tool was clearly outperformed by our tool.

In practice, however, we do not believe that the unambiguous determinization construction needs to be applied often. Most specifications commonly used consists of a large safety fragment that can be handled even by the simple subset construction. Nearly all other formulas occurring in practice belong to the class $\mathsf{TL}_{\mathsf{Streett}}$ so that the efficient breakpoint construction can be used. In that case, when the overall specification is made up of relatively small subformula, our approach scales rather well and we are able to synthesize controllers for industry-sized specifications which we have demonstrated with the AMBA case study.

This case study gave us also some interesting insights into some optimizations that are from a theoretical point of view less interesting, but are necessary to put controller synthesis forward to a valuable tool. Using the optimizations described in Chapter 6, we were even able to outperform the tool Anzu [12] that uses the much simpler algorithm from [79] for the solution of games. This advantage of our algorithm has to do with two optimizations on the generalized parity algorithm that we explored in Chapter 6: once the ability to solve safety games separately and secondly a re-coloring of (sub-) games that do not contain all colors .

The drawback of the tool Anzu is that the specifications that can be automatically handled are very limited and every other specification must be manually brought into the desired form. The approach of [79] assumes as input two sets of Büchi conditions Φ and Ψ that represent assumptions and guarantees, respectively. Thus, it is possible to combine our approach of generating deterministic (co)-Büchi automata by the breakpoint construction with the approach of [79] to obtain more efficient synthesis procedures.

This thesis shows the broad range of research options in LTL controller synthesis. First there is a need in better determinization constructions. In particular, a determinization construction that can be minimized on-the-fly would be rather useful for the general case. Finally, new application domains like fault detection and automatic repair of systems bear many interesting topics for future work. As other ways of future work, we already mentioned the solution of infinite games that are the backbone of the controller synthesis constructions. We expect that future research in the field of game solving will lead to large improvements. In combination with the work done in this thesis somewhere in the future controller synthesis may become a valuable tool accompanying the well-established model checking.

Bibliography

[1] M. Ajtai, J. Komlos, and E. Szemeredi. An $o(n \log(n))$ sorting network. In *Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1983.

[2] C. Allauzen and M. Mohri. An efficient pre-determinization algorithm. In O.H. Ibarra and Z. Dang, editors, *Conference on Implementation and Application of Automata (CIAA)*, volume 2759 of *LNCS*, pages 83–95, Santa Barbara, California, USA, 2003. Springer.

[3] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Transactions on Computational Logic (TOCL)*, 5(1):1–15, 2004.

[4] C. André. SyncCharts: A visual representation of reactive behaviors. Research Report tr95-52, University of Nice, Sophia Antipolis, France, 1995.

[5] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M.Y. Vardi. Efficient LTL compilation for SAT-based model checking. In *International Conference on Computer-Aided Design (ICCAD)*, pages 877–884, San Jose, California, USA, 2005. ACM/IEEE Computer Society.

[6] Michael Bauland, Martin Mundhenk, Thomas Schneider, Henning Schnoor, Ilka Schnoor, and Heribert Vollmer. The tractability of model-checking for ltl: The good, the bad, and the ugly fragments. *Electron. Notes Theor. Comput. Sci.*, 231:277–292, 2009.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] J. Bernet, D. Janin, and I. Walukiewicz. Permissive strategies: from parity games to safety games. *RAIRO Theoretical Informatics and Applications*, 36:251–275, 2002.

[9] G. Berry. The Esterel v5 language primer. `http://www-sop.inria.fr/esterel.org/`, July 2000.

[10] R. Bloem, A. Cimatti, I. Pill, M. Roveri, and S. Semprini. Symbolic implementation of alternating automata. In O.H. Ibarra and H.-C. Yen, editors, *Conference on Implementation and Application of Automata (CIAA)*, volume 4094 of *LNCS*, pages 208–218, Taipei, Taiwan, 2006. Springer.

[11] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In R. Lauwereins and J. Madsen, editors, *Design, Automation and Test in Europe (DATE)*, pages 1188–1193, Nice, France, 2007. IEEE Computer Society.

[12] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190:3–16, 2007.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science (LICS)*, pages 1–33, Washington, DC, USA, 1990. IEEE Computer Society.

[14] J.R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6(1):66–92, 1960.

[15] J.R. Büchi. Weak second order arithmetic and finite automata. *Z. Math. Logik Grundlagen Math.*, 6:66–92, 1960.

[16] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[17] O. Carton and M. Michel. Unambiguous Büchi automata. *Theoretical Computer Science (TCS)*, 297(1-3):37–81, 2003.

[18] E.Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In W. Kuich, editor, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 623 of *LNCS*, pages 474–486, Vienna, Austria, 1992. Springer.

[19] K. Chatterjee, T.A. Henzinger, and N. Piterman. Generalized parity games. In H. Seidl, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 4423 of *LNCS*, pages 153–167, Braga, Portugal, 2007. Springer.

[20] A. Church. Logic, arithmetic and automata. In *International Cong. Math*, pages 23–35, Stockholm, Sweden, 1962.

[21] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design (FMSD)*, 10(1):47–71, February 1997.

[22] L. de Alfaro, T.A. Henzinger, and R. Majumdar. From verification to control: Dynamic programs for omega-regular objectives. In *Logic in Computer Science (LICS)*, pages 279–290, Boston, Massachusetts, USA, 2001. IEEE Computer Society.

[23] D.L. Dill, A.J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation preorders. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification (CAV)*, volume 575 of *LNCS*, pages 255–265, Aalborg, Denmark, 1992. Springer.

[24] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier, 1990.

[25] E.A. Emerson and E.M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[26] E.A. Emerson and C.S. Jutla. Tree automata, μ-calculus and determinacy. In *Foundations of Computer Science (FOCS)*, pages 368–377, San Juan, Puerto Rico, 1991.

[27] E.A. Emerson and A.P. Sistla. Deciding branching time logic. In *Symposium on Theory of Computing (STOC)*, pages 14–24, 1984.

[28] K. Etessami, T. Wilke, and R.A. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM Journal on Computing (SICOMP)*, 34(5):1159–1175, 2005.

[29] K. Fisler and S.D. Johnson. Integrating design and verification environments through a logic supporting hardware diagrams. In *Computer Hardware Description Languages and Their Applications (CHDL)*, pages 669–674. IEEE Computer Society, 1995. CHDL proceedings pp. 493-696 of the "ACV'95" held August 29 to September 1, 1995, Chiba, Japan.

[30] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In O.H. Ibarra and Z. Dang, editors, *Conference on Implementation and Application of Automata (CIAA)*,

volume 2759 of *LNCS*, pages 35–48, Santa Barbara, California, USA, 2003. Springer.

[31] C. Fritz. Concepts of automata construction from LTL. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3835 of *LNCS*, pages 728–742, Montego Bay, Jamaica, 2005. Springer.

[32] C. Fritz. *Simulation-Based Simplification of omega-Automata*. PhD thesis, Technischen Fakultät der Christian-Albrechts-Universität zu Kiel, Germany, 2005.

[33] C. Fritz and T. Wilke. State space reductions for alternating Büchi automata. In M. Agrawal and A. Seth, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2556 of *LNCS*, pages 157–168, Kanpur, India, 2002. Springer.

[34] C. Fritz and T. Wilke. Simulation relations for alternating parity automata and parity games. In O.H. Ibarra and Z. Dang, editors, *Developments in Language Theory (DLT)*, volume 4036 of *LNCS*, pages 59–70. Springer, 2006.

[35] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 53–65, Paris, France, 2001. Springer.

[36] Embedded Systems Group. The averest toolset. Website. Available online at `http://www.averest.org`;.

[37] E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*. Springer, Dagstuhl, Germany, 2002.

[38] D.P. Guelev. A syntactical proof of the canonical reactivity form for past linear temporal logic. *Journal of Logic and Computation*, 18(9):615–623, 2008.

[39] Y.S. Gurevich and L. Harrington. Trees, automata, and games. In *Symposium on Theory of Computing (STOC)*, pages 60–65, San Francisco, California, USA, 1982.

[40] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In E. Brinksma and K.G. Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 610–623, Copenhagen, Denmark, 2002. Springer.

[41] S. Gurumurthy, O. Kupferman, F. Somenzi, and M.Y. Vardi. On complementing nondeterministic Büchi automata. In D. Geist and E. Tronci, editors, *Correct*

Hardware Design and Verification Methods (CHARME), volume 2860 of *LNCS*, pages 96–110, L'Aquila, Italy, 2003. Springer.

[42] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[44] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[45] T.A. Henzinger, O. Kupferman, and S.K. Rajamani. Fair simulation. In A. Mazurkiewicz and J. Winkowski, editors, *Concurrency Theory (CONCUR)*, volume 1243 of *LNCS*, pages 273–287, Warsaw, Poland, 1997. Springer.

[46] T.A. Henzinger and N. Piterman. Solving games without determinization. In Z. Ésik, editor, *Computer Science Logic (CSL)*, volume 4207 of *LNCS*, pages 395–410, Szeged, Hungary, 2006. Springer.

[47] A.J. Hu and D.L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation Conference (DAC)*, pages 266–271, Dallas, Texas, USA, 1993. ACM.

[48] B. Jobstmann. *Applications and Optimizations for LTL Synthesis.* PhD thesis, IST - Institute for Software Technology, TU Graz, Graz, Austria, February 2007.

[49] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In A. Gupta and P. Manolios, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 117–124, San Jose, California, USA, 2006. IEEE Computer Society.

[50] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 258–262, Berlin, Germany, 2007. Springer.

[51] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S.K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 226–238, Edinburgh, UK, 2005. Springer.

[52] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences (JCSS)*, 2009.

[53] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *LNCS*, pages 1–16, Aalborg, Denmark, 1998. Springer.

[54] J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science (TCS)*, 363(2):182–195, October 2006. http://dx.doi.org/10.1016/j.tcs.2006.07.022.

[55] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1998.

[56] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science (TCS)*, 27(3):333–354, December 1983.

[57] O. Kupferman. Avoiding determinization. In *Logic in Computer Science (LICS)*, pages 243–254, Seattle, Washington, USA, 2006. IEEE Computer Society.

[58] O. Kupferman, N. Piterman, and M.Y. Vardi. Safraless compositional synthesis. In T. Ball and R.B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 31–44, Seattle, Washington, USA, 2006. Springer.

[59] O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Logic in Computer Science (LICS)*, pages 81–92, Indianapolis, Indiana, USA, 1998. IEEE Computer Society.

[60] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science (FOCS)*, pages 531–540. IEEE Computer Society, 2005.

[61] L.H. Landweber. Decision problems for ω-automata. *Mathematical Systems Theory*, 3(4):376–384, 1969.

[62] ARM Ltd. Amba specification (rev. 2). Website, 1999. Available online at `http://www.arm.com`;.

[63] M. Maidl. The common fragment of CTL and LTL. In *Foundations of Computer Science (FOCS)*, pages 643–652, 2000.

[64] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *LNCS*, pages 124–164, Altrincham, UK, 1989. Springer.

[65] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Principles of Distributed Computing (PODC)*, pages 377–408, Quebec City, Quebec, Canada, 1990. ACM.

[66] Z. Manna and A. Pnueli. *The temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[67] R. McNaughton and S. Papert. *Counter-free Automata*. MIT Press, 1971.

[68] S. Merz and A. Sezgin. Emptiness of linear weak alternating automata. Technical report, LORIA, 2003.

[69] R. Milner. An algebraic definition of simulation between programs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 481–489, 1971.

[70] S. Miyano and T. Hayashi. Alternating automata on ω-words. *Theoretical Computer Science (TCS)*, 32:321–330, 1984.

[71] A. Morgenstern and K. Schneider. From LTL to symbolically represented deterministic automata. In F. Logozzo, D.A. Peled, and L.D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4905 of *LNCS*, pages 279–293, San Francisco, California, USA, 2008. Springer.

[72] A. Morgenstern, K. Schneider, and S. Lamberti. Generating deterministic ω-automata for most LTL formulas by the breakpoint construction. In C. Scholl and S. Disch, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 119–128, Freiburg, Germany, 2008. Shaker.

[73] A.W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, editor, *Computation Theory*, volume 208 of *LNCS*, pages 157–168. Springer, 1984.

[74] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree, and its complexity. In L. Kott, editor, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 226 of *LNCS*, pages 275–283, Rennes, France, 1986. Springer.

[75] D.E. Muller and P.E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems by Rabin, McNaughton, and Safra. *Theoretical Computer Science (TCS)*, 141(1-2):69–108, 1995.

[76] I. Parberry. A computer assisted optimal depth lower bound for sorting networks with nine inputs. In *Supercomputing*, pages 152–161, Reno, Nevada, USA, 1989. ACM. http://doi.acm.org/10.1145/76263.76280.

[77] R. Pelánek and J. Strejcek. Deeper connections between LTL and alternating automata. In J. Farré, I. Litovsky, and S. Schmitz, editors, *Conference on Implementation and Application of Automata (CIAA)*, volume 3845 of *LNCS*, pages 238–249, Sophia Antipolis, France, 2006. Springer.

[78] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *Logic in Computer Science (LICS)*, pages 255–264, Seattle, Washington, USA, 2006. IEEE Computer Society.

[79] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E.A. Emerson and K.S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, pages 364–380, Charleston, South Carolina, USA, 2006. Springer.

[80] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57, Providence, Rhode Island, USA, 1977. IEEE Computer Society.

[81] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, Austin, Texas, USA, 1989. ACM.

[82] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1025–1029, 1969.

[83] M.O. Rabin. Automata on infinite objects and Church's problem. In *Regional Conference Series in Mathematics*, volume 13. American Mathematical Society, 1972.

[84] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:115–125, 1959.

[85] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization (SICON)*, 25(1):206–230, 1987.

[86] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, The Weizmann Institute of Science, Israel, Rehovot, Israel, 1992.

[87] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In D. Bosnacki and S. Edelkamp, editors, *Model Checking Software (SPIN)*, volume 4595 of *LNCS*, pages 149–167, Berlin, Germany, 2007. Springer.

[88] S. Safra. On the complexity of ω-automata. In *Foundations of Computer Science (FOCS)*, pages 319–327, 1988.

[89] S. Schewe. Solving parity games in big steps. In V. Arvind and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4855 of *LNCS*, pages 449–460, New Delhi, India, 2007. Springer.

[90] K. Schneider. Improving automata generation for linear temporal logic by considering the automata hierarchy. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2250 of *LNAI*, pages 39–54, Havana, Cuba, 2001. Springer.

[91] K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[92] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[93] K. Schneider and T. Kropf. The C@S system: Combining proof strategies for system verification. In T. Kropf, editor, *Formal Hardware Verification - Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 248–329. Springer, state of the art report edition, August 1997.

[94] K. Schneider and G. Logothetis. Abstraction of systems with counters for symbolic model checking. In M. Mutz and N. Lange, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 31–40, Braunschweig, Germany, 1999. Shaker.

[95] R. Sebastiani and S. Tonetta. More deterministic vs. smaller Büchi automata for efficient LTL model checking. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *LNCS*, pages 126–140, L'Aquila, Italy, 2003. Springer.

[96] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, July 1985.

[97] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for LTL games. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 77–84, Austin, Texas, USA, 2009. IEEE Computer Society.

[98] S. Sohail, F. Somenzi, and K. Ravi. A hybrid algorithm for LTL games. In F. Logozzo, D.A. Peled, and L.D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4905 of *LNCS*, pages 309–323, San Francisco, California, USA, 2008. Springer.

[99] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 248–263, Chicago, Illinois, USA, 2000. Springer.

[100] L. Staiger and K.W. Wagner. Automatentheoretische Charakterisierungen topologischer Klassen regulärer Folgenmengen. *Elektronische Informationsverarbeitung und Kybernetik*, 10:379–392, 1974.

[101] R.S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1-2):121–141, 1982.

[102] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 4, pages 133–191. Elsevier, 1990.

[103] W. Thomas. Infinite trees and automaton definable relations over ω-words. In C.Choffrut and T.Lengauer, editors, *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 415 of *LNCS*, pages 263–277. Springer, 1990.

[104] W. Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 900 of *LNCS*, pages 1–13, Munich, Germany, 1995. Springer.

[105] T. Tuerk and K. Schneider. Relationship between alternating omega-automata and symbolically represented nondeterministic omega-automata. Internal Report 340, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, November 2005.

[106] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency - Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

[107] M.Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 1–22, Genoa, Italy, 2001. Springer.

[108] J. Vöge and M. Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 202–215, Chicago, Illinois, USA, 2000. Springer.

[109] K. Wagner. On ω-regular sets. *Information and Control*, 43(2):123–177, 1979.

[110] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In O.H. Ibarra and Z. Dang, editors, *Conference on Implementation and Application of Automata (CIAA)*, volume 2759 of *LNCS*, pages 11–22, Santa Barbara, California, USA, 2003. Springer.

[111] B.W. Watson. Directly constructing minimal DFAs: Combining two algorithms by Brzozowski. In S. Yu and A. Păun, editors, *Conference on Implementation and Application of Automata (CIAA)*, volume 2088 of *LNCS*, pages 311–317, London, Ontario, Canada, 2001. Springer.

[112] P.L. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* PhD thesis, University of California, Palo Alto, California, USA, 1982.

[113] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Aït Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 219–231, Vancouver, British Columbia, Canada, 1998. Springer.

[114] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs. In M.K. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*, pages 233–247, Palo Alto, California, USA, 1996. Springer.