Dissertation

Applications and Optimizations for LTL Synthesis

Barbara Jobstmann

26. Februar 2007

durchgeführt am

IST - Institut für Softwaretechnologie
Technische Universität Graz

Betreuer:	Univ.Ass. Roderick Bloem, Ph.D.
1. Begutachter:	Univ.Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
2. Begutachter:	Prof. Orna Kupferman, Ph.D.

Abstract

LTL synthesis is the process of generating a reactive finite-state system from a formal specification written in Linear Temporal Logic (LTL). The idea of synthesis is to automatically construct a functional correct system from a behavioral description of the system. Even though the idea is nearly fifty years old and the underlying theory is well established, is has not be adapted to practice yet. The main reasons are the high complexity of the problem and intrinsic algorithms.

This thesis follows along three different paths to bring LTL synthesis closer to practice. First, we present our synthesis tool Lily, a Linear Logic Synthesizer. Lily is the first implementation of a synthesis tool for full LTL. It is based on a recent approach to synthesis by Kupferman and Vardi that goes through a variety of alternating tree automata. Lily implements this approach and a set of optimizations necessary to make the approach applicable. Lily can only handle small examples, but since it imposes no restriction on the syntax of the LTL formula, it is easy to use and can help users to get familiar with LTL.

In the second part, we state the repair problem and solve it for finite-state systems using synthesis techniques. In the repair problem, given a specification and faulty system, we search for slightly modifications of the system that make it correct with respect to the specification. We consider the repair problem as a game, which consists of a modified version of the faulty system and an automaton representing the specification. Every winning strategy of the game corresponds to a repair. However, a strategy with memory results in a repair that adds variables to the program, which we argue is undesirable. In order to avoid extra variables, we need a memoryless strategy. We show that the problem of finding a memoryless strategy is NP-complete and present a heuristic to compute such strategies. We have implemented the approach symbolically and present initial experimental results to show its usefulness.

Finally, in the third part of the thesis we focus on the main application of synthesis:

constructing a correct system from formal specification. Due to a new algorithm by Piterman et al. synthesis has recently made a huge process in terms of efficiency. We extend this algorithm to generate compact circuits and we show how to synthesize a generalized buffer and an arbiter for ARM's AMBA AHB bus from specifications given in LTL. These examples constitute the first industrial designs that have been synthesized automatically from their specifications.

Zusammenfassung

LTL Synthese beschäftigt sich mit dem automatischen Erzeugen von reaktiven Systemen ausgehend von einer formalen Spezifikation. Die Spezifikation beschreibt neben den Eingangs- und Ausgangssignalen, das gewünschte zeitliche Verhalten des Systems mit Hilfe von Formeln in LTL (Linear Temporal Logic). Spezifikationen dieser Art werden schon seit einiger Zeit sehr erfolgreich zum formalen Verifizieren von System (mittels Modelchecking) verwendet. Im Gegensatz zur Verifikation, in der Systeme auf ihre Konsistenz mit der Spezifikation überprüft werden, geht es in der Synthese darum, das eigentliche Systeme zu generieren. Obwohl die Idee schon über 50 Jahre alt ist, und die dahinterliegende Theorie gut ausgebaut ist, hat LTL Synthese den Sprung in die Praxis noch nicht geschafft. Gründe dafür sind die hohe Komplexität des Problems, sowieso äußerst komplexe Algorithmen, die einen praktischen Einsatz bis jetzt verhindert haben.

Das Ziel der vorliegenden Arbeit ist es, Synthese trotz dieser Schwierigkeiten einen Schritt näher zur Praxis zu bringen. Um dieses Ziel zu erreichen, schlagen wir drei recht unterschiedliche Wege ein. Zuerst präsentieren wir unser Syntheseprogramm Lily, einen Linear Logic Synthesizer. Lily ist das erste Syntheseprogramm, das LTL Spezifikationen in seinem vollen Umfang synthetisieren kann und sich nicht auf eine Teilmenge der Logik konzentriert. Lily baut auf einen von Kupferman und Vardi kürzlich entwickelten Synthesealgorithmus auf, der ausgehend von einer LTL Formel mehrere verschiedene alternierende Baumautomaten konstruiert. Lily implementiert diesen Algorithmus und eine Menge von Optimierungen, die für eine moderate Laufzeit unabdinglich sind. Die Spezifikationen, die Lily synthetisieren kann, sind relativ klein. Da jedoch weder die Struktur noch der Typ der Formeln eingeschränkt wird, ist Lily sehr einfach zu verwenden und gut geeignet um den Umgang mit LTL zu erlernen.

Im zweiten Teil, beschäftigen wir uns mit dem Reparieren von fehlerhaften Systemen mit endlichen Zustandsräumen. Ein System ist fehlerhaft, wenn es seine Spezifikation

verletzt. Das Reparaturproblem ist die Suche nach kleinen Veränderungen des Systems, sodass das modifizierte System die Spezifikation erfüllt. Wir reduzieren das Reparaturproblem auf ein vereinfachtes Syntheseproblem, indem wir Teile das Systems entfernen und dann diese Teile neu synthetisieren. Dazu fassen wir das Problem als ein Spiel auf, das aus dem reduzierten System und einem Automaten, der die Spezifikation repräsentiert, besteht. Jede gewinnende Strategie für dieses Spiel entspricht einer gültigen Reparatur. Reparaturen, die aus Strategien mit Gedächtnis erzeugt werden, erweitern den Zustandsraum des ursprünglichen System und verändern dieses dadurch wesentlich, was wir als nicht wünschenswert erachten. Deshalb bauen unsere Reparaturen auf Strategien ohne Gedächtnis auf, wodurch der Zustandsraum unverändert bleibt. Da gedächtnislose Strategien nicht immer existieren oder schwer zu finden sind, präsentieren wir Heuristiken zum Finden dieser Strategien. Mittels einer symbolischen Implementierung und einfachen Beispielen zeigen wir die Anwendbarkeit unserer Ansatzes.

Der dritte und letzte Teil dieser Arbeit konzentriert sich wieder auf das eigentliche Syntheseproblem: das Erzeugen eines korrekten Systems ausgehend von einer formalen Spezifikation. Im Gegensatz zum ersten Teil, der sich auf die Algorithmik konzentriert, liegt der Fokus hier in der Anwendung. Durch einen neuartigen Algorithmus von Piterman u.a. für eine Teilmenge der Logik LTL, ist Synthese viel effizienter geworden. Aufbauend auf diesen Ansatz erzeugen wir sequentielle Schaltungen und zeigen wie man einen Buffer und einen Arbiter für ARMs AMBA AHB Bus von den jeweiligen LTL Spezifikationen synthetisieren kann. Dies sind die ersten industrienahen Designs, die je von einer formalen Spezifikation synthetisiert wurden.

Acknowledgments

I would like to express my deep gratitude to my supervisor Roderick Bloem for providing support whenever I needed it. Roderick always had time to answer one of my endless questions and was supportive when I was confronted with a problem. Many fruitful discussions and his enthusiasm for new ideas made it a pleasure to work with him. I could not have wished for a better supervisor.

I am indebted to Prof. Franz Wotawa, head of the Institute of Software Technology, for building up a large and active group. I had a great work environment with an excellent working atmosphere and many opportunities for development.

Special thanks go to my colleagues and friends Andreas Griesmayer, Ingo Pill, and Stefan Staber. We shared an office and did our studies together, which positively influenced my work and made my life as a graduate student great.

I would like to thank Martin Weiglhofer, Stefan Galler, and Karin Greiml for their interest in the synthesis topic and their work on the tool Anzu.

Furthermore, I would like to thank Prof. Orna Kupferman and Prof. Amir Pnueli for patient explanations of their work and many comments regarding my work. Their expertise deeply impressed me, and if anyone would ask about my role models, Orna would definitely be one of them.

Finally, I would like to thank my family and my boyfriend Mathias Mayrhofer for supporting me in every situation of my life. Mathias always had a warm shoulder to lean against and his patient encouragements eased up many stressful days.

Thanks a million,

Barbara Jobstmann

Acknowledgments

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis focuses on specification-based synthesis, which aims to automatically construct a reactive system from a formal specification. The specification is given in *linear temporal logic (LTL)* and describes logical properties of the system. For simplicity, we will refer to this form of high-level synthesis as *LTL synthesis* or simply *synthesis*, but emphasize that it should not be confused with the synthesis of a gate-level description from VERILOG, RTL code, or from a high-level behavioral description. This chapter starts with a motivation followed by the history of synthesis and the structure of the document. Finally, we summarize our contributions.

1.1 Motivation

Automatically constructing a reactive system from a logical specification has been an ambitious dream in computer science for about half a century now [Chu62]. The obvious benefit is that we only have to give a list of desired behaviors and a synthesis tool comes up with a state model that takes all demanded properties into account. For systems that have no further constraints (e.g., on timing or space consumption) a synthesis tool would completely avoid hand-coding. A less ambitious benefit is the possibility to construct rapid prototypes from specification. These functional prototypes could be used for early test integration and would allow to "simulate the specification". Most developers rely on simulation to check if the constructed system meets their intents. In the hardware community, people claim that formal verification can only be integrated in the mainstream design flow, if it looks and feels like simulation [Bau06]. A synthesis tool could provide a simulation-like feeling for a formal specification, which would help people in understand-

ing and writing a formal specification (since faults in the specification become apparent immediately). Synthesis is an extremely good way to validate and debug a specification, something that will gain importance as formal specification starts to be used as the basis for a manual implementation.

Although a synthesis tool has many benefits and synthesis is solved in theory (see Section 1.2), the idea did not get accepted in practice due to three major reasons. The first reason is that synthesis of LTL properties is 2EXPTIME-complete [Ros92]. The second is that the solution to LTL synthesis [PR89] uses an intricate determinization construction [Saf88] that is hard to implement and very hard to optimize. Thirdly, the solution to synthesis is not compositional and therefore does not reflect the usually iterative process of writing a complete specification. The first reason should not prevent one from implementing the approach. After all, the bound is a lower bound as shown in [Ros92], so there are specifications for which the smallest correct system is doubly exponentially larger than the specification. Thus, the worst case complexity of verifying the specification on a manual implementation is also 2EXPTIME in terms of the (full) specification. In combination with the second reason, however, the argument gains strength. For many specifications, a doubly-exponential blow up is not necessary, but can only be avoided through careful use of optimization techniques, which is hard to achieve in combination with Safra's algorithm. Recently developed algorithms by Kupferman et al. [KV05, KPV06] and Piterman et al. [PPS06] follow along completely different paths and give new hope for synthesis.

Motivated by those developments, the research in this thesis focuses on techniques to bring LTL synthesis from theory closer to practice. On the one hand, we concentrate on those state-of-the-art algorithms to get closer to automatic system construction, which we see as the long-term goal of synthesis. On the other hand, we search for ways to make synthesis useful in short-term by providing techniques to support the current design or verification process.

Currently, in a formal verification process, model checking proves whether a system adheres to its specifications. If not, the user is typically presented with a counterexample showing an execution of the system that violates the specification. The user needs to find and correct the fault in the system, which is a nontrivial task.

The problem of locating a fault in a misbehaving system has been the attention of recent research [SW96, JRS02, BNR03, GV03, Gro04]. Given a suspicion of the fault location, it may still not be easy to repair the system. There may be multiple suggestions, only one of which is the actual fault and knowing the fault is not the same as knowing a fix. The

repair problem goes one step beyond fault localization. Given a set of *suspect components*, it looks for a modification of the system that satisfies its specifications. It can be used to find the actual fault among the suggestions of a fault localization tool and a correction, while avoiding the tedious debugging that would normally ensue.

The repair problem is closely related to the synthesis problem. Essentially, solving the repair problem is like synthesizing only a small part of a system. In order to automatically synthesize a system, a complete specification is needed. For the repair problem, we need as much of the specification as is necessary to decide the correct repair, just as for model checking we do not need a full specification to detect a fault. (This has the obvious drawback that an automatic repair may violate an unstated property and needs to be reviewed by a designer.) A further benefit is that the modification is limited to a small portion of the design. The structure and logic of the system are left untouched, which makes it amenable to further modification by the user. Due to the restricted modifications we can apply heuristics. So the repair problem can be seen as the light version of the synthesis problem. Both problems allow for a lot of interesting research and are subject of this thesis.

1.2 History of Synthesis

Synthesis aims to transform a specification into a system that is guaranteed to satisfy the specification. The theory behind synthesis of reactive systems is well established and goes back to Church, who stated the *Synthesis Problem* [Chu62] using different fragments of *restricted recursive arithmetic* (S1S) as specification. Nowadays this problem is also know as *Realizability or Church's Problem* and defined as follows: Given a relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$ defined by restricted recursive arithmetic or another logic (e.g., LTL), we search for a function $f : (2^I)^* \rightarrow 2^O$ that generates for all sequences $x = x_0, x_1, x_2, ... \in (2^I)^\omega$ a sequence $y = y_0, y_1, y_2, ... \in (2^O)^\omega$ with $y_i = f(x_0, x_1, ..., x_{i-1})$ for $i > 0$ such that $R(x, y)$ holds. We can view I and O as the sets of input and output signals of a reactive system. The relation R can be seen as a linear specification including all pairs of input and output sequences that define the correct behavior of the system. The function f (called *strategy*) then maps every possible input sequence to a correct output sequence, and represents a correct system.

In the following years, Büchi and Landweber [BL69] and Rabin [Rab72] presented independent solutions to Church's problem. The first is based on infinite game theory, where

the latter uses tree automata. It took nearly ten years until researchers discovered the similarities between games and tree automaton [GH82].

Specifying the behavior of reactive systems in S1S is cumbersome. So there was an urgent need for new specification languages. A very successful proposal is that of *temporal logic* [Pnu77, EC82], which is now widely used in the formal verification community and provides the basis for commercial specifications languages as *PSL (Property Specification Language)* or *SVA (System Verilog Assertions)*.

Essentially, temporal logic comes in two flavors: *linear* or *branching time*. *Computation Tree Logic (CTL)* [EC82] is a branching time logic. Given a CTL formula and a design, we can check efficiently if the design fulfills the formula. However, the restricted syntax of CTL limits the expressive power and makes writing specifications in CTL rather complicated. Specifying is easier in *Linear Temporal Logic (LTL)* [Pnu77, MP91], which is also more suitable for compositional reasoning [KVW00, Var01]. In theory, the model checking problem for LTL is PSPACE-complete [SC85], but in practice (cf. [EH00, SB00]), it takes exponential time and space in the size of the LTL formula assuming P $\neq$ PSPACE .

The introduction of temporal logic gave rise to further developments in the area of synthesis of reactive systems. Emerson and Clarke [EC82] and Manna and Wolper [MW84] considered the problem for temporal specifications given in CTL and LTL, respectively. Both concluded that if a specification φ is satisfiable, we can construct a system that adheres to the specification using the model that satisfies φ. Due to the reduction to satisfiability the approaches are limited to constructing *closed systems*, which lead to systems that are only guaranteed to work correctly in cooperative environments (environments that help to satisfy φ).

In the late 80s, Pnueli and Rosner [PR89] reconsidered the topic for LTL and provided a solution for constructing *open systems* (reactive systems). The key observation (also observed in [Rab72]) is that even though the specification can be represented as infinite sequences (words) over the input and output signals, the solution to the synthesis problem is an infinite tree. Furthermore, Rosner proved that synthesis of LTL properties is 2EXPTIME-complete [Ros92]. The first exponent derives from the translation of the LTL formula into a non-deterministic Büchi automaton. The second exponent is due to the determinization of the automaton. In theory LTL synthesis was classified as solved but it was said to be hopelessly intractable and so the topic was dropped for a decade.

At the same time, Ramadge and Wonham [RW89] introduced the problem of *controller synthesis*, which deals with constructing a *controller* for a *plant*. They considered speci-

fication of the form $\mathsf{G}\,p$, and proved that the controller can be synthesized in linear time for such specifications.

In order to overcome the complexity issues of LTL synthesis people started to concentrate on subsets of the language. E.g., Maidl [Mai00] and Alur and La Torre [AL01] identified subsets of LTL for which deterministic automaton of less than doubly exponential size can be constructed. Wallmeier et al. [WHT03] provided an efficient symbolic algorithm to synthesize *request-response* specification, which are of the form $\mathsf{G}(p_i \rightarrow \mathsf{F}\,q_i)$ for $i \in \{0..n\}$. Harding et al. [HRS05] observed that by leaving out the determinization step, they obtain an efficient but incomplete symbolic algorithm. In [JGB05], we made the same observations independently in the context of using LTL synthesis to repair finite state systems (see Chapter 4).

Recently, Pnueli, Piterman, and Sa'ar [PPS06] proposed a efficient symbolic algorithm to automatically synthesize designs from LTL formulas belonging to the class of *generalized reactivity* of rank 1 (GR(1)), in time N^3 where N is the size of the state space of the design. The class GR(1) covers the vast majority of properties using in practice. GR(1) formulas have the form $(\bigwedge_{i\in\{0\cdots n\}} \mathsf{G}\,\mathsf{F}(E_i)) \rightarrow (\bigwedge_{j\in\{0\cdots m\}} \mathsf{G}\,\mathsf{F}(S_j))$, where E_i and S_j are Boolean formulas over atomic propositions representing the signals of the system. We can transform most specifications into that form using deterministic monitors (see Section 5.1.1 for a detailed explanation). Following this approach for an LTL subset, real-life designs can be synthesized as we show in Chapter 5.

Work concentrating on full LTL is sparse and has not be pursued for a long time. A major issue in Rosner's solution to LTL synthesis is the construction of a deterministic automaton for the given formula, which includes constructing a nondeterministic Büchi automaton and determinizing it. The first translation for LTL to nondeterministic Büchi automaton has be proposed by [WVS83], and since then translators from LTL to automaton have improved a lot (cf. [DGV99, SB00, GO01]). Unfortunately, constructions to determinize (arbitrary) Büchi automaton did not make such progress. In 1988, Safra [Saf88] was the first to provide a determinization construction that was asymptotical optimal. The lower bound to determinization [Löd99] was extended from the lower bound of [Mic88] to complementation of nondeterministic Büchi automata. ([KV98] shows a doubly-exponential lower bound when one starts from LTL.) Later, Muller and Schupp [MS95] and Klarlund [Kla91] provide different constructions that also match the lower bound. These algorithms turned out to be quite resistant to efficient implementations [TBK95, KB05, ATW06]. So their implementations can determinize automaton with

approximately ten states and Rosner's solution to the Synthesis Problem was never implemented.

In 2005, Kupferman and Vardi [KV05] proposed an alternative solution that goes through universal co-Büchi and weak alternating tree automata. In contrast to previous approaches it avoids Safra's determinization constructions. We developed optimizations for the *Safraless approach* and built the first implementation of a synthesis algorithm for full LTL (see Chapter 3). In [KPV06], Kupferman et al. proposed a compositional version of the Safraless approach, which is based on generalized universal co-Büchi automata.

Recently, Piterman proposed improvements to Safra's construction that lead to better complexity bounds and deterministic Parity Automata [Pit06]. In [HP06], Henzinger and Piterman introduced nondeterministic automata that are *good for games (GFG)*. Those automata fair simulate their deterministic equivalent and can be used to solve the synthesis problem. Henzinger and Piterman provide a simple algorithm to construct GFG automata from nondeterministic Büchi automata. The construction is a replacement for a determinization construction and so it has the same worst case complexity. However, since it can be implemented symbolically, it is expected to perform better in practice.

1.3 Structure of the Thesis

This thesis summarized our research in the area of LTL synthesis in line with the European Union project *PROSYD (Property-Based System Design)*. The thesis is based on several papers and technical reports we published in the last three years. The general motivation presented in Section 1.1 has appeared in part in [Job06, AMJB05]. Section 1.2 and 1.4 summarizes the related work and the contributions presented in [JGB05], [JB06b], and [BGJ$^+$07a]. Chapter 2 introduces the necessary concepts to understand this work. Beside the introduction and the preliminary chapter the thesis consists of three main parts: Chapter 3 summarizes our work on **optimizations for LTL synthesis**, which was presented in [JB06a] and [JB06b]. Along with this research, we developed the tool Lily [BJ06], a linear logic synthesizer, which constitutes the first implementation of a synthesis tool for full LTL. Chapter 4 and Chapter 5 present two applications to LTL synthesis. In Chapter 4, we show how to use synthesis to **repair finite-state systems**. This chapter is based on [JGB05], [BJP05], and [JSGB07]. Chapter 5 summarizes our effort to **synthesize real-life designs** from a formal specification. This work was first presented in [BGJ$^+$06] and will be published in [BGJ$^+$07a] and [BGJ$^+$07b]. In Chapter 6, we conclude

and given an outlook to future work.

1.4 Contributions

This section reflects the structure of the thesis and is split into three parts. First we summarize our contribution in the area of synthesis for full LTL, then we discuss our work on repairing finite-state systems, and finally, we report our achievements in synthesizing circuits from formal specifications.

Optimizations for LTL Synthesis

Our first contribution is based on the Safraless approach of Kupferman and Vardi [KV05]. Starting from a specification φ, they generate, through the nondeterministic Büchi word automaton for $\neg\varphi$, a universal co-Büchi tree automaton (see Section 2.2) that accepts all trees satisfying φ. From this automaton they construct an alternating weak tree automaton accepting at least one (regular) tree satisfying φ (or none, if φ is not realizable). Finally, the alternating automaton is converted to a nondeterministic Büchi tree automaton with the same language. A witness for the nonemptiness of this automaton is an implementation of φ. The approach is applicable to any linear logic that is closed under negation and can be compiled to nondeterministic Büchi word automaton.

We propose an approach that allows for optimizations at all steps. First of all, to generate the nondeterministic Büchi word automaton, we use the optimizations present in Wring [SB00]. The conversion to the universal co-Büchi tree automaton is relatively simple. If we consider the universal co-Büchi tree automaton as a game between the environment (which drives I) and the system (which drives O), states that are winning for the environment represent unrealizable specifications and can be removed. On the weak alternating tree automaton that is created next, we can perform the same optimization. Furthermore, we extend the concept of simulation to alternating tree automata and use it to optimize the automaton. Next, we compute the states of the nondeterministic automaton in a breadth-first manner and compute the game at every step. Thus, we may avoid expanding many of the states of the nondeterministic automaton. Finally, we use a simulation-based optimization to minimize the size of the resulting finite state machine. As suggested in [KV05], we perform the construction of the weak alternating automaton and the nondeterministic Büchi automaton incrementally. We build an increasingly large part of the weak alternating automaton and reuse results obtained using the smaller

automata for the larger ones.

We developed a tool, Lily (Linear Logic Synthesizer), that takes as input an LTL formula and a partition of the atomic propositions into input and output signals. If the specification is realizable, it delivers a Verilog file as output. Our implementation is the first to handle full LTL. It does not impose any syntactic requirements on the specification.

Repair of Finite-state Systems

The second contribution is an approach to automatically fix faults in a finite state system by considering the repair problem as a game. The idea is to look for a slight modification of the systems such that the system can satisfy its specification. We assume that a system consists of several components and one of them is responsible for the fault. The choice of components, which is up to the user, depends on the level of abstraction in which the system is given. In a high-level representation like VERILOG source code, components can be expressions. On the lower gate level either a gate or a set of gates (e.g., full adder) are considered as a component. The actual fault model depends on the choice of the components. In our examples, components are expressions. These can occur either as conditions (e.g., of an if statement) or on the right-hand side of an assignment. We have chosen this fault model for the purpose of illustration, and our method applies equally well to other fault models.

We assume that the specification is given in LTL. Given a set of suspect components, we extend the faulty system to an LTL game that captures the possible repairs, by making some components "unknown". The game is played between the environment, which provides the inputs, and the system, which provides the correct value for the unknown expression. The game is won if for any input sequence the system can provide a sequence of values for the unknown expression such that the specification is satisfied. A winning strategy fixes the proper values for the unknown expression and thus corresponds to a repair.

In order to find a strategy, we construct a Büchi game that is the product of the constructed game and the standard nondeterministic automaton for the specification. If the product game is won, so is the original LTL game, but because of the nondeterminism in the automaton, the converse does not hold. In many cases, however, we can find a winning finite state strategy anyway, and the nondeterministic automaton may be exponentially smaller than a deterministic equivalent.

To implement the repair corresponding to a finite state strategy, we may need to add

state to the system, mirroring the specification automaton. Such a repair is unlikely to please the developer as it may significantly alter the system, inserting new variables and new assignments throughout the code. Instead, we look for a memoryless strategy, which corresponds to a repair that changes only the suspected lines and does not introduce new variables.

We obtain a conservative algorithm that yields valid repairs and is complete for invariants. It may, however, fail to find a memoryless repair for other types of properties, either because of nondeterminism in the automaton or because of the heuristic that constructs a memoryless strategy. We describe a symbolic method to extract a repair from the strategy. We have implemented the algorithm in VIS [B$^+$96] and present initial experiences with the algorithm.

Our work is related to controller synthesis [RW89]. The controller synthesis problem, however, does not assume that the plant is malfunctioning, and our repair application is novel. Also, we study the problem of finding a memoryless repair, which corresponds to a controller that is "integrated" in the plant. Buccafurri et al. [BEGL99] consider the repair problem for CTL as an abductive reasoning problem and present an approach that is based on calling the model checker once for every possible repair to see if it is successful. Our approach needs to consider the problem only once, considering all possible repairs at the same time, and is likely to be more efficient.

Model-based diagnosis [Rei87, CFTD93] can also be used to suggest repairs for broken programs by incorporating proper fault models into the diagnosis problem. Stumptner and Wotawa [SW96] discuss this approach for functional programs. The approach appears to be able to handle only a small amount of possible repairs, and bases its conclusions on a few failed test cases (typically one) instead of the specification. This approach was combined with model-checking techniques and applied to sequential system by Staber et al. [SFBD06] and to C programs by Griesmayer et al. [GSB06]. Both approaches provide potential fault locations but do not find repairs.

In [SJB05], Staber et al. show how to extend our approach to finding and fixing faults. The idea is to introduce an additional initial step to the program game, in which the system chooses the faulty component that is going to be replaced.

In follow-up work, Griesmayer et al. [GBC06] adapt the repair idea to infinite state systems using push-down games. Their approach handles Boolean programs (incl. recursions), which appear in Software model checking (cf. [BMMR01]).

Synthesis of Real-Life Designs

As our third contribution, we demonstrate the viability of the synthesis approach for the derivation of a correct gate-level implementation directly from a specification written in the LTL.

Even though synthesis from LTL specifications in general is considered intractable, there exist several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved efficiently (cf. Section 1.2). Major progress has been achieved in [PPS06], where the authors present an efficient symbolic algorithm for GR(1) specifications.

We have implemented the approach of [PPS06] in a tool called Anzu, and extended it to produce not only a BDD representing a set of possible implementations, but also an actual circuit.

We demonstrate the application of the synthesis method by means of two examples. The first is a generalized buffer from IBM, a tutorial design for which a good specification is available. The second is the arbiter for one of the AMBA buses [Ltd99], a characteristic industrial design that is not too big. Previous work on synthesis has only considered toy examples such as a simple mutual exclusion protocol, an elevator controller, or a traffic light controller [HRS05, PPS06, JB06b]. This is the first time a realistic industrial design has been tackled.

Chapter 2

Preliminaries

In this chapter, we describe the necessary theoretical background for our work. We start with an introduction to Linear Temporal Logic (LTL) (Section 2.1). In Section 2.2 we introduce tree and word automata. Afterwards, we talk about infinite game theory (Section 2.3). In Section 2.4, we explain how we solve infinite games. Finally, in Section 2.5 we introduce the synthesis problem and explain how it relates to infinite game and automata theory.

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [Pnu77] is a linear-time logic to express properties over paths. LTL formulas are built from the atomic propositions in AP, their negations, the temporal modalities X (next), U (until), and R (releases), and Boolean conjunction and disjunction.

Given an atomic propositions $p \in AP$, we define the syntax of an LTL formula in negation normal form as

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \varphi \,\mathsf{R}\, \varphi$$

We can compute the negation of a formula by recursively applying De Morgan's Laws and the identities $\neg \mathsf{X}\varphi = \mathsf{X}\neg\varphi$ and $\neg(\varphi \,\mathsf{U}\, \psi) = \neg\varphi \,\mathsf{R}\, \neg\psi$.

We define the semantics of LTL with respect to paths over 2^{AP}. Let $\pi = \pi_0, \pi_1, \pi_2, ..$ be an infinite path over 2^{AP} and $\pi^i = \pi_i, \pi_{i+1}, \pi_{i+2}, ..$ be the path starting at π_i. Then the semantics of formula φ is defined as follows

- $\pi \models \mathsf{true}$,

- $\pi \not\models \mathsf{false}$,
- $\pi \models \varphi$ iff $\varphi \in \pi_0$ for $\varphi \in AP$,
- $\pi \models \neg\varphi$ iff $\varphi \notin \pi_0$ for $\varphi \in AP$,
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $\pi \models \psi$,
- $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$,
- $\pi \models \mathsf{X}\,\varphi$ iff $\pi^1 \models \varphi$,
- $\pi \models \varphi\,\mathsf{U}\,\psi$ iff $\exists i \geq 0 : (\pi^i \models \psi$ and $\forall 0 \leq j < i : \pi^j \models \varphi)$, and
- $\pi \models \varphi\,\mathsf{R}\,\psi$ iff $\forall i \geq 0 : \pi^i \models \psi$, or $\exists i \geq 0 : (\pi^i \models \varphi$ and $\forall 0 \leq j \leq i : \pi^j \models \psi)$.

We write $L(\varphi)$ for the set of infinite words that satisfy φ. We use the usual definition of the Boolean connectives $\rightarrow$ and $\leftrightarrow$, and the abbreviations F (eventually), G (globally), and W (weakuntil), which are defined as follows

- $\mathsf{F}\,\varphi = \mathsf{true}\,\mathsf{U}\,\varphi$,
- $\mathsf{G}\,\psi = \mathsf{false}\,\mathsf{R}\,\psi$, and
- $\varphi\,\mathsf{W}\,\psi = (\varphi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\varphi$.

In Chapter 5, we use further abbreviations to indicate the raising or the falling edge of a signal, and state nested weak until formulas. The abbreviations are inspired by abbreviations used in PSL [EF06]. Given an atomic propositions p and two LTL formula φ and ψ, we define

- $\mathtt{raise}(p) = \neg p \wedge \mathsf{X}\,p$,
- $\mathtt{fall}(p) = p \wedge \mathsf{X}\,\neg p$, and
- $\varphi\,\mathsf{W}\,[i]\psi = \varphi\,\mathsf{W}(\psi \wedge \mathsf{X}(\varphi\,\mathsf{W}\,[i-1]\psi))$ for $i > 1$ and $\varphi\,\mathsf{W}[1]\psi = \varphi\,\mathsf{W}\,\psi$.

2.2 Moore Machines, Trees, and Automata

Moore Machines A *Moore machine* with output alphabet Σ and input alphabet D is a tuple $M = (\Sigma, D, S, s_0, f, g)$ such that S is a finite set of states, $s_0 \in S$ is the initial state, $f : S \times D \rightarrow S$ is the transition function, and $g : S \rightarrow \Sigma$ is the output

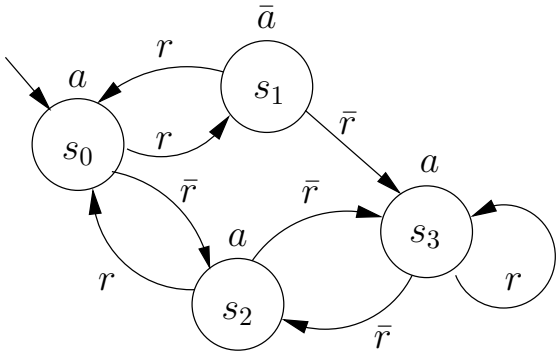

Figure 2.1: Moore machine with $\Sigma = \{a, \bar{a}\}$ and $D = \{r, \bar{r}\}$.

function. We extend f to the domain $S \times D^*$ with the recursive definition $f(s, d_0 d_1 ... d_n) = f(f(s, d_0), d_1 ... d_n)$. The *input/output language* of M is $L(M) = \{\pi \in (\Sigma \cup D)^\omega \mid \pi = (\sigma_0 \cup d_0, \sigma_1 \cup d_1, \dots) \text{ and } \sigma_i = g(f(q_0, d_0 ... d_{i-1}))\}$.

For instance, Figure 2.1 depicts a Moore machine with $\Sigma = \{a, \bar{a}\}$ and $D = \{r, \bar{r}\}$. Circles denote states and arrows denote transitions. The labeling on the circles defines the output function g, and the labels on the arrows are part of the transition function f. The machine starts in s_0 and outputs a. Depending on the input (r or $\bar{r}$) it moves to s_1 or s_2 and outputs $\bar{a}$ or a respectively. From the next state (s_1 or s_2) the machine proceeds similarly. Each path through the Moore machine corresponds to a word in its input/output language, e.g., the path $s_0, s_1, s_0, s_1, \dots$ corresponds to $ar, \bar{a}r, ar, \bar{a}r, \dots$.

Trees Let Σ be an *alphabet* and let D be a set of *directions*, then a Σ-*labeled* D-*tree* is a pair (T, τ) such that $T \subseteq D^*$ is prefix-closed and $\tau : T \to \Sigma$. (See Figure 2.2.) We call $t \in T$ a *node* of the tree T, and the empty word ε is the *root* of T. The tree is *complete* if $T = D^*$. The set of all Σ-labeled D-trees is denoted by $T_{\Sigma,D}$. Given a word language $L \in (\Sigma \cup D)^\omega$, let $\Lambda(L) \subseteq T_{\Sigma,D}$ be the set of trees T such that all paths of T are in L. Similarly, we will write $\Lambda(\varphi)$ for the set of trees T such that every path of T satisfies the LTL formula φ.

We will use Σ-labeled D-trees to model Moore machines with input alphabet D and output alphabet Σ. Let π be a *path* of T, then $\varepsilon \in \pi$ and for all nodes $t \in \pi$ holds that either t is a leaf or there exists a unique direction $d \in D$ such that $t \cdot d \in \pi$. Given a node $t' = t \cdot d$, we call d the direction of the node t' and denote it by $d(t')$. Then a path $\pi = t_0 t_1 \dots$ of a Σ-labeled D-tree induces a word $w = w_0 w_1 \dots \in (\Sigma \cup D)^\omega$ with $w_i = (\tau(t_i) \cup d(t_{i+1}))$ for $i \geq 0$. Intuitively, we merge the label of the node with the

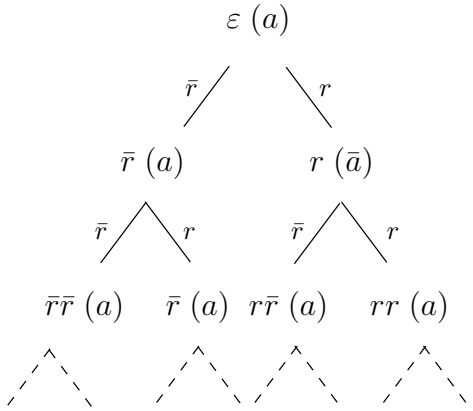

Figure 2.2: Infinite tree with $\Sigma = \{a, \bar{a}\}$ and $D = \{r, \bar{r}\}$.

direction following it in the path. Recall that labels are elements of the output alphabet and directions are elements of the input alphabet. So the definition of w corresponds to the behavior of a Moore machine, which determines its current output value before it reads the current input value (cf. Figure 2.1).

In fact, every Moore machine corresponds to a complete Σ-labeled D-tree for which every node $t \in D^*$ is labeled with $g(f(q_0, t))$. For instance, the Moore machine of Figure 2.1 corresponds to the $\{a, \bar{a}\}$-labeled $\{r, \bar{r}\}$-tree shown in Figure 2.2. Thus, every tree language $L \subseteq T_{\Sigma,D}$ defines a set $\mathcal{M}(L)$ of Moore machines: those machines M for which $\Lambda(L(M)) \in L$. Note that not every tree can be defined by a Moore machine, because not all trees are *regular*. (A tree is regular if it is the unwinding of some Moore machine.) Thus there are tree languages L for which $\bigcup\{\Lambda(L(M)) \mid M \in \mathcal{M}(L)\} \neq L$.

Automata An *alternating tree automaton* for Σ-labeled D-trees is a tuple $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ such that Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \to 2^{2^{D \times Q}}$ is the *transition relation* (an element $C \in 2^{D \times Q}$ is a *transition*) and $\alpha \subseteq Q$ is the *acceptance condition*. We denote by A^q, for $q \in Q$, the automaton A with the initial state q.

In Figure 2.3, we show an example of an alternating tree automaton that accepts Σ-labeled D-trees with $\Sigma = \{a, \bar{a}\}$ and $D = \{r, \bar{r}\}$. Circles denote states, two concentric circles denote accepting states, and boxes denote transitions. Transitions are split into two parts: green and red edges. Green edges start at circles and are labeled with subsets of the alphabet Σ. We use a Boolean formula over the variable a to represent a subset, e.g., true represents the set $\{a, \bar{a}\}$, while $\neg\mathsf{a}$ stands for the set $\{\bar{a}\}$. Similarly, we label red edges

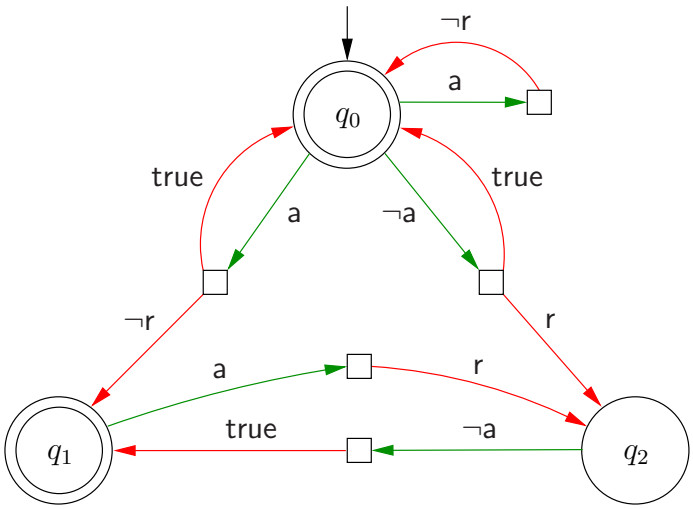

Figure 2.3: An alternating Büchi tree automaton

starting at boxes with subsets of D, and use the variable r to describe them. The transition corresponding to a box C consists of all pairs $(d, q) \in D \times Q$ for which there is an edge from C to q such that d is included in the label on the edge. For instance, the box between q_0 and q_1 reachable from state q_0 with letter a corresponds to $\{(\bar{r}, q_0), (r, q_0), (r, q_1)\}$.

A *run* (R, ρ) of A on a Σ-labeled D-tree (T, τ) is a $(T \times Q)$-labeled $\mathbb{N}$-tree satisfying the following constraints:

1. $\rho(\varepsilon) = (\varepsilon, q_0)$.
2. If $r \in R$ is labeled (t, q), then there is a set $\{(d_1, q_1), \ldots, (d_k, q_k)\} \in \delta(q, \tau(t))$ such that r has k children labeled $(t \cdot d_1, q_1), \ldots, (t \cdot d_k, q_k)$.

Intuitively, the nodes of the run on a tree T are labeled with pairs (t, q) meaning that A is in state q in node t of T. Because A is alternating, for a given node t there can be multiple q_i and nodes labeled (t, q_i) in R. The automaton starts at the root node in state q_0. If it is in state q in node t of T, and t is labeled σ, then $\delta(q, \sigma)$ tells A what to do next. The automaton can nondeterministically choose a transition $C \in \delta(q, \sigma)$. Then, for all $(d', q') \in C$, A moves to node $t \cdot d'$ in state q'. (The transition relation $\delta(q, \sigma)$ can be considered as a DNF formula over $D \times Q$.) Note that there are no runs with a node (t, q) for which $\delta(q, \tau(t)) = \emptyset$. On the other hand, a run that visits a node t does not need to visit all of its children; there are no restrictions on subtrees with roots that are not visited. In particular, a node (t, q) such that $\delta(q, \tau(t)) = \{\emptyset\}$ does not have any children and there

are no restrictions on the subtree with root t.

For instance, a run of the automaton in Figure 2.3 on the tree of Figure 2.2 starts in state q_0. The automaton reads the letter a of the first node ε. Now the automaton can choose between the transitions $\{(\bar{r}, q_0)\}$ and $\{(\bar{r}, q_0), (\bar{r}, q_1), (r, q_1)\}$. The latter tells it to move to q_0 and q_1 in direction $\bar{r}$ and to q_1 with r and the run branches to the three nodes $(\bar{r}, q_0)$, $(\bar{r}, q_1)$, and (r, q_1). In each node the automaton reads again the corresponding letter of the tree, chooses a matching transition, and moves accordingly. Note that the automaton needs not to visit all children of a tree node, e.g., if the automaton chooses the first transition from above, which tell the automaton to move to q_0 in direction $\bar{r}$ only, the path of the run corresponding to direction r ends and is therefore finite.

We consider two acceptance conditions: Büchi and co-Büchi. A run (R, ρ) of a Büchi (co-Büchi) automaton is accepting if all *infinite* paths of (R, ρ) have infinitely many states in α (only finitely many states in α, resp.). The language $L(A)$ of A is the set of trees for which there exists an accepting run.

An alternating tree automaton induces a graph. The states of the automaton are the nodes of the graph and there is an edge from q to q' if (d', q') occurs in $\delta(q, \sigma)$ for some $\sigma \in \Sigma$ and $d' \in D$. A Büchi automaton is *weak* if each strongly connected component (SCC) contains either only states in α or only states not in α.

An automaton is *universal* if $|\delta(q, \sigma)| = 1$. A universal automaton has at most one run for a given input. An automaton is *nondeterministic* if for all $q \in Q, \sigma \in \Sigma, C \in \delta(q, \sigma)$ and $(d_i, q_i), (d_j, q_j) \in C$ we have $d_i = d_j$ implies $q_i = q_j$. That is, the automaton can only send one copy in each direction and every run is isomorphic to the input tree. An automaton is *deterministic* if it is both universal and nondeterministic.

An automaton is a *word automaton* if $|D| = 1$. In that case, we can leave out D altogether. We will leave out the boxes when we draw deterministic or nondeterministic word automata (cf. Figure 3.1).

We distinguish automata according to their branching mode (alternating/nondeterministic/universal/deterministic), their acceptance condition (Büchi/co-Büchi/weak), and the input element (tree/word) they run on. Following this distinction we will abbreviate automata as three letter acronyms: {A/N/U/D} {B/C/W} {T/W}, e.g., NBW stands for nondeterministic Büchi word automaton.

NBW from LTL Given an LTL formula φ over atomic propositions AP, we can construct a nondeterministic Büchi word automaton A with the alphabet 2^{AP} that accepts all words

in φ. For more details we refer to [WVS83, LP85, SB00].

Following the construction proposed in [SB00] we get a *generalized* NBW, which has more than one Büchi acceptance condition and labels on the states. We use automata with labels on transitions rather than states. The translation between those two types is straight forward. Furthermore, we can easily get rid of multiple Büchi conditions with help of the well-known counting construction [Cho74]. The size of the resulting automaton is exponential in the length of the formula in the worst case. In Figure 3.1, we show an example of NBW with labels on transitions constructed from an LTL formula.

2.3 Games

Mealy games We define a *(Mealy) game* G over set of atomic propositions AP as a tuple $(S, s_0, I, C, \delta, \lambda, F)$, where S is a finite set of states, $s_0 \in S$ is the initial state, I and C are finite sets of environment inputs and system choices, $\delta : S \times I \times C \rightharpoonup S$ is the partial transition function, $\lambda : S \to 2^{AP}$ is the labeling function, and F is the winning condition defining obligations on the system. The winning conditions of *safety*, *Büchi,* and *co-Büchi games* are defined with respect to a target set $A \subseteq S$. Safety games require that a play stays within the target set A, while Büchi (co-Büchi) games require that a play visits the target set A infinitely often (only finitely many times, respectively). For such games, we will write A for F. The winning condition of an *LTL game* is the set of plays satisfying an LTL formula φ. In this case, we will write φ for F. In general, $F \subseteq S^\omega$ is given as a set of infinite sequences of states. With the exception of this section and Section 4.4, we will assume that δ is a complete function.

Intuitively, a game is an incompletely specified finite state machine together with a specification. The environment chooses the inputs (as usual), and the system choices C represent the freedom of the implementation. The challenge of the system (the protagonist) is to find proper values for C such that F is satisfied, while the environment (the antagonist) tries to force a violation of F. We look at games and the corresponding definitions from the point of the protagonist. All definitions can be stated dually for the antagonist.

Given a game $G = (S, s_0, I, C, \delta, \lambda, F)$, a *(finite state) strategy* is a tuple $\sigma = (Q, q_0, \mu)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\mu : Q \times S \times I \to 2^{C \times Q}$ is the *move function*. Intuitively, a strategy automaton 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, and we can easily extract a deterministic strategy by fixing the strategy to one of the possible responses. We need nondeterminism in Chapter 4 in order to have maximal freedom when we attempt to convert a finite state strategy to a memoryless strategy.

For the strategy to be winning, a winning play has to ensue for any nondeterministic choices of the strategy.

A *play* on G according to σ is a finite or infinite sequence $\pi = q_0 s_0 \xrightarrow{i_0 c_0} q_1 s_1 \xrightarrow{i_1 c_1} \dots$ such that $(c_i, q_{i+1}) \in \mu(q_i, s_i, i_i)$, $s_{i+1} = \delta(s_i, i_i, c_i)$, and either the play is infinite, or there is an n such that $\mu(q_n, s_n, i_n) = \emptyset$ or $\delta(s_n, i_n, c_n)$ is not defined, which means that the play is finite. A play is *winning* if it is infinite and $s_0 s_1 \dots \in F$. (If $\mu(q_n, s_n, i_n) = \emptyset$, the strategy does not suggest a proper system choice and the game is lost.) A strategy σ is *winning* on G if all plays according to σ on G are winning. A state s is *winning* if there is winning strategy starting at s. A game is *determined* if each state $s \in S$ is either winning or *lost*, that is winning for the environment. The set of all winning states is the *winning region*. The game G is *winning* (or *won*) if the initial state s_0 is in the winning region.

A *memoryless strategy* is a finite state strategy with only one state. We will write a memoryless strategy as a function $\sigma : Q \times I \to 2^C$ and a play of a memoryless strategy as a sequence $s_0 \xrightarrow{i_0 c_0} s_1 \xrightarrow{i_1 c_1} \dots$, leaving out the state of strategy automaton. Safety, Büchi, and co-Büchi games are determined and from each state either the system or the environment has a winning memoryless strategy (cf. [Tho95]).

We extend the labeling function λ to plays: the *output word* is $\lambda(\pi) = \lambda(s_0)\lambda(s_1)\dots$. Likewise, the *input word* is $\iota(\pi) = i_0 i_1 \dots$, the sequence of environment inputs. The *output language (input language)* $L(G)$ ($I(G)$) of a game is the set of all $\lambda(\pi)$ ($\iota(\pi)$) with π winning. Note that if a play π is winning, it does not imply that π can be generated by a winning strategy. So $w \in I(G)$ does not imply that the system necessarily wins on input w.

Games and Tree Automata Games and tree automata (see Section 2.2) are closely related. We can view a nondeterministic tree automaton as a game between two players, the *automaton* and the *pathfinder* [GH82]. The automaton controls the label and the transition and the pathfinder chooses the direction. In order to check language emptiness the automaton and the pathfinder play against each other in a round-based manner. Each play induces an infinite sequence of states. The automaton wins if this infinite sequence satisfies the acceptance condition, otherwise the pathfinder wins. A winning strategy of

the automaton corresponds to a Σ-labeled D-tree that is accepted. If the pathfinder has a winning strategy the language of the tree automaton is empty. Note that we use a different order in which the protagonist and the antagonist move in the definitions of games and tree automata. In a game we allow the antagonist (environment) to move first, while with trees the protagonist (automaton) moves first. The order in which the players move determines if the generated system can react immediately or not. The two types of system are essentially equivalent and can be transformed into each other.

Moore games In Chapter 3, we will use games in which the protagonist moves first. We call such a game a *Moore game*. A Moore game is a tuple $G_M = (Q, q_0, C, I, \delta, F)$, where Q is a set of states, q_0 is the initial state, C is the set of choices of the protagonist, I is the set of inputs the antagonist can choose from, $\delta : Q \times C \times I \to Q$ is a transition function, and $F \subseteq Q$ either defines a Büchi or a co-Büchi winning condition. Each Moore game $G_M = (Q, q_0, C, I, \delta, F)$ corresponds to a Mealy game $(Q', q_0', I', C', \delta', \lambda', F')$, with $Q' = Q$, $q_0' = q_0$, $I' = C$, $C' = I$, $\delta' = \delta$, $\lambda'(q) = q$ for all $q \in Q'$, and $F' = F$ but with the dual winning condition. Büchi Moore games correspond to co-Büchi Mealy games, and co-Büchi Moore games to Büchi Mealy games. Intuitively, we switch the role of the two players and take the dual winning condition. Plays, winning regions, and strategies are defined accordingly.

Whenever we talk about a game we refer to a Mealy game unless stated differently.

2.4 Solving Games

In order to solve games, we introduce a slightly modified version of the *μ-calculus*. The μ-calculus is first-order predicate logic with a *least* (μ) and a *greatest* (ν) *fixpoint operator*. For a detailed description we refer to [Koz83].

Given a game $G = (S, s_0, I, C, \delta, \lambda, F)$ and a set $A \subseteq S$, then the set $\mathsf{MX}\, A = \{s \mid \forall i \in I\, \exists c \in C : \delta(s, i, c) \in A\}$ is the set of states from which the system can force a visit to a state in A in one step. The set $\mathsf{M}A\,\mathsf{U}\,B$ is defined by the μ-calculus formula $\mu Y.\, B \cup \mathsf{MX}\,(A \cap Y)$. It defines the set of states from which the system can force a visit to B without leaving A. The *iterates* of this computation are $Y_0 = B$ and $Y_{j+1} = Y_j \cup (A \cap \mathsf{MX}\, Y_{j-1})$ for $j > 0$. From Y_j the system can force a visit to B in at most j steps. Note that if the system is finite, there can only be finitely many iterations. Dually, we define the set $\mathsf{M}A\,\mathsf{R}\,B$ by the formula $\nu Y.\, B \cap \mathsf{MX}\,(A \cup Y)$, which is the set of states from which the system can force to

stay in B forever or eventually visit a state in $A \cap B$. We define $\mathsf{MG}\, A = \nu Z.A \cap \mathsf{MX}\, Z$, the set of states from which the system can avoid leaving A.

For a Büchi game, we define $W_B = \nu Z.\mathsf{MX}\,(\mathsf{M}Z\,\mathsf{U}(Z \cap B))$. The set W_B is the set of states from which the system can win the Büchi game. Similarly, given a co-Büchi game, we define the set $W_C = \mu Z.\,\mathsf{MX}\,(\mathsf{M}Z\,\mathsf{R}(Z \cup (Q \setminus B)))$ as the set of from which the system can win the co-Büchi game. Note that these fixpoints are similar to the ones used in model checking of fair CTL and are easily implemented symbolically.

Using these characterizations, we can compute memoryless strategies for safety, Büchi, and co-Büchi games [Tho95]. For a safety game with condition A, the strategy $\sigma(s,i) = \{c \in C \mid \delta(s,i,c) \in \mathsf{MG}\, A\}$ is winning if and only if $s_0 \in \mathsf{MG}\, A$. For a Büchi game, let $W = \nu Z.\mathsf{MX}\,(\mathsf{M}Z\,\mathsf{U}(Z \cap B))$. Let Y_0 through Y_n with $Y_0 = \emptyset$ and $Y_n = W$ be the set of distinct iterates of $\mathsf{M}W\,\mathsf{U}(W \cap B) = W$. We define the *attractor strategy* for B to be

$$\begin{aligned}\sigma(s,i) \;=\; & \{c \in C \mid \exists\, 1 < j \leq n, k < j : s \in Y_j \setminus Y_{j-1} \wedge \delta(s,i,c) \in Y_k\} \cup \\ & \{c \in C \mid s \in Y_1 \wedge \delta(s,i,c) \in W\}.\end{aligned}$$

The attractor strategy brings the system ever closer to B, and then brings it back to a state from which it can force another visit to B.

For a co-Büchi game, let $W = \mu Z.\,\mathsf{MX}\,(\mathsf{M}Z\,\mathsf{R}(Z \cup (Q \setminus B)))$ and let $Z_0, \ldots Z_n$ with $Z_0 = \emptyset$ and $Z_n = W$ be the iterates of the outer fixpoint. We define the *co-Büchi strategy* for B to be

$$\begin{aligned}\sigma(s,i) \;=\; & \{c \in C \mid \exists\, 1 < j \leq n, k < j : s \in B \wedge s \in Z_j \setminus Z_{j-1} \wedge \delta(s,i,c) \in Z_k\} \cup \\ & \{c \in C \mid \exists\, 1 < j \leq n, k \leq j : s \notin B \wedge s \in Z_j \setminus Z_{j-1} \wedge \delta(s,i,c) \in Z_k\} \cup \\ & \{c \in C \mid s \in Z_1 \wedge \delta(s,i,c) \in Z_1 \setminus B\} \cup\end{aligned}$$

In order to compute the winning region and a winning strategy of a Moore game $G_M = (Q, q_0, C, I, \delta, F)$, we simply replace MX with a corresponding operator $\mathsf{MX}_T\,(A) = \{q \mid \exists c \in C, \forall i \in I : \delta(q,c,i) \in A\}$ and use the same fixpoint formulas.

2.5 LTL Synthesis

We will use LTL to specify the behavior of a system. Properties will use the set $I \cup O$ of atomic propositions, where I and O are disjoint sets denoting the input and output signals, respectively.

The key to the solution of LTL Synthesis is the observation that a program with input signals I and output signals O can be seen as a complete Σ-labeled D-tree with $\Sigma = 2^O$ and $D = 2^I$: the label of node $t \in D^*$ gives the output after input sequence t. The solution proposed in [PR89] is to build a nondeterministic Büchi word automaton for the specification and then to convert this automaton to a deterministic Rabin automaton that recognizes all complete trees satisfying the specification. A witness to the nonemptiness of the automaton is an implementation of the specification.

In a game-theoretic of view, synthesis is a infinite game between two players, the environment and the system. The environment controls the input signals I and the system controls the output signals O. The winning conditions of the system is to fulfill the specification, which is defined over $I \cup O$. The environment aims to force a violation. A winning strategy for the system defines for each finite input sequence a correct output values. Intuitively, it tells the system how to fulfill the specification and therefore correspond to an implementation of the specification.

For more details about the particular approaches our work is based on, we refer to Section 3.2.1, 4.2, and 5.1.1.

Chapter 3

Optimization for LTL Synthesis

This chapter is based on [JB06a] and [JB06b].

In this chapter, we present an approach to automatic synthesis of specifications given in arbitrary LTL formulas. The approach is based on a translation through universal co-Büchi tree automata and alternating weak tree automata [KV05]. By careful optimization of all intermediate automata, we achieve a major improvement in performance.

In Section 3.1, we start with presenting optimization techniques for alternating tree automata, including a game-based approximation to language emptiness and a simulation-based optimization. In Section 3.2, we recall the construction of Kupferman and Vardi [KV05], and discuss further optimizations applicable to the construction. All our optimizations are computed in time polynomial in the size of the automaton on which they are computed. We have applied our implementation to several examples and show a significant improvement over the straightforward implementation in Section 3.3.

3.1 Simplifying tree automata

In this section we discuss two optimizations that can be used for any tree automaton.

3.1.1 Simplification Using Games

We define a sufficient (but not necessary) condition for language emptiness of A^q. Our heuristic views the alternating automaton as a Moore game (see Section 2.3) which is played in rounds. In each round, starting at a state q, the protagonist decides the label $\sigma \in \Sigma$ and a set $C_q \subseteq \delta(q, \sigma)$ and the antagonist chooses a pair $(d, q') \in C_q$. The next

round starts in q'. If $\delta(q, \sigma)$ or C_q are empty the play is finite and the player who has to choose from an empty set loses the game. If a play is infinite the winner is determined by the acceptance condition. For an ABT (ACT), the protagonist wins the play if the play visits the set of accepting states α infinitely often (only finitely often, resp.). Formally, we define the following Moore game with Büchi (co-Büchi) winning condition.

Definition 3.1. *Given an ABT (ACT) $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, we construct a Moore game $G = (Q', q_0', C', I', \delta', \alpha')$ with state space $Q' = Q \cup \{w, l\}$, choices of the protagonist $C' = \Sigma \times 2^{D \times Q}$, inputs of the antagonist $I' = D \times Q$, and a Büchi (co-Büchi) winning condition $\alpha' = \alpha \cup \{w\}$ ($\alpha' = \alpha \cup \{l\}$, resp.). The transition function $\delta' : Q' \times C' \times I' \to Q'$, is defined as*

$$\delta'(q, (\sigma, C_q), (d, q')) = \begin{cases} l & \text{if } q = l, \\ w & \text{if } q = w, \\ l & \text{if } q \in Q \wedge C_q \notin \delta(q, \sigma), \\ w & \text{if } q \in Q \wedge C_q \in \delta(q, \sigma) \wedge (d, q') \notin C_q, \text{ and} \\ q' & \text{otherwise.} \end{cases}$$

Since the game is a Büchi (co-Büchi) game, we can compute the winning region as shown in Section 2.4. Note that Büchi (co-Büchi) games are determined, and for each state either the protagonist or the antagonist has a winning memoryless strategy (cf. 2.3). Recall that given a Moore game $G = (Q, q_0, C, I, \delta, \alpha)$, a memoryless strategy s (for the protagonist) maps from a state $q \in Q$ to a choice $c \in C$, while a strategy for the antagonist maps from a state q and a choice c to an input $i \in I$.

Lemma 3.2. *Given an ABT (ACT) $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, let $G = (Q', q_0', C', I', \delta', \alpha')$ be a Moore game with Büchi (co-Büchi) winning condition defined as above, and let $W \subseteq Q'$ be the winning region. If $q \notin W$ then $L(A^q) = \emptyset$.*

Proof. In order to proof that $L(A^q) = \emptyset$ if $q \notin W$, we show that every (infinite) Σ-labeled D-tree (T, τ) is rejected by A^q. Note that a tree (T, τ) is rejected iff all runs of A^q on (T, τ) have at least one path that is rejected. Consider a run of A^q on (T, τ). A path in the run is a play in G that stays in $Q' \setminus \{l\}$. (Note that for a node r in a run with label (t, q), we have $\delta'(q, (\sigma, C_q), (d, q')) \neq l$ in the corresponding play because $C_q \in \delta(q, \tau(t))$ always holds. If a path ends at node r with direction d then for all $q' \in Q$ it holds that $(d, q') \notin C_q$ and the path is accepting. The corresponding play loops in w and is winning.) In fact, in a run there is one path for every sequence of choices of the antagonist. Since

the antagonist has a winning strategy from q, one such path does not visit α' infinitely often (only finitely many times, resp.) and the run is rejected. □

In the case of an NBT or NCT the converse holds as well [GH82], but in the alternating case it does not: A counterexample is a word automaton such that (1) $\delta(q_0, \sigma) = q_1 \wedge q_2$ for all σ, (2) $L(A^{q_1}) \cap L(A^{q_2}) = \emptyset$, and (3) the games corresponding to A^{q_1} and A^{q_2} are won. (Computing a necessary and sufficient condition in polynomial time is not possible as it would give us an EXPTIME algorithm for deciding realizability.)

In an ABT (ACT) with acceptance condition α, we can discard the states outside of W, where W is the winning region of the corresponding game as defined in Definition 3.1.

Theorem 3.3. *Given an ABT (ACT) $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, let G be a game as defined in Definition 3.1, and let W be the winning region of G. If $q_0 \in W$, let the ABT (ACT) $A' = (\Sigma, D, Q', q_0, \delta', \alpha')$ with $Q' = Q \cap W$, $\alpha' = \alpha \cap W$, and $\delta'(q, \sigma) = \{C \in \delta(q, \sigma) \mid \forall (d, q') \in C, q' \in W\}$. If $q_0 \notin W$, let A' consist of a single non-accepting state.*

We have $L(A^q) = L(A'^q)$ for all $q \in Q'$ and thus $L(A) = L(A')$.

Proof. If $q_0 \notin W$, A' is a single non-accepting state and $L(A') = \emptyset$. From Lemma 3.2 and $q_0 \notin W$ follows that $L(A^{q_0}) = L(A) = \emptyset$. If $q_0 \in W$, we have for each state $q \in Q'$ and letter $\sigma \in \Sigma$ that $\delta'(q, \sigma) = \{C \in \delta(q, \sigma) \mid \forall (d, q') \in C, q' \in W\}$. That implies that for all $C_r \in \delta(q, \sigma)$ with $C_r \notin \delta'(q, \sigma)$, holds that $\exists (d, q') \in C_r, q' \notin W$. Since $q' \notin W$ implies that $L(A^q) = \emptyset$ (Lemma 3.2), every run that follows transition C_r has at least one non-accepting path and is therefore rejected. Removing the conjunct C_r from δ has no influence on the accepting runs, which implies that $L(A^q) = L(A'^q)$. With Lemma 3.2 for $q \in Q \setminus W$ follows that $L(A) = L(A')$. □

3.1.2 Simplification Using Simulation Relations

The second optimization uses (direct) simulation minimization on alternating tree automata. Our construction generalizes that for alternating word automata [AHKV98, FW02, GKSV03].

Definition 3.4. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. The* direct simulation relation *$\preceq \subseteq Q \times Q$ is the largest relation such that $u \preceq v$ implies that*

1. *$u \in \alpha \rightarrow v \in \alpha$ and*
2. *$\forall \sigma \in \Sigma \; \forall C_u \in \delta(u, \sigma) \; \exists C_v \in \delta(v, \sigma) \; \forall (d', v') \in C_v \; \exists u' \in Q$ such that $(d', u') \in C_u$ and $u' \preceq v'$.*

If $u \preceq v$, we say that u is *simulated by* v. If additionally, $u \succeq v$, we say that u and v are *simulation equivalent*, denoted $u \simeq v$.

Lemma 3.5. *If $u \preceq v$ then $L(A^u) \subseteq L(A^v)$.*

Proof. Given an ABT $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, and two states $u, v \in Q$ with $u \preceq v$, we prove that $L(A^u) \subseteq L(A^v)$. Let (T, τ) be a Σ-labeled D-tree in $L(A^u)$, then there exists an accepting run (R_u, ρ_u) of A^u on (T, τ). If $u \preceq v$, we can build a run (R_v, ρ_v) of A^v that accepts (T, τ). We build the run inductively and show that every node $r_v \in (R_v, \rho_v)$ with $\rho_v(r_v) = (t, q_v)$ has a corresponding node $r_u \in (R_u, \rho_u)$ with $\rho_u(r_u) = (t, q_u)$, s.t. $q_u \preceq q_v$. (1) Initially, we have $\rho_v(\varepsilon) = (\varepsilon, v)$, and (ε, v) corresponds to (ε, u), since $u \preceq v$. (2) For every corresponding pair r_u, r_v with $\rho_u(r_u) = (t, q_u)$ and $\rho_v(r_v) = (t, q_v)$, suppose $r_1, ...r_k$ are the children of r_u labeled with $(t \cdot d_1, q_1), ...(t \cdot d_k, q_k)$, then there exists a conjunct $C_u = \{(d_1, q_1), ...(d_k, q_k)\} \in \delta(q_u, \tau(t))$. Since $q_u \preceq q_v$ (by induction hypothesis), there exists a conjunct $C_v \in \delta(q_v, t_v)$, such that $\forall (d', q') \in C_v \ \exists (d', q) \in C_u : q \preceq q'$. Suppose C_v is the set $\{(d'_1, q'_1), ...(d'_j, q'_j)\}$, then r_v has j' children $(t \cdot d'_1, q'_1), ...(t \cdot d'_j, q'_j)$. Because $q_u \preceq q_v$, each child $(t \cdot d', q')$ corresponds to a child $(t \cdot d, q)$ of r_u with $q \preceq q'$. Since every node in (R_v, ρ_v) corresponds to a node in (R_u, ρ_u), each path in (R_v, ρ_v) has a corresponding path in (R_u, ρ_u). All paths in (R_u, ρ_u) visit an accepting state infinitely often. Since we have a direct simulation relation all paths in (R_v, ρ_v) do the same. It follows that (R_v, ρ_v) is an accepting run, and $(T, \tau) \in L(A^v)$. □

The following theorems are tree-automaton variants of those presented in [GKSV03] for optimizing alternating word automata. The first theorem tells us that we can drop states from a transition if they are not minimal with respect to the simulation relation and the second theorem tells us that we can drop entire transitions if there are other transitions that allow for a larger language.

Theorem 3.6. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. For $C \subseteq D \times Q$, let $\min(C) = \{(d, v) \in C \mid \neg\exists u \neq v$ such that $u \preceq v$ and $(d, u) \in C\}$. Let $A' = (\Sigma, D, Q, q_0, \delta', \alpha)$, where for all q and σ we have $\delta'(q, \sigma) = \{\min(C) \mid C \in \delta(q, \sigma)\}$. We have $L(A) = L(A')$.*

Proof. First, we prove that every state in A is equivalent to the state in A' with the same name. For clarity we mark states in A' with a tilde (e.g., $\tilde{q}$), we do likewise with sets and transitions. We have to show that for all $q \in Q$, we have $q \preceq \tilde{q}$ and $\tilde{q} \preceq q$.

It should be clear that for every state q, we have $q \preceq \tilde{q}$, since given a label $\sigma \in \Sigma$, any transition $\tilde{C} \in \delta'(\tilde{q}, \sigma)$ is a subset or equal to the corresponding transition $C \in \delta(q, \sigma)$.

Due to Lemma 3.5, we have $L(A) \subseteq L(A')$. For the other direction, let $S = \{(\tilde{q}, q) \mid q \in Q\} \cup \{(\tilde{q}, r) \mid q \preceq r\}$. We show that S is a simulation relation. Suppose $(\tilde{q}, q) \in S$. Let $\sigma \in \Sigma$, $\tilde{C} \in \delta'(\tilde{q}, \sigma)$, then there is a $C \in \delta(q, \sigma)$ s.t. $\tilde{C} = \min(C)$ (by definition of δ'). Thus, for any $(d, u) \in C$, there is a $u' : (d, u') \in \min(C), u' \preceq u$, so $(\tilde{u'}, u) \in S$. Now, suppose $(\tilde{q}, r) \in S$ and $q \preceq r$. Let $\sigma \in \Sigma$, $\tilde{C} \in \delta'(\tilde{q}, \sigma)$. Then, there exists $C \in \delta(q, \sigma)$ s.t. $\tilde{C} = \min(C)$, then because $q \preceq r$, there is a $C_r \in \delta(r, \sigma)$ such that for all $(d, t) \in C_r$, there is a $(d, s) \in C$ with $s \preceq t$. Let s' be minimal in C s.t. $s' \preceq s \preceq t$ by transitivity of $\preceq$, we have $s' \preceq t$, then $(d, s') \in \tilde{(}C)$. It follows that S is a simulation relation. Since $(\tilde{q_0}, q_0) \in S$, it follows from Lemma 3.5 that $L(A') \subseteq L(A)$. □

Theorem 3.7. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. Suppose $C, C' \in \delta(q, \sigma)$, $C \neq C'$, and for all $(d, q') \in C'$ there is a $(d, q) \in C$ such that $q \preceq q'$. Let $A = (\Sigma, D, Q, q_0, \delta', \alpha)$ be an ABT for which δ' equals δ except that $\delta'(q, \sigma) = \delta(q, \sigma) \setminus C$. We have $L(A) = L(A')$.*

Proof. Again, we first show that every state q in A is equivalent to the state $\tilde{q}$ in A' with the same name. Note that we mark states in A' with a tilde. We have to show that for any states q, we have $q \preceq \tilde{q}$ and $\tilde{q} \preceq q$.

Let $\sigma \in \Sigma$ and $q \in Q$. Since $\delta'(\tilde{q}, \sigma) \subseteq \delta(q, \sigma)$, $\tilde{q} \preceq q$ holds trivially, and it follows from Lemma 3.5 that $L(A') \subseteq L(A)$. For the other direction, let $S = \{(q, \tilde{q}) \mid q \in Q\} \cup \{(s, \tilde{t}) \mid s \preceq t\}$. We show that S is a simulation relation. Suppose $(q, \tilde{q}) \in S$. Let $\sigma \in \Sigma$ and $C \in \delta(q, \sigma)$, then there is a $\tilde{C} \in \delta'(\tilde{q}, \sigma)$ s.t. either $C = \tilde{C}$, or there exists a $C' \in \delta(q, \sigma)$ s.t. forall $(d, q') \in C'$ there is a $(d, u) \in C$ with $u \preceq q'$ by definition of δ', and we have forall $(d, q') \in \tilde{C'}$ there is a $(d, u) \in C$ with $u \preceq q'$. Now, suppose $(s, \tilde{t}) \in S$ and $s \preceq t$. Let $\sigma \in \Sigma$ and $C_s \in \delta(s, \sigma)$, then there is a $C_t \in \delta(t, \sigma)$ s.t. forall $(d, t') \in C_t$, exists a $(d, s') \in C_s : s' \preceq t'$ (due to $s \preceq t$). Let $\tilde{C_t}$ be the transition in $\delta'(\tilde{t}, \sigma)$ that corresponds to C_t. Then we have $\forall (d, \tilde{t'}) \in \tilde{C_t}$ exists a $(d, s') \in C_s : s' \preceq \tilde{t'}$ due to transitivity of $\preceq$. It follows that S is a simulation relation, and with Lemma 3.5 we have $L(A) \subseteq L(A')$. □

The following theorem allows us to restrict the state space of an ABT to a set of representatives of every equivalence class under $\simeq$.

Theorem 3.8. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT, let $u, v \in Q$, and suppose $u \simeq v$ and $u \neq v$. Let $A' = (\Sigma, D, Q', q_0', \delta', \alpha')$, where $Q' = Q \setminus \{u\}$, $q_0' = v$ if $q_0 = u$ and $q_0' = q_0$ otherwise, δ' is obtained from δ by replacing u by v everywhere, and $\alpha' = \alpha \setminus \{u\}$. Then, $L(A) = L(A')$.*

Proof. When $u \simeq v$ and $(d, u) \in C$ for some transition $C \in \delta$, we can add (d, v) to C and then remove (d, u), both by Theorem 3.7. This means that u becomes unreachable and can be removed. □

We can simplify an ABT by repeated application of the last two theorems and removal of states that are no longer reachable from the initial state. The simulation relation can be computed in polynomial time, as can the optimizations. (Application of the theorems does not alter the simulation relation.)

3.2 Optimizations for Synthesis

3.2.1 Synthesis Algorithm

The goal of synthesis is to find a Moore machine M implementing an LTL specification φ (or to prove that no such M exists). Our approach follows that of [KV05], introducing optimizations that make synthesis much more efficient. The flow is as follows.

1. Construct an NBW A_{NBW} with $L(A_{\mathrm{NBW}}) = \{w \in (\Sigma \cup D)^\omega \mid w \not\models \varphi\}$. Let n' be the number of states of A_{NBW}. Note that in the worst case n' is exponential in $|\varphi|$ [WVS83].

2. Construct a UCT A_{UCT} with $L(A_{\mathrm{UCT}}) = T_{\Sigma,D} \setminus \Lambda(A_{\mathrm{NBW}}) = \Lambda(\varphi)$. Let n be the number of states of A_{UCT}; we have $n \leq n'$,

3. Perform the following steps for increasing k, starting with $k = 0$.

   a) Construct an AWT $A_{\mathrm{AWT}k}$ such that $L(A_{\mathrm{AWT}k}) \subseteq L(A_{\mathrm{UCT}})$ and $A_{\mathrm{AWT}k}$ has at most $n \cdot k$ states.

   b) Construct an NBT $A_{\mathrm{NBT}k}$ such that $L(A_{\mathrm{NBT}k}) = L(A_{\mathrm{AWT}k})$; $A_{\mathrm{NBT}k}$ has at most $(k+1)^{2n}$ states.

   c) Check for the nonemptiness of $L(A_{\mathrm{NBT}k})$. If the language is nonempty, proceed to Step 4.

   d) If $k = 2n^{2n+2}$, stop: φ is not realizable. Otherwise, proceed with the next iteration of the loop. (The bound on k follows from [Pit06].)

4. Compute a witness for the nonemptiness of $A_{\mathrm{NBT}k}$ and convert it to a Moore machine.

If the UCT constructed in Step 2 is weak, synthesis is much simpler: we complement the acceptance condition of A_{UCT} turning it into a UWT, a special case of an AWT. Then, we convert the UWT into an NBT A_{NBT} as in Step 3b. If $L(A_{\mathrm{NBT}})$ is nonempty, the witness is a Moore machine satisfying φ, if it is empty, φ in unrealizable. In this case, we avoid increasing k and the size of the NBT is at most 2^{2n}.

It turns out that in practice, for realizable specifications, the algorithm terminates with very small k, often around three. It should be noted that it is virtually impossible to prove the specification unrealizable using this approach, because of the high bound on k. The one exception is if the UCT is weak, because in that case we avoid the dependence on k altogether, as explained above.

In the following, we will describe the individual steps, discuss the optimizations that we use at every step, and show how to reuse information gained in one iteration of the loop for the following iterations.

3.2.2 NBW & UCT

We use Wring [SB00] to construct a nondeterministic generalized Büchi automaton for the negation of the specification. We then use the classic counting construction and the optimizations available in Wring to obtain a small NBW A_{NBW} with $L(A_{\mathrm{NBW}}) = (D \cup \Sigma)^\omega \setminus L(\varphi)$.

We construct a UCT A_{UCT} over Σ-labeled D-trees with $L(A_{\mathrm{UCT}}) = \Lambda((\Sigma \cup D)^\omega \setminus L(A_{\mathrm{NBW}}))$.

Definition 3.9. *[KV05] Given an NBW $A_{NBW} = (\Sigma, D, Q, q_0, \delta, \alpha)$, let UCT $A_{UCT} = (\Sigma, D, Q, q_0, \delta', \alpha)$, with for every $q \in Q$ and $\sigma \in \Sigma$*

$$\delta'(q, \sigma) = \big\{\{(d, q') \mid d \in D, q' \in \delta(q, d \cup \sigma)\}\big\}.$$

We have $L(A_{\mathrm{UCT}}) = T_{\Sigma,D} \setminus \Lambda(A_{\mathrm{NBW}})$.

We can reduce the size of $L(A_{\mathrm{UCT}})$ using game-based simulation and Theorem 3.3. Optimizing the UCT reduces the time spent optimizing the AWT and, most importantly, it may make the UCT weak, which means that we avoid the expensive construction of the AWT discussed in the next section. Because the UCT is small in comparison to the AWT and the NBT, optimization comes at a small cost.

Specifications are often of the form $\varphi \rightarrow \psi$, where φ is an assumption on the environment and ψ describes the allowed behavior of the system. When the system assertion ψ has

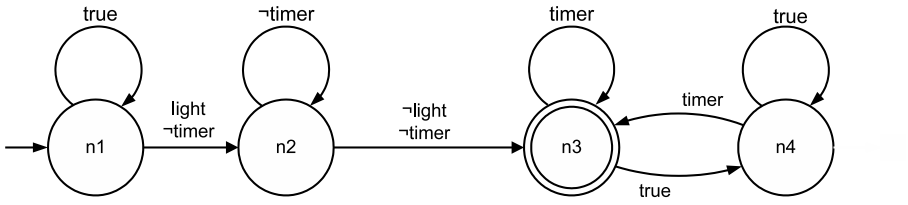

Figure 3.1: NBW for $\neg\varphi = \mathsf{G}(\mathsf{F}(\text{timer})) \wedge \mathsf{F}(\text{light} \wedge (\neg\text{light}\,\mathsf{R}\,\neg\text{timer}))$

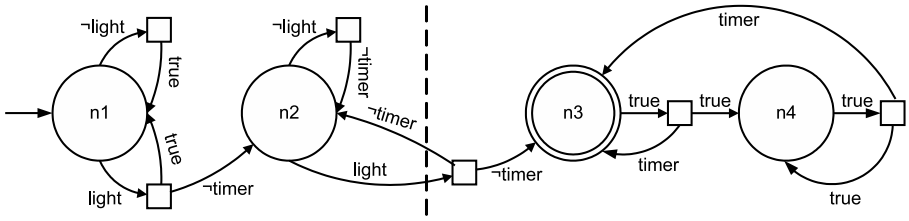

Figure 3.2: UCT for $\varphi = \mathsf{G}(\mathsf{F}(\text{timer})) \rightarrow \mathsf{G}(\text{light} \rightarrow (\text{light}\,\mathsf{U}\,\text{timer}))$

been violated, it may still be possible to satisfy the specification by a violation of the environment assumption. However, since the system cannot control the environment, states that check that φ is violated once the system assertion ψ has been violated are not necessary. Such states, among others, are removed by the game-based optimization.

Example 3.10. *Let $\varphi = \mathsf{G}\,\mathsf{F}\,\mathit{timer} \rightarrow \mathsf{G}(\mathit{light} \rightarrow (\mathit{light}\,\mathsf{U}\,\mathit{timer}))$. This formula is part of the specification of a traffic light controller and states that if the timer signal is set regularly, the light does not make a transition to zero unless the timer is high. The atomic propositions are partitioned into $I = \{\mathit{timer}\}$ and $O = \{\mathit{light}\}$. Fig. 3.1 shows a minimal NBW A_{NBW} accepting all words in $\neg\varphi$. The edges in the figure (and in the implementation) are labeled with cubes over the set of atomic propositions $I \cup O$. (A cube is a conjunct consisting of possibly negated atomic propositions.) An edge labeled with the cube c summarizes a set of edges, each labeled with a letter $w \subseteq I \cup O$ that is compatible with c.*

The UCT A_{UCT} that accepts all 2^O-labeled 2^I-trees not in $T(A_{NBW})$ is shown in Fig. 3.2. Circles denote states and boxes denote transitions. We label edges starting at circles with cubes over O ($\Sigma = 2^O$)) and edges from boxes with cubes over I ($D = 2^I$). The transition corresponding to a box C consists of all pairs (d, q) for which there is an edge from C to q such that d satisfies the label on the edge. In particular, if d satisfies none of the labels, the branch in direction d is finite, e.g., in state n_2 with light=0 and timer=1. Recall that

finite branches are accepting.

Even though the NBW is optimized, the UCT is not minimal: The tree languages $L({A_{UCT}}^{n3})$ *and* $L({A_{UCT}}^{n4})$ *are empty. Our algorithm finds both states and replaces them by transitions to* false*, removing the part of* A_{UCT} *to the right of the dashed line. Note that the optimizations cause the automaton to become weak.*

3.2.3 AWT

From the automaton A_{UCT} we construct an AWT $A_{\mathrm{AWT}k}$ such that $L(A_{\mathrm{AWT}k}) \subseteq L(A_{\mathrm{UCT}})$

Definition 3.11. *[KV05] Let* $A_{UCT} = (\Sigma, D, Q, q_0, \delta, \alpha)$*, let* $n = |Q|$ *and let* $k \in \mathbb{N}$*. Let* $[k]$ *denote* $\{0, \ldots, k\}$*. We construct* $A_{AWTk} = (\Sigma, D, Q', q_0', \delta', \alpha')$ *with*

$$\begin{aligned}
Q' &= \{(q,i) \in Q \times [k] \mid q \notin \alpha \text{ or } i \text{ is even}\},\\
q_0' &= (q_0, k),\\
\delta'((q,i),\sigma) &= \big\{\{(d_1,(q_1,i_1)),\ldots,(d_k,(q_k,i_k))\} \mid\\
&\qquad \{(d_1,q_1),\ldots,(d_k,q_k)\} \in \delta(q,\sigma),\\
&\qquad i_1,\ldots,i_k \in [i], \forall j : (q_j,i_j) \in Q'\big\}\\
\alpha' &= Q \times \{1,3,\ldots,2k-1\}.
\end{aligned}$$

We call i *the* rank *of an AWT state* (q,i)*.*

If $k = 2n^{n+2}$ we have $L(A_{\mathrm{AWT}k}) = \emptyset$ implies $L(A_{\mathrm{UCT}}) = \emptyset$ [KV05, Pit06].

We improve this construction in three ways: by using games, by merging directions, and by using simulation relations.

Game Simulation

We can use Theorem 3.3 to remove states from $A_{\mathrm{AWT}k}$.

Example 3.12. *Consider the UCT in Fig. 3.3 and the corresponding AWT in Fig. 3.4, using* $k = 5$*. The UCT (an artificial example) has been optimized using the techniques discussed in Section 3.2.2, and the AWT has been optimized in three ways: We have removed states that are not reachable from the initial state, we have merged directions, and we have removed edges. (The last two optimizations are explained in the next subsections). Still, there is ample room for improvement of the AWT.*

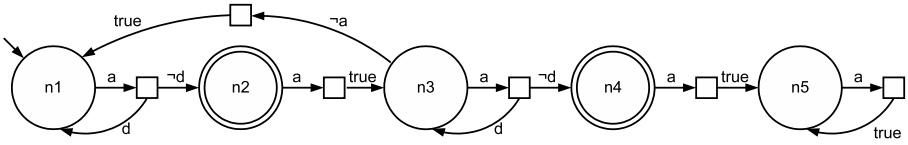

Figure 3.3: UCT that requires rank 5. Edges that are not shown (for instance from n_4 with label $\neg a$) correspond to labels that are not allowed.

Application of Theorem 3.3 removes the 12 states below the dashed line on the bottom left and the incident edges. This is a typical situation: each UCT state has an associated minimum rank.

It should be noted that $A_{\mathrm{AWT}k}$ has a layered structure: there are no states with rank j with a transition back to a state with a rank $i > j$. Furthermore, $A_{\mathrm{AWT}k+1}$ consists of $A_{\mathrm{AWT}k}$ plus one layer of states with rank $k+1$. This implies that game information computed for $A_{\mathrm{AWT}k}$ can be reused for $A_{\mathrm{AWT}k+1}$. A play is won (lost) in $A_{\mathrm{AWT}k+1}$ if it reaches a states that is won (lost) in $A_{\mathrm{AWT}k}$. Furthermore, if (q, j) is won, then so is (q, i) for $i > j$ when i is odd or j is even, which allows us to reuse some of the information computed for states with rank k when adding states with rank $k+1$. This follows from the fact that (q, i) simulates (q, j), as will be discussed in Subsection 3.2.3.

Merging Directions

Note that δ' may be drastically larger than δ: a single transition $C \in \delta(q, \sigma)$ yields $i^{|C|}$ transitions out of state $(q, i) \in Q'$. However, it turns out that it is not necessary to include conjuncts that send a copy to a (q, j) and (q, j') for $j \neq j'$. This is fortunate because it allows us to treat edges labeled with cubes over I as if they were labeled with directions.

Theorem 3.13. *Let $A''_{AWTk} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ be definite like A_{AWTk} in Definition 3.11, but with*

$$\delta''((q,i),\sigma) \;=\; \{C \;\in\; \delta'((q,i),\sigma) \;\mid\; \forall (d,(q,j)),(d',(q,j')) \;\in\; C,\ \textit{we have}\ j \;=\; j'\}.$$

We have $L(A''_{AWTk}) = L(A_{AWTk})$.

Proof. Because $\delta''(q, \sigma) \subseteq \delta'(q, \sigma)$, any tree accepted by $A''_{\mathrm{AWT}k}$ is also accepted by $A_{\mathrm{AWT}k}$.

Let R be a run of $A_{\mathrm{AWT}k}$, we will build a run R'' of $A''_{\mathrm{AWT}k}$. Run R'' is isomorphic to R, using a bijection that maps node v of R to node v'' of R''. Run R'' has the same labels

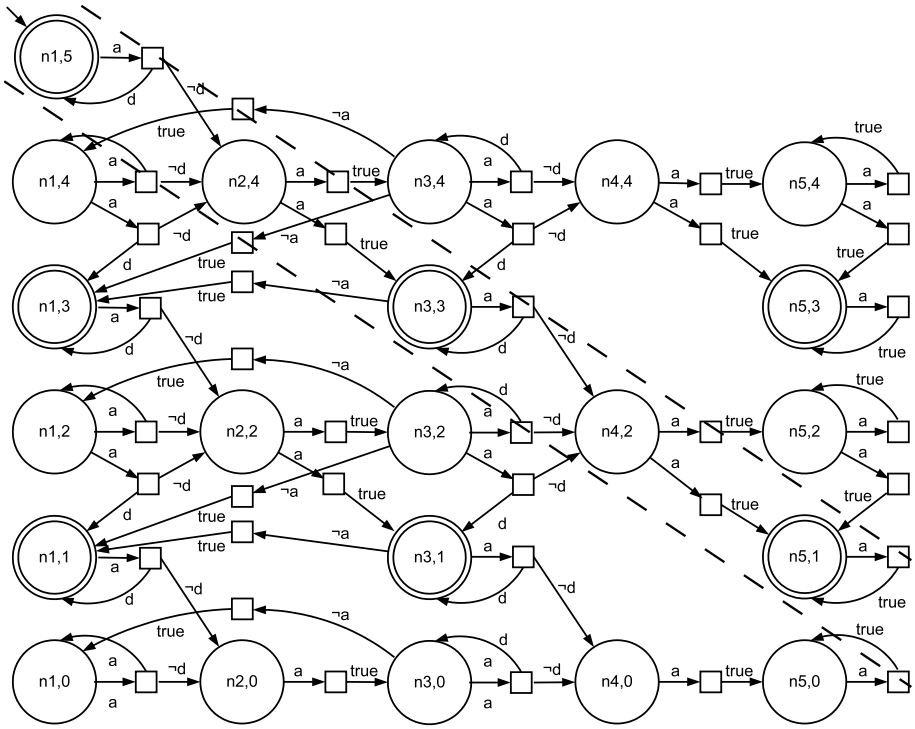

Figure 3.4: AWT for UCT in Figure 3.3.

as R with the following exception. If node v in R is labeled $(t,(q,i))$ and has children $(t',(q',i'))$ and $(t'',(q',i''))$ with $i' > i''$, then the corresponding children of node v'' of R'' are labeled $(t',(q',i'))$ and $(t'',(q',i'))$.

Because in $A_{\mathrm{AWT}k}$ state (q',i') has all transitions that (q',i'') has, R'' is a run of $A_{\mathrm{AWT}k}$, and because it satisfies the extra condition on δ'' it is also a run of $A''_{\mathrm{AWT}k}$. If R is accepting, then every infinite path π in R gets stuck in an odd rank w from some level l onwards. So starting from l, all children of nodes on π have rank at most w. That implies that the nodes on π in R'' have rank w starting at rank $l+1$ at the latest. Thus, π is still accepting, and since π is arbitrary, R'' is accepting as well. □

This theorem is key to an efficient implementation as it allows us to represent a set of pairs $\{(d_1,q),\ldots,(d_k,q)\}$ as $(\{d_1,\ldots,d_k\},q)$ whenever $\{d_1,\ldots,d_k\}$ can efficiently be represented by a cube over the input signals I.

Simulation minimization

We compute the simulation relation on $A_{\mathrm{AWT}k}$ and use Theorems 3.8, 3.6, and 3.7 to optimize the automaton. We would like to point out one optimization in particular.

Lemma 3.14. *For $(q,i),(q,j) \in Q'$ with $i \geq j$ such that i is odd or j is even, we have $(q,i) \succeq (q,j)$.*

Proof. If $i \geq j$ and i is odd or j is even, then if follows from Def. 3.11 that $\forall \sigma \in \Sigma : \delta((q,j),\sigma) \subseteq \delta((q,i),\sigma)$ and $(q,j) \in \alpha' \rightarrow (q,i) \in \alpha'$ holds, which implies $(q,i) \succeq (q,j)$ (Def. 3.4). □

Thus, for any σ, if i is even, we can remove all transitions $C \in \delta((q,i),\sigma)$ that include a pair (q',j) for $j \leq i-2$. If i is odd we can additionally remove all transitions that contain a pair (q',j) with $q \notin \alpha$ and $j = i-1$. That is, odd states become deterministic and for even states there are at most two alternatives to choose from.

Theorem 3.15. *Let $A''_{AWTk} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ be defined like A_{AWTk} in Definition 3.11, but with*

$$\delta''((q,i),\sigma) = \big\{C \in \delta(q,\sigma) \mid \forall (d',(q',i')) \in C : \\ i' \in \{i-1,i\} \wedge (i \text{ is even} \vee q' \in \alpha \vee i' = i) \wedge \forall (d'',(q',i'')) \in C : i' = i''\}.$$

then $L(A''_{AWTk}) = L(A_{AWTk})$.

Proof. Let $A'''_{\mathrm{AWT}k} = (\Sigma, D, Q', q'_0, \delta''', \alpha')$ be defined like the $A_{\mathrm{AWT}k}$ in Definition 3.11 but with $\delta''((q,i),\sigma) = \big\{C \in \delta(q,\sigma) \mid \forall (d',(q',i')) \in C : i' \in \{i-1,i\} \wedge (i \text{ is even} \vee q' \in \alpha \vee i' = i)\}$. We first prove that $L(A'''_{\mathrm{AWT}k}) = L(A_{\mathrm{AWT}k})$. Note that $\delta''' \subseteq \delta'$. Vice versa, we will associate a transition in δ''' to every transition in δ'. Given a state $(q,i) \in Q'$ and a label $\sigma \in \Sigma$, let $\{(d_1,(q_1,i_1)),\ldots,(d_k,(q_k,i_k))\}$ be a transition in $\delta'((q,i),\sigma)$. Suppose i is odd. For all $j \in \{1,\ldots,k\}$, if $q_j \in \alpha$, let $i'_j = i-1$, else let $i'_j = i$ then $(q_j,i'_j) \in Q'$ and $(q_j,i'_j) \succeq (q_j,i_j)$. Suppose i is even. For all $j \in \{1,\ldots,k\}$, if i_j is odd, take $i'_j = i-1$, else let $i'_j = i$. Again, $(q_j,i'_j) \in Q'$ and $(q_j,i'_j) \succeq (q_j,i_j)$. It should be clear that $\{(d_1,(q_1,i'_1)),\ldots,(d_k,(q_k,i'_k))\}$ is a transition in $\delta'''((q,i),\sigma)$. Thus, $L(A'''_{\mathrm{AWT}k}) = L(A_{\mathrm{AWT}k})$. It follows from Theorem 3.13 that $L(A''_{\mathrm{AWT}k}) = L(A_{\mathrm{AWT}k})$.

□

Example 3.16. *States* $(n_4, 4)$*,* $(n_5, 4)$*, and* $(n_5, 3)$ *(top right) are simulation equivalent with* $(n_4, 2)$*,* $(n_5, 2)$*, and* $(n_5, 1)$*, respectively. Using Theorem 3.8, we can remove states* $(n_4, 4)$*,* $(n_5, 4)$*, and* $(n_5, 3)$*, and redirect incoming edges to equivalent states.*

Furthermore, the previous removal of the states on the bottom left implies that $(n_3, 4) \preceq (n_3, 3)$*. Since* $(n_2, 4)$ *has identical transitions to* $(n_3, 4)$ *and* $(n_3, 3)$*, Theorem 3.7 allows us to remove the transition to* $(n_3, 4)$*. Thus,* $(n_3, 4)$ *becomes unreachable and can be removed. The same holds for* $(n_5, 2)$ *for a similar reason. (This optimization also allows us to remove states* $(n_4, 4)$*,* $(n_5, 4)$*, and* $(n_5, 3)$*, but Theorem 3.7 is not in general stronger than Theorem 3.8.)*

The optimization of the edges due to Theorem 3.15 is already shown in Fig. 3.4. Consider, for instance, the transition from $(n_2, 4)$ *to* $(n_3, 4)$*.*

Altogether, we have reduced the number of states in the AWT from 22 to 5. The removal of edges is equally important as it reduces nondeterminism and makes the translation to an NBT more efficient.

3.2.4 NBT

The next step is to translate $A_{\mathrm{AWT}k}$ to an NBT $A_{\mathrm{NBT}k}$ with the same language. Assume that $A_{\mathrm{AWT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. We first need some additional notation. For $S \subseteq Q$ and $\sigma \in \Sigma$ let

$$sat(S, \sigma) = \{C \in 2^{D \times Q} \mid C \text{ is a minimal set such that } \forall q \in S \, \exists C_q \in \delta(q, \sigma) : C_q \subseteq C\}.$$

For $(S, O) \in 2^Q \times 2^Q$, let

$$sat((S, O), \sigma) = \{(S', O') \in 2^Q \times 2^Q \mid S' \in sat(S, \sigma), O' \in sat(O, \sigma), O' \subseteq S'\}.$$

Furthermore, let $S_d = \{s \mid (d, s) \in S\}$, let $O_d = \{s \mid (d, s) \in O\}$. Let $C_N(S, O) = \{(d, (S_d, O_d \setminus \alpha)) \mid d \in D\}$ and let $C_\emptyset(S) = \{(d, (S_d, S_d \setminus \alpha)) \mid d \in D\}$.

Definition 3.17. *[KV05, MH84] Let* $A_{NBTk} = (\Sigma, D, Q', q'_0, \delta', \alpha')$ *with*

$$\begin{aligned} Q' &= 2^Q \times 2^{Q \setminus \alpha} \\ q'_0 &= (\{q_0\}, \emptyset) \\ \delta'((S,O),\sigma) &= \begin{cases} \{C_N(S',O') \mid (S',O') \in sat((S,O),\sigma)\} & \text{if } O' \neq \emptyset \\ \{C_\emptyset(S') \mid S' \in sat(S,\sigma)\} & \text{otherwise} \end{cases} \\ \alpha' &= 2^Q \times \{\emptyset\} \end{aligned}$$

We have $L(A_{\mathrm{NBT}k}) = L(A_{\mathrm{AWT}k})$.

We improve this construction in three ways. First, we make use of the simulation relation on the AWT to reduce the size of the NBT. Second, we remove *inconsistent states*, and third, we compute the NBT on the fly.

Simulation-Based Optimization

We can use the simulation relation that we have computed on $A_{\mathrm{AWT}k}$ to approximate the simulation relation on $A_{\mathrm{NBT}k}$. This is a simple extension of Fritz' result for word automata [Fri03].

Given a direct simulation relation $\preceq_{\mathrm{AWT}}$ for $A_{\mathrm{AWT}k}$, we define the simulation relation $\preceq' \subseteq Q' \times Q'$ on $A_{\mathrm{NBT}k}$ as

$$(S_1, O_1) \preceq' (S_2, O_2) \text{ iff } \forall q_2 \in S_2 \; \exists q_1 \in S_1 : q_1 \preceq_{\mathrm{AWT}} q_2 \wedge (q_2 \in O_2 \rightarrow q_1 \in O_1).$$

Lemma 3.18. *Let* $\preceq$ *be the full (direct) simulation relation on* A_{NBTk}*, then* $(S_1, O_1) \preceq' (S_2, O_2)$ *implies* $(S_1, O_1) \preceq (S_2, O_2)$.

Proof. We prove that $\preceq$' is a simulation relation (Def. 3.4) on $A_{\mathrm{NBT}k}$ by showing that $(S_1, O_1) \preceq' (S_2, O_2)$ implies (1) $(S_1, O_1) \in \alpha' \rightarrow (S_2, O_2) \in \alpha'$, (2) $\forall \sigma \in \Sigma \; \forall C_1 \in \delta((S_1, O_1), \sigma) \; \exists C_2 \in \delta((S_2, O_2), \sigma) \; \forall (d', (S'_2, O'_2)) \in C_2 \; \exists (d', (S'_1, O'_1) \in C_1 : (S'_1, O'_1) \preceq' (S'_2, O'_2)$.

(1) $(S_1, O_1) \in \alpha'$ implies that $O_1 = \emptyset$. Due to $(S_1, O_1) \preceq' (S_2, O_2)$, $O_2 = \emptyset$, which implies that $(S_2, O_2) \in \alpha'$.

(2) Suppose $O_1 \neq \emptyset$. Given a letter $\sigma \in \Sigma$ and a transition $C_1 \in \delta'((S_1, O_1), \sigma)$. We have $C_1 = C_N(S'_1, O'_1) = \{(d, (S'_{1d}, O'_{1d} \setminus \alpha)) \mid d \in D\}$, where $S'_1 \in sat(S_1, \sigma)$, $O'_1 \in sat(O_1, \sigma)$, and $O'_1 \subseteq S'_1$ (Def. 3.17). Thus, $\forall q_1 \in S_1 \; \exists C_{q_1} \in \delta(q_1, \sigma) : C_{q_1} \subseteq S'_1$. Now, we will pick transitions for all $q_2 \in S_2$. Since $\forall q_2 \in S_2 \; \exists q_1 \in S_1 : q_1 \preceq_{\mathrm{AWT}} q_2$, there

is a transition $C_{q_2} \in \delta(q_2, \sigma)$ s.t. $\forall (d, q_2') \in C_{q_2}\ \exists (d, q_1') \in C_{q_1} : q_1' \preceq_{\text{AWT}} q_2'$. So there exists a set $S_2' \in sat(S_2, \sigma)$ such that $\forall (d, q_2') \in S_2'\ \exists (d, q_1') \in S_1' : q_1' \preceq_{\text{AWT}} q_2'$. In an analogous manner, since $O_2 \subseteq S_2$ and $\forall q_2 \in O_2\ \exists q_1 \in O_1 : q_1 \preceq_{\text{AWT}} q_2$, we derive an $O_2' \subseteq S_2'$ s.t. $O_2' \in sat(O_2, \sigma)$ and $\forall (d, q_2') \in O_2'\ \exists (d, q_1') \in O_1' : q_1' \preceq_{\text{AWT}} q_2'$. Now we build $C_2 = \{(d, (S_{2d}', O_{2d}' \setminus \alpha)) \mid d \in D\} = C_N(S_2', O_2') \in \delta'((S_2, O_2), \sigma)$. Furthermore, $\forall (d, (S_2', O_2')) \in C_2\ \exists (d, (S_1', O_1')) \in C_1\ \forall q_2' \in S_2'\ \exists q_1' \in S_1' : q_1' \preceq_{\text{AWT}} q_2'$ and $q_2' \in O_2' \rightarrow q_1' \in O_1'$, and thus $(S_1', O_1') \preceq' (S_2', O_2')$. For $O_1 = \emptyset$, the proof is similar. □

In particular, for a state $(S, O) \in Q'$, if $q, q' \in S$, $q \preceq_{\text{AWT}} q'$, and $q' \in O \rightarrow q \in O$, then $(S, O) \simeq (S \setminus \{q'\}, O \setminus \{q'\})$. Thus, by Theorem 3.8, we can remove q' from such sets. Likewise, if $A_{\text{NBT}k}$ contains two simulation equivalent states (S, O) and (S', O') we keep only one (preferring the one with smaller cardinality). Finally, we can use Theorem 3.7 to remove states that have a simulating sibling.

Removing Inconsistent States

In [KV05], it is shown that it is not necessary to include states (S, O) such that (q, i) and $(q, j) \in S$ with $i \neq j$. This implies that we can use the following optimization.

Theorem 3.19. *[KV05] Let $A'_{NBTk} = (\Sigma, D, Q'', (\{q_0\}, \emptyset), \delta'', 2^Q \times \emptyset)$ be as in Definition 3.17, with $Q'' = Q \setminus \{(S, O) \mid \exists (q, i), (q, j) \in S : i \neq j\}$. The transition relation δ'' is obtained from δ' by replacing, for all $C \in \delta'(q, \sigma)$ and all $(S, O) \in C$, state (S, O) by (S', O') where S' is obtained from S by removing all states (q, j) with j not minimal and O' is obtained from O by replacing $(q, j) \in O$ by (q, j') if $(q, j) \notin S'$ and $(q, j) \in S'$.*

We have $L(A'_{NBTk}) = L(A_{NBT})$.

This is an important theorem as it reduces the number of states in the NBT to $(k+1)^{2n}$ instead of 2^{nk}, where n is the number of states in A_{UCT}.

On-the-Fly Computation

Suppose $A_{\text{NBT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. Instead of building $A_{\text{NBT}k}$ in full, we construct an NBT $A'_{\text{NBT}}[k] = (\Sigma, D, Q', q_0, \delta', \alpha \cap Q')$ such that $q_0 \in Q' \subseteq Q$ and for $q \in Q'$, either $\delta'(q, \sigma) = \delta(q, \sigma)$ for all σ or $\delta'(q, \sigma) = \emptyset$ for all σ. Thus, $L(A'_{\text{NBT}k}) \subseteq L(A_{\text{NBT}k})$. If $L(A'_{\text{NBT}k}) \neq \emptyset$, the witness of nonemptiness of $L(A'_{\text{NBT}k})$ is a witness of nonemptiness of $L(A_{\text{NBT}k})$. Otherwise, we select a state $q \in Q'$ with $\delta'(q, \sigma) = \emptyset$ and *expand* it, setting $\delta'(q, \sigma) = \delta(q, \sigma)$, introducing the necessary states to Q'.

Our current heuristic expands states in a breadth first manner, which is quite effective. It may be beneficial to expand certain state first, say states with a low cardinality or with high ranks.

3.2.5 Moore Machine

We use the game defined in Section 3.1.1 to compute language emptiness on the $A_{\mathrm{NBT}k}$. Since $A_{\mathrm{NBT}k}$ is nondeterministic, all states in the winning region have a nonempty language. If the initial state is in the winning region, the language of $A_{\mathrm{NBT}k}$ is not empty and we extract a witness.

Since $A_{\mathrm{NBT}k}$ is a subset of $A_{\mathrm{NBT}k+1}$, we can reuse all results obtained when computing language emptiness on $A_{\mathrm{NBT}k}$ to compute language emptiness on $A_{\mathrm{NBT}k+1}$.

Moreover, it follows from Miyano and Hayashi's construction that if $L(A^{(S,O)}) \neq \emptyset$ and $S \subseteq S'$, then $L(A^{(S',O')}) \neq \emptyset$. We may use this fact to further speed up the computation of language emptiness, and especially to reuse information obtained computing language emptiness on $A_{\mathrm{NBT}k}$ for larger k.

A witness for nonemptiness corresponds to a winning *attractor strategy* [Tho95] (cf. Section 2.4). The winning strategy follows the μ-iterations of the final ν-computation of $W_B(\alpha)$: From a state $q \notin \alpha$ we go to a state q' from which the protagonist can force a shorter path to an accepting state. In an accepting state we move back to an arbitrary state in the winning region.

If a strategy exists, it corresponds to a complete Σ-labeled D-tree and thus to a Moore machine M. The states of M are the states of $A_{\mathrm{NBT}k}$ that are reachable when the strategy is followed, and the edges are given by the strategy.

To minimize the strategy, we compute the simulation relation and apply Theorem 3.8, which is equivalent to using the classical FSM minimization algorithm [HU79]. Thus, the optimized strategy is guaranteed to be minimal with respect to its given I/O language. The output of our tool is a state machine described in Verilog that implements this strategy.

3.3 Experimental Results

Our tool, Lily[1], is an implementation of the synthesis algorithm and the optimizations described in this paper. It is implemented on top of Wring [SB00, GBS02] and written in

[1] http://www.ist.tugraz.at/staff/jobstmann/lily/

Perl. We have run our experiments on a Linux machine with a 2.8 GHz Pentium 4 CPU and 2 GB of RAM.

Example 3.20. *Consider the following formula.*

$$\mathsf{G}\,\mathsf{F}\,\mathit{timer} \rightarrow \big(\mathsf{G}(\mathit{hl} \rightarrow (\mathit{hl}\,\mathsf{U}\,\mathit{timer})) \wedge \mathsf{G}(\mathit{fl} \rightarrow (\mathit{fl}\,\mathsf{U}\,\mathit{timer})) \wedge \mathsf{G}(\neg \mathit{hl} \vee \neg \mathit{fl}) \wedge \mathsf{G}(\mathit{car} \rightarrow \mathsf{F}\,\neg \mathit{car} \vee \mathit{fl}) \wedge \mathsf{G}\,\mathsf{F}\,\mathit{hl} \wedge \mathsf{G}(\mathit{hl} \rightarrow (\mathit{hl}\,\mathsf{W}\,\mathit{car}))\big).$$

The formula specifies a small traffic light system consisting of two lights. The highway light is green iff $\mathit{hl} = \mathsf{true}$*, and similarly for the crossing farm road and fl. Signals hl and fl form the output. The input signal car indicates that a car is waiting at the farm road and timer represents the expiration of a timer. The specification assumes that the timer expires regularly. It requires that a green light stay green until the timer expires. Furthermore, one of the lights must always be red, every car at the farm road is eventually allowed to drive on (unless it disappears), the highway light is regularly set to green, and it does not become red until there is a car that wants to cross the highway. The specification is realizable and the design generated by Lily is shown in Figure 3.5.*

We show the effectiveness of the various optimizations by synthesizing 20 handwritten formulas, mostly different arbiters and some traffic light controllers. Our examples are small, but we show a significant improvement over the straightforward implementation.

For realizable formulas, we have verified the output of our tool with a model checker. We can verify that a formula is unrealizable by synthesizing an environment that forces the specification to be violated. Since a system is a Moore machine, an environment is a Mealy machine. Note that φ can be realized by a Mealy machine iff φ' can be realized by a Moore machine, where φ' is obtained from φ by replacing all occurences of an output o by $\mathsf{X}\,o$. This means that we can apply our approach to check that the environment is realizable.

We show our results in Table 3.1. (The specification in example 3.20 has number 9.) In the column labeled *T,B,AP*, we provide the number of temporal operators, the number of Boolean operators, and the number of atomic propositions in the formula. We also give the strength in Column *strength* and the number of states and edges of the optimized NBW in Column *NBW* using the format states(edges).

```
module tlc(hl,fl,clk,car,timer);
  input  clk,car,timer;
  output hl,fl;
  wire clk,hl,fl,car,timer;
  reg [0:0] state;

  assign hl = (state == 0);
  assign fl = (state == 1);

  initial begin
    state = 0;
  end
  always @(posedge clk) begin
    case(state)
    0: begin
      if (timer==0) state = 0;
      if (timer==1 && car==1) state = 1;
      if (car==0) state = 0;
    end
    1: begin
      if (timer==1) state = 0;
      if (timer==0) state = 1;
    end
    endcase
  end
endmodule //tlc
```

Figure 3.5: Generated design for a simple traffic light

Table 3.1: Experimental results for Lily

No	*T,B,AP*	*Strength*	*NBW*	*Plain*	*Plain+dm*	*UCT+dm*	*AWT+dm*	*NBT+dm*	*All*	*NBT-plain+dm*	*NBT-all*	*Witness*
1	12,5,4	weak	8(14)	2.94 s	1.96 s	0.71 s	0.73 s	2.53 s	0.38 s	48(192)	0(0)	n. r.
2	6,3,4	strong	7(15)	> mem	1689.13 s	575.70 s	2.20 s	2.43 s	1.25 s	3943(973764)	6(28)	2(3)
3	4,4,4	strong	12(44)	> mem	> mem	3.73 s	>3600 s	> 3600 s	3.95 s	-	0(0)	n. r.
4	9,5,4	strong	6(11)	12.90 s	3.14 s	2.63 s	0.61 s	0.74 s	0.66 s	95(1104)	8(37)	6(11)
5	9,6,4	weak	7(19)	0.61 s	0.62 s	0.68 s	0.64 s	0.62 s	0.69 s	14(65)	14(65)	10(25)
6	15,12,4	weak	9(30)	3.43 s	2.94 s	3.01 s	2.97 s	3.26 s	3.15 s	58(502)	58(502)	43(145)
7	13,5,5	strong	7(15)	69.14 s	20.07 s	12.83 s	1.23 s	3.42 s	1.24 s	384(9564)	26(164)	15(31)
8	20,9,7	strong	9(22)	> 3600 s	258.81 s	294.29 s	6.41 s	13.32 s	5.98 s	1810(75264)	80(811)	41(109)
9	11,9,4	strong	9(19)	> mem	113.08 s	9.29 s	14.90 s	5.57 s	8.90 s	1079(81700)	101(840)	2(5)
10	12,5,4	weak	8(12)	3.64 s	1.28 s	0.62 s	0.70 s	1.31 s	0.35 s	28(116)	0(0)	n. r.
11	23,21,5	strong	24(90)	> mem	> mem	> 3600 s	17.57 s	94.29 s	15.91 s	-	31(167)	6(13)
12	40,24,8	weak	14(84)	201.18 s	219.38 s	58.82 s	61.02 s	37.95 s	32.41 s	251(88016)	26(456)	5(41)
13	10,9,4	strong	24(134)	> mem	> 3600 s	> 3600 s	522.21 s	> 3600 s	46.37 s	360(440093)	23(127)	17(75)
14	14,9,4	weak	13(34)	7.15 s	6.09 s	4.08 s	4.44 s	4.54 s	2.39 s	82(730)	21(92)	7(22)
15	16,10,10	weak	24(68)	90.63 s	68.49 s	13.60 s	13.10 s	17.00 s	8.29 s	618(8326)	27(142)	n. r.
16	16,13,4	weak	18(50)	8.65 s	6.33 s	6.45 s	7.23 s	7.71 s	3.93 s	142(1492)	17(112)	8(31)
17	19,13,4	weak	25(69)	24.52 s	21.88 s	11.28 s	12.97 s	14.46 s	8.18 s	368(3716)	25(162)	12(54)
18	16,17,4	weak	17(45)	11.14 s	9.14 s	4.58 s	5.51 s	6.05 s	4.14 s	118(730)	13(53)	8(23)
19	28,14,9	strong	11(30)	> mem	> mem	> 3600 s	72.78 s	483.38 s	39.26 s	755(83106)	242(3834)	124(444)
20	8,6,2	strong	7(14)	3.21 s	2.98 s	1.24 s	1.13 s	0.82 s	0.73 s	112(1187)	5(10)	2(3)

The next six columns report time used with different combinations of optimizations. We write "> mem" if the run has exceeded the memory limit and "> 3600s" if it did not finish within one hour.

In Column *Plain* we give the time using only one optimization: transitions from a state with rank i go only to states with ranks $i-1$ and $i-2$, not to smaller ranks. Without this optimization, synthesis is impossible on most examples. Column *Plain+dm* shows the time used if we apply Theorem 3.13, which allows us to merge directions. In Column *UCT+dm*, we give the time usage of runs in which we applied game optimization (Theorem 3.3) on the UCT and we merged the directions. We show the results for applying all the optimizations suggested in Section 3.2.3 on the AWT in Column *AWT+dm*. Column *UCT+dm* shows the time used if we apply the NBT optimizations and merge directions.

In the Column *All* we give the results for combining all optimizations. For realizable formulas, the number of states and edges of the design generated during those runs is given in the column labeled with *Witness*. We write n.r. if a formula is not realizable. The generated designs are minimized as described in Section 3.2.5.

In Column *NBT-all* we give the size (states(edges)) of the NBT using all optimizations. In contrast, in Column *NBT-plain+dm*, we show the size of the NBTs generated by the runs where we used only direction merging.

Our examples show that if the NBW for $\neg\varphi$ is strong and we have to construct the AWT, the straightforward implementation often fails to complete. (See Column *Plain*.) Five examples exceed the memory limit due to the expensive construction of the transition relation of the AWT. The optimizations due to Theorem 3.13, which allows us to merge directions, are necessary to overcome this limit. (See Column *Plain+dm*.)

If we optimize the UCT according to Theorem 3.3, we speed up about half of the examples by a factor of two or more. (Compare Column *Plain+dm* and *UCT+dm*, e.g., Line 2,3, or 9). The game based minimization is very effective if the formula or part of it is not realizable and is very cheap to compute. In particular for Examples 3 the difference is huge because the language of the optimized UCT is empty.

None of the optimizations on the AWT was very effective on their own. For example we still had two timeouts with game based optimization. The same holds for simulation based optimization. Further simplification of δ' according to Theorem 3.15 resulted in one timeout. Nevertheless, these optimizations are very efficient when combined as can be seen in Column *AWT+dm*.

The simulation-based optimizations for the NBT (explained in Section 3.2.4) typically

reduce the size of the resulting NBT between 60% and 90%. For example, in Example 9 the size of the NBT is reduced from about 1000 states to 300. For this example, the on-the-fly game computation further reduces the number of NBT-states to about 100. Only in the small examples is the entire state space of the NBT needed to compute a witness. (Cf. Column *NBT-plain+dm,NBT-all,Witness*).

In our examples, the UCT optimizations were crucial once to turn a strong UCT into a weak one (in this case with an empty language). In all other cases, they are outperformed by the AWT optimizations. The results of the NBT optimization are mixed: they can fail (Example 3 and 13) or perform better than any other optimization (Example 9 and 12). The combination of all optimizations is needed to finish all examples.

Chapter 4

Repair of Finite-State Systems

This chapter is based on [JGB05], [BJP05], and [JSGB07].

In this chapter we present a conservative method to automatically fix faults in a finite state program by considering the repair problem as a game. The game consists of the product of a modified version of the program and an automaton representing the LTL specification.

In 4.1, we describe how to obtain a program game from a program and a suspicion of a fault. The product of the program game and the automaton for the LTL formula is a Büchi game. If the product game is winning, it has a memoryless winning strategy. In 4.2 we show how to construct a finite state strategy for the program game from the strategy for the product game and we discuss under which conditions we can guarantee that the product game is winning. A finite state strategy for the program game corresponds to a repair that adds states to the program. Since we want a repair that is as close as possible to the original program, we search for a memoryless strategy. In 4.3, we show that it is NP-complete to decide whether a memoryless strategy exists, and in 4.4, we present a heuristic to construct a memoryless strategy and show how to extract a repair from a memoryless strategy. In Section 4.7 we present our experimental results.

4.1 Constructing a Game

Suppose that we are given a program that does not fulfill its LTL specification φ. Suppose furthermore that we have an idea which variables or lines may be responsible for the failure, for instance, from a diagnosis tool.

A program corresponds to an LTL game $K = (S, s_0, I, \{c\}, \delta, \lambda, \varphi)$. The set of system choices is a singleton (the game models a deterministic system) and the acceptance condition is the specification. Given an expression e in which the right-hand side (RHS) may be incorrect, we turn K into a *program game* G by *freeing* the value of this expression. That is, if Ω is the domain of the expression e, we change the system choice to $C' = C \times \Omega$ and let the second component of the system choices define the value of the expression. If we can find a winning memoryless strategy for G, we have determined a function from the state of the program to the proper value of the RHS, i.e., a repair.

We do not consider other fault models, but these can be easily added. The experimental results show that we may find good repairs even for programs with faults that we do not model.

4.2 Finite State Strategies

Given two games $G = (S, s_0, I_G, C_G, \delta_G, \lambda_G, F_G)$ and $A = (Q, q_0, 2^{AP}, C_A, \delta_A, \lambda_A, F_A)$, let the *product game* be $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, F)$, where $\delta((s, q), i_G, (c_G, c_A)) = \big(\delta_G(s, i_G, c_G), \delta_A(q, \lambda_G(s), c_A)\big)$, $\lambda(s, q) = \lambda_G(s)$, and $F = \{(s_0, q_0), (s_1, q_1), \cdots \mid s_0, s_1, \cdots \in F_G \text{ and } q_0, q_1, \cdots \in F_A\}$. Intuitively, the output of G is fed to the input of A, and the winning conditions are conjoined (see Figure 4.1). Therefore, the output language of the product is the intersection of the output language of the first game and the input language of the second.

Lemma 4.1. *For games G, A, $L(G \triangleright A) = L(G) \cap I(A)$.*

Lemma 4.2. *Let G and A be games. If a finite state winning strategy σ for $G \triangleright A$ exists, then there is a finite state winning strategy σ' for G such that for all plays π of G according to σ', $\lambda(\pi) \in L(G)$ and $\lambda(\pi) \in I(A)$.*

Proof. Let $\sigma = (V, v_0, \mu)$ be a winning finite state strategy for $G \triangleright A$. Note that $\mu : (S \times Q) \times V \times I_G \to 2^{(C_G \times C_A) \times V}$. Let $\sigma' = (Q \times V, (q_0, v_0), \mu')$ with $\mu' : S \times (Q \times V) \times I_G \to 2^{C_G \times (Q \times V)}$ such that

$$\mu'(s, (q, v), i_G) = \big\{(c_G, (q', v')) \mid \exists c_A : ((c_G, c_A), v') \in \mu((s, q), v, i_G) \wedge q' = \delta_A(q, \lambda_G(s), c_A)\big\}.$$

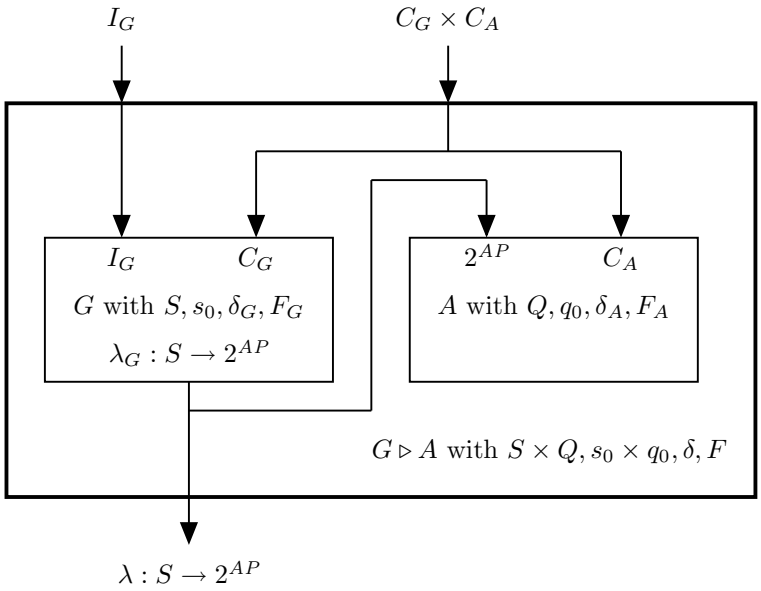

Figure 4.1: Product game $G \triangleright A$

Let $\pi' = s_0(q_0, v_0) \overset{i_{G0}c_{G0}}{\longrightarrow} s_1(q_1, v_1)\dots$ be a play of G according to σ'. Then there are c_{Ai} such that $\pi = (s_0, q_0)v_0 \overset{i_{G0}(c_{G0},c_{A0})}{\longrightarrow} (s_1, q_1)v_1\dots$ is a play of $G \triangleright A$ according to σ. Because σ is winning, π is winning and $\lambda(\pi) \in L(G) \cap I(A)$. Since $\lambda(\pi') = \lambda(\pi)$ it follows that $\lambda(\pi') \in L(G) \cap I(A)$, and because π' is arbitrary, this implies that σ' is winning. □

The finite state strategy σ' is the product of A and the finite state strategy σ for $G \triangleright A$. If $F_G = S^\omega$, then σ' is the winning strategy for the game G with the winning condition defined by A. The following theorem is an example of *game simulation* (cf. [Tho95]).

Theorem 4.3. *Let $G = (S, s_0, I, C, \delta, \lambda, \varphi)$ be an LTL game, let G' be as G but with the winning condition S^ω, and let $A = (Q, q_0, 2^{AP}, C, \delta, \lambda, B)$ be a Büchi game constructed from an NBW for φ by modeling the nondeterminism of the automaton with the system choice. (Note that the input language of A is the set of words satisfying φ). If there is a memoryless winning strategy for the Büchi game $G' \triangleright A$ then there is a finite state winning strategy for G.*

Proof. Follows from Lemma 4.2. □

Note that the converse of the theorem does not hold. In fact, Harding [Har05] shows that we are guarantee to find a winning strategy iff the game fulfills the property and the

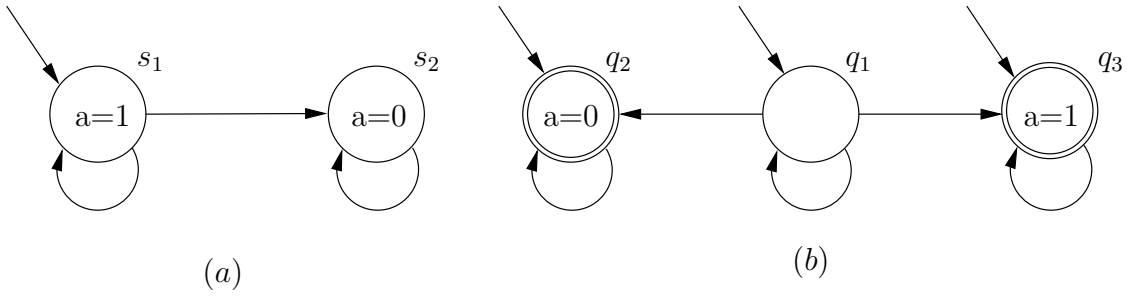

Figure 4.2: (a) Game in which the environment can assign the value for variable a. (b) automaton for $\mathsf{F}\,\mathsf{G}(a = 1) \vee \mathsf{F}\,\mathsf{G}(a = 0)$.

automaton is *trivially determinizable*, i.e., we can make it deterministic by removing edges without changing the language.

For example, there is no winning strategy for the game shown in Fig. 4.2. If the automaton for the property $\mathsf{F}\,\mathsf{G}(a = 1) \vee \mathsf{F}\,\mathsf{G}(a = 0))$ moves to the state q_3, the environment can decide to move to s_2 (set $a = 0$), a move that the automaton cannot match. If, on the other hand, the automaton waits for the environment to move to s_2, the environment can stay in s_1 forever and thus force a non-accepting run. Hence, although the game fulfills the formula, we cannot give a strategy. Note that this problem depends not only on the structure of the automaton, but also on the structure of the game. For instance, if we remove the edge from s_1 to s_2, we can give a strategy for the product.

In general, the translation of an LTL formula to a deterministic automaton requires a doubly exponential blowup and the best known upper bound for deciding whether a translation is possible is EXPSPACE [KV98]. To prevent this blowup, we can either use heuristics to reduce the number of nondeterministic states in the automaton [ST03], or we can use a restricted subset of LTL. Maidl [Mai00] shows that translations in the style of [GPVW95] (of which we use a variant [SB00]) yield deterministic automata for the formulas in the set $\mathrm{LTL}^{\mathrm{det}}$, which is defined as follows: If φ_1 and φ_2 are $\mathrm{LTL}^{\mathrm{det}}$ formulas, and p is a predicate, then p, $\varphi_1 \wedge \varphi_2$, $\mathsf{X}\,\varphi_1$, $(p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)$, $(p \wedge \varphi_1)\,\mathsf{U}(\neg p \wedge \varphi_2)$ and $(p \wedge \varphi_1)\,\mathsf{W}(\neg p \wedge \varphi_2)$ are LTL^{det} formulas. Note that this set includes invariants $(\mathsf{G}\,p)$ and $\neg p\,\mathsf{U}\,p = \mathsf{F}\,p$. $\mathrm{LTL}^{\mathrm{det}}$ describes the intersection of LTL and CTL. In fact, deterministic Büchi automata describe exactly the properties expressibly in the alternation-free μ-calculus, a superset of CTL [KV98].

Alur and La Torre [AL01] define a set of LTL fragments for which we can compute

deterministic automata using a different tableau construction. They are classified by means of the operators used in their subformulas. (On the top level, negation and other Boolean connectives are always allowed.) Alur and La Torre give appropriate constructions for the classes LTL($\mathsf{F}, \wedge$) and LTL($\mathsf{F}, \mathsf{X}, \wedge$). In contrast, for LTL($\mathsf{F}, \vee, \wedge$) and LTL($\mathsf{G}, \mathsf{F}$) they show that the size of a corresponding deterministic automaton is necessarily doubly exponential in the size of the formula. Since trivially deterministic automata can be made deterministic by removing edges, they can be no smaller than the smallest possible deterministic automaton and thus there are no exponential-size trivially deterministic automata for the latter two groups.

4.3 Memoryless Strategies are NP-Complete

The finite state strategy corresponding to the product game defined in the last section may be quite awkward to implement as it requires the program to keep track of the state of the automaton. This means that we have to ad extra state, which we have to update whenever a variable changes that is referenced in the specification. Instead, we wish to construct a memoryless strategy. Such a strategy corresponds to a repair that does not require additional state.

In this Section we prove that deciding whether a game with a Büchi acceptance condition has a memoryless strategy is NP complete. From this it follows that there is no algorithm to decide whether an LTL game has a memoryless strategy that is polynomial in the size of the game graph, unless P = NP.

Theorem 4.4. *Deciding whether a game with a winning condition defined by a Büchi automaton has a memoryless winning strategy is NP-complete.*

Proof. It follows from the results of Fortune, Hopcroft, and Wyllie [FHW80] that given a directed graph G and two nodes v and w, it is NP-complete to compute whether there are node-disjoint paths from v to w and back. Assume that we build a game G' based on the graph G. The acceptance condition is that v and w are visited infinitely often, which can easily be expressed by a Büchi automaton. Since the existence of a memoryless strategy for G' implies the existence of two node-disjoint paths from v to w and back, we can deduce the Theorem 4.4. □

4.4 Heuristics for Memoryless Strategies

Since we cannot compute a memoryless strategy in polynomial time, we use a heuristic. Given a memoryless strategy for the product game, we construct a strategy that is common to all states of the automaton, which is our candidate for a memoryless strategy on the program game. Then, we compute whether the candidate is a winning strategy, which is not necessarily the case. Note that invariants have an automaton consisting of one state and thus the memoryless strategy for the product game is a memoryless strategy for the program game.

Recall that the product game is $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, B)$. Let $\sigma : (S \times Q) \times I_G \to 2^{C_G \times C_A}$ be the attractor strategy for condition B. Note that the strategy is immaterial on nodes that are either not reachable (under any choice of the system) or not winning (and thus will be avoided by the system). Let R be the set of reachable states of the product game, and let W be the set of winning states. We define

$$\tau'(s, i_G) = \big\{c_G \;\big|\; \forall q \in Q : \big((s,q) \notin R \cap W \text{ or } \exists c_A \in C_A : (c_G, c_A) \in \sigma((s,q), i_G)\big)\big\}.$$

Intuitively, we obtain τ' by taking the moves common to all reachable, winning states of the strategy automaton.[1]

If τ' is winning, then so is σ, but the converse does not hold. To check whether τ' is winning, we construct a game G' from G by restricting the transition relation to adhere to τ': $\delta' = \{(s, i, c, s') \in \delta \mid c \in \tau'(s, i)\}$. This may introduce states without a successor. We see whether we can avoid such states by computing $W' = \mathsf{MG}\, S$. If we find that $s_0 \notin W'$, we cannot avoid visiting a dead-end state, and we give up trying to find a repair. If, on the other hand, $s_0 \in W'$, we get our final memoryless strategy τ by restricting τ' to W', which ensures that a play that starts in W' remains there and never visits a dead-end. We thus reach our main conclusion in the following theorem.

Theorem 4.5. *If $s_0 \in W'$ then τ is a winning strategy of G.*

Proof. Take a play $\pi = s_0 \overset{i_{G0} c_{G0}}{\longrightarrow} s_1 \overset{i_{G1} c_{G1}}{\longrightarrow} \dots$ of G according to τ. We will build a winning play $\pi' = (s_0, q_0) \overset{i_{G0}(c_{G0} c_{A0})}{\longrightarrow} (s_1, q_1) \dots$ of $G \triangleright A$ according to σ. It then follows that, since σ is winning, π' is winning. Thus, $s_0, \dots \in F_G$ and π is a winning play. Since π is arbitrary, this means that τ is winning.

[1] We may treat multiple Büchi constraints, if present, in the same manner. This is equivalent to using the counting construction.

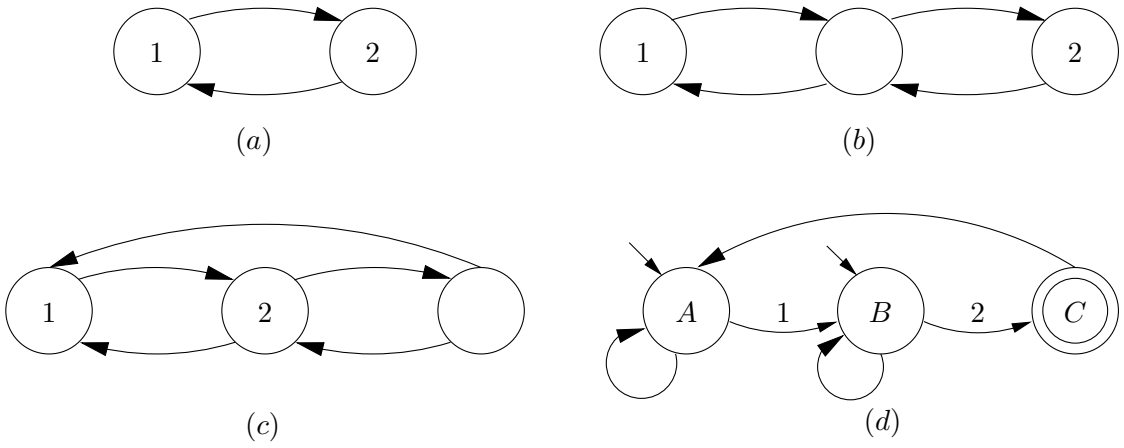

Figure 4.3: Fig. a, b, c show three games with the winning condition that states 1 and 2 are both visited infinitely often. Multiple outgoing arcs from a state model a system choice. The winning condition is defined by the Büchi automaton shown in Fig. d. For Fig. a, the strategies for States A, B, and C coincide, and a memoryless strategy exists. For Fig. b, no memoryless strategy exists, and for Fig. c, a memoryless strategy exists, but it is not equal to the intersection of all the strategies for states A, B, and C. (The strategies are contradictory for the state on the right.)

We will build π' inductively. Suppose that π' is built according to σ up to (s_j, q_j). Thus, (s_j, q_j) is reachable and since σ is winning, $(s_j, q_j) \in W$. Then, since $\tau(s_j, i_{Gj}) = s_{j+1}$, c_G is chosen in such a way that $\forall q \in Q : \big((s_j, q) \notin R \cap W$ or $\exists c_A \in C_A : (c_G, c_A) \in \sigma((s_j, q), i_G)\big)$, and we can choose q_{j+1} such that $\delta((s_j, q_j), i_{Gj}) = (s_{j+1}, q_{j+1})$. □

4.5 Extracting a Repair

This section shows a symbolic method to extract a repair statement from a memoryless strategy. We determinize the strategy by finding proper assignments to the system choices that can be used in the suspect locations. For any given state of the program, the given strategy may allow for multiple assignments, which gives us room for optimization.

We may not want the repair to depend on certain variables of the program, for example, because they are out of the scope of the component that is being repaired. In that case, we can universally quantify these variables from the strategy and its winning region and check that the strategy still supplies a valid response for all combinations of state and

input.

For each assignment to the system choice variables, we calculate a set $P_j \subseteq S \times I$ for which the assignment is a part of the given strategy. We can use these sets P_j to suggest the repair "`if P0 then assign0 else if P1 then ...`", in which `Pj` is an expression that represents the set P_j. The expression `Pj`, however, can be quite complex: even for small examples it can take over a hundred lines, which would make the suggested repair inscrutable.

We exploit the fact that the sets P_j can overlap to construct new sets A_j that are easier to express. We have to ensure that we still cover all winning and reachable states using the sets A_j. Therefore, A_j is obtained from P_j by adding or removing states outside of a *care set*. The care set consists of all states that cannot be covered by A_j because they are not in P_j and all states that must be covered by A_j because they are neither covered by an A_k with $k < j$, nor by a P_k with $k > j$. We then replace `Pj` with an expression for A_j to get our repair suggestion.

For simultaneous assignment to many variables, we may consider generating repairs for each variable seperately, in order to avoid enumerating the domain. For example, we could assign the variables one by one instead of simultaneously.

Extracting a simple repair is similar to multi-level logic synthesis in the presence of satisfiability don't cares and we may be able to apply multi-level minimization techniques [HS96]; the problem of finding the smallest expression for a given relation is NP-hard by reduction from 3SAT. One optimization we may attempt is to vary the order of the `Aj`s, but in our experience, the suggested repairs are typically quite readable.

4.6 Complexity

The complexity of the algorithm is polynomial in the number of states of the system, and exponential in the length of the formula, like the complexity of model checking. A symbolic implementation needs a quadratic number of preimage computations to compute the winning region of a Büchi game, the most expensive operation, like the Emerson-Lei algorithm typically used for model checking [RBS00]. For invariants, model checking and repair both need a linear number of preimage computations. Although the combination of universal and existential quantification makes preimage computations more expensive and we have to do additional work to extract the repair, we expect that repair is feasible for a large class of designs for which model checking is possible.

In our current implementation, we build the strategy as a monolithic BDD, which may use a lot of memory. We are still researching ways to compute the strategy in a partitioned way.

4.7 Examples

In this section we present initial experimental results supporting the applicability of our approach on real (though small) examples.

We have implemented our repair approach in the VIS model checker [B+96] as an extension of the algorithm of [JRS02]. The examples below are finite state programs given in pseudo code. They are translated to Verilog before we feed them to the repair algorithm. Suspect expressions are freed and a new system choice is added with the same domain as the expression. Assertions are replaced by `if(...) error=1` and the property $\mathsf{G}(\text{error} = 0)$. In the current version, this translation and the code augmentation are done manually.

4.7.1 Locking Example

We start with the abstract program shown in Fig. 4.4 [GV03]. This programs abstracts a class of concrete programs with different if and while conditions, all of which perform simple lock request/release operations. The method `lock()` checks that the lock is available and requests it. Vice versa, `unlock()` checks that the lock is held and releases it. The `if(*)` in the first line causes the lock to be requested nondeterministically, and the `while(*)` causes the loop to be executed an arbitrary number of times. The variable `got_lock` is used to keep track of the status of the lock (Lines 4 and 5). The assertions in Lines 11 and 21 constitute a safety property that is violated, e.g., if the loop is executed twice without requesting the lock. The fault is that the statement `got_lock--` should be placed within the scope of the preceding `if`.

Model-based diagnosis can be used to find a candidate for the repair [MSW00]. A diagnosis of the given example was performed in [CKW05] and localizes the fault in Lines 1, 6, or 7. We reject the possibility of changing Line 1 or 7 because we want the repair to work regardless of the if and while conditions in the concrete program. Instead, we look for a faulty assignment to `got_lock`. Thus, we free the RHS in Lines 3 and 6. The algorithm suggests a correct repair, `got_lock=1` for Line 3 and `got_lock=0` for Line 6. Note that we repair the program using a different fault model than the one which caused it, i.e., after

```
  int got_lock = 0;
  do{
1   if (*) {
2     lock();
3     got_lock++; }
4   if (got_lock != 0) {
5     unlock();}
6   got_lock--;
7 } while(*)

  void lock() {
11  assert(L = 0);
12  L = 1; }

  void unlock(){
21  assert(L = 1);
22  L = 0; }
```

Figure 4.4: Locking Example

```
1  int least = input1;
2  int most = input1;
3  if(most < input2){
4     most = input2; }
5  if(most < input3){
6     most = input3;}
7  if(least > input2){
8     most = input2; }
9  if(least > input3){
10    least = input3;}
11 assert (least <= most);
```

Figure 4.5: MinMax Example

the repair the program is correct, even though we did not suggest to move `got_lock--` inside the scope of the `if`.

4.7.2 MinMax

To present a more general fault model we show a simple program which assigns the minimal and maximal values out of three input values to `least` and `most`, resp. [Gro04].

The fault is located in Line 8 of Fig. 4.5, where `input2` is assigned to `most` (instead of `least`), which was one of five single fault diagnoses found by a model based debugger based on [MSW00]. To find the correct repair, we replace the assignments in lines 4, 6, 8, and 10 with switch-statements over the system choice that selects whether to assign to `least`, to `most`, or to replace the RHS. The algorithm correctly suggests to assign to variable `most` in Lines 4 and 6, and to `least` in Lines 8 and 10.

4.7.3 Critical Sections

Fig. 4.6 demonstrates how to cope with problems when testing properties that have no deterministic automaton (see Section 4.2). The example from [BEGL99] depicts two processes that share `flag` and `turn` variables, which are used to avoid concurrent access to the

Process A

```
1  flag1A = true;
2  turn1B = false;
3  while(flag1B && turn1B);
4  x = x && y;
5  flag1A = false;
6  if(turn1B){
7    flag2A = true;
8    turn2B = true;
9    while(flag2B && turn2B);
10   y = false;
11   flag2A = false;}
12 goto 1;
```

Process B

```
1  flag1B = true;
2  turn1B = false;
3  while(flag1A && !turn1B);
4  x = x && y;
5  flag2B = true;
6  turn2B = false;
7  while(flag2A && !turn2B);
8  y = !y;
9  x = x || y;
10 flag2B = false;
11 flag1B = false;
12 goto 1;
```

Figure 4.6: Critical Section Example

variables `x` and `y`. The process contains an arbiter (not shown) that nondeterministically yields control to either Process A or B, and records its choice in the variable `arbiter`. The fault is that `turn1B` is set to `false` in Line 2 of Process A. The correct value is `true`. This can cause both a deadlock and a violation of the critical region of `x`.

To check if Process B is eventually allowed to access `x` when it is waiting for it, we check the property FairArbiter $\rightarrow$ G(Bwaiting $\rightarrow$ F $\neg$Bwaiting) where FairArbiter = G F(`arbiter` $= A$) $\wedge$ G F(`arbiter` $= B$)) and Bwaiting is true whenever Process B is in Lines 3 or 7. As the implication leads to an negation of FairArbiter we get a nondeterministic automaton. Our algorithm cannot find a strategy for the product game of the program and this automaton (See Fig. 4.2).

We solve this problem by manually changing the arbiter to switch processes infinitely often. Freeing `turn1B` in Line 2 of Process A with domain `{false,true}` now leads to the correct answer, `turn1B = true`. Note that this repair also works for the original model. This repair can also be found by checking for violations of the critical section, which can be stated as a simple invariant and therefore does not require a modification of the system.

4.7.4 Processor

In order to compare the efficiency of repair algorithm to that of model checking, we have introduced a fault in a 16-bit version of a simple unpipelined DLX-style processor. The fault is in the ALU and the property checks that the ALU works correctly.

On a 2.8GHz Linux machine with 2GB of RAM, the model checking run needs 230 seconds to check that the property does not hold on the incorrect version. The repair algorithm finds a repair in 200 seconds, and the repair is verified to be correct by the model checker (an unnecessary precaution) in 210 seconds; all runs use around 1.2GB.

Chapter 5

Synthesis of Real-Life Designs

This chapter is based on [BGJ+06], [BGJ+07a], and [BGJ+07b].

In this chapter we propose to use a formal specification language as a high-level hardware description language. Formal languages allow for compact, unambiguous representations and yield designs that are correct by construction. The idea of automatic synthesis from specifications is old, but used to be completely impractical. Recently, great strides towards efficient synthesis from specifications have been made. We extend these recent methods to generate compact circuits and we show their practicality by synthesizing a generalized buffer and an arbiter for ARM's AMBA AHB bus from specifications given in LTL. These are the first industrial examples that have been synthesized automatically from their specifications.

In Section 5.1, we describe how to synthesize a circuit from specifications. In Section 5.2, we describe the Generalized Buffer, give its formal specification, and show the resuts of synthesizing them. In 5.3, we do the same for the AMBA AHB arbiter and we discuss lessons learned in 5.4.

5.1 Synthesis

In this section, we discuss how circuits can be obtained automatically from their LTL specifications.

5.1.1 Synthesis of GR(1) Properties

We briefly review the results presented in [PPS06] on synthesizing GR(1) properties. We are interested in the question of *realizability* of LTL specifications (cf. [PR89]). Assume two sets of Boolean variables $\mathcal{X}$ and $\mathcal{Y}$. Intuitively $\mathcal{X}$ is the set of input variables controlled by the environment and $\mathcal{Y}$ is the set of system variables. *Realizability* amounts to checking whether there exists an *open controller* that satisfies the specification. Such a controller can be represented as an automaton which, at any step, reads values of the $\mathcal{X}$ variables and outputs values for the $\mathcal{Y}$ variables.

Here we concentrate on a subset of LTL for which realizability and synthesis can be solved efficiently. The specifications we consider are of the form $\varphi = \varphi^e \rightarrow \varphi^s$. We require that φ^α for $\alpha \in \{e, s\}$ can be rewritten as a conjunction of the following parts.

- φ_i^α – a Boolean formula which characterizes the initial states of the implementation.
- φ_t^α – a formula of the form $\bigwedge_i \mathsf{G}\, B_i$ where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\mathsf{X}\, v$ where $v \in \mathcal{X}$ if $\alpha = e$, and $v \in \mathcal{X} \cup \mathcal{Y}$ otherwise.
- φ_g^α – has the form $\bigwedge_{i \in I} \mathsf{G}\,\mathsf{F}\, B_i$ where each B_i is a Boolean formula.

In order to allow formulas of other forms (e.g., $\mathsf{G}(p \rightarrow (q\, \mathsf{W}\, r))$ where p, q, and r are Boolean), we augment the set of variables by adding *deterministic monitors*. Deterministic monitors are Büchi automata whose behavior is deterministic according to the choice of the inputs and the outputs. These monitors follow the truth value of the expression nested inside the G operator. Deterministic automata are easily represented in LTL by a three sets of formulas: (1) One formula for each edge of the automaton, of the form $\mathsf{G}(s \wedge i \rightarrow \mathsf{X}(s'))$, where s and s' identify states and i is an input, (2) a Boolean formula representing the initial state, and (3) a formula of the form $\mathsf{G}\,\mathsf{F}(B)$ to represent the fairness condition, where B is a Boolean formula representing a set of states. (An example can be found in Section 5.2.3.) It should be noted that even with these restrictions, all possible (finite state) designs can be expressed as a set of properties.

We reduce the realizability problem of a LTL formula to the decision of the winner in an infinite two-player game played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment. A *game structure* is a multi-graph whose nodes are all the truth assignments to $\mathcal{X}$ and $\mathcal{Y}$. A node v is connected by edges to all the nodes v' such that the truth assignments to $\mathcal{X}$ and $\mathcal{Y}$ satisfy $\varphi_t^e \wedge \varphi_t^s$, where v supplies the assignments to the current values and v' to the next values. We then group all the edges that agree on the assignment of $\mathcal{X}$ in v' to

one multi-edge. A play starts by the environment choosing an assignment to $\mathcal{X}$ and the system choosing a state in $\varphi_i^e \wedge \varphi_i^s$ that agrees with this assignment. A play proceeds by the environment choosing a multi-edge and the system choosing one of the nodes connected to this multi-edge. The system wins if this interaction produces an infinite play that satisfies $\varphi_g^e \rightarrow \varphi_g^s$.

We *solve* the game, attempting to decide whether the game is winning for the environment or the system. If the environment is winning the specification is *unrealizable*. If the system is winning, we *synthesize* a winning strategy. This strategy, a BDD, is a nondeterministic representation of a working implementation. Formally, we have the following.

Theorem 5.1. [PPS06] *Given sets of variables $\mathcal{X}$ and $\mathcal{Y}$ and a LTL formula φ of the form presented above with m and n conjuncts, we can determine using a symbolic algorithm whether φ is realizable in time proportional to $(mn2^{d+|\mathcal{X}|+|\mathcal{Y}|})^3$ where d is the number of variables added by the monitors for φ.*

5.1.2 Generating Circuits from BDDs

In this section, we describe how to construct a circuit from the strategy. The strategy is a BDD over the variables $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{X}'$, and $\mathcal{Y}'$, where $\mathcal{X}$ are input variables, $\mathcal{Y}$ are output variables, and the primed versions represent next state variables. The corresponding circuit contains $|\mathcal{X}| + |\mathcal{Y}|$ flipflops to store the values of the inputs and outputs in the last clock tick. (See Figure 5.1.) In every step, the circuit reads the next input values $\mathcal{X}'$ and determines the next output values $\mathcal{Y}'$ using combinational logic with inputs `I` $= \mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}'$. Note that the strategy does not prescribe a unique combinational output for every combinational input. In most cases, multiple outputs are possible, in states that do not occur when the system adheres to the strategy, no outputs may be allowed.

We have attempted two methods to build the combinational logic, one based on [KS00] and one based on computing cofactors.

The approach of [KS00] yields a circuit that can generate, for a given input, any output allowed by the strategy. To this end, it uses a set of extra inputs to the combinational logic. Note that this is more general than what we need: a circuit that always yields one valid output given an input. We will see later that this generality comes at a heavy price in terms of the size of the logic.

The second method to build the combinational logic uses the pseudo code shown in Figure 5.2. We write `o` $\in \mathcal{Y}'$ for a combinational output and `i` $\in$ `I` for a combinational

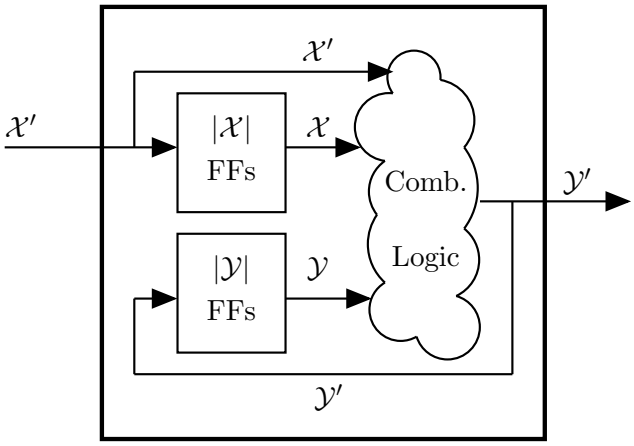

Figure 5.1: Diagram of generated circuit

input. The strategy is denoted by `S` and `O\o` is the set of combinational outputs excluding output `o`. For every combinational output `o` we construct a function f in terms of `I` that is compatible with the given strategy BDD. The algorithm proceeds through the combinational outputs `o` one by one: First, we build `S'` to get a BDD that restricts only `o` in terms of `I`. Then we build the *positive* and *negative cofactors* (`p,n`) of `S'` with respect to `o`, that is, we find the sets of inputs for which `o` can be 1 (0, respectively). For the inputs that occur in the positive and in the negative cofactor, both values are allowed. The combinational inputs that are neither in the positive nor in the negative cofactor are outside of the winning region and thus represent situations that cannot occur (as long as the environment satisfies the assumptions). Thus, f has to be 1 in `p` $\cap$ `!n` and 0 in (`!p` $\cap$ `n`), which give us the set of care states. We minimize the positive cofactors with the care set to obtain the function f. Finally, we substitute variable `o` in `S` by f, and proceed with the next variable. The substitution is necessary since a combinational output may also depend on other combinational outputs.

The resulting circuit is constructed by writing the BDDs for the functions using CUDD's DumpBlif command [Som]. We then optimize the result using ABC [Ber] and map it to a library of standard cells. We also use ABC to estimate the size of the circuits. (Although the absolute numbers thus obtained may not be very meaningful, the relative numbers are.)

In the following we describe two extensions that are simple and effective. (Cf. Section 5.2.3 and Section 5.3.3.)

```
for all combinational outputs o do
  S' = exists O\o . S
  p = positive cofactor of o in S'
  n = negative cofactor of o in S'
  // (*)
  // note: p and n in general incomparable
  careset = p*!n + !p*n
  f[o] = p minimized wrt. careset
  // keep relation between outputs
  S = S in which o is substituted by f[o]
od
```

Figure 5.2: Algorithm to construct a circuit from a BDD

```
p = p * !n
n = n * !p
// where p and n overlap, output can be anything
for all inputs i
  p' = exists i. p
  n' = exists i. n
  if p' * n' = 0 then
    p = p'; n = n';
  fi
end
```

Figure 5.3: Extension to algorithm in Figure 5.2

Optimizing the Cofactors

The algorithm presented in Figure 5.2 generates a function in terms of the combinational inputs for every combinational output. Some outputs may not depend on all inputs and we would like to remove unnecessary inputs from the functions. Given the positive and the negative cofactor of a variable `o`, if the cofactors do not overlap when we existentially quantify variable `i`, variable `i` is not needed to distinguish between the states where `o` has to be 1 and where `o` has to be 0, and we can simply leave it out. We adapt the algorithm in Figure 5.2 by inserting the code shown in Figure 5.3 at the spot marked with `(*)`.

Removing Dependent Variables

After computing the combinational logic, we perform *dependent variables analysis* [HD93] on the set of reachable states to simplify the generated circuit.

Definition 5.2. *[HD93] Given a Boolean function f over $x_0, x_1, \cdots x_n$, a variable x_i is* functionally dependent *in f iff $\forall x_i.f = 0$.*

Note that if x_i is functionally dependent, it is uniquely determined by the remaining variables of f and can be replaced by a function $g(x_0, \cdots x_{i-1}, x_{i+1} \cdots x_n)$.

Suppose our generated circuit has the set $R(\mathcal{X} \cup \mathcal{Y})$ of reachable states. If a state variable s is functionally dependent in R, we can remove the corresponding flipflop in the circuit, and instead compute its value as a function of the values of the other flipflops.

5.2 Generalized Buffer Case Study

5.2.1 Description of the Generalized Buffer

The generalized buffer (henceforth *GenBuf*) is a design that has been developed by IBM as a tutorial for the Rulebase verification tool[1]. GenBuf comes with a relatively complete specification in PSL.

Figure 5.4 contains a block diagram of the design and its interface. Dashed boxes represent the environment. GenBuf is a family of buffers parameterized by a number n. It transmits data from n senders to two receivers. Data is offered by the senders in an arbitrary order, and is received by the receivers in round-robin order. The buffer has a handshake protocol with each sender and each receiver. For each sender i, GenBuf has an input StoB_REQ(i) (sender to buffer request), which signals a request to send, and an output BtoS_ACK(i) (buffer to sender acknowledge). Furthermore, each sender has a 32-bit databus to send data to the buffer. The buffer contains a four-slot FIFO to hold the data.

On the receiver side, a similar interface exists. It connects the buffer to each receiver using the output BtoR_REQ(j) (buffer to receiver request) and the input RtoB_ACK(j) (receiver to Buffer acknowledge). The receivers share a single 32-bit data bus.

Genbuf consists of a controller, a FIFO, and a multiplexer. We synthesize the controller from its specification, while assuming that the implementation of the FIFO and the multiplexer are given. FIFOs and multiplexers are standard pieces of logic and synthesizing

[1] See http://www.haifa.ibm.com/projects/verification/RB_Homepage/tutorial3/.

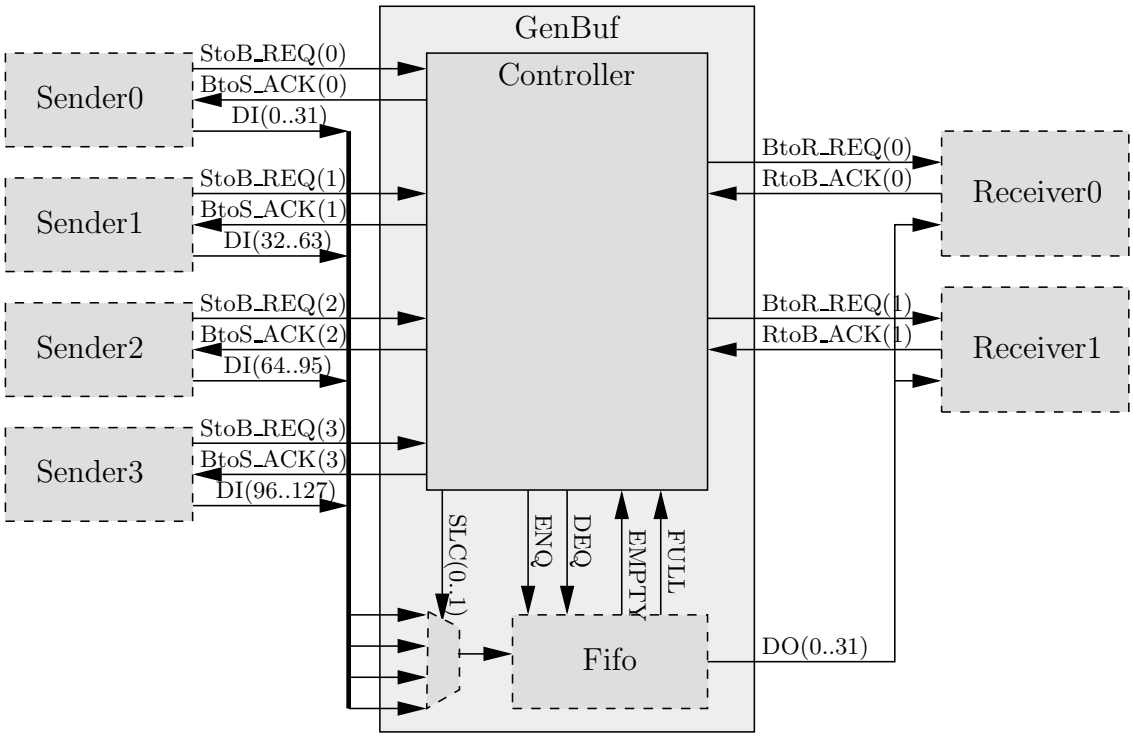

Figure 5.4: Block diagram of GenBuf with four senders

them from specifications would make the task unnecessarily complex, especially because they involve 32-bit data buses.

The control logic communicates with the FIFO through two outputs and two inputs. The outputs ENQ (enqueue data) and DEQ (dequeue oldest data) are used to fill and empty the FIFO. The inputs FULL and EMPTY tell the controller whether the FIFO is ready to receive or send data. The controller communicates with the multiplexer using a multi-bit output called SLC determines which signal from the clients is loaded when ENQ is asserted.

Handshake between Senders and GenBuf

The interface between a sender and GenBuf is a four-phase handshake: illustrated in Figure 5.5:

1. Sender i initiates the transfer by raising StoB_REQ(i). One cycle later, it puts its data on the bus.

2. At least one tick after StoB_REQ(i) is raised, GenBuf raises BtoS_ACK(i) and reads the data.
3. One tick after BtoS_ACK(i) is raised, the sender lowers StoB_REQ(i). From this time on, it is no longer required to keep the data on the bus.
4. GenBuf eventually lowers BtoS_ACK(i). It may take several cycles to do so. A new transfer may not be initiated by sender i until one cycle after BtoS_ACK(i) is lowered.

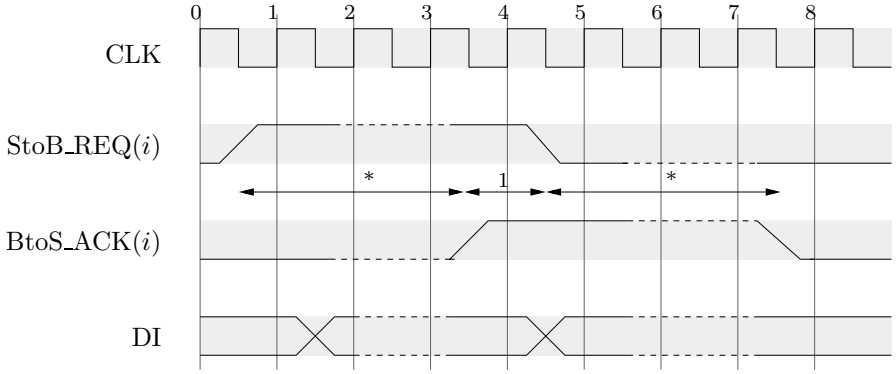

Figure 5.5: An example of a Sender-to-GenBuf handshake

Handshake between Receivers and GenBuf

The handshake between GenBuf and the receivers also consists of four phases, illustrated in Figure 5.6.

1. When GenBuf wants to send data to a receiver, say receiver j, it asserts BtoR_REQ(j) and puts the data on the D0 bus.

2. When receiver j is ready to receive data, it raises RtoB_ACK(j) and reads the data.

3. One cycle later, GenBuf lowers BtoR_REQ(j). It may now remove the data from the bus.

4. One cycle later, receiver j finalizes the transfer by lowering RtoB_ACK(j). GenBuf does not initiate another transfer until one cycle after RtoB_ACK(j) has been lowered.

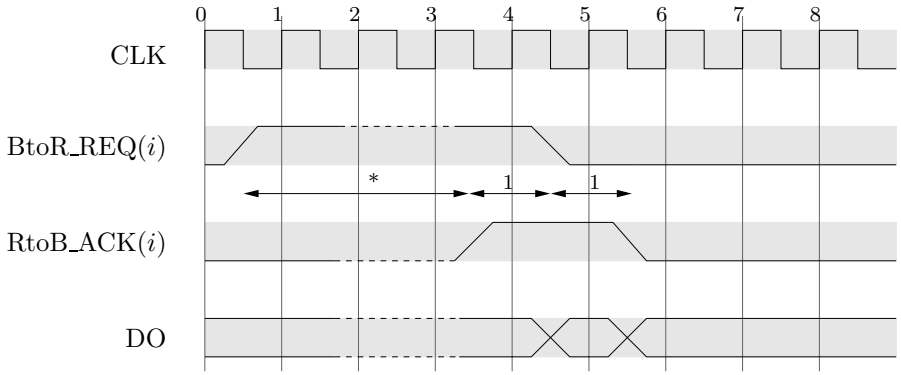

Figure 5.6: An example of a GenBuf-to-Receiver handshake

5.2.2 Formal Specification

We will now present the specification that we have developed for GenBuf. It is closely related to IBM's original specification. Since we do not synthesize the FIFO and multiplexer automatically, we have removed the specifications that stated that they work correctly and we added formulas that specify the interaction with the FIFO and multiplexer.

The LTL formulas for the specification are summarized in Table 5.1. In the table, we use $i \in \{0, \ldots, n\}$ to denote the number of a sender. We use $j \in \{0, 1\}$ to denote a receiver.

Communication with Senders

Guarantee 1. *A request from a sender is always acknowledged.*

$$\bigwedge_i \mathsf{G}(\mathit{StoB\_REQ}(i) \rightarrow \mathsf{F}(\mathit{BtoS\_ACK}(i)))$$

Furthermore, the acknowledgement is eventually lowered.

$$\bigwedge_i \mathsf{G}(\neg \mathit{StoB\_REQ}(i) \rightarrow \mathsf{F}(\neg \mathit{BtoS\_ACK}(i)))$$

Guarantee 2. *Immediate acknowledges are forbidden, because the data of the sender are not valid until one step after the assertion of request.*

$$\bigwedge_i \mathsf{G}(\mathtt{raise}(\mathit{StoB\_REQ}(i)) \rightarrow \mathsf{X}(\neg \mathit{BtoS\_ACK}(i)))$$

Guarantee 3. *There is no acknowledgement without a request.*

$$\bigwedge_i \mathsf{G}(\texttt{raise}(BtoS\_ACK(i)) \rightarrow StoB\_REQ(i))$$

Guarantee 4. *An acknowledge is not deasserted unless the sender deasserts its request first.*

$$\bigwedge_i \mathsf{G}((BtoS\_ACK(i) \wedge StoB\_REQ(i)) \rightarrow \mathsf{X}(BtoS\_ACK(i)))$$

Assumption 1. *A request is not lowered until it is served.*

$$\bigwedge_i \mathsf{G}((StoB\_REQ(i) \wedge \neg BtoS\_ACK(i)) \rightarrow \mathsf{X}(StoB\_REQ(i)))$$

The signal $StoB\_REQ(i)$ is lowered one cycle after $BtoS\_ACK(i)$ is raised and it cannot be raised until one cycle after $BtoS\_ACK(i)$ is lowered.

$$\bigwedge_i \mathsf{G}(BtoS\_ACK(i) \rightarrow \mathsf{X}(\neg StoB\_REQ(i)))$$

Guarantee 5. *Only one sender sends data at any one time.*

$$\bigwedge_i \bigwedge_{i' \neq i} \mathsf{G}\,\neg(BtoS\_ACK(i) \wedge BtoS\_ACK(i'))$$

Communication with Receivers

Assumption 2. *A request from the buffer is always acknowledged.*

$$\bigwedge_j \mathsf{G}(BtoR\_REQ(j) \rightarrow \mathsf{F}(RtoB\_ACK(j)))$$

Furthermore, the acknowledgement is lowered one tick after the request is lowered.

$$\bigwedge_j \mathsf{G}(\neg BtoR\_REQ(j) \rightarrow \mathsf{X}(\neg RtoB\_ACK(j)))$$

Assumption 3. *An acknowledgement is not deasserted unless the buffer deasserts its request first.*

$$\bigwedge_j \mathsf{G}((BtoR\_REQ(j) \wedge RtoB\_ACK(j)) \rightarrow \mathsf{X}(RtoB\_ACK(j)))$$

Assumption 4. *There is no acknowledgement without a request.*

$$\bigwedge_j \mathsf{G}(\mathsf{X}(\mathit{RtoB\_ACK}(j)) \rightarrow \mathit{BtoR\_REQ}(j))$$

Guarantee 6. *A request is not lowered until it is served.*

$$\bigwedge_j \mathsf{G}((\mathit{BtoR\_REQ}(j) \wedge \neg \mathit{RtoB\_ACK}(j)) \rightarrow \mathsf{X}(\mathit{BtoR\_REQ}(j)))$$

The request is lowered one cycle after the acknowledgement is raised and it cannot be raised until one cycle after the acknowledgement is lowered.

$$\bigwedge_j \mathsf{G}(\mathit{RtoB\_ACK}(j) \rightarrow \mathsf{X}(\neg \mathit{BtoR\_REQ}(j)))$$

Guarantee 7. *GenBuf does not request both receivers simultaneously.*

$$\mathsf{G}\,\neg(\mathit{BtoR\_REQ}(0) \wedge \mathit{BtoR\_REQ}(1)).$$

GenBuf will not make two consecutive requests to any receiver. (This guarantees round-robin scheduling.)

$$\mathsf{G}(\mathtt{raise}(\mathit{BtoR\_REQ}(0)) \rightarrow \mathsf{X}(\neg\,\mathtt{raise}(\mathit{BtoR\_REQ}(0))\,\mathsf{W}\,\mathtt{raise}(\mathit{BtoR\_REQ}(1))))$$
$$\mathsf{G}(\mathtt{raise}(\mathit{BtoR\_REQ}(1)) \rightarrow \mathsf{X}(\neg\,\mathtt{raise}(\mathit{BtoR\_REQ}(1))\,\mathsf{W}\,\mathtt{raise}(\mathit{BtoR\_REQ}(0))))$$

Guarantee 8. *GenBuf will deassert its request to receiver j one cycle after receiver j acknowledged the request.*

$$\bigwedge_j \mathsf{G}(\mathit{RtoB\_ACK}(j) \rightarrow \mathsf{X}(\neg \mathit{BtoR\_REQ}(j)))$$

Interface to the FIFO and the Multiplexer

Guarantee 9. *The select and enqueue signals follow the acknowledgements to the senders.*

$$\mathsf{G}(\textstyle\bigvee_i \mathtt{raise}(\mathit{BtoS\_ACK}(i)) \leftrightarrow \mathsf{X}(\mathit{ENQ}))$$
$$\textstyle\bigwedge_i \mathsf{G}(\mathtt{raise}(\mathit{BtoS\_ACK}(i)) \rightarrow \mathsf{X}(\mathit{SLC} = i))$$

Guarantee 10. *Data is dequeued when the transfer to the receiver has completed.*

$$\mathsf{G}((\texttt{fall}(\mathit{RtoB\_ACK}(0)) \vee \texttt{fall}(\mathit{RtoB\_ACK}(1))) \leftrightarrow \mathsf{X}(\mathit{DEQ}))$$

Guarantee 11. *No enqueue when the FIFO is full and we do not dequeue data, and no dequeue when it is empty.*

$$\mathsf{G}((\mathit{FULL} \wedge \neg \mathit{DEQ}) \rightarrow \neg \mathit{ENQ})$$
$$\mathsf{G}(\mathit{EMPTY} \rightarrow \neg \mathit{DEQ})$$

Guarantee 12. *If the FIFO is not empty, a dequeue will ensue eventually.*

$$\mathsf{G}(\neg \mathit{EMPTY} \rightarrow \mathsf{F}(\mathit{DEQ}))$$

Assumption 5. *The FIFO behaves correctly. If we enqueue and dequeue simultaneously or not at all, the status of the FIFO does not change. If data is only enqueued (dequeued, resp.), the FIFO must not be empty (full) in the next cycle.*

$$\mathsf{G}((\mathit{DEQ} \leftrightarrow \mathit{ENQ}) \rightarrow (\mathit{EMPTY} \leftrightarrow \mathsf{X}(\mathit{EMPTY})))$$
$$\mathsf{G}((\mathit{DEQ} \leftrightarrow \mathit{ENQ}) \rightarrow (\mathit{FULL} \leftrightarrow \mathsf{X}(\mathit{FULL})))$$
$$\mathsf{G}((\mathit{ENQ} \wedge \neg \mathit{DEQ}) \rightarrow \mathsf{X}(\neg \mathit{EMPTY}))$$
$$\mathsf{G}((\mathit{DEQ} \wedge \neg \mathit{ENQ}) \rightarrow \mathsf{X}(\neg \mathit{FULL}))$$

Initially, the buffer we synthesized from the specification above ignored the FIFO. Instead it would wait until it could send data to a receiver before accepting data from a sender. Hence, we added the following property, which ensures that the FIFO is used.

Guarantee 13. *If the FIFO is not full and a sender requests to send data, the data is enqueued either in this or in the next step.*

$$\mathsf{G}((\neg \mathit{FULL} \wedge \exists i : \mathit{StoB\_REQ}(i)) \rightarrow (\mathit{ENQ} \vee \mathsf{X}(\mathit{ENQ})))$$

5.2.3 Synthesis

As explained in Section 5.1.1, not all LTL specifications can be synthesized directly. We first have to translate Guarantees 1, 2, 7, 12 and Assumption 2 into a suitable form.

Taking the Guarantee 4, 6, and Assumption 4 into account, we can combine Guarantee 1

Table 5.1: LTL specification of the GenBuf

G1	$\bigwedge_i \mathsf{G}(\text{StoB\_REQ}(i) \rightarrow \mathsf{F}(\text{BtoS\_ACK}(i)))$ $\bigwedge_i \mathsf{G}(\neg\text{StoB\_REQ}(i) \rightarrow \mathsf{F}(\neg\text{BtoS\_ACK}(i)))$
G2	$\bigwedge_i \mathsf{G}(\texttt{raise}(\text{StoB\_REQ}(i)) \rightarrow \mathsf{X}(\neg\text{BtoS\_ACK}(i)))$
G3	$\bigwedge_i \mathsf{G}(\texttt{raise}(\text{BtoS\_ACK}(i)) \rightarrow \text{StoB\_REQ}(i))$
G4	$\bigwedge_i \mathsf{G}((\text{BtoS\_ACK}(i) \wedge \text{StoB\_REQ}(i)) \rightarrow \mathsf{X}(\text{BtoS\_ACK}(i)))$
A1	$\bigwedge_i \mathsf{G}((\text{StoB\_REQ}(i) \wedge \neg\text{BtoS\_ACK}(i)) \rightarrow \mathsf{X}(\text{StoB\_REQ}(i)))$ $\bigwedge_i \mathsf{G}(\text{BtoS\_ACK}(i) \rightarrow \mathsf{X}(\neg\text{StoB\_REQ}(i)))$
G5	$\bigwedge_i \bigwedge_{i' \neq i} \mathsf{G}\,\neg(\text{BtoS\_ACK}(i) \wedge \text{BtoS\_ACK}(i'))$
A2	$\bigwedge_j \mathsf{G}(\text{BtoR\_REQ}(j) \rightarrow \mathsf{F}(\text{RtoB\_ACK}(j)))$ $\bigwedge_j \mathsf{G}(\neg\text{BtoR\_REQ}(j) \rightarrow \mathsf{X}(\neg\text{RtoB\_ACK}(j)))$
A3	$\bigwedge_j \mathsf{G}((\text{BtoR\_REQ}(j) \wedge \text{RtoB\_ACK}(j)) \rightarrow \mathsf{X}(\text{RtoB\_ACK}(j)))$
A4	$\bigwedge_j \mathsf{G}(\mathsf{X}(\text{RtoB\_ACK}(j)) \rightarrow \text{BtoR\_REQ}(j))$
G6	$\bigwedge_j \mathsf{G}((\text{BtoR\_REQ}(j) \wedge \neg\text{RtoB\_ACK}(j)) \rightarrow \mathsf{X}(\text{BtoR\_REQ}(j)))$ $\bigwedge_j \mathsf{G}(\text{RtoB\_ACK}(j) \rightarrow \mathsf{X}(\neg\text{BtoR\_REQ}(j)))$
G7	$\mathsf{G}\,\neg(\text{BtoR\_REQ}(0) \wedge \text{BtoR\_REQ}(1))$. $\mathsf{G}(\texttt{raise}(\text{BtoR\_REQ}(0)) \rightarrow \mathsf{X}(\neg\,\texttt{raise}(\text{BtoR\_REQ}(0))\,\mathsf{W}\,\texttt{raise}(\text{BtoR\_REQ}(1))))$ $\mathsf{G}(\texttt{raise}(\text{BtoR\_REQ}(1)) \rightarrow \mathsf{X}(\neg\,\texttt{raise}(\text{BtoR\_REQ}(1))\,\mathsf{W}\,\texttt{raise}(\text{BtoR\_REQ}(0))))$
G8	$\bigwedge_j \mathsf{G}(\text{RtoB\_ACK}(j) \rightarrow \mathsf{X}(\neg\text{BtoR\_REQ}(j)))$
G9	$\mathsf{G}(\bigvee_i \texttt{raise}(\text{BtoS\_ACK}(i)) \leftrightarrow \mathsf{X}(\text{ENQ}))$ $\bigwedge_i \mathsf{G}(\texttt{raise}(\text{BtoS\_ACK}(i)) \rightarrow \mathsf{X}(\text{SLC} = i))$
G10	$\mathsf{G}((\texttt{fall}(\text{RtoB\_ACK}(0)) \vee \texttt{fall}(\text{RtoB\_ACK}(1))) \leftrightarrow \mathsf{X}(\text{DEQ}))$
G11	$\mathsf{G}((\text{FULL} \wedge \neg\text{DEQ}) \rightarrow \neg\text{ENQ})$ $\mathsf{G}(\text{EMPTY} \rightarrow \neg\text{DEQ})$
G12	$\mathsf{G}(\neg\text{EMPTY} \rightarrow \mathsf{F}(\text{DEQ}))$
A5	$\mathsf{G}((\text{DEQ} \leftrightarrow \text{ENQ}) \rightarrow (\text{EMPTY} \leftrightarrow \mathsf{X}(\text{EMPTY})))$ $\mathsf{G}((\text{DEQ} \leftrightarrow \text{ENQ}) \rightarrow (\text{FULL} \leftrightarrow \mathsf{X}(\text{FULL})))$ $\mathsf{G}((\text{ENQ} \wedge \neg\text{DEQ}) \rightarrow \mathsf{X}(\neg\text{EMPTY}))$ $\mathsf{G}((\text{DEQ} \wedge \neg\text{ENQ}) \rightarrow \mathsf{X}(\neg\text{FULL}))$
A13	$\mathsf{G}((\neg\text{FULL} \wedge \exists i : \text{StoB\_REQ}(i)) \rightarrow (\text{ENQ} \vee \mathsf{X}(\text{ENQ})))$

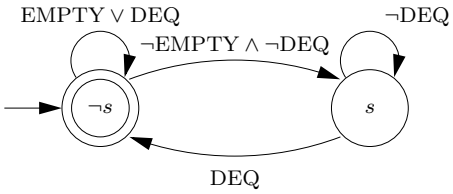

Figure 5.7: Monitor for Guarantee 12

and 2 to

$$\bigwedge_i \mathsf{G}\,\mathsf{F}(\text{StoB\_REQ}(i) \leftrightarrow \text{BtoS\_ACK}(i))$$

and we can rewrite Assumption 2 to

$$\bigwedge_i \mathsf{G}\,\mathsf{F}(\text{BtoR\_REQ}(i) \leftrightarrow \text{RtoB\_ACK}(i)).$$

For Guarantee 12 and the second part of Guarantee 7 we have to build deterministic monitors. Although there are formulas for which no deterministic monitor exists, and constructing such monitors is hard in general [KV98], constructing them is very simple for the formulas considered in this document.

For instance, Figure 5.7 shows the deterministic automaton for Guarantee 12 stating that $\mathsf{G}(\neg\text{EMPTY} \rightarrow \mathsf{F}\,\text{DEQ})$. We used the standard approach to construct Büchi automata from LTL formulas (e.g., [SB00]) with a slightly modified form of the standard expansion rules. In particular, we used the expansion rule $\mathsf{F}\,q$ equals $q \vee (\neg q \wedge \mathsf{X}\,\mathsf{F}\,q)$ and the fact that $\neg\text{EMPTY} \rightarrow \varphi$ equals $\text{EMPTY} \vee (\neg\text{EMPTY} \wedge \varphi)$.

After the specification has been brought into the proper form, it is synthesized using the algorithm described in Section 5.1. In Figure 5.8 we show the time needed to synthesize GenBuf for different numbers of senders, excluding the time taken by ABC to optimize the circuit, which is typically a few seconds. We have plotted the time taken to generate the circuits using the method based on [KS00], the time needed by our algorithm, our algorithm with the optimization of the cofactors, and our algorithm with optimization of the cofactors and removal of dependent variables. (See Section 5.1.2.) The time for synthesis remains under one minute and is similar for all methods. (We can not explain why synthesis is much faster when we have nine senders.)

In Figure 5.9 we show the area of the resulting circuits after optimization by ABC, measured using ABC's standard-cell grid count. The method based on [KS00] yields

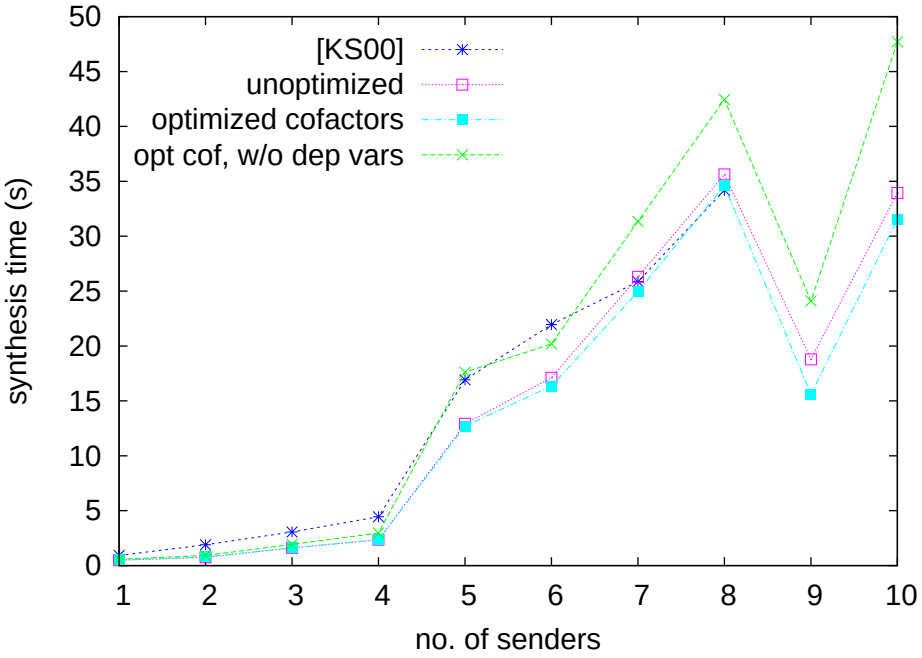

Figure 5.8: Time to synthesize GenBuf

circuits that are about an order of magnitude larger than ours. (For more than 6 senders, this method yields circuits that are too large for ABC to handle.) Optimizing the cofactors yields about 16%. Removing the dependent variables does not change the area much and reduces the number of latches by 5% to 12%. Which dependent variables are found is hard to predict, but usually includes ENQ and some or all of the SLC signals.

In Figure 5.9, we also show the area of the corresponding handwritten designs. The automatically generated design for one sender uses roughly the same area as the handwritten version. The area of the handwritten designs increases only by a factor of 5 from one to ten senders, while the area of the automatically generated designs increase by a factor of 75.

Optimization by ABC yields an improvement in area of about 20%. It should be noted that the growth of the circuit is well-behaved, but a circuit of size 200 000 is still very large.

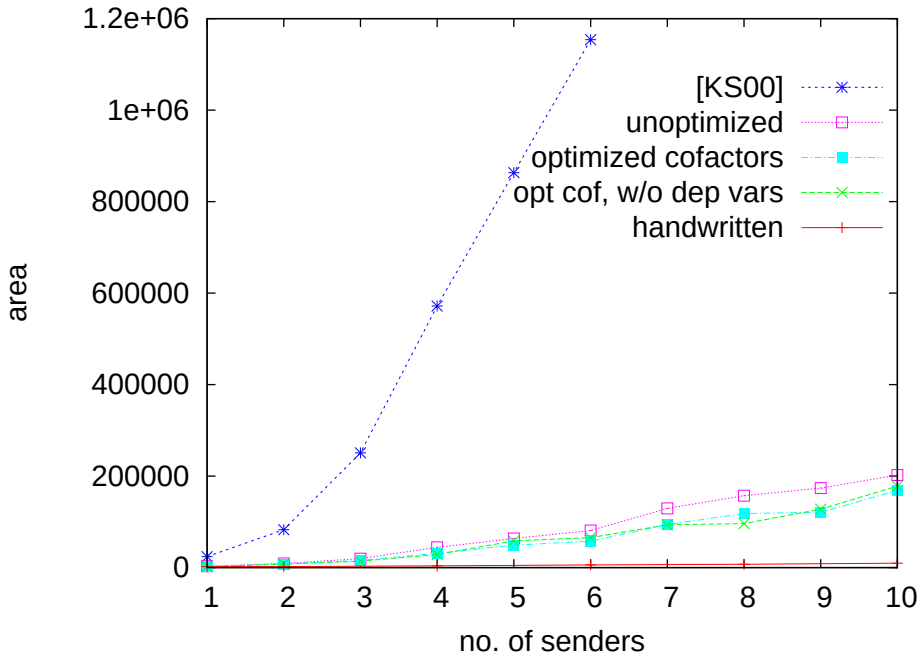

Figure 5.9: Size of the GenBuf circuits

5.3 AMBA AHB Case Study

In the section we summarize a case study that we performed on one of the AMBA (*Advanced Microcontroller Bus Architecture*) [Ltd99] busses of ARM.

5.3.1 Protocol

ARM's *Advanced Microcontroller Bus Architecture* (AMBA) [Ltd99] defines the *Advanced High-Performance Bus* (AHB), an on-chip communication standard connecting such devices as processor cores, cache memory, and DMA controllers. Up to 16 *masters* and up to 16 *slaves* can be connected to the bus. The masters initiate communication (read or write) with a slave of their choice. Slaves are passive and can only respond to a request. Master 0 is the *default master* and is selected whenever there are no requests for the bus.

The AHB is a pipelined bus. This means that different masters can be in different stages of communication. At one instant, multiple masters can request the bus, while another master transfers address information, and a yet another master transfers data. A bus *access* can be a single *transfer* or a *burst*, which consists of a specified or unspecified

number of transfers. Access to the bus is controlled by the *arbiter*, which is the subject of this section. All devices that are connected to the bus are Moore machines, that is, the reaction of a device to an action at time t can only be seen by the other devices at time $t + 1$.

The AMBA standard leaves many aspects of the bus unspecified. The protocol is at a logic level, which means that timing and electric parameters are not specified; neither are aspects such as the arbitration protocol.

We will now introduce the signals used in the AHB. 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 from Master i to receive a locked (uninterruptible) access to the bus. (Raised in combination with HBUSREQ[i].) Driven by the masters.
- HMASTER[3:0] – The master that currently owns the address bus (binary encoding). Driven by the arbiter.
- HREADY – High if the slave has finished processing the current data. Change of bus ownership and commencement of transfers only takes place when HREADY is high. Driven by the slave.
- 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. If this signal is low, a burst access may be interrupted when the bus is assigned to a different master. Driven by the arbiter

The following set of signals is multiplexed using HMASTERas the control signal. For instance, although every master has an address bus, only the address provided by the currently active master is visible on HADDR.

- HADDR[31:0] – The address for the next transfer. The address determines the destination slave.
- HBURST[1:0] – One of SINGLE (a single transfer), BURST4 (a four-transfer burst access), or INCR (unspecified length burst).

The list of signals does not contain the data transfer signals as these do not concern the arbiter. (Ownership of the data bus follows ownership of the address bus in a straightforward manner.) Bursts of length 8 or 16 are not taken into account, nor are the different addressing types for bursts. Adding longer bursts only lengthens the specification and the addressing types do not concern the arbiter. Furthermore, as an optional feature of the

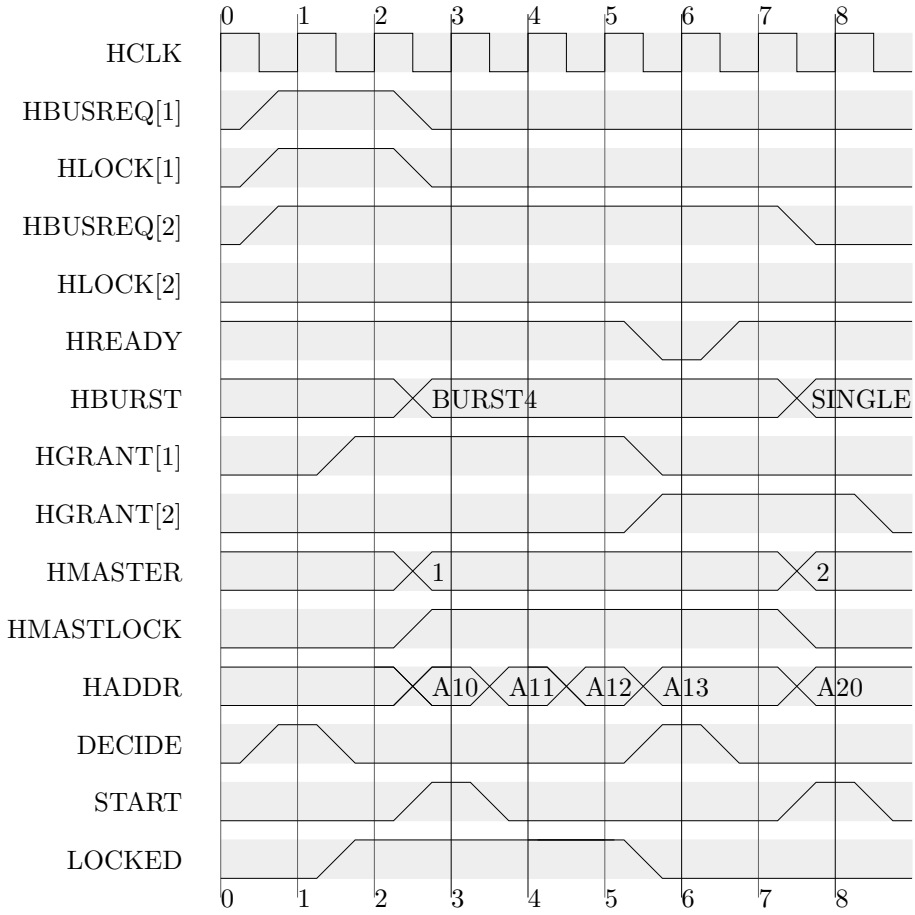

Figure 5.10: An example of AMBA bus behavior

AHB, a slave is allowed to "split" a burst access and request that it be continued later. We have left this feature out for simplicity, but it can be handled by our approach.

A typical set of accesses is shown in Figure 5.10. (Please ignore the DECIDE, START, and LOCKED signals for now.) At time 1, Masters 1 and 2 request an access. Master 1 requests a locked transfer. The access is granted to Master 1 at the next time step, and Master 1 starts its access at time 3. Note that HMASTER changes and HMASTLOCK goes up. The access is a BURST4 that cannot be interrupted. At time 6, when the last transfer in the burst starts, the arbiter prepares to hand over the bus to Master 2 by changing the grant signals. However, HREADY is low, so the last transfer is extended and the bus is only handed over in time step 8, after HREADY has become high again.

5.3.2 Formal Specification

This section contains the specification of the arbiter. To simplify the specification, we have added three auxiliary variables, START, LOCKED, and DECIDE, which are driven by the arbiter. Signal START indicates the start of an access. In Figure 5.10, for instance, START is high in Step 3 and 8 and low otherwise. The master only switches when START is high. The signal LOCKED indicates if the bus will be locked at the next start of an access. Signal DECIDE is described below.

We group the properties into three sets. The first set of properties defines when a new access is allowed to start, the second describes how the bus has to be handed over, and the third describes which decisions the arbiter makes. We distinguish *guarantees*, which are properties that the arbiter must fulfill, and *assumptions*, which are properties that the arbiter's environment must fulfill. A summary of the formal LTL specification is given in Table 5.2.

Starting an Access

Assumption 1. *During a locked unspecified length burst, leaving HBUSREQ[i] high locks the bus. This is forbidden by the standard.*

$$\mathsf{G}((\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{INCR}) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg \mathit{HBUSREQ[HMASTER]})$$

The expression HBUSREQ[HMASTER] is not part of the LTL syntax. We can either rewrite the formula above to $\bigwedge_i \mathsf{G}((\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{INCR} \wedge \mathit{HMASTER} = i) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg \mathit{HBUSREQ[i]})$*, or we introduce a new variable (e.g., BUSREQ) for this expression and obtain the formulas* $\mathsf{G}((\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{INCR}) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg \mathit{BUSREQ})$ *and* $\bigwedge_i \mathsf{G}(\mathit{HMASTER} = i \rightarrow (\mathit{BUSREQ} \leftrightarrow \mathit{HBUSREQ[i]}))$*. We chose the later option, since it made the synthesis computation more efficient.*

Assumption 2. *Leaving HREADY low locks the bus, the standard forbids it.*

$$\mathsf{G}\,\mathsf{F}\,\mathit{HREADY}$$

Assumption 3. *The lock signal is asserted by a master at the same time as the bus request signal.*

$$\bigwedge_i \mathsf{G}(\mathit{HLOCK[i]} \rightarrow \mathit{HBUSREQ[i]})$$

Guarantee 1. *A new access can only start when HREADY is high.*

$$\mathsf{G}(\neg \mathit{HREADY} \rightarrow \mathsf{X}(\neg \mathit{START}))$$

Guarantee 2. *When a locked unspecified length burst starts, a new access does not start until the current master (HMASTER) releases the bus by lowering HBUSREQ[HMASTER].*

$$\mathsf{G}((\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{INCR} \wedge \mathit{START}) \rightarrow$$
$$\mathsf{X}(\neg \mathit{START} \mathsf{W} (\neg \mathit{START} \wedge \neg \mathit{HBUSREQ[HMASTER]})))$$

We tread the expression HBUSREQ[HMASTER] in the same way as in Assumption 1.

Guarantee 3. *When a length-four locked burst starts, no other accesses start until the end of the burst. We can only transfer data when HREADY is high, so the current burst ends at the fourth occurrence of HREADY. (In the formula, we treat the cases where HREADY is true initially separately from the case in which it is not.)*

$$\mathsf{G}((\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{BURST4} \wedge \mathit{START} \wedge \mathit{HREADY}) \rightarrow$$
$$\mathsf{X}(\neg \mathit{START} \mathsf{W}[3](\neg \mathit{START} \wedge \mathit{HREADY})))$$
$$\mathsf{G}((\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{BURST4} \wedge \mathit{START} \wedge \neg \mathit{HREADY}) \rightarrow$$
$$\mathsf{X}(\neg \mathit{START} \mathsf{W}[4](\neg \mathit{START} \wedge \mathit{HREADY})))$$

Granting the Bus

Guarantee 4. *The HMASTER signal follows the grants: When HREADY is high, HMASTER is set to the master that is currently granted. This implies that no two grants may be high simultaneously and that the arbiter cannot change HMASTER without giving a grant.*

$$\bigwedge_i \mathsf{G}(\mathit{HREADY} \rightarrow (\mathit{HGRANT}[i] \leftrightarrow \mathsf{X}(\mathit{HMASTER} = i)))$$

Guarantee 5. *Whenever HREADY is high, the signal HMASTLOCK copies the signal LOCKED.*

$$\mathsf{G}(\mathit{HREADY} \rightarrow (\mathit{LOCKED} \leftrightarrow \mathsf{X}(\mathit{HMASTLOCK})))$$

Guarantee 6. *If we do not start an access in the next time step, the bus is not reassigned*

and HMASTLOCK does not change.

$$\bigwedge_i \mathsf{G}(\mathsf{X}(\neg \mathit{START}) \rightarrow \quad ((\mathit{HMASTER} = i \leftrightarrow \mathsf{X}(\mathit{HMASTER} = i)) \wedge (\mathit{HMASTLOCK} \leftrightarrow \mathsf{X}(\mathit{HMASTLOCK}))))$$

Deciding the Next Access

Signal DECIDE indicates the time slot in which the arbiter decides who the next master will be, and whether its access will be locked. The decision is based on HBUSREQ[i] and HLOCK[i]. (For instance, DECIDE is high in Step 1 and 6 in Figure 5.10.) Note that a decision is executed at the next START signal, which can occur at the earliest two time steps after the HBUSREQ[i] and HLOCK[i] signals are read. (See Figure 5.10, the signals are read in Step 1 and the corresponding access starts at Step 3.)

Guarantee 7. *When the arbiter decides to grant the bus, it uses LOCKED to remember whether a locked access was requested.*

$$\bigwedge_i \mathsf{G}((\mathit{DECIDE} \wedge \mathsf{X}(\mathit{HGRANT}[i])) \rightarrow (\mathit{HLOCK}[i] \leftrightarrow \mathsf{X}(\mathit{LOCKED})))$$

Guarantee 8. *We do not change the grant or locked signals if DECIDE is low.*

$$\mathsf{G}(\neg \mathit{DECIDE} \rightarrow \bigwedge_i (\mathit{HGRANT}[i] \leftrightarrow \mathsf{X}(\mathit{HGRANT}[i])))$$
$$\mathsf{G}(\neg \mathit{DECIDE} \rightarrow (\mathit{LOCKED} \leftrightarrow \mathsf{X}(\mathit{LOCKED})))$$

Guarantee 9. *We have a fair bus. Note that this is not required by the AMBA standard, and there are valid alternatives, such as a fixed-priority scheme. (Without this property, there is no need for the arbiter to serve any master at all.)*

$$\bigwedge_i \mathsf{G}(\mathit{HBUSREQ}[i] \rightarrow \mathsf{F}(\neg \mathit{HBUSREQ}[i] \vee \mathit{HMASTER} = i))$$

Guarantee 10. *We do not grant the bus without a request, except to Master 0. If there are no requests, the bus is granted to Master 0.*

$$\bigwedge_{i \neq 0} \mathsf{G}(\neg \mathit{HGRANT}[i] \rightarrow (\neg \mathit{HGRANT}[i] \mathsf{W}\, \mathit{HBUSREQ}[i]))$$
$$\mathsf{G}((\mathit{DECIDE} \wedge \bigwedge_i \neg \mathit{HBUSREQ}[i]) \rightarrow \mathsf{X}(\mathit{HGRANT}[0]))$$

Guarantee 11. *An access by Master 0 starts in the first clock tick and simultaneously, a decision is taken. Thus, the signals DECIDE, START, and HGRANT[0] are high and all*

Table 5.2: LTL specification of the AMBA arbiter

A1	$\mathsf{G}((\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR}) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg\text{HBUSREQ[HMASTER]})$
A2	$\mathsf{G}\,\mathsf{F}\,\text{HREADY}$
A3	$\bigwedge_i \mathsf{G}(\text{HLOCK[i]} \rightarrow \text{HBUSREQ[i]})$
G1	$\mathsf{G}(\neg\text{HREADY} \rightarrow \mathsf{X}(\neg\text{START}))$
G2	$\mathsf{G}((\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{START}) \rightarrow$ $\mathsf{X}(\neg\text{START}\,\mathsf{W}(\neg\text{START} \wedge \neg\text{HBUSREQ[HMASTER]})))$
G3	$\mathsf{G}((\text{HMASTLOCK} \wedge \text{HBURST} = \text{BURST4} \wedge \text{START} \wedge \text{HREADY}) \rightarrow$ $\mathsf{X}(\neg\text{START}\,\mathsf{W}[3](\neg\text{START} \wedge \text{HREADY})))$ $\mathsf{G}((\text{HMASTLOCK} \wedge \text{HBURST} = \text{BURST4} \wedge \text{START} \wedge \neg\text{HREADY}) \rightarrow$ $\mathsf{X}(\neg\text{START}\,\mathsf{W}[4](\neg\text{START} \wedge \text{HREADY})))$
G4	$\bigwedge_i \mathsf{G}(\text{HREADY} \rightarrow (\text{HGRANT[i]} \leftrightarrow \mathsf{X}(\text{HMASTER} = i)))$
G5	$\mathsf{G}(\text{HREADY} \rightarrow (\text{LOCKED} \leftrightarrow \mathsf{X}(\text{HMASTLOCK})))$
G6	$\bigwedge_i \mathsf{G}(\mathsf{X}(\neg\text{START}) \rightarrow ((\text{HMASTER} = i \leftrightarrow \mathsf{X}(\text{HMASTER} = i)) \wedge$ $(\text{HMASTLOCK} \leftrightarrow \mathsf{X}(\text{HMASTLOCK}))))$
G7	$\bigwedge_i \mathsf{G}((\text{DECIDE} \wedge \mathsf{X}(\text{HGRANT[i]})) \rightarrow (\text{HLOCK[i]} \leftrightarrow \mathsf{X}(\text{LOCKED})))$
G8	$\mathsf{G}(\neg\text{DECIDE} \rightarrow \bigwedge_i(\text{HGRANT[i]} \leftrightarrow \mathsf{X}(\text{HGRANT[i]})))$ $\mathsf{G}(\neg\text{DECIDE} \rightarrow (\text{LOCKED} \leftrightarrow \mathsf{X}(\text{LOCKED})))$
G9	$\bigwedge_i \mathsf{G}(\text{HBUSREQ[i]} \rightarrow \mathsf{F}(\neg\text{HBUSREQ[i]} \vee \text{HMASTER} = i))$
G10	$\bigwedge_{i\neq 0} \mathsf{G}(\neg\text{HGRANT[i]} \rightarrow (\neg\text{HGRANT[i]}\,\mathsf{W}\,\text{HBUSREQ[i]}))$ $\mathsf{G}((\text{DECIDE} \wedge \bigwedge_i \neg\text{HBUSREQ[i]}) \rightarrow \mathsf{X}(\text{HGRANT[0]}))$
G11	$\text{DECIDE} \wedge \text{START} \wedge \text{HGRANT[0]} \wedge \text{HMASTER} = 0 \wedge \neg\text{HMASTLOCK} \wedge$ $\bigwedge_{i\neq 0} \neg\text{HGRANT[i]}$
A4	$\bigwedge_i(\neg\text{HBUSREQ[i]} \wedge \neg\text{HLOCK[i]}) \wedge \neg\text{HREADY}$

others are low.

$$\mathit{DECIDE} \wedge \mathit{START} \wedge \mathit{HGRANT[0]} \wedge \mathit{HMASTER} = 0 \wedge \neg\mathit{HMASTLOCK} \wedge \bigwedge_{i\neq 0} \neg\mathit{HGRANT[i]}$$

Assumption 4. *We assume that all input signals are low initially.*

$$\bigwedge_i(\neg\mathit{HBUSREQ[i]} \wedge \neg\mathit{HLOCK[i]}) \wedge \neg\mathit{HREADY}$$

5.3.3 Synthesis

As explained in Section 5.1.1, not all LTL specifications can be synthesized directly. Rather, we first have to build deterministic monitors for the formulas A1, G2, G3,

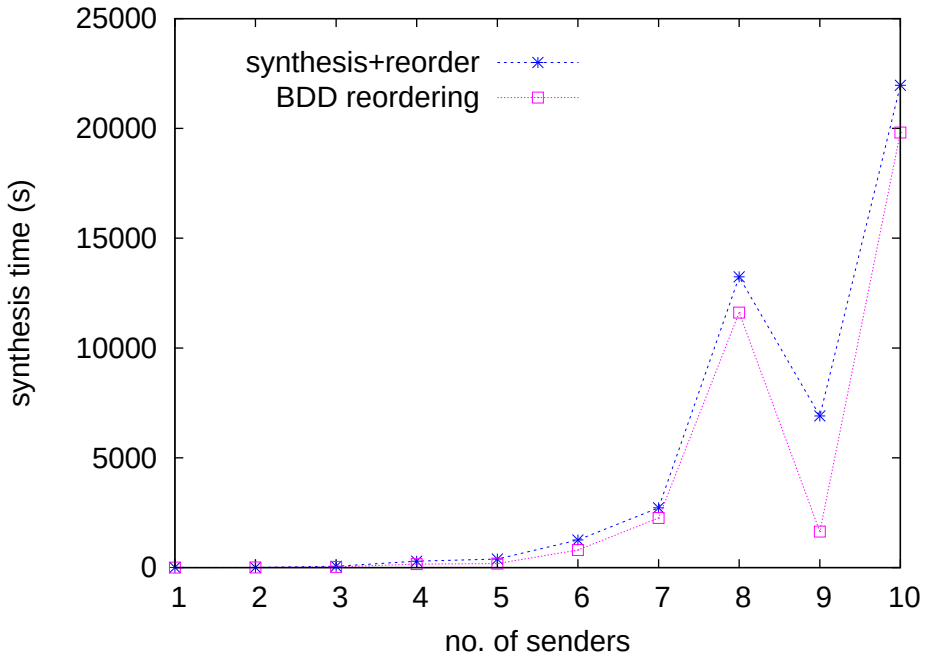

Figure 5.11: Time to synthesize AMBA Arbiter

and G10.

After the specification has been brought into the proper form using the techniques discussed in Section 5.2.3, it is synthesized and a circuit is constructed as described in Section 5.1.2. Subsequently, the circuit is optimized and mapped to standard cells using ABC [Ber].

In our initial experiments ([BGJ+07a]), we were only able to synthesize arbiters for up to four masters, for larger arbiters the synthesis algorithm ran out of memory when building the strategy. (2GB of memory were available.) After rewriting the specification (see Assumption 1) we can handle up to ten masters. The time for synthesis is shown in Figure 5.11 and ranges from a few second to 6.5 hours. Most of the time is spent in reordering BDDs. (We do not know why synthesis for nine masters is faster then for eight.)

In Figure 5.12, we show the areas of the arbiter as a function of the number of masters using our algorithm compared with a manual implementation. For one master the manual and the automatically generated implementation have approximately the same size. The automatically generated implementations grow rapidly with the number of masters, while

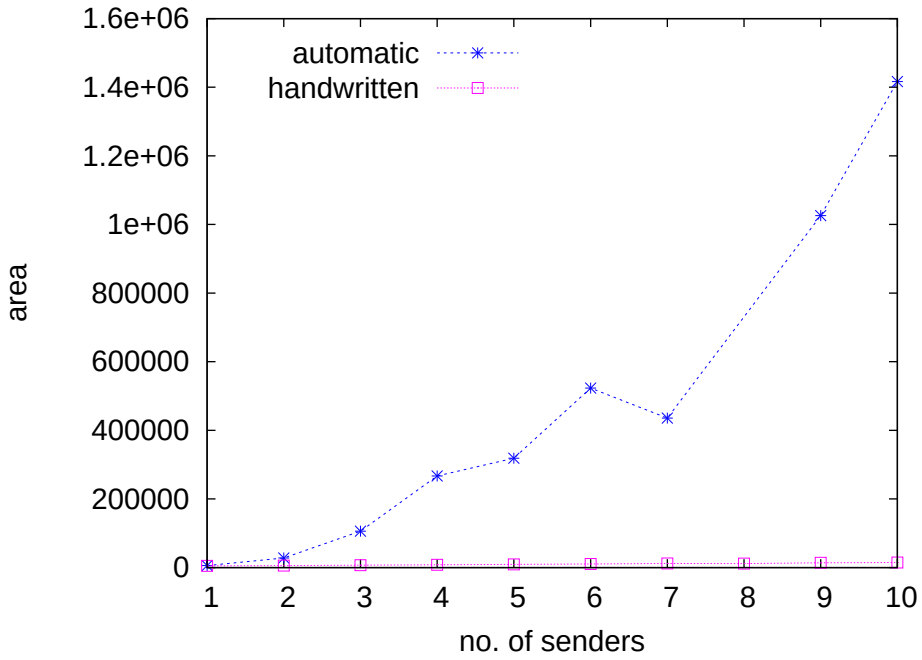

Figure 5.12: Size of the arbiter circuits

the manual implementations are nearly independent of the number of masters. The automatically generated implementation for ten master is about a hundred times larger than the manual implementation.

The automatically generated arbiter implements a round-robin arbitration scheme. This can be explained from the construction of the strategy in the synthesis algorithm, but it is also the simplest implementation of a fair arbiter. We have validated our specification by combining the resulting arbiter with manually written masters and clients, with which it cooperates without problems.

5.4 Discussion

In this section we discuss the most important benefits and drawbacks of automatic synthesis, as we perceive them.

Writing the formal specification for the generalized buffer was straightforward and posed few surprises. This may be ascribed in part to the simplicity of the block and in part to the crisp informal specification given by IBM and in part to the fact that a good part of

the formal specification had been provided (albeit not quite free of mistakes). The need to be able to succinctly specify its behavior may have influenced the design of GenBuf positively.

On the other hand, writing a complete formal specification for the AMBA arbiter was not trivial. Many aspects of the arbiter are not defined in ARM's standard. Such ambiguities would lead to long discussions on how someone implementing a bus device could read the standard, and which behavior the arbiter should allow. Note that the same problem occurs when writing a VERILOG implementation for the arbiter.

Second, it was not trivial to translate the informal specification to formulas. One of the important insights when writing the specification of the arbiter was that additional signals were needed. This problem also occurs when we attempt to formally verify a manually coded arbiter, in which case the same signals are useful. (In fact, these signals occur, in one form or other, in our manual implementation as well.)

The effort for a manual implementation of the generalized buffer does not depend much on the number of senders. (Similarly for the AMBA arbiter and the number of masters.) The same is not true for automatic synthesis: the time to synthesize GenBuf grows quickly with the number of masters as does the size of the generated circuit. Unfortunately, the generated gate-level output is complicated and cannot be changed by hand. The resulting circuit can likely be improved further by using more intelligent methods to generate the circuits, which will be important if this methodology is to become accepted. The problem is related to synthesis of partially specified functions [HS96] with the important characteristic that the space of allowed functions is very large.

On the upside, the resulting LTL specification is short, readable, and easy to modify, much more so than a manual implementation in VERILOG. For the arbiter in particular, we expect that it is easier to learn the way the design functions from the formal specification than from a manual VERILOG implementation. The synthesis algorithm was also a great tool to get the specifications to be consistent and complete. We doubt we would have gotten to a complete and consistent specification without the synthesis tool.

Automatic synthesis is first and foremost applicable to control circuitry. We are looking into methods to beneficially combine manually coded data paths with automatically synthesized control circuitry.

Although this approach removes the need for verification of the resulting circuit, the specification itself still needs to be validated. The lack of tools for debugging specifications was apparent in our exercise. Some work on such tools has taken place [PSC+06], but

further research, in particular in connection with realizability, is needed.

Chapter 6

Conclusion and Outlook

Automatic synthesis of reactive systems from (temporal) logical specifications has always engaged the imagination of computer scientists and has been considered as one of the most ambitious and challenging problems in system design. However once the high complexity of the problem was established, synthesis did not attract interest anymore for a long time. Only in the last two years, new very promising algorithms [KV05, PPS06] have been developed. The work presented here is partly based on those algorithms and has the general goal to bring LTL synthesis closer to practice. It consists of three parts.

In the first part, we have presented the first implementation of a synthesis tool for full LTL. We described a set of optimizations for tree automata, including a game-based heuristic to check language emptiness and simulation-based state-space reductions. We have shown how these optimizations make a major difference in the efficiency of the implementation. We have evaluated our tool on a set of hand-written examples. Although our examples are still small, we believe that the optimizations discussed here form an important step towards making LTL synthesis practical.

More work remains to be done on further increasing the efficiency, debugging of specifications (especially of unrealizable ones), and effectively combining specifications with hand-written HDL code. It would also be interesting to investigate in how far our optimizations can be applied to the compositional version of the Safraless approach [KPV06].

In the second part, we have stated the repair problem and showed its relation to the synthesis problem. The repair problem is concerned with fixing a faulty system such that it adheres to its specification. Assuming a given suspect, we proceed by building the product of a game corresponding to the broken system and the automaton reflecting the specification. If the product game has a winning strategy, we can repair the system.

However, a strategy may not exist for the product even if a repair exists because of nondeterminism in the automaton. We could circumvent this problem by determinizing the automaton, but the cost is exponential and for many combinations of program and specification, nondeterminism turns out not to be problematic.

A winning finite state strategy correspond to a repair that introduces new state. We reject the possibility of changing the system logic and instead turn to the problem of finding a memoryless strategy. We have shown that deciding whether a memoryless strategy exists is NP-complete, and we have presented a conservative heuristic that conjoins the strategies for the different states of the automaton. We have described a heuristic that finds an efficient repair for a given memoryless strategy.

The algorithm is of a complexity that is comparable to that of model checking, which makes us optimistic as to the practical applicability of the approach. We have implemented a symbolic version of the algorithm and the initial experimental results show that the algorithm finds readable repairs in acceptable time, though improvements in the implementation are still possible.

In follow-up work, Staber et al. [SJB05] combine the approach with fault localization and provide a way to find and fix faults. Furthermore, Griesmayer et al. [GBC06] extend the approach to push-down games. Their approach handles recursions that appear in Boolean programs generated by a SLAM-like abstraction/refinement process [BR01].

An important question, which remains open, is in how far we can minimize the negative effects of using a finite state strategy, e.g., by using a dependent variable analysis [HD93] (cf. Section 5.1.2) to minimize the amount of added state. Extending the idea to a more general framework, where multiple repair locations are allowed in combination with cost functions for "good" repairs and least-cost games, could be also worth while. We are also looking into further improvements in the efficiency of the implementation.

In the third part, we have presented how automatic synthesis of hardware designs can actually work. When specifications are available early, synthesis can be used to obtain a first implementation, yielding a functional test environment when critical blocks are replaced by manual implementations. Furthermore, these implementations function as a valuable sanity check for the specification, which is very important when a future manual implementation is to be based on the formal specification.

The circuits we synthesized are quite large, but the approach is still very young and only a few avenues for optimization have been pursued. We attempted to generate circuits using an approach by [KS00]. A second attempt using cofactors yielded circuits that are

an order of magnitude smaller, and optimizations to that approach reduce the size of the circuit up to a third. We expect that future research will yield further large improvements, making automatic synthesis a real alternative to manual coding of some types of circuits.

The three parts of this thesis show the broad range of research options in LTL synthesis. Starting from a rather theoretical work on optimizations for LTL synthesis, via the new repair application, to system construction as the classical application of synthesis. Bringing synthesis to practice holds a variety of interesting problems and even though the way to practice is still long, it is definitely short than before this thesis.

Bibliography

[AHKV98] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *Proc. 9th Conferance on Concurrency Theory*, pages 163–178, Nice, September 1998. Springer-Verlag. LNCS 1466.

[AL01] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. In *Symposium on Logic in Computer Science (LICS'01)*, pages 291–302, 2001.

[AMJB05] G. Auerbach, M. Moulin, B. Jobstmann, and R. Bloem. Property-based design and implementation. Technical Report Prosyd D 2.1/1, 2005.

[ATW06] C. Schulte Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of Büchi automata. *Theoretical Computer Science*, 363:224–233, 2006.

[B+96] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[Bau06] J. Baumgartner. Integrating FV into main-stream verification: The IBM experience, 2006. Invited Talk at the Conference on Formal Methods in Computer Aided Design (FMCAD'06).

[BEGL99] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artificial Intelligence*, 112:57–104, 1999.

[Ber] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification, release 61208. http://www.eecs.berkeley.edu/~alanmi/abc/.

[BGJ+06] R. Bloem, S. Galler, B. Jobstmann, A. Pnueli, and M. Weiglhofer. Evaluation of tools and methodology for property-based logic synthesis. Technical Report Prosyd D 2.3/1, Graz University of Technology, 2006.

[BGJ+07a] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[BGJ+07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, pages 3–16, 2007.

[BJ06] R. Bloem and B. Jobstmann. Manual for property-based synthesis tool. Technical Report Prosyd D 2.2/3, 2006.

[BJP05] R. Bloem, B. Jobstmann, and A. Pnueli. Property-based logic synthesis for rapid design prototyping. Technical Report Prosyd D 2.2/1, 2005.

[BL69] J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[BNR03] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *30th Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, January 2003.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *8th International SPIN Workshop*, pages 103–122, Toronto, May 2001. Springer-Verlag. LNCS 2057.

[CFTD93] L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1499. Morgan-Kaufmann, 1993.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[Chu62] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[CKW05] R. Chen, D. Köb, and F. Wotawa. A comparison of fault explanation and localization. unpublished, 2005.

[DGV99] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear time temporal logic. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 249–260. Springer-Verlag, Berlin, 1999. LNCS 1633.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer-Verlag, 2006.

[EH00] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Proc. 11th International Conference on Concurrency Theory (CONCUR2000)*, pages 153–167. Springer, 2000. LNCS 1877.

[FHW80] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[Fri03] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *Conference on Implementation and Application of Automata (CIAA'03)*, pages 35–48, 2003. LNCS 2759.

[FW02] C. Fritz and T. Wilke. State space reductions for alternating Büchi automata. In *Foundations of Software Technology and Theoretical Computer Science*, pages 157–168, Kanpur, India, December 2002. Springer-Verlag. LNCS 2556.

[GBC06] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In *18th Conference on Computer Aided Verification (CAV'06)*, pages 358–371, 2006. LNCS 4144.

[GBS02] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 610–623. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[GH82] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proc. 14th ACM Symp. Theory of Comp.*, pages 60–65, San Francisco, CA, 1982.

[GKSV03] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In *Correct Hardware Design and Verification Methods (CHARME'03)*, pages 96–110, Berlin, October 2003. Springer-Verlag. LNCS 2860.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Thirteenth Conference on Computer Aided Verification (CAV '01)*, pages 53–65. Springer-Verlag, 2001. LNCS 2102.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[Gro04] A. Groce. Error explanation with distance metrics. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 108–122, Barcelona, Spain, March-April 2004. LNCS 2988.

[GSB06] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science*, 174:95–111, 2006. Workshop on Verification and Debugging (V&D'06).

[GV03] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking of Software: 10th International SPIN Workshop*, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[Har05] A. Harding. *Symbolic Strategy Synthesis For Games With LTL Winning Conditions*. PhD thesis, University of Birmingham, 2005.

[HD93] A. J. Hu and D. Dill. Reducing BDD size by exploiting functional dependencies. In *Proceedings of the Design Automation Conference*, pages 266–271, Dallas, TX, June 1993.

[HP06] T. A. Henzinger and N. Piterman. Solving games without determinization. In *15th Conference on Computer Science Logic*, pages 394–409, Szeged, Hungary, September 2006. LNCS 4207.

[HRS05] A. Harding, M. Ryan, and P.-Y. Schobbens. A new algorithm for strategy synthesis in LTL games. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 477–492, Edinburgh, UK, 2005. LNCS 3440.

[HS96] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, MA, 1996.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[JB06a] B. Jobstmann and R. Bloem. Game-based and simulation-based improvements for LTL synthesis. In *3rd Workshop on Games in Design and Verification (GDV'06)*, 2006.

[JB06b] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[JGB05] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.

[Job06] B. Jobstmann. Property synthesis. In *9th SIGDA Ph.D. Forum at the Design Automation Conference (DAC 2006)*, San Francisco, California, USA, 2006.

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 445–459, Grenoble, France, April 2002. LNCS 2280.

[JSGB07] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 2007. Accepted for publication.

[KB05] J. Klein and C. Baier. Experiments with deterministic omega-automata for formulas of linear temporal logic. In *Conference on Implementation and Application of Automata (CIAA'05)*, pages 199–212, 2005. LNCS 3845.

[Kla91] N. Klarlund. Progress measures for complementation of ω-automata with application to temporal logic. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 358–367, San Juan, 1991.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KPV06] O. Kupferman, N. Piterman, and M. Y. Vardi. Safraless compositional synthesis. In *Conference on Computer Aided Verification (CAV'06)*, pages 31–44, 2006.

[KS00] J. H. Kukula and T. R. Shiple. Building circuits from relations. In *12th Conference on Computer Aided Verification (CAV'00)*, pages 113–123, 2000. LNCS 1855.

[KV98] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[KV05] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Symposium on Foundations of Computer Science (FOCS'05)*, pages 531–542, 2005.

[KVW00] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[Löd99] C. Löding. Optimal bounds for transformations of omega-automata. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 97–109, 1999.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM*

Symposium on Principles of Programming Languages, pages 97–107, New Orleans, January 1985.

[Ltd99] ARM Ltd. AMBA specification (rev. 2). Available from www.arm.com, 1999.

[Mai00] M. Maidl. The common fragment of CTL and LTL. In *Proc. 41th Annual Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words. Manuscript, CNET, Paris, 1988.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems *Specification**. Springer-Verlag, 1991.

[MS95] D. E. Muller and P. E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141:69–107, 1995.

[MSW00] C. Mateis, M. Stumptner, and F. Wotawa. A value-based diagnosis model for Java programs. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis*, 2000.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions Programming Languages and Systems*, 6:68–93, 1984.

[Pit06] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21st Symposium on Logic in Computer Science*, pages 255–264, Seattle, WA, August 2006.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[PSC+06] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *Design Automation Conference*, pages 821–826, 2006.

[Rab72] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*. Regional Conference Series in Mathematics. American Mathematical Society, Providence, RI, 1972.

[RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, November 2000. LNCS 1954.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[Saf88] S. Safra. On the complexity of ω-automata. In *Symposium on Foundations of Computer Science*, pages 319–327, October 1988.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 3(32):733–749, 1985.

[SFBD06] S. Staber, G. Fey, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. In *Second International Haifa Verification Conference (HVC 2006)*, pages 50–64, 2006.

[SJB05] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In D. Borrione and W. Paul, editors, *13th Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, pages 35–49. Springer-Verlag, 2005. LNCS 3725.

[Som] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[ST03] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *Correct Hardware Design and Verification Methods (CHARME'03)*, pages 126–140, Berlin, October 2003. Springer-Verlag. LNCS 2860.

[SW96] M. Stumptner and F. Wotawa. A model-based approach to software debugging. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*, 1996.

[TBK95] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, April 1995.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13. Springer-Verlag, 1995. LNCS 900.

[Var01] M. Y. Vardi. Branching vs. linear time: Final showdown. *Lecture Notes in Computer Science*, 2031:1–22, 2001.

[WHT03] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proceedings of the International Conference on the Implementation and Application of Automata*. Springer-Verlag, 2003.

[WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.