Symbolic LTL Reactive Synthesis

Master Thesis by

Remco Abraham

Supervised by dr. T. van Dijk and dr. rer. nat. S. Sickert

UNIVERSITY OF TWENTE.

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Formal Methods and Tools (FMT)
Enschede, the Netherlands

2021

Graduation Committee:

prof. dr. M. Huisman (Chair)
Formal Methods and Tools
University of Twente

dr. T. van Dijk
Formal Methods and Tools
University of Twente

dr. rer. nat. S. Sickert
School of Computer Science & Engineering
The Hebrew University of Jerusalem

dr. A. Skopalik
Discrete Mathematics and Mathematical Programming
University of Twente

Acknowledgments

I am especially and equally grateful to both my supervisors, Tom van Dijk and Salomon Sickert, for their excellent support, feedback and suggestions. Tom introduced me to Salomon and many others in the field of reactive synthesis, which provided me with many valuable resources and contacts that I would not have been able to acquire alone. He has provided valuable feedback and technical support throughout the whole process. Even though this thesis was written during a pandemic, he ensured that the complete process was an enjoyable experience.

I thank Salomon for the suggestion to combine the LTL normalization with the results of Boker et al. to construct a DPW. It is the key and novel element of this construction. I furthermore thank him for his intensive involvement in this thesis. He provided various pointers to recent developments in the field and has supported the development of this construction as part of the LTL and ω-automata library Owl. During development, he has provided much technical feedback which has made the development of the construction a valuable learning experience not only for research but also for engineering.

Finally, I would like to acknowledge others that have been of great help in this thesis. I thank Marieke Huisman for her extensive feedback on the draft of this thesis. I also gratefully acknowledge the reviewing efforts of Rick de Vries during the preliminary literature study. I would like to recognize the assistance of Swen Jacobs, who has introduced me to the idea of bounded synthesis, and Philipp Meyer, who has occasionally provided technical support during the development of the construction. Lastly, I acknowledge the indirect help of Orna Kupferman, who has suggested using the work of Boker et al. to Salomon.

Abstract

A major goal in theoretical computer science is to obtain systems, such as software or hardware, that are provably correct. One approach to reach this goal is to *synthesize* systems directly from their specifications. In particular, we consider reactive systems, which are systems that respond to events in their environment, such as infrastructure controllers, medical devices and communication protocols. Properties of reactive systems are typically described in temporal logics such as linear temporal logic (LTL). Unfortunately synthesizing a controller from an LTL formula is computationally expensive and was deemed infeasible by many. However, recent developments raise a renewed interest in this topic, aiming at scalable LTL synthesis for use beyond small academic examples.

In nearly all domains of verification and synthesis, the state space explosion problem dominates: to reason about slightly larger systems, we need to consider exponentially many more possible system states. A method that can tackle this state space explosion is symbolic reasoning with e.g. binary decision diagrams (BDDs). BDDs can be used to represent sets of states efficiently, and allow for reasoning over these sets directly without explicitly enumerating the states within them.

In this thesis, we apply and evaluate a novel and partially symbolic LTL synthesis construction that uses a recently discovered normalization technique for LTL formulas. This allows us to divide the formula into fragments that are in lower classes of the temporal logics hierarchy which are relatively easy to translate to deterministic ω-automata. We symbolically combine these ω-automata to obtain a symbolic deterministic parity automaton which we convert to a game and solve. This results in a strategy that can be trivially translated to a Mealy machine adhering to the specification.

We develop the construction in Otus, which is a new prototypical LTL reactive synthesis tool based on the LTL and ω-automata library Owl. We evaluate it against Strix, which is another reactive synthesis tool applying a similar but explicit approach. Strix represents the state of the art as it has won the 2018, 2019 and 2020 SYNTCOMP synthesis competition and is, therefore, an interesting comparison target. The results are mixed but very promising: although Otus is not optimized, we find examples where Otus is in the order of 10 times faster than Strix.

Contents

CHAPTER 1

Introduction

Computer systems play an ever-increasing role in our daily lives. We use these systems not only for simple conveniences but also to manage critical infrastructure and to guarantee our safety. It is therefore of vital importance that we can rely on these systems to behave correctly. One of the main goals in theoretical computer science and the primary goal in the field of formal methods is therefore the ability to produce provably correct systems.

One popular approach involves proving for an existing system that it adheres to some specification. Usually, this entails formally modelling the system such that the model accurately reflects the system's behaviour. Automata-based models are frequently used to model systems because their semantics are close to real-world systems and they can be verified against some formal specification. Modelling a system and verifying it against a specification is well-known as *model-checking* [25]. The benefit of model-checking is that it is usually fully automatic. The major problem with this approach, however, is that the models of the system become extremely large in practice. This gives rise to the infamous state space explosion problem. Nevertheless, model-checking has seen successful applications in practice already [55].

Instead of verifying a system against its specification, another approach is to construct a system directly from it. Assuming that the constructed system is of good quality (i.e. is reasonably small) and it is possible to provide a complete specification, this approach is preferable over model-checking as it eliminates the need for the development of a system in the first place. Although these assumptions are still challenges on their own [48,51], we focus on the scalability of this technique since the state space explosion problem dominates here as well.

The process of automatically transforming a specification to a system that adheres to the specification is known as *synthesis*. Of particular interest are those systems that continuously respond to stimuli from their environment, as these systems are often found in domains where a malfunction can have catastrophic consequences. Example domains include but are not limited to medical devices, infrastructure systems such as railway and aviation controllers, and spacecrafts. This branch of synthesis is commonly referred to as *reactive* synthesis.

In this thesis, we consider the synthesis of reactive systems from linear temporal logic (LTL), which is a logic that allows the expression of properties over time. A typical approach consists of translating the LTL specification to a deterministic parity automaton, which is an automaton that runs on infinite words. This is then converted to a 2-player game where the system plays against its environment such that the specification is satisfiable if and only if the winner of this

game is the system player. The winning strategy of the system player is then converted to an implementation of a reactive system that adheres to the specification.

Unfortunately, this approach does not scale well. To improve the scalability, *symbolic* methods are often applied. These are methods that allow for a compact representation of sets. A primary example of such a method is the encoding of sets in *binary decision diagrams* [56]. A binary decision diagram (BDD) is an efficient representation for propositional formulas, which can also be used to encode sets. Using BDDs has shown to be very successful in model-checking [80]. It has also already been applied to reactive synthesis by Sohail et al. [87] and Morgenstern [70].

Both do not translate the full specification to a deterministic parity automaton directly as this is not efficiently implementable symbolically. Instead, they try to decompose the specification into fragments of LTL that have a simple translation to automata. They translate each fragment separately and then symbolically compute the product to obtain a game for the full specification. However, if there is no trivial decomposition, they use less efficient approaches that involve a direct translation of the full specification.

Contribution. In this thesis, we revisit the approach by Sohail et al. and Morgenstern. Instead of relying on the specifications being trivially decomposable, we use a new normalization technique for LTL [86] to extract LTL fragments from any specification. This means that we can decompose any LTL formula into fragments and no longer need some inefficient fallback strategy.

Apart from applying this normalization technique, our work differs from previous work in the way we construct the game. Whereas previously the constructed game was a generalized version of a so-called *parity game*, we use the work of Boker et al. [13] to construct a deterministic parity automaton, which we convert to a normal parity game instead. This is beneficial as most parity game algorithms are not designed for generalized parity games and finding a solution for a normal parity game is more efficient. So far, no other implementation has applied this result even though it lends itself for a symbolic implementation, unlike many other approaches that rely on some form of appearance records to construct a deterministic parity automaton.

We implement this new approach in a new prototypical reactive synthesis tool Otus.[1] We compare it against Strix [59], the winner of the 2018-2020 SYNTCOMP reactive synthesis competitions [39, 41]. Even though the tool is not fully optimized, we observe that for certain specifications Otus is in the order of $10\times$ faster than Strix, showing that the approach is promising.

Outline. We structure the rest of the thesis as follows. In Chapter 2, we cover the preliminaries that are needed for a more detailed discussion on previous work and our contribution to it. We then describe the related work, and we motivate and discuss our approach in Chapter 3. Chapters 4-6 then describe the complete construction in detail. Next, we provide details on the implementation in Chapter 7 and an empirical evaluation is presented in Chapter 8. Finally, we reflect on our approach and provide pointers for future research in Chapter 9, and conclude in Chapter 10.

[1] The tool and all results are available on `http://doi.org/10.5281/zenodo.5046346`.

CHAPTER 2

Preliminaries

This chapter discusses a few preliminaries that are essential for a fruitful discussion about the related work in reactive synthesis and our contribution to it. The reader needs to have a good understanding of these topics before proceeding with the remainder of the thesis. Experienced readers may want to skim through this chapter to get familiar with the definitions and notations that will recur throughout this thesis.

This chapter is structured as follows. We start with some remarks on our notation. We then introduce linear temporal logic, which is used to provide the specifications for reactive synthesis. Finally, we discuss ω-automata and parity games as they form the foundation of many reactive synthesis approaches, including ours.

Since this thesis builds upon a lot of preliminary knowledge, we only cover the most essential topics in this chapter. Any remaining preliminaries are covered in later chapters before they are first used such as to not overwhelm the reader in this chapter.

2.1 Notational Remarks

In this thesis, we use various concepts from set theory, propositional and first-order logic for which we use conventional notation that need no additional explanation. We assume the reader is familiar with these concepts. Furthermore, we assume an undergraduate level of understanding of language and automata theory, and refer to Sudkamp [90] for a recap if needed. Notation for new concepts is introduced when it is first used but we introduce some generic notation here.

First of all, we use a non-standard notation for lists. We denote a list of two elements x_0 and x_1 by $[x_0, x_1]$. Furthermore, let L be a list, then L_x is the xth element in L starting at 0. We use $\sqcup$ to denote an append operation, so given two lists L and M of sizes l and m respectively, $L \sqcup M = [L_0, L_1, \ldots, L_{l-1}, M_0, M_1, \ldots, M_{m-1}]$. Finally, given elements x_i for $0 \leq i < n$, we use $\bigsqcup_{i=0}^{n}[x_i]$ to denote the list $[x_0, x_1, \ldots, x_n]$.

Next, we occasionally use a compact notation to represent sets. Suppose we have some set of two elements $\{a, b\}$, then we sometimes use ab as a shorthand notation instead. Furthermore in the context of $\{a, b\}$, $a\bar{b}$ represents the subset $\{a\}$. This is used in particular as a shorthand notation for a symbol in a word since a symbol is a subset of atomic propositions as will become clear in subsequent sections.

2.2 Linear Temporal Logic

Temporal logics are logics that allow reasoning over time. They are well suited for expressing specifications for reactive systems. We roughly identify two categories of temporal logics: those that reason about states in the system, i.e. branching-time temporal logics, and those that reason about paths, i.e. linear-time temporal logics. Although both types have their limitations [97], linear-time temporal logics are more popular nowadays [11]. In this thesis, we therefore also focus on a linear-time temporal logic, namely *linear temporal logic* (LTL). We will first define LTL and then we define a hierarchy of LTL fragments.

2.2.1 Definition

LTL comes in many variations, but all variations are constructed from propositional logic formulas combined with some temporal operators. Many temporal operators exist, and many of them are in some way equivalent to some other temporal operators. In this thesis, we define LTL as follows:

DEFINITION 2.1 *Given some finite set of atomic propositions AP, the following grammar defines LTL recursively:*

$$\varphi ::= a \mid \bar{a} \mid \top \mid \bot \mid \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \boldsymbol{X}\, \varphi \mid \varphi\, \boldsymbol{U}\, \varphi \mid \boldsymbol{F}\, \varphi \mid \boldsymbol{G}\, \varphi \mid \varphi\, \boldsymbol{R}\, \varphi \mid \varphi\, \boldsymbol{M}\, \varphi \mid \varphi\, \boldsymbol{W}\, \varphi$$

where a is any element of AP

The atomic propositions AP define the alphabet $\Sigma = 2^{AP}$. LTL formulas reason about infinite words where each symbol is in Σ. Suppose we have a word $w = w_0 w_1 \ldots$ and $AP = \{a\}$, then the semantics of the LTL formula a are such that $w \models a \iff a \in w_0$. Similarly, $w \models \bar{a} \iff a \notin w_0$. The semantics of $\mathbf{X}\, \varphi$ are such that $w \models \mathbf{X}\, \varphi \iff w_1 w_2 \ldots \models \varphi$, i.e. $\mathbf{X}\, \varphi$ is true if and only if φ is true for the ne**X**t symbol of the word. $\varphi\, \mathbf{U}\, \psi$ is true if and only if there is some future point in time i such that for all states from the current state until (but not including) i, φ is true, and at time i, ψ is true. Thus, $\varphi\, \mathbf{U}\, \psi$ expresses that φ is true **U**ntil ψ is true. Formally:

$$w \models \varphi\, \mathbf{U}\, \psi \iff \exists\, i \geq 0.\ (w_i w_{i+1} \ldots \models \psi) \wedge \forall\, 0 \leq k < i.\ (w_k w_{k+1} \ldots \models \varphi)$$

.Finally, $\varphi\, \mathbf{R}\, \psi$ is true if and only if ψ is true up until and including the point that φ becomes true, or φ never becomes true and ψ remains true forever. Thus, φ **R**eleases ψ. Formally:

$$\begin{aligned} w \models \varphi\, \mathbf{R}\, \psi \iff & (\exists\, i \geq 0.\ (w_i w_{i+1} \ldots \models \varphi) \wedge \forall\, 0 \leq k \leq i.\ (w_k w_{k+1} \ldots \models \psi)) \\ & \vee (\forall\, i \geq 0.(w_i w_{i+1} \ldots \models \psi) \wedge (w_i w_{i+1} \cdots \not\models \varphi)) \end{aligned}$$

We define the other temporal operators in terms of the until and release operators, but first, we give some intuition for their meaning.

$\mathbf{G}\, \varphi$ means φ holds in all symbols (i.e. holds **G**lobally) and $\mathbf{F}\, \varphi$ means φ holds some time

in the future (i.e. **F**inally holds). $\varphi\ \mathbf{M}\ \psi$ is the same as $\varphi\ \mathbf{R}\ \psi$ except that $\varphi\ \mathbf{M}\ \psi$ requires ψ to eventually become true, i.e. a *strong release*. Finally $\varphi\ \mathbf{W}\ \psi$ is the **W**eak until operator: it is the same as $\varphi\ \mathbf{U}\ \psi$ but ψ may also never become true, in which case φ has to be true forever. We formalize the semantics in terms of **U** and **R** as follows:

LEMMA 2.2 [3] *Semantics of LTL operators in terms of **U** and **R**:*

$$\boldsymbol{F}\ \varphi \equiv \top\ \boldsymbol{U}\ \varphi \qquad \varphi\ \boldsymbol{M}\ \psi \equiv \psi\ \boldsymbol{U}\ (\varphi \wedge \psi)$$
$$\boldsymbol{G}\ \varphi \equiv \bot\ \boldsymbol{R}\ \varphi \qquad \varphi\ \boldsymbol{W}\ \psi \equiv \psi\ \boldsymbol{R}\ (\varphi \vee \psi)$$

The remaining operators allow for boolean combinations of the temporal operators. Note that we only allow for negation on an atomic proposition. At first, this may seem restrictive, but it turns out that it is possible to transform any LTL formula with negations on higher levels to an LTL formula with negations only on the atomic proposition by applying the so-called *duality rules* [3]. This form is often referred to as positive normal form or negated normal form. Since this transformation has a linear complexity and yields a formula of the same size, we can assume w.lo.g. that every LTL formula is in this form. We, therefore, define negation simply as a recursive syntactic substitution as below. Furthermore, this allows us to use implication and bi-implication operators defined as usual in terms of negation, conjunction and disjunction.

DEFINITION 2.3 [3] *For any LTL formula φ we define the negation $\neg\varphi$ by a recursive application of the duality rules as:*

$$\neg\bot = \top$$
$$\neg\top = \bot$$
$$\neg a = \bar{a}$$
$$\neg\bar{a} = a$$
$$\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$$
$$\neg(\varphi_1 \vee \varphi_2) = \neg\varphi_1 \wedge \neg\varphi_2$$
$$\neg\boldsymbol{X}\ \varphi_1 = \boldsymbol{X}\ \neg\varphi_1$$

$$\neg(\boldsymbol{F}\ \varphi_1) = \boldsymbol{G}\ \neg\varphi_1$$
$$\neg(\boldsymbol{G}\ \varphi_1) = \boldsymbol{F}\ \neg\varphi_1$$
$$\neg(\varphi_1\ \boldsymbol{U}\ \varphi_2) = \neg\varphi_1\ \boldsymbol{R}\ \neg\varphi_2$$
$$\neg(\varphi_1\ \boldsymbol{R}\ \varphi_2) = \neg\varphi_1\ \boldsymbol{U}\ \neg\varphi_2$$
$$\neg(\varphi_1\ \boldsymbol{M}\ \varphi_2) = \neg\varphi_1\ \boldsymbol{W}\ \neg\varphi_2$$
$$\neg(\varphi_1\ \boldsymbol{W}\ \varphi_2) = \neg\varphi_1\ \boldsymbol{M}\ \neg\varphi_2$$

Although the definition of LTL that we use here is fairly common, there are many variations. For example, several other operators are commonly used in literature: most notably a past-time operator "previous" which comes with past-time equivalents of all future-time operators. Although certainly helpful in some cases, adding past-time operators to LTL does not increase the expressiveness [57]. Still, LTL with past-time operators is exponentially more succinct than LTL without past-time operators [61]. For simplicity, however, we do not consider those operators.

EXAMPLE 2.4 A classic example to illustrate LTL is by means of a coffee machine. Suppose we have a coffee machine that after insertion of a coin gives out a cup of coffee. If we assume that our set of atomic propositions is $\{coin, coffee\}$ where *coin* indicates that we inserted a coin and *coffee* that we received a coffee, then we can formalize the property that after an insertion of a

coin we receive a coffee as $\mathbf{G}\ (\mathit{coin} \rightarrow \mathbf{F}\ \mathit{coffee})$. The property expresses that at any time point in time, if a coin is inserted then sometime later a coffee is received.

Note that the property does not state that each coin gives exactly one coffee, since inserting five coins and receiving a single coffee still satisfies this specification, as does receiving unlimited amounts of coffee without inserting any coins. This specification is therefore not complete for the typical behaviour of a coffee machine. □

2.2.2 Hierarchy

LTL formulas can be classified in various classes of expressiveness using the temporal property hierarchy [60]. We will see that this classification is essential for an efficient construction of automata from these formulas. We can use a simple syntactic definition for these fragments of LTL:

DEFINITION 2.5 [60] *Given some set of atomic propositions AP, we define LTL fragment X as the set of LTL formulas generated from the grammar rule X below, where φ stands for any propositional formula over AP.*

$$\Sigma_1 ::= \varphi \mid \boldsymbol{X}\ \Sigma_1 \mid \Sigma_1\ \boldsymbol{M}\ \Sigma_1 \mid \Sigma_1\ \boldsymbol{U}\ \Sigma_1 \mid \boldsymbol{F}\ \Sigma_1 \mid \Sigma_1 \wedge \Sigma_1 \mid \Sigma_1 \vee \Sigma_1$$
$$\Pi_1 ::= \varphi \mid \boldsymbol{X}\ \Pi_1 \mid \Pi_1\ \boldsymbol{R}\ \Pi_1 \mid \Pi_1\ \boldsymbol{W}\ \Pi_1 \mid \boldsymbol{G}\ \Pi_1 \mid \Pi_1 \wedge \Pi_1 \mid \Pi_1 \vee \Pi_1$$
$$\Delta_1 ::= \Sigma_1 \mid \Pi_1 \mid \Delta_1 \wedge \Delta_1 \mid \Delta_1 \vee \Delta_1$$
$$\Sigma_2 ::= \boldsymbol{X}\ \Sigma_2 \mid \Sigma_2\ \boldsymbol{M}\ \Sigma_2 \mid \Sigma_2\ \boldsymbol{U}\ \Sigma_2 \mid \boldsymbol{F}\ \Sigma_2 \mid \Sigma_2 \wedge \Sigma_2 \mid \Sigma_2 \vee \Sigma_2 \mid \Delta_1$$
$$\Pi_2 ::= \boldsymbol{X}\ \Pi_2 \mid \Pi_2\ \boldsymbol{R}\ \Pi_2 \mid \Pi_2\ \boldsymbol{W}\ \Pi_2 \mid \boldsymbol{G}\ \Pi_2 \mid \Pi_2 \wedge \Pi_2 \mid \Pi_2 \vee \Pi_2 \mid \Delta_1$$
$$\Delta_2 ::= \Sigma_2 \mid \Pi_2 \mid \Delta_2 \wedge \Delta_2 \mid \Delta_2 \vee \Delta_2$$

These fragments form a hierarchy as shown on the left side of Figure 2.1. Using terminology from [60], we can give an intuitive description for each of the classes. We start with the classes that are lowest in the hierarchy, namely Π_1 and Σ_1. These classes represent *safety* and *guarantee* properties respectively. Each Π_1-formula is equivalent to some LTL formula of the form $\mathbf{G}\ \varphi$, where φ only reasons about finite prefixes. Similarly, Σ_1-formulas are equivalent to an LTL formula of the form $\mathbf{F}\ \varphi$. Δ_1-formulas, also known as *obligation* properties, are boolean combinations of Σ_1- and Π_1-formulas. Any Δ_1-formula is equivalent to an LTL formula that is a conjunction of disjunctions of a Π_1- and a Σ_1-formula.

The next layer of the hierarchy contains Σ_2- and Π_2-formulas. They represent *persistence* and *recurrence* properties respectively, with each property having a respective equivalent formula of the form $\mathbf{FG}\ \varphi$ and $\mathbf{GF}\ \varphi$ (where φ again only reasons about finite prefixes). The specification of Example 2.4 is an example of a recurrence property. Finally, boolean combinations of persistence and recurrence properties give rise to the most general class of LTL formulas, namely Δ_2 for *reactivity* properties, for which each formula is equivalent to some conjunction of disjunctions of a Σ_2- and Π_2-formula.

It is known that any LTL formula is equivalent to some Δ_2-formula [21], however until

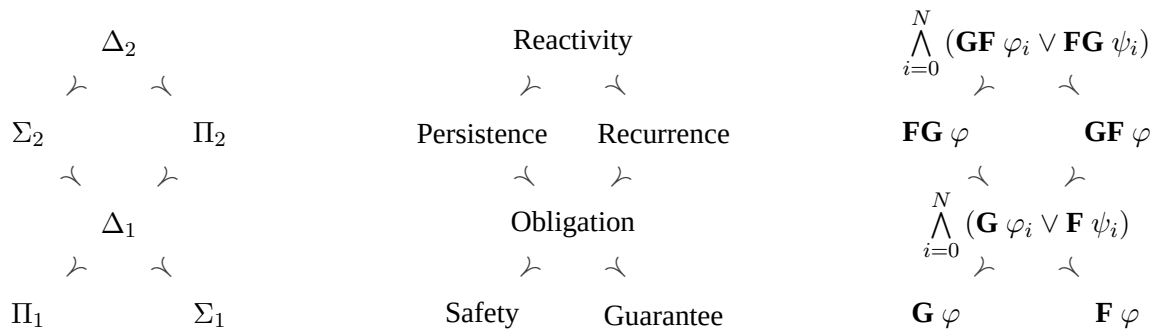


FIGURE 2.1 The LTL hierarchy in three equivalent representations [60]. $A \prec B$ means that the language of A is contained in the language of B.

recently there was no efficient approach to find such a formula. With a recent breakthrough, it is now possible to convert any LTL formula to an equivalent formula in Δ_2 using a normalization [86]. We will see later how this can be used to efficiently construct automata from an LTL formula. For that purpose, we now proceed with defining ω-automata.

2.3 ω-Automata and Parity Games

In this section, we describe the various types of automata that we use in the construction. We cover different kinds of ω-automata, which are automata that run on infinite words. ω-Automata are closely related to LTL in that they essentially form another specification language. Although ω-automata could be used instead of LTL to represent a specification directly, they are much less convenient to write. However, an automata-based representation is useful as an intermediate step because of its executional nature.

We will first formally give a generic definition of ω-automata. We then discuss several types of ω-automata and their relation to LTL. Finally, we extend the definition of one particular type of ω-automata to games.

2.3.1 Definition

An ω-automaton is a finite automaton that runs on infinite words as input. Consequently, the commonly seen acceptance condition of a finite automaton on finite words, namely ending up in a so-called "accepting" state [90], is no longer applicable. Instead, various other acceptance conditions have been defined for ω-automata, and each of them comes with its own expressiveness. Before enumerating these acceptance conditions, let us formally define ω-automata. We adapt the definition from Kupferman [52] but we use the unified notation of the Hanoi Omega-Automata format [2] to describe the acceptance condition.

DEFINITION 2.6 **[2, 52]** *Given a set of atomic propositions AP, an ω-automaton is a 5-tuple $A = (Q, \Sigma, \mathcal{I}, \delta, \alpha)$ where*

- *Q is a set of states,*
- *$\Sigma = 2^{AP}$ is the alphabet,*

- $\mathcal{I} \subseteq Q$ *is a set of initial states,*
- $\delta \subseteq Q \times \Sigma \times Q$ *is a transition relation,*
- α *is the acceptance condition derived from* $f ::= \bot \mid \top \mid \mathit{inf}(s) \mid \mathit{fin}(s) \mid f \wedge f \mid f \vee f$ *where* $s \subseteq Q$*.*

An infinite word $w = w_0, w_1 \ldots$, where each $i \in \mathbb{N}$ and $w_i \in \Sigma$, is accepted by an ω-automaton $(Q, \Sigma, \mathcal{I}, \delta, \alpha)$ if there is at least one infinite run through the automaton satisfying α. A run is an infinite sequence of states $s_0, s_1, \ldots$ such that $s_0 \in \mathcal{I}$ and for each w_i where $i > 0$, there is a $s_i \in Q$ such that $(s_{i-1}, w_{i-1}, s_i) \in \delta$. The semantics of α are then such that $\mathit{inf}(s)$ is true if and only if there is a run such that s is visited infinitely often. Similarly, $\mathit{fin}(s)$ is true if and only there is a run such that s is visited finitely often. The semantics of the remaining symbols of $\mathcal{F}$ are defined as normal propositional logic to create boolean combinations of these conditions.

For simplicity, we often give just a partial description of the transition relation of an ω-automaton. We can however assume, w.l.o.g., that every ω-automaton is complete, i.e. for every $s_1 \in Q$ and $w \in \Sigma$ we have that there is some $s_2 \in Q$ such that $(s_1, w, s_2) \in \delta$, since we can always introduce a (non-accepting) "sink state" $\lightning$ and change the transition relation to δ' such that δ' is defined as:

$$\delta' = \delta \cup \{(s, w, \lightning) \mid \neg \exists s'.(s, w, s') \in \delta\}$$

EXAMPLE 2.7 Consider the automaton $(\{A, B, C, D\}, 2^{\{a,b\}}, \{A, B\}, \delta, \mathit{inf}(\{C\}))$ where

$$\delta = \{(A, \{a, b\}, C), (B, \{a, b\}, C), (B, \emptyset, B), (C, \{a\}, D), (D, \{a\}, C)\}$$

The example is depicted in Figure 2.2. Now consider a word that starts with the symbol ab and then continues with an infinite sequence of $a\bar{b}$ (e.g. $ab, a\bar{b}, a\bar{b}, a\bar{b}, \ldots$). This word is accepted by this automaton because we infinitely often visit C. Next, consider the word that is an infinite sequence of ab symbols. This word is not accepted, even though we reach C, because we get stuck in the state C and therefore end up in the implicit sink state $\lightning$. Finally, consider the word that consists of an infinite repetition of $\bar{a}\bar{b}$. The automaton then non-deterministically decides to start in B and will not get stuck, but the acceptance condition $\mathit{inf}(\{B\})$ does not hold so this word is also not accepted. □

2.3.2 Types of ω-Automata

We define the acceptance condition of an ω-automaton using notation from the Hanoi Omega-Automata format [2]. ω-Automata with an arbitrarily complex acceptance condition expressible in this format are known as Emerson-Lei automata, named after Emerson and Lei who originally suggested this canonical form [32].

More often, we do not need or want this much freedom in the acceptance condition as this freedom comes at the cost of more complex and computationally expensive translations. We,

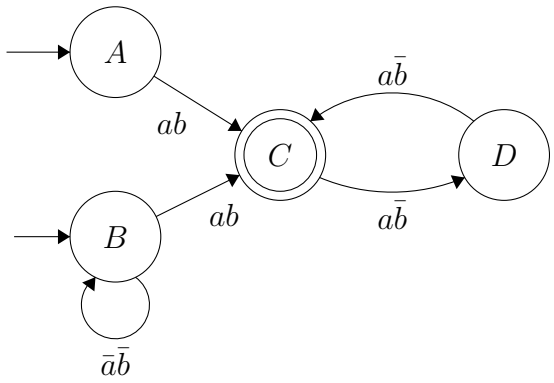


FIGURE 2.2 An example of an ω-automaton with acceptance condition $inf(\{C\})$

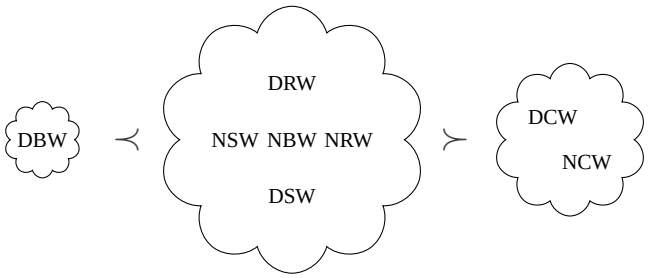


FIGURE 2.3 Automata classes and their expressiveness [85]. Automata classes of equal expressiveness are in the same cloud. $A \prec B$ indicates that the language of the classes in A is contained in the language of the classes in B.

therefore, identify various acceptance conditions that restrict this freedom such that the automata become easier to work with but also become possibly less expressive or succinct.

We start with the most restricted acceptance conditions that we will use in this thesis, namely those of the form $inf(s)$ and $fin(s)$ where s is a set of states. ω-Automata with acceptance conditions of the form $inf(s)$ are automata are often referred to as Büchi automata [17]. Observe that the example provided in Section 2.3.1 is also a Büchi automaton. Automata with $fin(s)$ acceptance conditions are also known as co-Büchi automata, reflecting that these automata are dual to Büchi automata. In other words, suppose we have some Büchi automaton $(Q, \Sigma, \mathcal{I}, \delta, inf(s))$ then the co-Büchi automaton $(Q, \Sigma, \mathcal{I}, \delta, fin(s))$ recognizes exactly the complement of the language of the Büchi automaton [85].

Next, we consider boolean combinations of pairs of these acceptance conditions as follows. Given sets of states s_i and r_i for $0 \leq i < N$, ω-automata where the acceptance condition is a disjunction of N pairs of the form $fin(s_i) \wedge inf(r_i)$ are referred to as Rabin automata [81]. Conversely, ω-automata with a conjunction of N pairs of $fin(s_i) \vee inf(r_i)$ as the acceptance condition are referred to as Streett automata [89]. These automata are also each other's dual [85].

A special case of the Rabin automaton is when its condition can be represented as an (ordered) list α of n sets of states such that the sets induce a partitioning over Q and such that a run is accepted if the least index i where $0 \leq i < n$ for which $\alpha[i]$ is satisfiable infinitely often is even. An automaton with this acceptance condition is called a parity automaton [74]. For parity automata, we use this list representation for the acceptance condition as it is much more convenient than the representation using fin and inf sets.

We have to note that there are many more automata classes. In particular, every class we

Name	Abbrev.	Acceptance condition	A run is accepted iff...
Büchi	B	$inf(s_0)$	s_0 is visited infinitely often
co-Büchi	C	$fin(s_0)$	s_0 is visited finitely often
Rabin	R	$\bigvee_{i=0}^{n-1} inf(s_i) \wedge fin(r_i)$	s_i is visited infinitely often and r_i is visited finitely often for some i such that $0 \leq i < n$
Streett	S	$\bigwedge_{i=0}^{n-1} fin(s_i) \vee inf(r_i)$	s_i is visited finitely often or r_i is visited infinitely often for all i such that $0 \leq i < n$
Parity	P	$\bigvee_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} \left(inf(s_{2j}) \wedge fin\left(\neg \bigvee_{i=2j}^{n-1} s_i \right) \right)$	The least index i such that s_i is visited infinitely often is even (with all s_i inducing a partition on Q)

TABLE 2.1 Acceptance conditions of ω-automata [85]

have seen so far has a corresponding *occurrence* version where instead of considering whether we visit a set of state infinitely or finitely often, we consider whether we visit the set at least once or not at all. This allows the expression of safety and co-safety properties, but since we do not use these automata in the construction, we do not formally define them and refer to Schneider [85] and Cerná et al. [20] for more information.

We define the following ω-automaton classes formally, where we use the naming convention NxW where N stands for non-deterministic, x is an abbreviation for the acceptance conditions and W stands for words. DxW is used to refer to the deterministic counterpart of the same automaton class, i.e. those automata where $|\mathcal{I}| = 1$ and for each state s_1 and word w, we have that there is exactly one s_2 such that $(s_1, w, s_2) \in \delta$.

DEFINITION 2.8 [85] *An ω-automaton $A = (Q, \Sigma, \mathcal{I}, \delta, \alpha)$ is referred to as NxW where x is the abbreviation as in Table 2.1 if α is of the given form, in which s_i and r_i for $0 \leq i < n$ are sets of states. Furthermore, we use DxW if additionally $|\mathcal{I}| = 1$ and for each state s_1 and word w, we have that there is exactly one s_2 such that $(s_1, w, s_2) \in \delta$.*

Unlike with finite automata on finite words, the deterministic version of an ω-automaton class is not necessarily equally expressive as the non-deterministic version of the same class. For example, a DBW is strictly less expressive than an NBW. Additionally, it is known that DRW, DSW, NRW, NSW and NBW automata are all equally expressive [85]. NCW and DCW automata are also equally expressive. We can visualize the expressiveness of the automata as shown in Figure 2.3 [85].

It is important to realize that the equality of the expressiveness of two automata classes does not imply that they can be efficiently translated to each other. For example, NCW and DCW automata are equally expressive, but we will see later that determinizing an NCW causes an exponential blowup in the number of states.

2.3.3 Parity Games

ω-Automata can be seen as 1-player games, in which a player decides the transitions to be taken in a run. The game is won if the run is accepting. In general, ω-automata can be extended to 2-player games if the state set is partitioned such that each state is controlled by one of the two players. One player wins if the run is accepting, the other wins if it is rejecting. In addition, we also assume that the alphabet Σ is partitioned such that each symbol is owned by one of the two players. Although we can generally extend any ω-automaton to a 2-player game [8], we only define it for the deterministic parity automaton as this is the only automaton that we will convert to a game later on:

DEFINITION 2.9 *A* parity game *is a 7-tuple* $G = (Q_c, Q_e, \Sigma_c, \Sigma_e, \mathcal{I}, \delta, \alpha)$ *where*

- Q_c *and* Q_e *are sets of states belonging to player* c *and* e *respectively,*
- Σ_c *and* Σ_e *are sets of symbols belonging to player* c *and* e *respectively,*
- $\mathcal{I} \in Q_c \cup Q_e$ *is the initial state,*
- $\delta \subseteq (Q_c \times \Sigma_c) \cup (Q_e \times \Sigma_e) \to Q_c \cup Q_e$ *is the transition function.*
- α *is an ordered partitioning of the states* $Q_c \cup Q_e$ *such that* α_i *represents the states with* priority i *(starting with* $i = 0$*).*

Furthermore, the arena *of the game is the underlying DPW* $(Q_c \cup Q_e, \Sigma_c \cup \Sigma_e, \mathcal{I}, \delta, \alpha)$

A parity game is played turn-based starting in $\mathcal{I}$, where the player that owns the current state determines the next move. Parity games can have finite and infinite plays, but just like with the ω-automata, we assume that the game is complete and therefore only consider infinite plays. Player c wins if the play is accepting on the arena, otherwise player e wins. Thus, the winner of an infinite play is player c (resp. e) if the minimal priority i such that α_i is visited infinitely often is even (resp. odd).

Solving a parity game involves determining, for each node, which player can be guaranteed to win after reaching this node. This is possible since it has been shown that parity games are *determined* [62], i.e. every node is either won by player c or by player e. The solution to a game thus partitions the nodes in two so-called *winning regions*. A solution to the parity game $(Q_c, Q_e, \Sigma_c, \Sigma_e, \mathcal{I}, \delta, \alpha)$ is, therefore, a subset W of $Q_c \cup Q_e$ where W is the winning region of player c and $(Q_c \cup Q_e) \setminus W$ of player e.

Apart from finding these winning regions, we also want to know what the *strategy* of the player should be to win these regions. It is known that these strategies are *memoryless* [30], i.e. the players only need to consider the current node and not any nodes seen before. A strategy is therefore a subset of the transitions of the game, containing just the transitions that are actually used. For each node, only a single outgoing transition is present in the set.

A parity game can be solved in many different ways [5,45,94,99]. Without detailing exactly how to obtain the solution for a parity game, we now provide an example and the corresponding solution.

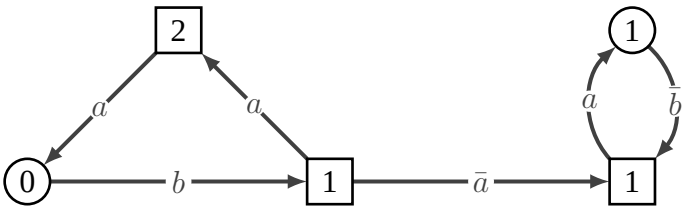


FIGURE 2.4 A parity game where the rectangular nodes are controlled by c and the circular nodes by e. The vertices are labelled with their priorities. The game starts in the node with priority 0.

EXAMPLE 2.10 An example of a parity game is presented in Figure 2.4. In the figure, we use rectangles for nodes owned by c and circles for those owned by e. In this example $\Sigma_c = \{a, \bar{a}\}$ and $\Sigma_e = \{b, \bar{b}\}$ and transitions are labelled with elements from Σ_c and Σ_e. Transitions outgoing from a node owned by c (resp. e) use symbols from Σ_c (resp. Σ_e). Vertices are labelled with their priorities and the vertex with priority 0 is the initial state.

This particular example is very simple in that there are only two possible strategies. Namely the strategy where c moves from the middle node to the right one using symbol $\bar{a}$ and that where c moves to top-left with an a. The latter is the winning strategy for c as that ensures that we never leave the triangle of three vertices. The lowest priority in that triangle is 0, which is even, and thus c wins the game. If c decides to move to the right instead, every play will enter the cycle on the right in which the minimal priority is odd and thus c loses. □

CHAPTER 3

Background

In this chapter, we discuss how the concepts introduced in the previous chapter are applied in reactive synthesis. We first describe exactly what reactive synthesis is. Then, we discuss different methods that have been applied in the past to do reactive synthesis and how we extend upon that.

3.1 Reactive Synthesis

Reactive synthesis is the process of transforming a formal specification to a reactive system that adheres to the specification. We often refer to such a system as a reactive controller (or controller for short). Before discussing how we synthesize a controller, we first need a more formal description of what a controller actually is.

There are two ways in which controllers are commonly modelled. One is through Mealy machines [63] and the other is through Moore machines [69]. Both are finite state machines but the difference is that the output of a Mealy machine depends on the current state and the input, whereas for a Moore machine it only depends on the current state. Both can be implemented in hardware circuits and thus serve a real-world purpose. They can also easily be converted to each other [77]. We only consider Mealy machines as it turns out that the semantics of a Mealy machine is better suited for our purposes than that of a Moore machine. We formally define Mealy machines, adapting notation from Ehlers [29], as follows:

DEFINITION 3.1 *A Mealy machine $\mathcal{M}$ is a 5-tuple $(S, \Sigma^I, \Sigma^O, s_0, \delta)$ where:*

- S *is a set of states*
- Σ^I *is a set of input symbols*
- Σ^O *is a set of output symbols*
- $s_0 \in S$ *is the initial state*
- $\delta : S \times \Sigma^I \to S \times \Sigma^O$ *is the transition function.*

Given an input word $i = i_0 i_1 i_2 \dots$ where all $i_k \in \Sigma^I$, $k \in \mathbb{N}$, a run of a Mealy machine is a sequence of states $\pi = \pi_0 \pi_1 \pi_2 \dots$ such that $\pi_0 = s_0$ and for all $k \in \mathbb{N}$, $\delta(\pi_k, i_k) = (\pi_{k+1}, o_k)$ for some $o_k \in \Sigma^O$. Thus, the result of a run is an output word $o = o_0 o_1 o_2 \dots$.

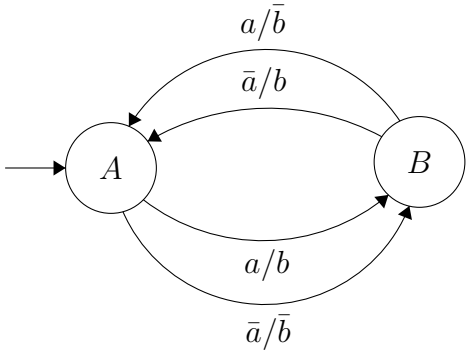


FIGURE 3.1 A Mealy machine

EXAMPLE 3.2 An example of a Mealy machine where $\Sigma^I = \{a, \bar{a}\}$ and $\Sigma^O = \{b, \bar{b}\}$ is given in Figure 3.1. The notation a/b on an edge from state A to B indicates that $\delta(A, a) = (B, b)$. This simple example essentially models a system that flips every second bit. To see why, consider the input word $i = aaaa\ldots$. This results in the output word $b = b\bar{b}b\bar{b}\ldots$. In terms of bits, we can consider an input a as 1 and $\bar{a}$ as 0. Similarly, output b ($\bar{b}$ resp.) indicates the output is 1 (0 resp.). Then it is clear that an input $i = 1111\ldots$ results in an output $o = 1010\ldots$. □

Now suppose we have some set of input and output variables Σ^I and Σ^O and we have some specification $\mathcal{S}$ over these variables that describe the desired properties of a controller. The process of reactive synthesis can then be understood as some function f that takes Σ^I, Σ^O and $\mathcal{S}$ and produces a Mealy machine $\mathcal{M}$ over Σ^I and Σ^O such that $\mathcal{M} \models \mathcal{S}$. The problem of finding such a function was originally introduced by Church [23, 24] and later solved by Büchi and Landweber [16].

In this thesis, we assume the specification $\mathcal{S}$ to be given as an LTL formula. Thus, we summarize the process of reactive synthesis as the construction of a Mealy machine out of an LTL formula such that the machine satisfies the formula. In the next section, we discuss several approaches and how ours differs from previous work.

3.2 Related Work

LTL reactive synthesis has been extensively researched in the past and many techniques exist to convert an LTL formula into a controller that satisfies the formula. A classic approach consists of converting an LTL specification to an NBW which can be determinized using Safra's determinization construction [84] to obtain a DPW. A DPW can then be converted to a parity game and solved. This parity game is a game in which the controller player c plays against the environment player e. If e wins the game, then it is not possible to construct a controller that satisfies the LTL specification, i.e. the specification is *unrealizable*. If c wins, then the specification is *realizable* and the winning strategy can be transformed to a Mealy machine that satisfies the specification.

Unfortunately, Safra's construction does not lend itself for an efficient implementation [1, 92]. Therefore "Safraless" approaches [53, 54] were invented that avoid the need for determinization. Unlike Safra's construction, these approaches can also be efficiently implemented

symbolically.

A symbolic implementation uses an abstract representation of the data. For example, instead of representing an ω-automaton explicitly as a tuple of sets like in Definition 2.6, symbolic implementations often use *binary decision diagrams* [56] to encode the sets, which is often much more compact and efficient. We cover this topic in much more detail when we apply it in Chapter 5.

A Safraless approach was successfully implemented in Lily [42], which was the first synthesis tool that supported full LTL. Unfortunately, even though the approach is implementable symbolically, Lily used an explicit implementation and could only be applied successfully on small examples.

Instead of aiming to avoid determinization, new implementations were created that use the observation that reactive systems are usually not specified as one large specification, but rather consist of a conjunction of several "subspecifications" [4]. This allows each subspecification to be independently converted to a deterministic automaton. This approach was first implemented by Sohail et al. [87], in which they use an improved version of Safra's construction by Piterman [78]. The automata are constructed explicitly but combined into a single symbolic generalized parity game, which is a parity game that consists of a conjunction of multiple parity conditions. They then symbolically solve the game using a generalized version [22] of the well-known recursive algorithm by Zielonka [99] for solving parity games.

The approach by Sohail et al. was later extended upon by Morgenstern et al. [72] by also constructing the automata for the subspecifications symbolically. They exploit the LTL hierarchy to obtain a fully symbolic construction: first, they identify the LTL class of each subspecification, and then they apply specialized constructions such as the breakpoint construction [68] for Π_2- and Σ_2-formulas, which they implemented symbolically [73], to obtain deterministic automata. For those specifications that are not suitable for the specialized constructions, they apply a general "fallback" symbolic determinization procedure that exploits special properties that are always present in ω-automata generated from LTL specifications [71]. Unfortunately, although their implementation outperformed Lily by their use of these specialized constructions, the fallback algorithm did not scale very well [70].

Morgenstern et al. use an approach that is in many ways very similar to the approach by Sohail et al. At first, it may seem that their approach is superior in that they use a fully symbolic construction. However, it is not clear which approach works better as the automata constructed from the subspecifications are expected to be relatively small. A symbolic approach may even be inferior to an explicit approach as symbolic algorithms tend to perform less well than explicit algorithms on small examples. For example, Lily outperformed the construction by Morgenstern et al. for small specifications [70]. Unfortunately, their work was never compared to the work by Sohail et al. because the implementation of the latter was never made publicly available.

A completely different approach to reactive synthesis was introduced by Schewe and Finkbeiner which they call *bounded synthesis* [36]. Their approach is based on a Safraless method [54] and sets a bound on the number of states. This simplifies the problem of reactive

synthesis so that it essentially becomes a search problem instead. Furthermore, the bound can be used to find small controllers by iteratively increasing the bound until a solution is found. More restrictions can be applied to find controllers of good quality, such as bounding the number of cycles as well [35].

This approach is very promising and has had various implementations so far [12, 28, 33, 35, 47]. It is particularly suitable for synthesizing distributed reactive systems [79] which is an even harder problem than normal synthesis and is generally undecidable [79]. However, the synthesis of distributed reactive systems is out of the scope of this thesis.

The final approach that we consider is that by Bloem et al. [10]. Their approach to improving the scalability of reactive synthesis is to limit the specifications such that only properties in a specific format can be expressed and is also known as *GR(1) synthesis*. By limiting the expressibility of LTL, they can synthesize reactive controllers in polynomial time. Although not all LTL formulas can be synthesized in this way, this approach has been successfully applied on an industrial scale using the tool Anzu [38, 43].

Although both bounded synthesis and GR(1) synthesis are very promising approaches, we focus in this thesis on the approach as also used by Sohail et al. and Morgenstern et al. The reason being that recent developments in this field raise new opportunities and that tools based on this technique have been performing surprisingly well in recent years. In particular, the tool Strix [59] has shown in the 2018, 2019 and 2020 SYNTCOMP reactive synthesis competitions [39, 41] that even an explicit parity-game-based approach can work very well when exploiting the temporal logics hierarchy.

The recently discovered normalization technique for LTL formulas [86] now makes it possible to convert any LTL formula into a Δ_2-formula, that is a boolean combination of Σ_2- and Π_2-formulas. This means it is no longer needed to assume that LTL specifications are provided as a conjunction of subspecifications, as Sohail et al. do in their work. Furthermore, the fallback strategy of Morgenstern et al. is also no longer needed, as we can now always apply, e.g., the breakpoint construction [68] in favour of Safra's construction [84] or Piterman's procedure [78].

Furthermore, instead of converting the automata to a generalized parity game, we can use the results of Boker et al. [13] to construct a normal parity game out of the product of a DRW and a DSW. This approach may be beneficial as this game is easier to solve [22] and it allows the use of a vast collection of recently discovered algorithms [5, 45, 94] in favour of the classic solution by Zielonka [99] that was generalized by Chatterjee et al. [22]. Furthermore, the approach presented by Boker et al. lends itself for a symbolic implementation, which makes it favourable over alternative techniques to construct a parity game, such as using an index appearance record [50], which has no obvious symbolic implementation.

We have implemented the approach in the new reactive synthesis tool Otus. In the next section, we give a detailed outline of the construction. Subsequent chapters will describe each part of the construction in more depth.

3.3 Outline of the Construction

Given some LTL specification φ, we summarize the construction as implemented in Otus in the following steps:

1) We start by normalizing φ using the results of Sickert et al. [86] to obtain an LTL formula in Δ_2, which itself consists of Σ_2- and Π_2-fragments. This Δ_2-formula is at most exponential in the size of φ.

2) Each Σ_2- and Π_2-fragment can then be translated to a so-called alternating weak automaton (AWW), which is similar to an ω-automaton but more convenient for this purpose. The sizes of the resulting automata are linear in the sizes of the formulas. They can then be determinized to DCWs and DBWs using a slightly adapted version of the breakpoint construction [68]. These automata are at most exponential in the sizes of the AWWs.

3) We now have a collection of explicit DCWs and DBWs. We convert them to a symbolic representation and combine them to obtain a symbolic product automaton where the acceptance condition is constructed such that it respects the Δ_2-formula of step 1. We will see that this results in a DRW.

4) We repeat all previous steps for the negated formula $\neg\varphi$ to obtain a DRW that represents the complement of the language of φ. Because DRW and DSW automata are each others dual, we can interpret the DRW as a DSW and we thus we obtain both a DRW and a DSW for the language of φ.

5) We now combine these two automate into a single symbolic DPW using the construction of Boker et al. [13].

6) Finally we convert the DPW to a symbolic parity game and solve the game using distraction fix-point iteration [58, 94] to obtain a winning strategy. This winning strategy can then be used to construct a Mealy machine respecting the original LTL formula.

The reason for constructing both a DRW and a DSW for the language is that Boker et al. [13] have shown that an ω-automaton which has both a DRW and a DSW condition that define the same language can be converted to a DPW in polynomial time. Taking the product of the DRW and the DSW while retaining only the acceptance condition of the DRW results in an automaton recognizing φ that always guarantees this property. However, it is not needed to know the DSW condition and thus occasionally we can skip the construction of the DSW altogether if it so happens that the DRW automaton already has an equivalent DSW condition on that structure, albeit unknown.

Furthermore, we have decided to construct the AWW automata and the DCW automata explicitly. The main reason is technical in that these constructions have already been implemented explicitly before [49] so we reuse them. However, as discussed in the previous section, it is not clear yet whether an explicit or symbolic approach is better as the work of Sohail et al. and Morgenstern et al. lack a comparison.

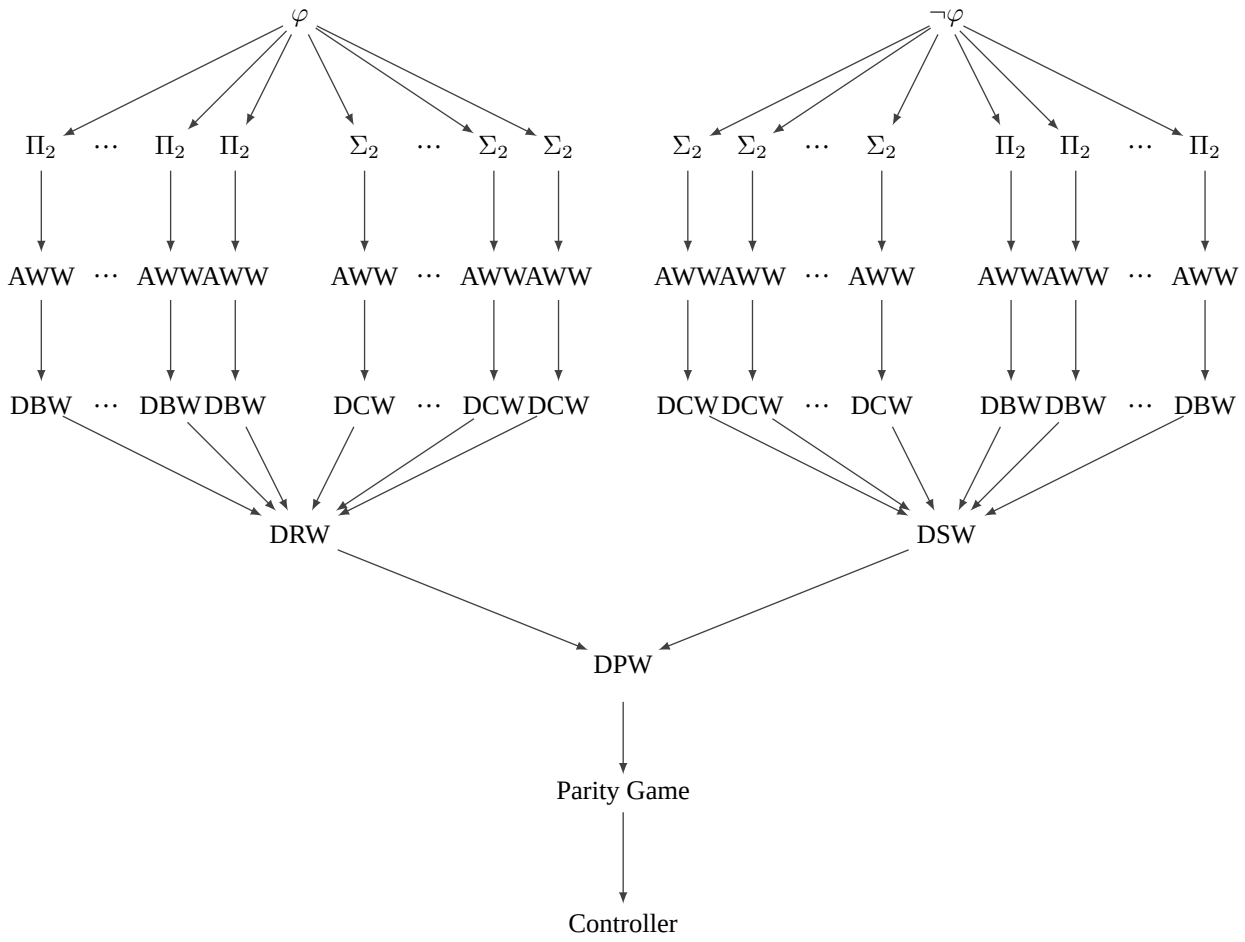


FIGURE 3.2 Visualization of the construction steps

We summarize the construction graphically in Figure 3.2 and note that the right part of the figure is sometimes skipped in line with the previous discussion. The next three chapters will describe the construction in more detail. Steps 1 and 2 will be discussed in Chapter 4. We then cover steps 3, 4 and 5 in Chapter 5. Finally, step 6 is discussed in Chapter 6.

CHAPTER 4

Constructing Explicit Automata

In this chapter, we show how we convert an LTL specification to a collection of DCW and DBW automata. The approach we present here is adapted from Sickert and Esparza [86]. We start with the normalization of an LTL specification into a Δ_2-formula from which we can extract Σ_2- and Π_2-fragments. We then introduce alternating weak automata and the conversion of Σ_2- and Π_2-fragment to them. Finally, we show how we determinize these automata to obtain DCW and DBW automata.

4.1 Normalization

In this section, we discuss the normalization procedure as proposed by Sickert and Esparza [86]. This normalization allows us to divide the formula into Σ_2- and Π_2-fragments that are easy to translate to automata. This section presents the results but for the proofs we refer to Sickert and Esparza [86].

We start with an LTL formula φ. The normalization procedure is then based on the observation that having information about the limit-behaviour of words allows for simplifications of the formula. Suppose $\varphi = a \,\mathbf{U}\, b$. If we know, for example, that for all words, $a \,\mathbf{U}\, b$ holds infinitely often, then we can simplify φ to a weaker version $\varphi' = a \,\mathbf{W}\, b$ since we know that b always eventually becomes true. We can use this concept systematically and enumerate all possible assumptions to obtain a normal form structured as follows:

$$\varphi \equiv \bigvee_{\textit{assumptions}} (\textit{simplified formula under assumption} \wedge \textit{the assumption holds})$$

This is how the normal form as proposed by Sickert and Esparza is structured. It makes various assumptions on the limit-behaviour of words, and simplifies the specification based on each assumption. They do so by partitioning the infinite set of all words according to their limit-behaviour. We briefly present their results and provide an example afterwards.

Let us first introduce some notation. We use $sf(\varphi)$ to denote the set of all subformulas of φ (including φ itself). We then define $\mu(\varphi)$ and $\nu(\varphi)$ in terms of $sf(\varphi)$ as follows:

DEFINITION 4.1 [86] *Let $\mu(\varphi)$ be defined as the set of all subformulas in $sf(\varphi)$ of the form $\varphi_1 \,\boldsymbol{U}\, \varphi_2$, $\varphi_1 \,\boldsymbol{M}\, \varphi_2$ or $\boldsymbol{F}\, \varphi_1$. Analogously, let $\nu(\varphi)$ be defined as the set of all subformulas in $sf(\varphi)$ of the form $\varphi_1 \,\boldsymbol{R}\, \varphi_2$, $\varphi_1 \,\boldsymbol{W}\, \varphi_2$ or $\boldsymbol{G}\, \varphi_1$.*

Now let $\mathcal{U}$ denote the set of all possible words, $\mathcal{GF}_w^\varphi$ denote all $\psi \in \mu(\varphi)$ that are infinitely often true for the word w and $\mathcal{FG}_w^\varphi$ denote all $\psi \in \nu(\varphi)$ that are eventually always true for the word w. We can now define the partitioning as follows:

DEFINITION 4.2 [86] *Let φ be an LTL formula, $M \subseteq \mu(\varphi)$ and $N \subseteq \nu(\varphi)$, then*

$$\mathcal{P}_{M,N}^\varphi = \{w \in \mathcal{U} \mid M = \mathcal{GF}_w^\varphi \wedge N = \mathcal{FG}_w^\varphi\}$$

This partitions the words such that all words with the same limit-behaviour are in the same partition. For each of these partitions, we can now derive a simplification for φ under the assumption that the words are in this partition. We first repeat the substitutions and then present the main theorem in Theorem 4.6.

DEFINITION 4.3 [86] *Let φ be a formula and $M \subseteq \mu(\varphi)$ be a set of subformulas of φ. Then we define $\varphi[M]_1^\Pi$ inductively by case distinction as follows:*

$$(\varphi_1 \,\boldsymbol{U}\, \varphi_2)[M]_1^\Pi = \begin{cases} (\varphi_1[M]_1^\Pi) \,\boldsymbol{W}\, (\varphi_2[M]_1^\Pi) & \text{if } \varphi_1 \,\boldsymbol{U}\, \varphi_2 \in M \\ \bot & \text{otherwise} \end{cases}$$

$$(\varphi_1 \,\boldsymbol{M}\, \varphi_2)[M]_1^\Pi = \begin{cases} (\varphi_1[M]_1^\Pi) \,\boldsymbol{R}\, (\varphi_2[M]_1^\Pi) & \text{if } \varphi_1 \,\boldsymbol{M}\, \varphi_2 \in M \\ \bot & \text{otherwise} \end{cases}$$

$$(\boldsymbol{F}\, \varphi_1)[M]_1^\Pi = \begin{cases} \top & \text{if } \boldsymbol{F}\, \varphi_1 \in M \\ \bot & \text{otherwise} \end{cases}$$

The remaining cases are defined homomorphically, e.g. $(\boldsymbol{G}\, \varphi_1)[M]_1^\Pi = \boldsymbol{G}\, (\varphi_1[M]_1^\Pi)$.

DEFINITION 4.4 [86] *Let φ be a formula and $N \subseteq \nu(\varphi)$ be a set of subformulas of φ. Then we define $\varphi[N]_1^\Sigma$ inductively by case distinction as follows:*

$$(\varphi_1 \,\boldsymbol{R}\, \varphi_2)[N]_1^\Sigma = \begin{cases} \top & \text{if } \varphi_1 \,\boldsymbol{R}\, \varphi_2 \in N \\ (\varphi_1[N]_1^\Sigma) \,\boldsymbol{M}\, (\varphi_2[N]_1^\Sigma) & \text{otherwise} \end{cases}$$

$$(\varphi_1 \,\boldsymbol{W}\, \varphi_2)[N]_1^\Sigma = \begin{cases} \top & \text{if } \varphi_1 \,\boldsymbol{W}\, \varphi_2 \in N \\ (\varphi_1[N]_1^\Sigma) \,\boldsymbol{U}\, (\varphi_2[N]_1^\Sigma) & \text{otherwise} \end{cases}$$

$$(\boldsymbol{G}\, \varphi_1)[N]_1^\Sigma = \begin{cases} \top & \text{if } \boldsymbol{G}\, \varphi_1 \in N \\ \bot & \text{otherwise} \end{cases}$$

The remaining cases are defined homomorphically.

DEFINITION 4.5 *Let φ be a formula and $M \subseteq \mu(\varphi)$ be a set of subformulas of φ. Then we define $\varphi[M]_2^\Sigma$ inductively by case distinction as follows:*

$$(\varphi_1 \,\boldsymbol{R}\, \varphi_2)[M]_2^\Sigma = (\varphi_1[M]_2^\Sigma \vee \boldsymbol{G}\, (\varphi_2[M]_1^\Pi)) \,\boldsymbol{M}\, (\varphi_2[M]_2^\Sigma)$$

$$(\varphi_1 \,\boldsymbol{W}\, \varphi_2)[M]_2^\Sigma = (\varphi_1[M]_2^\Sigma) \,\boldsymbol{U}\, ((\varphi_2[M]_2^\Sigma) \vee \boldsymbol{G}\, (\varphi_1[M]_1^\Pi))$$

$$(\boldsymbol{G}\ \varphi_1)[M]_2^{\Sigma} = (\boldsymbol{G}\ (\varphi_1[M]_1^{\Pi}))\ \boldsymbol{M}\ (\varphi_1[M]_2^{\Sigma})$$

The remaining cases are defined homomorphically

THEOREM 4.6 [86] *Let φ be an LTL formula, then*

$$\varphi \equiv \bigvee_{\substack{M\subseteq\mu(\varphi)\\ N\subseteq\nu(\varphi)}} \left(\varphi[M]_2^{\Sigma} \wedge \bigwedge_{\psi\in M} \boldsymbol{GF}\ (\psi[N]_1^{\Sigma}) \wedge \bigwedge_{\psi\in N} \boldsymbol{FG}\ (\psi[M]_1^{\Pi})\right)$$

The proof for this theorem is given by Sickert and Esparza [86] and outside the scope of this thesis. The key idea is that we iterate over the partitions, which are essentially assumptions on the limit behaviour of the words. For each assumption, $\varphi[M]_2^{\Sigma}$ is the simplification of φ given that the assumption holds. The remainder of the disjunct then ensures that the assumption indeed holds. We now proceed with an example.

EXAMPLE 4.7 [86] Let $\varphi = \mathbf{F}\ (a \wedge \mathbf{G}\ (b \vee \mathbf{F}\ c)))$. This formula is clearly not in Δ_2. We first illustrate informally how the normalization procedure can convert this formula to an equivalent formula in Δ_2, and then we partially apply the formal procedure to see how it arrives at the same result.

To change φ to a Δ_2-formula, we need to get rid of the innermost finally operator $\mathbf{F}\ c$. We therefore make a case distinction for the limit behaviour of $\mathbf{F}\ c$. First, we assume $\mathbf{F}\ c$ holds infinitely often, i.e. $\mathbf{GF}\ c$ holds. Then, we know that $\mathbf{G}\ (b \vee \mathbf{F}\ c)$ also holds. Thus, under the assumption that $\mathbf{GF}\ c$ holds, we can simplify φ to $\mathbf{F}\ a$. Now assume $\neg\mathbf{GF}\ c$, i.e. $\mathbf{FG}\ \bar{c}$ holds. Then we know that at some point in time $(b \vee \mathbf{F}\ c)$ is equivalent to b. This means that we can simplify $\mathbf{G}\ (b \vee \mathbf{F}\ c)$ to $(b \vee \mathbf{F}\ c)\ \mathbf{U}\ \mathbf{G}\ b$ since there is a point in time after which $\mathbf{G}\ (b \vee \mathbf{F}\ c)$ can only be satisfied by b and not by $\mathbf{F}\ c$.Combining the two assumptions then gives us the formula φ' such that $\varphi \equiv \varphi'$ and $\varphi' \in \Delta_2$:

$$\begin{aligned}\varphi' &= \mathbf{FG}\ \bar{c} \wedge \mathbf{F}\ (a \wedge ((b \vee \mathbf{F}\ c)\ \mathbf{U}\ \mathbf{G}\ b)) \vee \mathbf{GF}\ c \wedge \mathbf{F}\ a\\ &\equiv \mathbf{F}\ (a \wedge ((b \vee \mathbf{F}\ c)\ \mathbf{U}\ \mathbf{G}\ b)) \vee \mathbf{F}\ a\end{aligned}$$

Using Theorem 4.6, we can obtain the same results as follows. We start by identifying the subformula sets $\mu(\varphi)$ and $\nu(\varphi)$. This gives us $\mu(\varphi) = \{\varphi, \mathbf{F}\ c\}$ and $\nu(\varphi) = \{\mathbf{G}\ (b \vee \mathbf{F}\ c)\}$. Then Theorem 4.6 will yield a disjunct for each of the $|2^{\mu(\varphi)}| * |2^{\nu(\varphi)}| = 8$ combinations of M and N. We denote each of these disjuncts by $\varphi'_{M,N}$. We take $M = \{\varphi\}$ and $N = \{\mathbf{G}\ (b \vee \mathbf{F}\ c)\}$ as an example and obtain $\varphi'_{\{\varphi\},\{\mathbf{G}\ (b\vee\mathbf{F}\ c)\}}$:

$$\varphi'_{\{\varphi\},\{\mathbf{G}\ (b\vee\mathbf{F}\ c)\}} = \varphi[\{\varphi\}]_2^{\Sigma} \wedge \mathbf{GF}\ (\varphi[\{\mathbf{G}\ (b \vee \mathbf{F}\ c)\}]_1^{\Sigma}) \wedge \mathbf{FG}\ ((\mathbf{G}\ (b \vee \mathbf{F}\ c))[\{\varphi\}]_1^{\Pi})$$

Let us now compute the result of each of the three substitutions in $\varphi'_{\{\varphi\},\{\mathbf{G}\ (b\vee\mathbf{F}\ c)\}}$ separately:

$$\begin{aligned}\varphi[\{\varphi\}]_2^{\Sigma} &= \mathbf{F}\ (a \wedge ((\mathbf{G}\ ((b \vee \mathbf{F}\ c)[\{\varphi\}]_1^{\Pi}))\ \mathbf{M}\ ((b \vee \mathbf{F}\ c)[\{\varphi\}]_2^{\Sigma}))\\ &= \mathbf{F}\ (a \wedge ((\mathbf{G}\ b)\ \mathbf{M}\ (b \vee \mathbf{F}\ c)))\end{aligned}$$

$$\varphi[\{\mathbf{G}\ (b \vee \mathbf{F}\ c)\}]_1^{\Sigma} = \mathbf{F}\ a$$
$$(\mathbf{G}\ (b \vee \mathbf{F}\ c))[\{\varphi\}_1^{\Pi}) = \mathbf{G}\ b$$

Combining the results, we obtain:

$$\varphi'_{\{\varphi\},\{\mathbf{G}\ (b \vee \mathbf{F}\ c)\}} = \mathbf{F}\ (a \wedge ((\mathbf{G}\ b)\ \mathbf{M}\ (b \vee \mathbf{F}\ c))) \wedge \mathbf{GF}\ (\mathbf{F}\ a) \wedge \mathbf{FG}\ (\mathbf{G}\ b)$$

Obviously, $\mathbf{GF}\ (\mathbf{F}\ a) \equiv \mathbf{GF}\ a$ and $\mathbf{FG}\ (\mathbf{G}\ b) \equiv \mathbf{FG}\ b$. Furthermore $\mathbf{GF}\ a \wedge \mathbf{FG}\ b$ implies $\mathbf{F}\ (a \wedge ((\mathbf{G}\ b)\ \mathbf{M}\ (b \vee \mathbf{F}\ c)))$ so we simplify and obtain:

$$\varphi'_{\{\varphi\},\{\mathbf{G}\ (b \vee \mathbf{F}\ c)\}} = \mathbf{GF}\ a \wedge \mathbf{FG}\ b$$

We repeat this for the remaining 7 disjunctions and after applying several simplification rules [85], we again obtain the formula φ' [86]. □

It is easy to see that the substitution $\varphi[M]_1^{\Pi}$ for some φ and M results in a Π_1-formula. Similarly, $\varphi[N]_1^{\Sigma}$ results in a Σ_1-formula for some φ and N, and $\varphi[N]_2^{\Sigma}$ results in a Σ_2-formula. This means that Theorem 4.6 normalizes any formula φ into a disjunction of pairs of Π_2- and Σ_2-formulas as follows:

$$\bigvee_{\substack{M \subseteq \mu(\varphi) \\ N \subseteq \nu(\varphi)}} \left(\underbrace{\varphi[M]_2^{\Sigma} \wedge \bigwedge_{\psi \in M} \mathbf{GF}\ (\psi[N]_1^{\Sigma})}_{\Sigma_2} \wedge \underbrace{\bigwedge_{\psi \in N} \mathbf{FG}\ (\psi[M]_1^{\Pi})}_{\Pi_2} \right)$$

We use this observation in the next section to convert these Σ_2- and Π_2-formulas into automata.

4.2 Translating LTL Fragments to Alternating Automata

In this section we introduce the construction from a Δ_2-formulas to alternating automata [86]. We first define alternating automata. Then we present the translation to an alternating automaton [75, 86].

4.2.1 Alternating Automata

Unlike the automata classes we have seen in Section 2.3.2, alternating automata do not differ from the other classes through the acceptance condition. Instead, alternating automata are ω-automata with a special transition relation which we define later. It is important to understand that alternating automata are not a class of ω-automata. Instead, for every ω-automata class we have seen so far, an equivalent alternating automata class exists. Here, we assume alternating automata to have a co-Büchi acceptance condition and we therefore only define them with such a condition.

Alternating automata are similar to ω-automata in that they run on infinite words. The

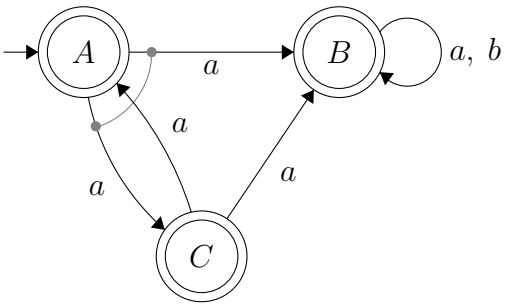


FIGURE 4.1 Alternating automaton $(\{A, B, C\}, \{a, b\}, A, \delta, \{A, B, C\})$

difference lies in what we consider a run. In an ω-automaton, a run is always a sequence of states. If multiple transitions are enabled for a symbol, a non-deterministic choice is made between these transitions. The existence of an accepting run for a word determines whether the word is accepted or not. In alternating automata, when multiple transitions are enabled, this may also be resolved through a non-deterministic choice but there is also the option to instead take all of the transitions simultaneously. This means that a run is no longer just a sequence of states, but rather an infinite tree where each node represents a state of the automaton. The acceptance condition is then defined such that a word is accepted if there exists a run such that every infinite path through the run eventually stays in some set of states forever. Let $\mathcal{B}^+(Q)$ denote the set of positive boolean formulas over Q, i.e. the closure of $Q \cup \{\top, \bot\}$ under disjunction and conjunction, then we formally define an alternating automaton as follows:

DEFINITION 4.8 *An alternating automaton is a 5-tuple $A = (Q, \Sigma, \mathcal{I}, \delta, \alpha)$ where*

- *Q is a set of states.*
- *Σ is a set of input symbols.*
- *$\mathcal{I} \in \mathcal{B}^+(Q)$ is the initial states formula.*
- *$\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the transition formula.*
- *$\alpha \subseteq Q$ is the (co-Büchi) acceptance condition.*

The transition formula distinguishes this automaton from a co-Büchi ω-automaton. For example, suppose $\delta(s, a) = \{\{q, r\}, \{s, t\}\}$ for some states $q, r, s, t \in Q$ and $a \in \Sigma$. This means that we non-deterministically decide whether we take the transitions to q and r, or we take the transitions to s and t. We represent this by encoding the range of δ using positive boolean formulas over the states where for some $s, t \in Q$ we have that $s \vee t$ denotes a non-deterministic choice between s and t, and $s \wedge t$ denotes that both transitions should be taken. We refer to the former as an existential transition and the latter as a universal transition.

EXAMPLE 4.9 Consider the automaton in Figure 4.1. We use gray arcs between transitions to indicate that they are universal. This automaton represents the language starting with aa and then infinitely repeating a or b (e.g. $aabbabba\ldots$). A possible run of this automaton is depicted in Figure 4.2 for the accepted word $aaaa\ldots$. Notice that we take both a transitions in A since

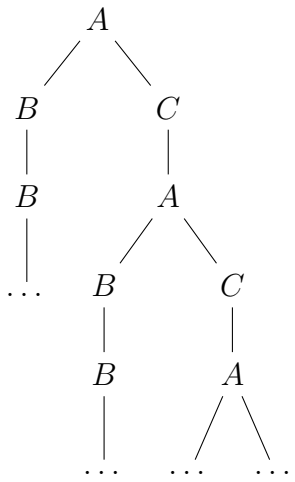


FIGURE 4.2 Example run for the word $aaaa\ldots$ of the alternating automaton depicted in Figure 4.1 (Input symbols have been omitted since we always read a in this example)

$\delta(A, a)$ is universal, whereas we only take a single transition in C since $\delta(C, a)$ is existential. □

Some special attention is needed for the cases that any of the propositional formulas is $\top$ or $\bot$. In case the initial state formula is $\top$, then the automaton accepts all words. Conversely, if the formula is $\bot$, then the automaton accepts no words. Runs where any branches reach a state where the next state formula is $\bot$ are rejecting, and branches of a run where the next state formula is $\top$ are accepting.

4.2.2 LTL to Alternating Automata

The main benefit of alternating automata is that it is almost trivial to obtain an alternating automaton recognizing any LTL formula. Given some LTL formula φ, the basic procedure works as follows. We associate each LTL formula to a state and the state associated with φ is the initial state. From there on, we build up the transition relation using a set of LTL equivalence rules called *expansion laws*.

LEMMA 4.10 [3] *For any LTL formula φ and ψ, the following expansion laws hold:*

$$\boldsymbol{F}\,\varphi \equiv \varphi \vee \boldsymbol{XF}\varphi \qquad \varphi\,\boldsymbol{R}\,\psi \equiv \psi \wedge (\varphi \vee \boldsymbol{X}\,(\varphi\,\boldsymbol{R}\,\psi))$$

$$\boldsymbol{G}\,\varphi \equiv \varphi \wedge \boldsymbol{XG}\varphi \qquad \varphi\,\boldsymbol{W}\,\psi \equiv \psi \vee (\varphi \wedge \boldsymbol{X}\,(\varphi\,\boldsymbol{W}\,\psi))$$

$$\varphi\,\boldsymbol{U}\,\psi \equiv \psi \vee (\varphi \wedge \boldsymbol{X}\,(\varphi\,\boldsymbol{U}\,\psi)) \qquad \varphi\,\boldsymbol{M}\,\psi \equiv \psi \wedge (\varphi \vee \boldsymbol{X}\,(\varphi\,\boldsymbol{M}\,\psi))$$

These laws allow a very intuitive extraction of the transition relation. This idea was first presented by Muller et al. [75], but we present a slight variation of the construction by Sickert and Esparza [86] because it is easier to determinize. However, we define it only for Δ_2-formulas as we only need this construction for Σ_2- and Π_2-formulas.

Let us start with some formula $\varphi \in \Delta_2$. We call the set of subformulas of φ that have a temporal operator at the top level or that are (negated) atomic propositions $s(\varphi)$. Then, we can construct the automaton by associating to each $\psi \in s(\varphi)$ states $\langle\psi\rangle_\Gamma$ where Γ is the least LTL

class in $\{\Pi_1, \Sigma_1, \Delta_1, \Pi_2, \Sigma_2, \Delta_2\}$ containing ψ. In case there is no single least class we create a state for every least class. For example, the states belonging to $\mathbf{X}\ a$ are $\langle \mathbf{X}\ a \rangle_{\Pi_1}$ and $\langle \mathbf{X}\ a \rangle_{\Sigma_1}$. Finally, we assign to each formula $\varphi \in \Delta_2$ a boolean combination of these states denoted $[\varphi]_{\leq \Gamma}$ as follows:

- $[\top]_{\leq \Gamma} = \top$
- $[\bot]_{\leq \Gamma} = \bot$
- $[\varphi_1 \vee \varphi_2]_{\leq \Gamma} = [\varphi_1]_{\leq \Gamma} \vee [\varphi_2]_{\leq \Gamma}$
- $[\varphi_1 \wedge \varphi_2]_{\leq \Gamma} = [\varphi_1]_{\leq \Gamma} \wedge [\varphi_2]_{\leq \Gamma}$
- Else $[\varphi]_{\leq \Gamma} = \bigvee_{\Gamma' \leq \Gamma} \langle \varphi \rangle_{\Gamma'}$

We now construct the automaton using the expansion laws from Lemma 4.10 such that the initial state is $[\varphi]_{\leq \Delta_2}$ and the acceptance condition contains all states of the form $\langle \psi \rangle_{\Pi_i}$ for $i \in \{1, 2\}$.

THEOREM 4.11 [86] *Given an LTL formula $\varphi \in \Delta_2$ over the atomic propositions AP, the alternating automaton recognizing the language of φ is $(Q, \Sigma, \mathcal{I}, \delta, \alpha)$ where:*

- $Q = \{\langle \psi \rangle_\Gamma \mid \psi \in s(\varphi), \Gamma \leq \Delta_2\}$
- $\Sigma = 2^{AP}$
- $\mathcal{I} = [\varphi]_{\leq \Delta_2}$
- $\alpha = \{\langle \psi \rangle_{\Pi_i} \mid i \in \{1, 2\}, \langle \psi \rangle_{\Pi_i} \in Q\}$
- *δ is the restriction to $Q \times \Sigma$ of the function $\delta' : \mathcal{B}^+(Q) \times \Sigma \to \mathcal{B}^+(Q)$ defined as follows:*

$$\delta'(\langle a \rangle_\Gamma, \sigma) = \begin{cases} \top & a \in \sigma \\ \bot & \textit{otherwise} \end{cases}$$

$$\delta'(\langle \bar{a} \rangle_\Gamma, \sigma) = \begin{cases} \top & a \notin \sigma \\ \bot & \textit{otherwise} \end{cases}$$

$$\begin{aligned}
\delta'(\top, \sigma) &= \top \\
\delta'(\bot, \sigma) &= \bot \\
\delta'(p \vee q, \sigma) &= \delta'(p) \vee \delta'(q) \\
\delta'(p \wedge q, \sigma) &= \delta'(p) \wedge \delta'(q) \\
\delta'(\langle \boldsymbol{X}\ \psi \rangle_\Gamma, \sigma) &= [\psi]_{\leq \Gamma} \\
\delta'(\langle \psi_1\ \boldsymbol{U}\ \psi_2 \rangle_\Gamma, \sigma) &= \delta'([\psi_2 \vee (\psi_1 \wedge \boldsymbol{X}\ (\psi_1\ \boldsymbol{U}\ \psi_2))]_{\leq \Gamma}, \sigma) \\
\delta'(\langle \psi_1\ \boldsymbol{W}\ \psi_2 \rangle_\Gamma, \sigma) &= \delta'([\psi_2 \vee (\psi_1 \wedge \boldsymbol{X}\ (\psi_1\ \boldsymbol{W}\ \psi_2))]_{\leq \Gamma}, \sigma) \\
\delta'(\langle \psi_1\ \boldsymbol{R}\ \psi_2 \rangle_\Gamma, \sigma) &= \delta'([\psi_2 \wedge (\psi_1 \vee \boldsymbol{X}\ (\psi_1\ \boldsymbol{R}\ \psi_2))]_{\leq \Gamma}, \sigma) \\
\delta'(\langle \psi_1\ \boldsymbol{M}\ \psi_2 \rangle_\Gamma, \sigma) &= \delta'([\psi_2 \wedge (\psi_1 \vee \boldsymbol{X}\ (\psi_1\ \boldsymbol{M}\ \psi_2))]_{\leq \Gamma}, \sigma) \\
\delta'(\langle \boldsymbol{F}\ \psi \rangle_\Gamma, \sigma) &= \delta'([\psi \vee \boldsymbol{X}\ \boldsymbol{F}\ \psi]_{\leq \Gamma}) \\
\delta'(\langle \boldsymbol{G}\ \psi \rangle_\Gamma, \sigma) &= \delta'([\psi \wedge \boldsymbol{X}\ \boldsymbol{G}\ \psi]_{\leq \Gamma})
\end{aligned}$$

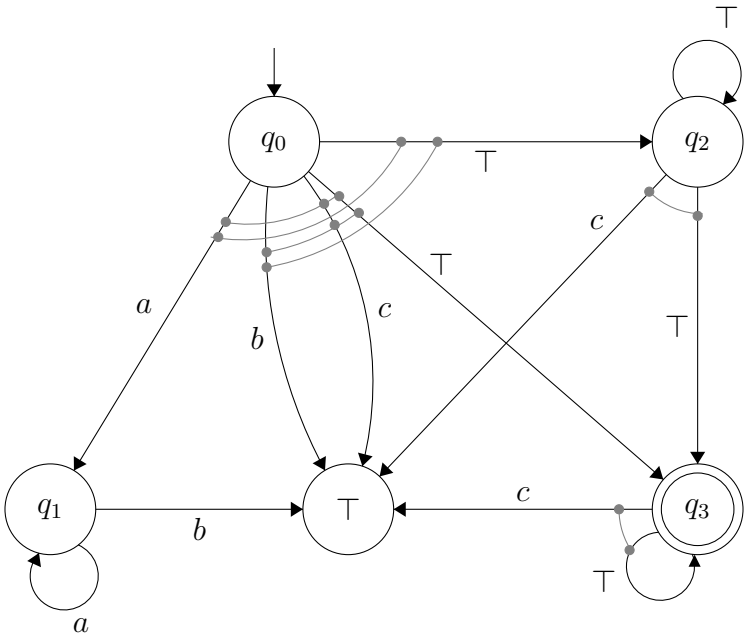


FIGURE 4.3 AWW for $\varphi = a \,\mathbf{U}\, b \wedge \mathbf{FG}\, c$ where $q_0 = \langle \varphi \rangle_{\Sigma_2}$, $q_1 = \langle a \,\mathbf{U}\, b \rangle_{\Sigma_1}$, $q_2 = \langle \mathbf{FG}\, c \rangle_{\Sigma_2}$ and $q_3 = \langle \mathbf{G}\, c \rangle_{\Pi_1}$

This results in an automaton that has at most twice as many states as there are subformulas in φ, meaning that this translation does not suffer from a blowup and is inexpensive. We will consider more properties of these automata in the next section, but we first present an example of the construction.

EXAMPLE 4.12 Consider the LTL formula $a \,\mathbf{U}\, b \wedge \mathbf{FG}\, c$. The AWW is presented in Figure 4.3. For notational purposes we include $\top$ as a state in the diagram. It should be realized however that this is not a real state and that it should be interpreted as discussed before. Furthermore, a transition label a is shorthand for any $\sigma \in \Sigma$ such that $a \in \sigma$. Finally, a transition labelled $\top$ is enabled for any $\sigma \in \Sigma$. □

4.2.3 Characteristics of the Alternating Automata

Alternating automata with a co-Büchi acceptance condition are more expressive than LTL formulas. To simplify the determinization, we identify a characteristic of the alternating automata that are created from the LTL formulas. It turns out that those automata are *alternating weak automata* [75]:

DEFINITION 4.13 *An alternating weak automaton (AWW) is an alternating automaton $(Q, \Sigma, \mathcal{I}, \delta, \alpha)$ where there exists an ordered partitioning $Q_1, \ldots Q_n$ of Q such that for each edge (q_1, σ, q_2) in any run of A, $q_1 \in Q_i$, $q_2 \in Q_j$ and $i \leq j$. Additionally, for each partition Q_i, either $Q_i \subseteq \alpha$ or $S_i \cap \alpha = \emptyset$.*

Intuitively, this means that we can partition the set of states such that we are guaranteed to always move in one direction through the partitions. In other words, any cycle in the AWW is guaranteed to be fully contained in one of the partitions. Furthermore, either all states or no states in a partition are accepting. For example, Figure 4.1 depicts an AWW where a valid

partitioning is the list $[\{A, C\}, \{B\}]$.

Next, we define the height of an AWW as the maximal number of times any run alternates between accepting and non-accepting states, plus one. For example, the height of Figure 4.1 is 1, since there is no alternation at all. The height of the AWWs obtained from the LTL construction is at most 2 [86]. In the next section, we show how we use this property to arrive at a simple determinization construction.

4.3 AWW Determinization

Since the height of the automaton obtained from the previous construction is at most 2, we know that for any accepted word there is a run through the automaton such that at some point in the run we stay in the set of accepting states forever [86]. Recall that this property is also present in NCWs. We can therefore adapt the determinization of NCWs to be used for these AWWs [86]. Let us first present how we can determinize an NCW. Then we adapt it slightly to work on AWWs with an alternation depth of 2.

4.3.1 NCW Breakpoint Construction

An explicitly represented NCW can be determinized using the breakpoint construction [68] to a DCW that is at most exponentially larger than the NCW. The breakpoint construction is an extension of the well known Rabin-Scott subset construction [82] for determinizing non-deterministic finite automata on finite words that is covered in any undergraduate's introduction to automata theory [90]. In the subset construction, each state in the deterministic automaton corresponds with the set of states that can be reached in the non-deterministic automaton when a certain symbol is read. Essentially, the construction collects all possible runs of the non-deterministic automaton and collects them in a single state for each input symbol.

Unfortunately, this approach is not sufficient for the determinization of an NCW. We take the example from Schneider [85] to illustrate why. Suppose we have a non-deterministic finite automaton as illustrated in Figure 4.4, then the deterministic automaton as constructed using the subset construction is shown in Figure 4.5.

The problem now arises when trying to find the acceptance condition for the DCW. The only information we have is whether the states of the DCW contain states from the acceptance condition of the NCW. Essentially, we have two options. We could define the acceptance condition of the DCW such that a state in the DCW is accepting if *all* accepting NCW states are in the subset of the state in the DCW. In that case, however, the language of the DCW in Figure 4.5 would be empty while the language of the NCW is clearly nonempty. The other option is that a state in the DCW is accepting if *any* accepting NCW states are in the subset of the state in the DCW. Also, this case would not work as that would mean that both states of the DCW in Figure 4.5 become accepting, which means $aaa\ldots$ becomes a word that is accepted by the DCW, while this word is not accepted by the NCW.

So, the subset construction alone is not sufficient. Observe that the problem arises because

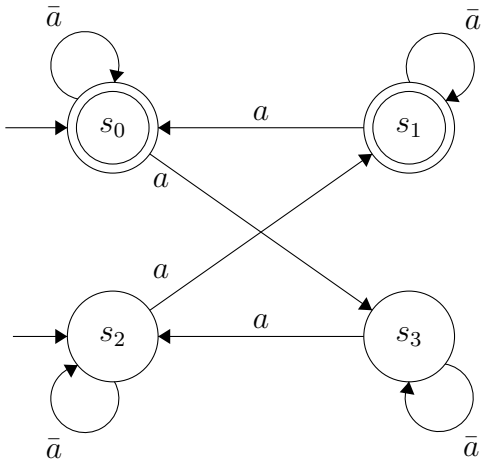


FIGURE 4.4 Example NCW where s_0 and s_2 are initial states and $fin(\{s_0, s_1\})$ the acceptance condition

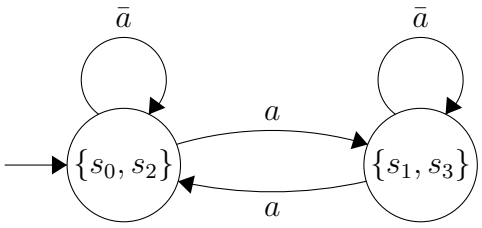


FIGURE 4.5 DCW constructed from Figure 4.4 using the subset construction. Regardless of the acceptance condition, this automaton does not accept the same language as Figure 4.4.

we lose information about entering and leaving the set of accepting states. For example, every run of $aaa\ldots$ in Figure 4.4 infinitely often visits the set of accepting states but in Figure 4.5 we lost this information and it seems as if we either never enter or never leave the set of accepting states (depending on which of the two above discussed options we take for defining the acceptance condition). Therefore, we need to keep track of the moments where we leave the set of accepting states. We call these moments *breakpoints*. We adapt the algorithm from Schneider [85] and present the breakpoint construction in Algorithm 1.

The algorithm works by exploring the state space. Just like with the subset construction, we collect all reachable states S_1' after reading some symbol σ. Additionally, we collect the accepting states that we have stayed in since the last breakpoint. If we just left a breakpoint, then this set is simply all accepting states of S_1'. Otherwise, it is the set of successors of S_2 that are accepting. Each state in the deterministic automaton then becomes a tuple where the first element is like in the subset construction and the second set is the set S_2'. The final acceptance condition of this automaton is the set of states for which the second element of the tuple is empty, i.e. the set of breakpoints since any accepting word can only finitely often visit such a breakpoint.

EXAMPLE 4.14 Consider again Figure 4.4. We now apply the breakpoint construction and obtain the DCW as shown in Figure 4.6. We see that in addition to the state sets from the subset construction, we now also track the states seen since the last breakpoint. The automaton is deterministic and recognizes the same language as the NCW. □

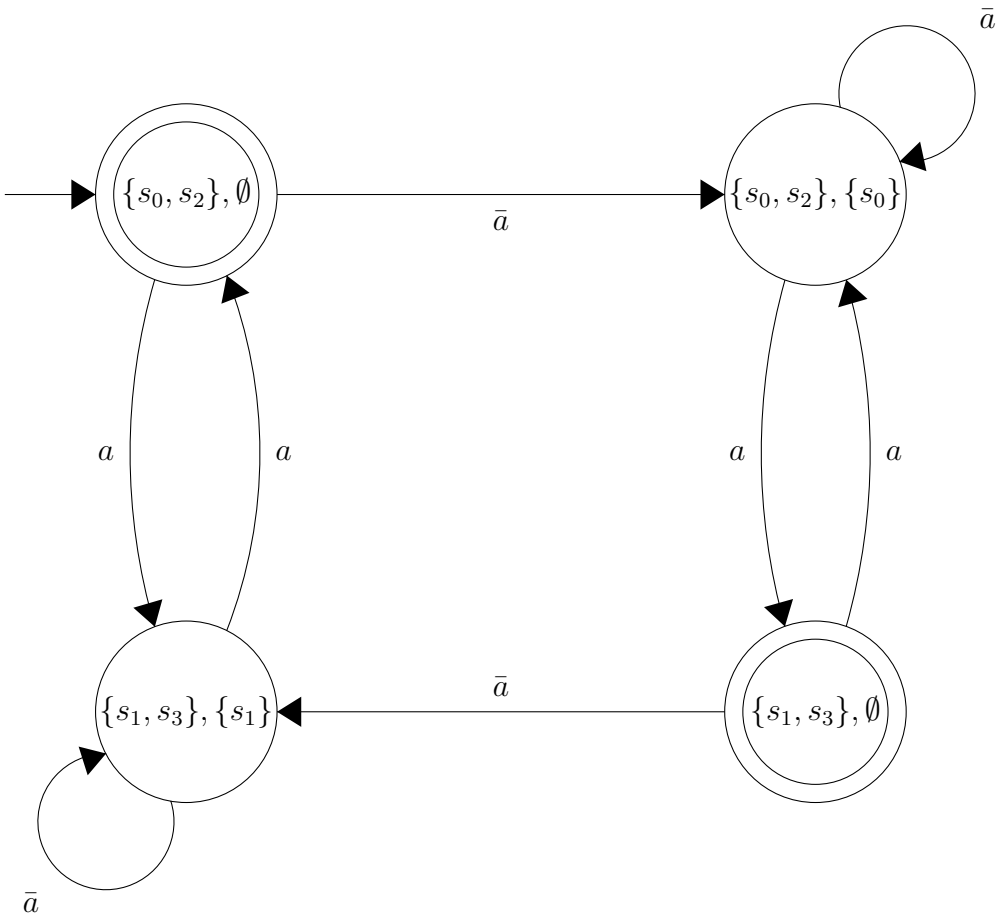


FIGURE 4.6 DCW constructed from Figure 4.4 using the breakpoint construction. Each state is labelled with a pair of sets of states from Figure 4.4. The first set is obtained from the subset construction and the second is the set of states seen since the last breakpoint.

Algorithm 1 `explBreakpoint`$(Q_A, \Sigma, \mathcal{I}_A, \delta_A, \mathit{fin}(\alpha_A))$

$L \leftarrow \{(\mathcal{I}_A, \emptyset)\}$
$S, \delta, \alpha \leftarrow \emptyset$
while $L \neq \emptyset$ **do**
    $(S_1, S_2) \leftarrow \texttt{pick}(L)$
    $S \leftarrow S \cup \{(S_1, S_2)\}$
    $L \leftarrow L \setminus \{(S_1, S_2)\}$
    for $\sigma \in \Sigma$ **do**
        $S_1' \leftarrow \{s | s' \in S_1, (s', \sigma, s) \in \delta_A\}$
        if $S_2 = \emptyset$ **then**
            $S_2' \leftarrow S_1' \cap \alpha_A$
        else
            $S_2' \leftarrow \{s | s' \in S_2, (s', \sigma, s) \in \delta_A\} \cap \alpha_A$
        end if
        $\delta \leftarrow \delta' \cup \{((S_1, S_2), \sigma, (S_1', S_2'))\}$
        if $(S_1', S_2') \notin S$ **then**
            $L \leftarrow L \cup \{(S_1', S_2')\}$
        end if
    end for
    if $S_2 = \emptyset$ **then**
        $\alpha \leftarrow \alpha \cup \{(S_1, S_2)\}$
    end if
end while
return $(S, \Sigma, \{(\mathcal{I}_A, \emptyset)\}, \mathit{fin}(\alpha))$

4.3.2 Adapting the Construction for AWWs

In the explicit breakpoint construction of an NCW, each state of the DCW is associated with two subsets of states of the NCW, representing which states we have seen since a point in time. This is possible because NCWs only support existential transitions. To apply the construction to AWWs we need to adapt it to support universal transitions. We can do so by using positive boolean formulas over the states instead of subsets just like in the transition relation of an AWW. For completeness, we repeat the algorithm in Algorithm 2 using this adjustment. The algorithm is essentially equivalent to Algorithm 1 but uses positive boolean formulas instead of subsets. Note also that we use $\varphi[\bot/(S_A \setminus \alpha_A)]$ to denote a substitution where we replace all non-accepting states by $\bot$ in φ.

It is important to realize that this adapted version of the breakpoint construction is not valid for all AWWs in general. It only works for AWWs constructed from Σ_2-formulas because these have an alternation depth of 2[1] and their initial state formulas only contain states that are not accepting (i.e. they are elements of $\mathcal{B}^+(Q \setminus \alpha)$). A dual construction exists for Π_2-formulas. We refer to Sickert and Esparza [86] for the proof and further details.

The results of this chapter show how we can transform a single LTL formula into a collection of DCW and DBW automata. In the next chapter, we show how we can encode these automata symbolically and how we combine them into a single symbolic DPW.

[1] This holds for AWWs constructed from Δ_2-formulas

Algorithm 2 aww2dcw$(S_A, \Sigma, \mathcal{I}_A, \delta_A, \alpha_A)$

$L \leftarrow \{(\mathcal{I}_A, \bot)\}$
$S, \delta \leftarrow \emptyset$
$\alpha \leftarrow \emptyset$
while $L \neq \emptyset$ **do**
    $(\varphi, \psi) \leftarrow$ pick(L)
    $S \leftarrow S \cup \{(\varphi, \psi)\}$
    $L \leftarrow L \setminus \{(\varphi, \psi)\}$
    for $\sigma \in \Sigma$ **do**
        $\varphi' \leftarrow \delta_A(\varphi, \sigma)$
        if $\psi \equiv \bot$ **then**
            $\psi' \leftarrow \varphi'[\bot/(S_A \setminus \alpha_A)]$
        else
            $\psi' \leftarrow \delta_A(\psi, \sigma)$
        end if
        $\delta \leftarrow \delta \cup \{((\varphi, \psi), \sigma, (\varphi', \psi'))\}$
        if $(\varphi', \psi') \notin S$ **then**
            $L \leftarrow L \cup \{(\varphi', \psi')\}$
        end if
    end for
    if $\psi \equiv \bot$ **then**
        $\alpha \leftarrow \alpha \cup \{(\varphi, \psi)\}$
    end if
end while
return $(S, \Sigma, \{(\mathcal{I}_A, \bot)\}, \alpha)$

CHAPTER 5

Constructing a Symbolic DPW

In this chapter, we show how we combine the automata from the previous chapter into a symbolic DPW. We will first introduce symbolic computation in general. Then, we show how we can encode ω-automata symbolically and how we then combine the automata into a symbolic DRW. Finally, we demonstrate how to symbolically convert a DRW to a DPW using the results from Boker et al. [13].

5.1 Symbolic Computation

We understand symbolic computation in our context as the strategy to encode data for some computation within a data structure that implicitly represents it. Operations on this data structure then map to operations on the original data but aim to use less space and time than when applied to the data directly. This technique can be very efficient, but, unfortunately not all computations can easily be implemented symbolically. Implementing an algorithm symbolically requires a suitable encoding of the data that supports all operations that the algorithm needs. Some data structures are well-suited for symbolic implementations whereas others are not.

Fortunately, most theory on reactive synthesis directly or indirectly relies on set theory and sets can be represented symbolically through propositional formulas. Many common set operations can be symbolically executed by operating on the formula. For example, consider the set $S = \{a, b, c, d, e, f, g, h\}$, we can represent this set and any subset of this set as propositional formulas over the variables p, q and r by associating each variable assignment with an element of S. Suppose we use the following encoding:

$$\{\ \underbrace{a}_{pqr}\ ,\ \underbrace{b}_{pq\bar{r}}\ ,\ \underbrace{c}_{p\bar{q}r}\ ,\ \underbrace{d}_{p\bar{q}\bar{r}}\ ,\ \underbrace{e}_{\bar{p}qr}\ ,\ \underbrace{f}_{\bar{p}q\bar{r}}\ ,\ \underbrace{g}_{\bar{p}\bar{q}r}\ ,\ \underbrace{h}_{\bar{p}\bar{q}\bar{r}}\ \}$$

We can then, for example, encode the subset $\{a, b, c, d\}$ as the propositional formula p. Generally, this means we can use x propositional variables to encode a set of 2^x elements with a propositional formula over x.

Encoding sets using propositional formulas is only useful if we have an efficient data structure to represent propositional formulas on which we can also apply operations that can be mapped back to set operations. Originally introduced by Lee in 1959 [56], binary decision di-

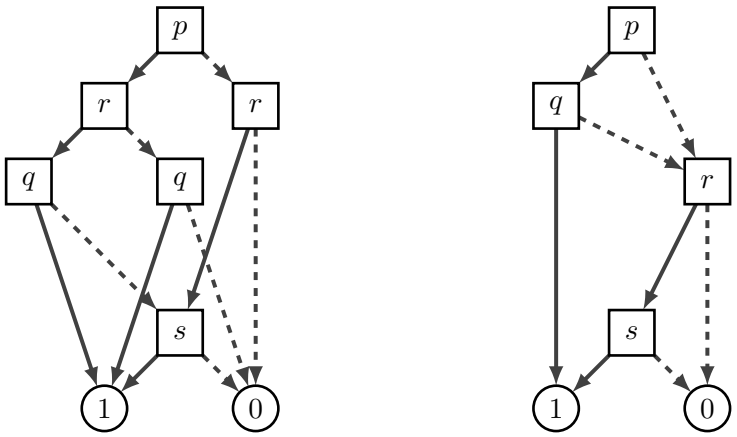


FIGURE 5.1 Two BDDs for the formula $p \wedge q \vee r \wedge s$ where the left and right BDD use the ordering p, r, q, s and p, q, r, s respectively

agrams have exactly this property. A *binary decision diagram* is a binary tree structure that can represent a propositional formula. It is based on the fact that any formula φ containing a variable x can be rewritten as "if x then $\varphi[x/\top]$ else $\varphi[x/\bot]$". This naturally allows for a tree structure to encode any propositional formula, where the leaves represent the truth value of the formula.

If we additionally structure the tree such that the order of evaluation of the variables is the same, regardless of which branches of the tree we follow, we obtain an ordered binary decision diagram. Furthermore, we can reduce this ordered binary decision diagram to obtain a reduced ordered binary decision diagram by removing all duplicate and redundant nodes [15]. When we consider binary decision diagrams (or BDDs for short), we assume the BDD is ordered and reduced.

Two examples of a BDD for the formula $p \wedge q \vee r \wedge s$ are shown in Figure 5.1. We always traverse a binary decision diagram top-down. An outgoing dashed edge indicates that the variable was false, and a solid edge indicates that it was true.

Observe that the size of a BDD is depending on the *variable ordering* we choose. For example, the left BDD in Figure 5.1 has 8 nodes but the right BDD only has 6 nodes. Unfortunately, deciding the optimal variable ordering for a BDD is NP-hard [14]. Still, using certain heuristics it is possible to find variable orderings which work well in practice.

We can estimate how well a variable ordering works by considering the *compression ratio* of the BDD, which can be computed as the number of satisfying assignments over the number of nodes in the BDD. The compression ratio essentially tells us how much information each node contains. Note that compression ratios below 1 are also possible. This occurs, for example, when the amount of variables exceeds the number of satisfying assignments. In these cases using *zero-suppressed decision diagram* can be more efficient [67].

In this research, we did not aim to find a good variable ordering. Instead, we use a rather naive variable ordering. We will see later that even a naive variable ordering can work very well in some cases. In Section 9.4 we discuss future work regarding the ordering of variables in the BDDs.

Given a good variable ordering, a BDD can be used to compactly represent a propositional

formula. There also exist algorithms that implement common operations on propositional formulas, such as negation, disjunction and conjunction, through BDDs. Additionally, there are algorithms for composition (i.e. replacing a variable in a BDD by another BDD) and for determining the number of satisfying variable assignments. The latter is useful because it directly maps to the number of elements in the set encoded by the BDD. Similarly, conjunction maps to intersection, disjunction to union and negation to complementation. Finally, an operation we will frequently encounter is the existential quantification over some set of variables. We denote the existential quantification over the variables $\mathcal{Q}$ in formula φ as $\exists_{\mathcal{Q}}\, \varphi$.

We will not go into the details of these algorithms. Efficient implementations of these algorithms are already widely available through libraries like BuDDy [26] and CUDD [88]. In our implementation, we have used Sylvan [93] which is a modern multi-threaded BDD library.

5.2 Symbolically Encoding ω-Automata

An ω-automaton can easily be represented symbolically by replacing sets with propositional formula as seen in the previous section. In practice, this allows for a representation using BDDs. In this chapter we abstract away the BDDs and use propositional formulas instead. In Chapter 7, we will discuss in more detail how we use BDDs in practice.

From here on, we redefine ω-automata symbolically. Instead of a set of states, we have a set of state variables that encode the states of the automaton. Additionally, we now use the atomic propositions AP to encode the alphabet $\Sigma = 2^{AP}$ of the automaton. To distinguish symbolic automata from explicit automata, we use a slightly different notation which is based on that used by Schneider [85].

DEFINITION 5.1 *A symbolic ω-automaton is a 5-tuple $A = (\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F})$ where*

- *$\mathcal{Q}$ is a set of state variables representing the set of states $2^{\mathcal{Q}}$,*
- *AP is a set of atomic propositions representing the alphabet $\Sigma = 2^{AP}$,*
- *$\mathcal{I}$ is a propositional formula over $\mathcal{Q}$, representing the set of initial states,*
- *$\mathcal{R}$ is a propositional formula over $\mathcal{Q} \cup AP \cup \{q_{\mathbf{x}} \mid q \in \mathcal{Q}\}$, representing the transition relation $\delta \subseteq 2^{\mathcal{Q}} \times 2^{AP} \times 2^{\mathcal{Q}}$, where the $q_{\mathbf{x}}$ variables encode the successor states,*
- *$\mathcal{F}$ is the acceptance condition derived from $f ::= \bot \mid \top \mid inf(\varphi) \mid fin(\varphi) \mid f \wedge f \mid f \vee f$ where φ is any propositional formula over $\mathcal{Q}$.*

EXAMPLE 5.2 We reconsider Example 2.7 and show its symbolic representation. For convenience we repeat it here and present it in Figure 5.2.

Recall Example 2.7 as $(\{A, B, C, D\}, \{a, b\}, \{A, B\}, \delta, inf(\{C\}))$ where

$$\delta = \{(A, \{a, b\}, C), (B, \{a, b\}, C), (B, \emptyset, B), (C, \{a\}, D), (D, \{a\}, C)\}$$

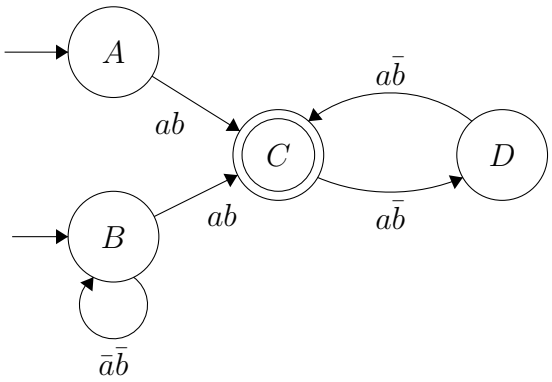


FIGURE 5.2 Example 2.7 repeated here for convenience

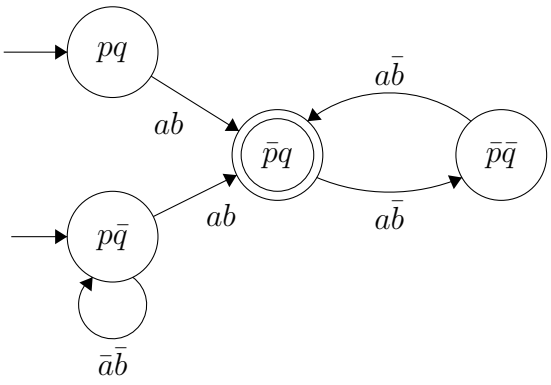


FIGURE 5.3 The automaton of Figure 5.2 with a symbolic state labelling

We now arbitrarily choose an encoding for the state set $\{A, B, C, D\}$ into state variables p and q as follows:

$$\{\ \underbrace{A}_{pq}\ ,\ \underbrace{B}_{p\bar{q}}\ ,\ \underbrace{C}_{\bar{p}q}\ ,\ \underbrace{D}_{\bar{p}\bar{q}}\ \}$$

We use propositional formulas over p and q to represent $\mathcal{I}$ and $\mathcal{F}$, and we additionally use a, b, $p_{\mathbf{x}}$, $q_{\mathbf{x}}$ for $\mathcal{R}$, from which we obtain the symbolic representation $(\{p, q\}, \{a, b\}, p, \mathcal{R}, \mathit{inf}(\bar{p} \wedge q))$ with

$$\mathcal{R} = p \wedge (a \wedge b \wedge \bar{p_{\mathbf{x}}} \wedge q_{\mathbf{x}} \vee \bar{q} \wedge \bar{a} \wedge \bar{b} \wedge p_{\mathbf{x}} \wedge \bar{q_{\mathbf{x}}}) \vee \bar{p} \wedge a \wedge \bar{b} \wedge (q \leftrightarrow \bar{q_{\mathbf{x}}})$$

□

Let us now introduce some useful notation. First of all, given a set of variables $\mathcal{Q}$, we use $\mathcal{Q}_{\mathbf{x}}$ to denote the set of next state variables $\{q_{\mathbf{x}} \mid q \in \mathcal{Q}\}$. Next, we use $\varphi[\mathbf{X}]$ where φ is a propositional formula to denote the substitution of all non-successor state variables q by the successor state variables $q_{\mathbf{x}}$. For example, suppose $\varphi = a \wedge q$ where q is a state variable and a an atomic proposition, then $\varphi[\mathbf{X}] = a \wedge q_{\mathbf{x}}$. We denote the inverse substitution of $\varphi[\mathbf{X}]$ by $\varphi[\mathbf{X}^{-1}]$.

These substitutions allows for an easy computation of the direct successors of some state formula φ over $\mathcal{Q}$ by $(\exists_{\mathcal{Q} \cup AP}(\mathcal{R} \wedge \varphi))[\mathbf{X}^{-1}]$. Similarly, we can compute the direct predecessors of φ by $\exists_{\mathcal{Q}_{\mathbf{x}} \cup AP}(\mathcal{R} \wedge (\varphi[\mathbf{X}]))$.

5.3 Combining the Automata

We now symbolically encode the DBW and DCW automata obtained in the previous chapter and combine them into a single Rabin automaton. For this, we observe that Theorem 4.6 can be reversely applied in the context of automata: We extracted from the specification a boolean combination of subspecifications, and translated each subspecification into a DBW or a DCW. Using that same boolean combination, we can combine all the DBWs and DCWs and obtain a DRW recognizing the full specification.

Recall that Theorem 4.6 has given us a disjunction of n conjunctions of a Σ_2- and a Π_2-formulas. Each Σ_2-formula φ_i is translated to a DCW $A_i = (\mathcal{Q}_{A_i}, AP, \mathcal{I}_{A_i}, \mathcal{R}_{A_i}, \mathcal{F}_{A_i})$ and each Π_2-formula ψ_i to a DBW $B_i = (\mathcal{Q}_{B_i}, AP, \mathcal{I}_{B_i}, \mathcal{R}_{B_i}, \mathcal{F}_{B_i})$. We construct the intersection of these automata as follows:

$$A_i \times B_i = (\mathcal{Q}_i = \mathcal{Q}_{A_i} \cup \mathcal{Q}_{B_i}, AP, \mathcal{I}_i = \mathcal{I}_{A_i} \wedge \mathcal{I}_{B_i}, \mathcal{R}_i = \mathcal{R}_{A_i} \wedge \mathcal{R}_{B_i}, \mathcal{F}_i = \mathcal{F}_{A_i} \wedge \mathcal{F}_{B_i})$$

Finally we take the disjunction of these automata over all $0 < i \leq n$ and we obtain the automaton C:

$$C = \left(\bigcup_{i=1}^{n} \mathcal{Q}_i, \bigwedge_{i=1}^{n} \mathcal{I}_i, \bigwedge_{i=1}^{n} \mathcal{R}_i, \bigvee_{i=1}^{N} \mathcal{F}_i \right)$$

Since this automaton has an acceptance condition that is a disjunction of a conjunction of a DCW and a DBW condition, this automaton is a Rabin automaton. Furthermore, since it is a product of deterministic automata, it itself is also deterministic and therefore a DRW. Finally, it results directly from Theorem 4.6 that this automaton recognizes the specification that we started with in the previous chapter.

In the next section, we continue with the construction from the DRW to the DPW. A crucial requirement for that is that there exists a Streett condition on the structure of the DRW that recognizes the same language as the Rabin condition. This is not generally true and for those cases where it is not, we proceed as follows:

Suppose our specification is the LTL formula φ. Then we repeat the complete construction for the negated specification $\neg\varphi$ up until this point. Because of the duality between DRW and DSW automata, the resulting DRW automaton can be interpreted as a DSW automaton recognizing φ by simply negating the acceptance condition to obtain a conjunction of disjunctions instead of a disjunction of conjunctions. Finally, we take the intersection of the DRW and the DSW automata like before, but we drop the DSW acceptance condition. The result is a DRW on which both a DRW and a DSW condition exist that recognize the same language.

Before we continue to transform this DRW to a DPW, we first need to discuss how we can symbolically compute the SCC decomposition of an automaton since this is a requirement for the algorithm that gives us a DPW. In the next section we present this symbolic SCC decomposition. The section thereafter will discuss the DPW construction.

5.4 SCC Decomposition

Decomposing a graph into its strongly connected components is a frequently occurring task in graph theory and has been thoroughly researched already. Two well known linear-time explicit SCC decomposition algorithms were invented by Tarjan [91] and Dijkstra [27]. These algorithms are not suitable for symbolic computation, however, since they require an enumeration of the state space. Instead, we use a semi-symbolic $O(n \log n)$ time algorithm based on the observation that an SCC containing a node v is exactly the intersection of the forward and backward reachability sets of v. The algorithm was originally invented by Xie and Beerel [98]

and was optimized by Bloem et al. [9] to reach $O(n \log n)$ time complexity.

The algorithm is presented in Algorithm 3 (where initially $V = \top$). It is semi-symbolic since it enumerates one state of each SCC. This also means that the algorithm performs better in the case that the automaton has a few large SCCs than when it has many small SCCs.

Intuitively, the algorithm recursively computes the SCCs by taking an arbitrary node and computing the forward and backward reachability sets of that node simultaneously, while keeping track of which set converges first.[1] The intersection of the two sets is the SCC C and the remaining SCCs are computed in two recursive calls. First, we compute the SCCs of the remaining elements of the set that converged first. Second, we compute the SCCs of the nodes that remain.

Algorithm 3 $\texttt{sccs}(A = (\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F}), V)$

if $V \equiv \bot$ **then**
    return $\emptyset$
end if
$F, B, S, P \leftarrow \texttt{pick}(V)$
while $(S \not\equiv \bot) \wedge (P \not\equiv \bot)$ **do**
    $S \leftarrow (\exists_{\mathcal{Q} \cup AP}\ \mathcal{R} \wedge S)[\mathbf{X}^{-1}] \wedge V \wedge \neg F$
    $P \leftarrow (\exists_{\mathcal{Q}_\mathbf{x} \cup AP}\ \mathcal{R} \wedge (P[\mathbf{X}])) \wedge V \wedge \neg B$
    $F \leftarrow F \vee S$
    $B \leftarrow B \vee P$
end while
if $S \equiv \bot$ **then**
    $J \leftarrow F$
else
    $J \leftarrow B$
end if
while $(S \wedge B \not\equiv \bot) \vee (P \wedge F \not\equiv \bot)$ **do**
    $S \leftarrow (\exists_{\mathcal{Q} \cup AP}\ \mathcal{R} \wedge S)[\mathbf{X}^{-1}] \wedge V \wedge \neg F$
    $P \leftarrow (\exists_{\mathcal{Q}_\mathbf{x} \cup AP}\ \mathcal{R} \wedge (P[\mathbf{X}])) \wedge V \wedge \neg B$
    $F \leftarrow F \vee S$
    $B \leftarrow B \vee P$
end while
$C \leftarrow F \wedge B$
return $\{C\} \cup \texttt{sccs}(A, J \wedge \neg C) \cup \texttt{sccs}(A, V \wedge \neg J)$

5.5 DPW Construction

Using the SCC decomposition, we can now present the DPW construction as given by Boker et al. [13] in the form of a constructive proof. We present it here in the form of a set of algorithms but we do not repeat the proof. An example is provided at the end of this section. For the details as to why this approach is correct, we refer to Boker et al.

[1] Recall that we symbolically compute the successor states of S by $(\exists_{\mathcal{Q} \cup AP}\ \mathcal{R} \wedge S)[\mathbf{X}^{-1}]$ and the predecessor states of P by $(\exists_{\mathcal{Q}_\mathbf{x} \cup AP}\ \mathcal{R} \wedge (P[\mathbf{X}]))$.

The construction is based on the notion of *hopeless* states. We will first define hopeless states. Afterwards, we present an algorithm that can compute these states using the SCC decomposition algorithm of the previous section.

DEFINITION 5.3 [13] *Given a DRW $A = (\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F})$, and a state $s \in 2^{\mathcal{Q}}$, s is said to be* hopeless *if and only if all runs of A containing s infinitely often are rejecting.*

Algorithm 4 `hopeless` $\left(A = \left(\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \bigvee_{i=0}^{N-1} \mathit{inf}(\varphi_i) \wedge \mathit{fin}(\psi_i)\right), V\right)$

```
1:  R ← V
2:  for 0 < i < N do
3:      C ← sccs(A, ¬ψ_i ∧ V)
4:      H ← ⊥
5:      for c ∈ C do
6:          if c ∧ φ_i ≡ ⊥ then
7:              H ← H ∨ c
8:          end if
9:      end for
10:     R ← R ∧ (ψ_i ∨ H)
11: end for
12: return R
```

Algorithm 4 symbolically computes all hopeless states for an automaton A restricted to some state formula V. We iterate through the Rabin pairs A. For each pair, we compute the SCCs of the automaton restricted to states $\neg\psi_i$. Any state satisfying ψ_i is hopeless for this pair since visiting it infinitely often would violate $\mathit{fin}(\psi_i)$.

Next, we check for each SCC whether the SCC contains no states that need to be visited infinitely often according to this pair. If so, this SCC is hopeless for this pair since there is no infinite run in this SCC that is accepting. We collect all these SCCs in the formula H. The hopeless states for this Rabin pair are then the states in H together with the states ψ_i. The hopeless state formula for A is then simply the disjunction of these for all pairs.

Using this algorithm, we can present the results of Boker et al. [13] in Algorithms 5 and 6.[2] The algorithm recursively computes the parity condition for each SCC of the automaton, and then merges these conditions back into a single condition. For each SCC, the algorithm is recursive on the number of pairs in the acceptance condition. When the acceptance condition consists of a single pair $\mathit{inf}(\varphi) \wedge \mathit{fin}(\psi)$, we can construct the parity condition as the list $[\bot, \psi \wedge V, \varphi \wedge \neg\psi \wedge V, \neg\varphi \wedge \neg\psi \wedge V]$.

If we have more than one pair then we look for a pair with index k such that $\psi_k \wedge V \equiv \bot$. The main result of Boker et al. is that such a pair always exists if there exists an equivalent Streett condition on the automaton. Thus, if we use a DRW constructed as described in Section 5.3 then the algorithm always succeeds.

We use this pair to split the acceptance condition $\mathcal{F}$ into two parts. The part that contains the words accepted by the kth pair, which is simply the words where φ_k is visited infinitely often,

[2] Remember that we use the notation introduced in Section 2.1 for lists.

Algorithm 5 $\texttt{dpwscc}\left(A = \left(\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F} = \bigvee_{i=0}^{N-1} inf(\varphi_i) \wedge fin(\psi_i)\right), V\right)$

if $N = 1$ **then**
  return $[\bot, \psi_0 \wedge V, \varphi_0 \wedge \neg\psi_0 \wedge V, \neg\varphi_0 \wedge \neg\psi_0 \wedge V]$
end if
$k \leftarrow \min_{i \geq 0} \begin{cases} i & \psi_i \wedge V \equiv \bot \\ N & \text{otherwise} \end{cases}$
if $k = N$ **then**
  return *failure*
end if
$\mathcal{F}' \leftarrow \bigvee_{i=0}^{k-1} (inf(\varphi_i \wedge V) \wedge fin((\psi_i \vee \varphi_k) \wedge V)) \vee \bigvee_{i=k+1}^{N-1} (inf(\varphi_i \wedge V) \wedge fin((\psi_i \vee \varphi_k) \wedge V))$
$H \leftarrow \texttt{hopeless}((\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F}'), V)$
$C \leftarrow \texttt{sccs}(A, \neg H \wedge V)$
$\gamma_S \leftarrow \bigcup_{c \in C} \{\texttt{dpwscc}((\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F}'), c)\}$
$n \leftarrow \max_{\gamma_c \in \gamma_S} |\gamma_c|$
$\gamma \leftarrow \bigsqcup_{i=0}^{n-1} \left[\bigvee_{\gamma_c \in \gamma_S} \gamma_c[i]\right]$ ▷ if $i \geq |y_c|$ then $\gamma_c[i] = \bot$
return $[\varphi_k \wedge V, \neg\varphi_k \wedge H] \sqcup \gamma$

Algorithm 6 $\texttt{dpw}\left(A = \left(\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F} = \bigvee_{i=0}^{N-1} inf(\varphi_i) \wedge fin(\psi_i)\right)\right)$

$H \leftarrow \texttt{hopeless}(A, \top)$
$C \leftarrow \texttt{sccs}(A, \neg H)$
$\gamma_S \leftarrow \bigcup_{c \in C} \{\texttt{dpwscc}(A, c)\}$
$n \leftarrow \max_{\gamma_c \in \gamma_S} |\gamma_c|$
$\gamma \leftarrow \bigsqcup_{i=0}^{n-1} \left[\bigvee_{\gamma_c \in \gamma_S} \gamma_c[i]\right]$ ▷ if $i \geq |y_c|$ then $\gamma_c[i] = \bot$
return $[\bot, H] \sqcup \gamma$

and the part that contains the words accepted by $\mathcal{F}$ but not accepted by the kth pair. The latter is represented by $\mathcal{F}'$ and consist of all but the kth pair where φ_k is not allowed to be visited infinitely often.

Next, we compute the SCCs of the automaton with acceptance condition $\mathcal{F}'$ but without the hopeless states. For each SCC, we recursively compute the parity condition for $\mathcal{F}'$. We can now compute the parity condition γ that is equivalent with the Rabin condition $\mathcal{F}'$ by "zipping" all parity conditions of the SCCs.

Finally, we combine γ with the parity condition of the kth pair. The parity condition of the kth pair is trivially $[\varphi_k \wedge V, \neg\varphi_k \wedge H]$. To combine this with γ we simply prepend this acceptance condition to γ. This preserves the partitioning induced by the parity condition, since φ_k is hopeless for γ and therefore disjoint from γ.

EXAMPLE 5.4 We will consider a DRW for the language of $\mathbf{GF}\, a \vee \mathbf{FG}\, b$ as presented in Figure 5.4. This example has a Rabin condition that consist of two pairs, namely $\mathit{inf}(p \vee q \vee r) \wedge \mathit{fin}(\bar{p} \wedge q)$ and $\mathit{inf}(\bar{p} \wedge (q \leftrightarrow \bar{r})) \wedge \mathit{fin}(\bot)$. Clearly, an equivalent Streett condition consisting of one pair exists on this automaton, namely $\mathit{fin}(\bar{p} \wedge q) \vee \mathit{inf}(\bar{p} \vee (q \leftrightarrow \bar{r}))$. Thus, we can apply the DPW construction as follows:

We start by computing the hopeless states of the automaton. We do this for each pair separately: the hopeless states of the first pair are $\bar{p} \wedge (q \vee (\bar{q} \wedge \bar{r})$ and for the second pair are $\bar{p} \wedge \bar{q} \wedge \bar{r}$. The intersection of both gives us the single hopeless state $\bar{p}\bar{q}\bar{s}$. To see why this state is hopeless, recall that the hopeless states are those states through which all infinite runs are rejecting. Since there are no infinite runs through $\bar{p}\bar{q}\bar{s}$, it is hopeless.

The next step is to compute the SCCs of the automaton without the hopeless states. It is trivial to see that after removal of $\bar{p}\bar{q}\bar{s}$, the automaton consists of a single SCC containing all states. We now call Algorithm 5 for this SCC.

On line 4 of Algorithm 5, we find the Rabin pair where the fin set is empty. In our case, this is trivial as one of our pairs is $\mathit{fin}(\bot)$. We now extract this pair from the acceptance condition on line 8. The result is a new acceptance condition:

$$\mathcal{F}' = \mathit{inf}(p \vee q \vee r) \wedge \mathit{fin}(\bar{p} \wedge q \vee \bar{p} \wedge (q \leftrightarrow \bar{r})) \equiv \mathit{inf}(p \vee q \vee r) \wedge \mathit{fin}(\bar{p} \wedge (q \vee r))$$

We now compute the hopeless states with respect to the SCC and this new acceptance condition and obtain $H = \bar{p} \wedge (q \vee r)$. This leaves us with a new SCC that is a single non-hopeless state with which we continue recursively on line 11. This time, we only have a single pair and return immediately with the parity condition to obtain γ_S:

$$\gamma_S = \{[\bot, \bot, p \wedge \bar{q} \wedge \bar{r}, \bot]\}$$

Line 12 and 13 then merge the parity sets for the different SCCs back together, but since we only have a single SCC and therefore only a single parity set, nothing changes and we return from Algorithm 5 with γ_S on line 3 of Algorithm 6:

$$\gamma_S = \{[\bar{p} \wedge (q \leftrightarrow \bar{r}), \bar{p} \wedge q \wedge r, \bot, \bot, p \wedge \bar{q} \wedge \bar{r}, \bot]\}$$

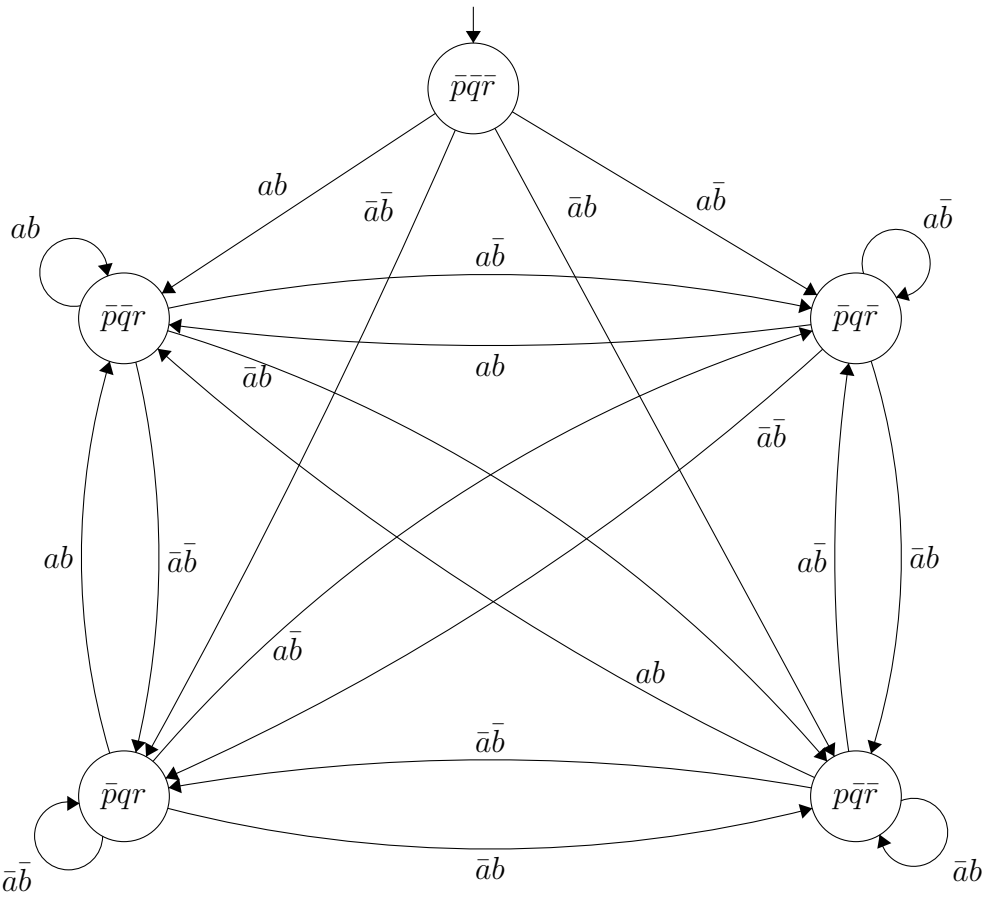


FIGURE 5.4 DRW of **GF** $a \vee$ **FG** b with condition $inf(p \vee q \vee r) \wedge fin(\bar{p} \wedge q) \vee inf(\bar{p} \wedge (q \leftrightarrow \bar{r})) \wedge fin(\bot)$. An equivalent Streett condition on the same structure is $fin(\bar{p} \wedge q) \vee inf(\bar{p} \vee (q \leftrightarrow \bar{r}))$.

Again, we do not need to merge the conditions as there is only one so we proceed and finally include the hopeless state $\bar{p}\bar{q}\bar{s}$ again so that the parity condition is indeed a partitioning of states, resulting in the parity condition:

$$\mathcal{F}_{parity} = [\bot, \bar{p} \wedge \bar{q} \wedge \bar{r}, \bar{p} \wedge (q \leftrightarrow \bar{r}), \bar{p} \wedge q \wedge r, \bot, \bot, p \wedge \bar{q} \wedge \bar{r}, \bot]$$

In other words, each state is added to a parity set with index as follows:

- $\bar{p}\bar{q}\bar{r} : 1$
- $\bar{p}\bar{q}r : 2$
- $\bar{p}q\bar{r} : 1$
- $\bar{p}qr : 3$
- $p\bar{q}\bar{r} : 6$

□

CHAPTER 6

Generating a Mealy Machine

In this chapter, we discuss how we convert the symbolic DPW to a Mealy machine. First, we convert the DPW to a parity game. Then we solve the game and the solution will give us a strategy. This strategy is then used to construct a Mealy automaton.

6.1 DPW to Parity Game

At this point in the construction, it becomes relevant which of the atomic propositions are inputs and which are outputs. Given this information, we can split the DPW into a parity game in which the system plays against its environment. We let the environment and the system control the inputs and outputs respectively. We first describe the approach informally. Then we illustrate it using an example and finally, we formalize it.

Suppose we have a symbolic DPW $(\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \mathcal{F})$. We denote the atomic propositions controlled by the environment and the system by AP_e and AP_c respectively. We now construct a game such that the environment first decides the valuation of AP_e after which the system decides the valuation of AP_c. This means that we will need intermediate states for each transition to represent the situations where the environment has decided on the valuation and the system still has to decide. These intermediate states hold the valuation of AP_e. They are therefore valuations of the variables $\mathcal{Q} \cup AP_e$. From these intermediate states, we have transitions over AP_c to non-intermediate states. To distinguish both sets of states, we introduce an extra variable i that is set to true for intermediate states. Finally, we assign the priorities such that all states where i is true are assigned priority $|\mathcal{F}| + 1$, and all other states are assigned the index of $\mathcal{F}$ for which the valuation is satisfying.

EXAMPLE 6.1 Consider the DPW in Figure 6.1 as an example. Now suppose a and b are controlled by the environment and the system respectively. We can convert this DPW to a parity game as in Figure 6.2. Starting in the initial state $pq\bar{i}\bar{a}$ with the environment player, the game is played by alternating between both players such that the environment decides whether to play a or $\bar{a}$ and the system decides b or $\bar{b}$. States are identified by state variables p and q. To "remember" the choices of the environment in the system states, the variables AP_e are also included as state variables (defaulting to false for environment states). Note that for simplicity, the implicit sink state as discussed in Section 2.3.1 is not shown. □

We now formalize this construction as follows. First, we convert Definition 2.9 to a sym-

bolic definition similarly to how we symbolically represent an ω-automaton as discussed in Section 5.2. Then we provide the lemma that formalizes the idea discussed above.

DEFINITION 6.2 *A* symbolic parity game *is a 7-tuple* $G = (\mathcal{Q}, \mathcal{Q}_c, AP_c, AP_e, \mathcal{I}, \mathcal{R}, \mathcal{F})$ *where*

- $\mathcal{Q}$ *is a set of state variables representing the set of states* $2^{\mathcal{Q}}$*,*
- $\mathcal{Q}_c$ *is a propositional formula representing the states controlled by the system,*
- AP_c *and* AP_e *are atomic propositions belonging to player* c *and* e *respectively,*
- $\mathcal{I}$ *is a propositional formula over* $\mathcal{Q}$ *representing the initial state of the game,*
- $\mathcal{R}$ *is a propositional formula over* $\mathcal{Q} \cup AP_c \cup AP_e \cup \{q_{\mathbf{x}} \mid q \in \mathcal{Q}\}$*, representing the transition relation where the* $q_{\mathbf{x}}$ *variables encode the successor states,*
- $\mathcal{F}$ *is a list of propositional formulas over* $\mathcal{Q}$ *that induce a partitioning of the states* $2^{\mathcal{Q}}$ *such that* $\mathcal{F}_i$ *represents the states with* priority i *(starting with* $i = 0$*).*

Furthermore, the arena *of the game is the underlying symbolic DPW* $(\mathcal{Q}, AP_c \cup AP_e, \mathcal{I}, \mathcal{R}, \mathcal{F})$

LEMMA 6.3 *Given a symbolic DPW* $(\mathcal{Q}_A, AP, \mathcal{I}_A, \mathcal{R}_A, \mathcal{F}_A)$ *and the environment variables* $AP_e \subseteq AP$ *such that* $AP_c = AP \setminus AP_e$*, the corresponding symbolic parity game is given by* $(\mathcal{Q}, \mathcal{Q}_c, AP_c, AP_e, \mathcal{I}, \mathcal{R}, \mathcal{F})$ *where*

$$
\begin{aligned}
\mathcal{Q} &= \mathcal{Q}_A \cup \{a' \mid a \in AP_e\} \cup \{i\} \\
\mathcal{Q}_c &= i \\
\mathcal{I} &= \mathcal{I}_A \wedge \bar{i} \wedge \bigwedge_{a \in AP_e} \bar{a'} \\
\mathcal{R} &= \mathcal{R}_c \vee \mathcal{R}_e \\
\mathcal{F} &= \left(\bigsqcup_{f \in \mathcal{F}_A} [f \wedge \bar{i}] \right) \sqcup [\bot, i]
\end{aligned}
$$

and where

$$
\begin{aligned}
\mathcal{R}_c &= \left(\bigwedge_{a \in AP_e} \bar{a} \wedge \bar{a'_{\mathbf{x}}} \right) \wedge i \wedge \bar{i_{\mathbf{x}}} \wedge \mathcal{R}_A[a/a'] \\
\mathcal{R}_e &= \left(\bigwedge_{a \in AP_e} \bar{a'} \wedge (a \leftrightarrow a'_{\mathbf{x}}) \right) \wedge \left(\bigwedge_{b \in AP_c} \bar{b} \right) \wedge \bar{i} \wedge i_{\mathbf{x}} \wedge \left(\bigwedge_{q \in \mathcal{Q}} q \leftrightarrow q_{\mathbf{x}} \right)
\end{aligned}
$$

and where a' *and* b' *denote a new variable representing the state variable corresponding to the atomic proposition of the environment and the system respectively. The substitution* $\mathcal{R}_A[a/a']$ *is the substitution of all environment atomic propositions* a *by the state variables* a'*.*

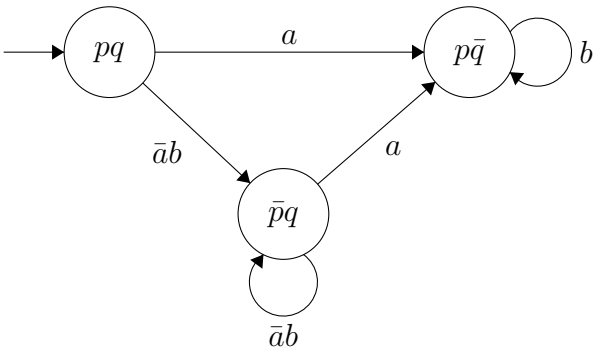


FIGURE 6.1 DPW where $\mathcal{F} = [p \wedge q, \bar{p}, p \wedge \bar{q}, \bot]$

The idea of the construction is simple, but its symbolic representation is difficult to decipher. Let us decompose the symbolic encoding of the transition relation to illustrate how this lemma implements the idea previously described informally. First of all, we split the transition relation into two parts, $\mathcal{R}_c$ and $\mathcal{R}_e$ representing the transitions controlled by the system and the environment respectively.

We start with $\mathcal{R}_e$. This formula contains the transitions from the environment states to the system states. They are therefore transitions from $\bar{i}$ to i, hence the term $\bar{i} \wedge i_{\mathbf{x}}$. Furthermore, on these transitions the system atomic propositions are unused so we set them to false by $\bigwedge_{b \in AP_c} \bar{b}$. Since these transitions move to intermediate states, the state variables remain the same and so we set each successor state variable $q_{\mathbf{x}}$ to the value of q in $\bigwedge_{q \in \mathcal{Q}} q \leftrightarrow q_{\mathbf{x}}$. Finally, we ensure the assignment of the environment atomic propositions $a \in AP_e$ are transferred to their new equivalent state variables a', and we set those state variables to false in the current state in $\bigwedge_{a \in AP_e} \bar{a_{\mathbf{x}}} \wedge (a \leftrightarrow a'_{\mathbf{x}})$. The intersection of these components results in a transition relation that simply moves from every non-intermediate (environment) state to an intermediate (system) state by copying over the assignment of environment atomic propositions to an assignment of their corresponding state variables.

In $\mathcal{R}_c$, we now move back from i to $\bar{i}$ so we obtain $i \wedge \bar{i_{\mathbf{x}}}$. We set the atomic propositions of the environment as they are not used for these transitions. Furthermore, we also set the successor equivalent of the state variable corresponding to the environment atomic propositions to false as these state variables are all set to false in the environment states. This gives us $\bigwedge_{a \in AP_e} \bar{a} \wedge \bar{a'_{\mathbf{x}}}$. Finally, we substitute each environment atomic proposition in the automaton's transition relation by the corresponding state variable as these state variables hold the environment's assignment of its atomic propositions. Again, the conjunction of these components gives us the transition relation for the system player.

6.2 Solving the Game

Now that we have a parity game, we have to determine the winner. Furthermore, we are interested in the strategy that the controller needs to adhere to in order to win, as this strategy is essentially an implementation of the controller. For this purpose, we use the *distraction fixpoint iteration* algorithm by van Dijk and Rubbens [94]. There are many other algorithms

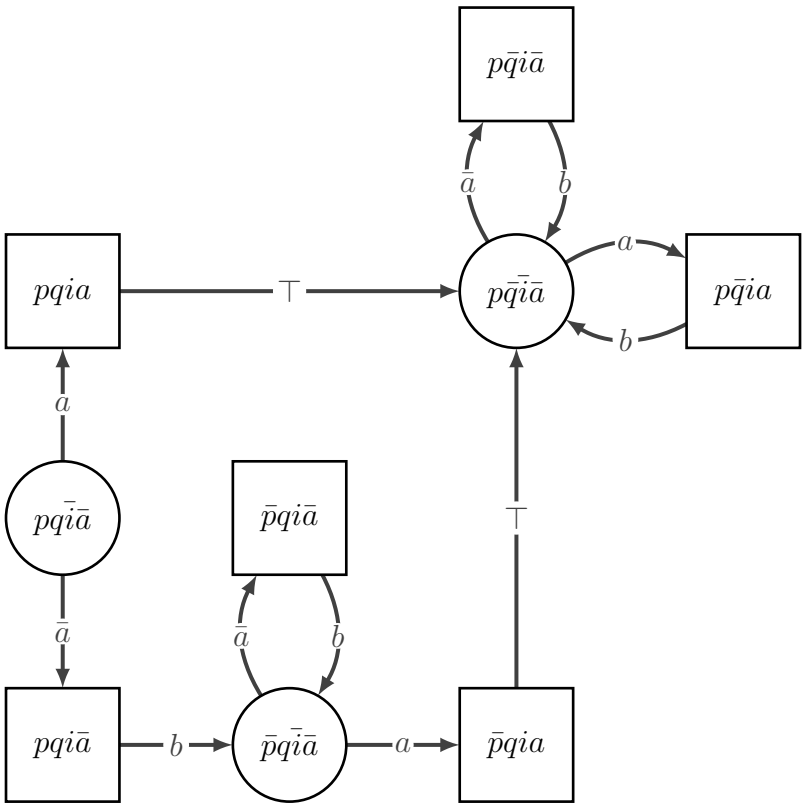


FIGURE 6.2 Parity game constructed from the DPW in Figure 6.1 where the circle nodes and square nodes are controlled by the environment and system respectively. Each node is labelled with its variable assigment. The acceptance condition is $\mathcal{F} = [q \wedge p \wedge \neg i, \neg p \wedge \neg i, \neg q \wedge p \wedge \neg i, \bot, \bot, i]$ and the game starts in $p\bar{q}i\bar{a}$.

for this purpose [5, 45, 99] but we use distraction fixpoint iteration because of its simplicity and because it is well-suited for an efficient symbolic implementation.

The algorithm works by iteratively evaluating for each immediate successor of every state of a certain priority, whether it is a good move for the player that owns that state or whether the successor is a so-called "distraction". In every iteration, the set of next distractions is updated based on the set of current distractions until a fixpoint is reached. The states are considered in the order of their priority, starting at the lowest priority (and thus the highest index) and moving to the highest priority. Every time the set of distractions changes, all distractions with a lower priority than that of the current iteration are removed and the algorithm starts again with the lowest priority (thus resulting in an exponential-time algorithm).

Given a symbolic parity game $G = \left(\mathcal{Q}, \mathcal{Q}_c, AP_c, AP_e, \mathcal{I}, \mathcal{R}, \bigsqcup_{p=0}^{d} \mathcal{F}_p\right)$, let V_{even} denote the states with even priority, i.e. $V_{\text{even}} = \bigwedge_{p=0}^{\lfloor \frac{d}{2} \rfloor} \mathcal{F}_{2p}$. Similarly, let V_{odd} denote the states with odd priority. Then given the set of distractions for some game G as a propositional formula Z, the winning regions of G can be estimated as follows:

DEFINITION 6.4 **[94]** *Given a game G, a propositional formula Z representing the set of distraction and a player a, the states estimated to be won by a are defined as:*

$$\mathit{wonBy}(G, Z, a) = \begin{cases} V_{even} \wedge \neg Z \vee V_{odd} \wedge Z & a = c \\ V_{even} \wedge Z \vee V_{odd} \wedge \neg Z & a = e \end{cases}$$

A state v with priority p is considered a distraction if p is even (resp. odd), the state is owned by c (resp. e) and all of its successors are in *wonBy*(G, Z, e) (resp. *wonBy*(G, Z, c)) or it is owned by e (resp. c) and there is a successor that is in *wonBy*(G, Z, e) (resp. *wonBy*(G, Z, c)). Initially, the even and odd states are distractions for e and c respectively. We can formalize a single distraction computation as follows.

DEFINITION 6.5 **[94]** *Given a parity game* $G = (\mathcal{Q}, \mathcal{Q}_c, AP_c, AP_e, \mathcal{I}, \mathcal{R}, \bigsqcup_{p=0}^{d} \mathcal{F}_p)$ *where* $AP = AP_c \cup AP_e$ *and a propositional formula representing the current set of distractions* Z*, the new set of distractions is defined as:*

$$\begin{aligned} \textit{onestep}(G, Z) = V_{\textit{even}} \wedge (\mathcal{Q}_c \wedge \neg (\exists_{\mathcal{Q}_{\mathbf{x}} \cup AP} \mathcal{R} \wedge \neg \textit{wonBy}(G, Z, e)[\boldsymbol{X}]) \\ \vee \mathcal{Q}_e \wedge (\exists_{\mathcal{Q}_{\mathbf{x}} \cup AP} \mathcal{R} \wedge \textit{wonBy}(G, Z, e)[\boldsymbol{X}])) \\ \vee V_{\textit{odd}} \wedge (\mathcal{Q}_c \wedge (\exists_{\mathcal{Q}_{\mathbf{x}} \cup AP} \mathcal{R} \wedge \textit{wonBy}(G, Z, c)[\boldsymbol{X}]) \\ \vee \mathcal{Q}_e \wedge \neg (\exists_{\mathcal{Q}_{\mathbf{x}} \cup AP} \mathcal{R} \wedge \neg \textit{wonBy}(G, Z, c)[\boldsymbol{X}])) \end{aligned}$$

We present the algorithm in Algorithm 7 where we use V to denote the set of reachable states.[1] It is heavily inspired by the symbolic algorithm presented by Lijzinga and van Dijk [58], which in turn is a symbolic adaption of the algorithm presented by van Dijk and Rubbens [94]. This algorithm does some additional bookkeeping to also obtain the winning strategies. We can use this algorithm to determine whether the original specification is realizable by verifying that the initial state of the game is in W_c, i.e. the initial state is in the system's winning region. If so, S_c is a strategy that adheres to the specification. If not, S_e is a counter-strategy that violates the specification. More details about the computation of these strategies and the algorithm in general can be found in the paper by van Dijk and Rubbens [94].

6.3 Strategy Determinization

As noted by Lijzinga and van Dijk [58], the algorithm of the previous section does not usually yield a single strategy. Instead, the propositional formula holding the strategy essentially encodes all possible winning strategies. In other words, for some states of the game, the strategy for some player a leaves the choice of the successor state open since no matter the choice, the game is won by a anyhow. Of course, this "nondeterminism" is inconvenient when the goal is to create a system of which the behaviour should be fixed. For that reason, we need to do a small postprocessing step to ensure that we have exactly one strategy.

Up until now, we have abstracted away from the BDDs that underlie the propositional formulas and used operations of which we know that efficient BDD equivalents exist. However, as we will see in a minute, for an efficient solution to this problem, this is not sufficient. Therefore, we design a custom BDD operation that "determinizes" our strategy. Let us first describe the problem in more detail and solve it naively.

Essentially, we need a symbolic representation of a function that takes the current state and

[1] This is symbolically computed using a fixpoint operation starting with $\mathcal{I}$ and iteratively computing the direct successors and adding them to the next iteration.

Algorithm 7 `dfi`$(G = (\mathcal{Q}, \mathcal{Q}_c, AP_c, AP_e, \mathcal{I}, \mathcal{R}, \bigsqcup_{p=0}^{d} \mathcal{F}_p))$

$Z \leftarrow \bot$
$C_0, \ldots, C_d \leftarrow \bot$
$S \leftarrow \bot$
$p \leftarrow d$
while $p \geq 0$ **do**
    $a \leftarrow \begin{cases} c & p \mod 2 = 0 \\ e & \textit{otherwise} \end{cases}$
    $X \leftarrow \mathcal{F}_p \wedge \neg Z \wedge \bigwedge_{i=0}^{d} \neg C_i$
    $Z' \leftarrow X \wedge \textit{onestep}(G, Z)$
    $Z \leftarrow Z \vee Z'$
    $S \leftarrow S \wedge \neg X$
        $\vee X \wedge \mathcal{Q}_c \wedge \mathcal{R} \wedge \textit{wonBy}(G, Z, c)[\mathbf{X}]$
        $\vee X \wedge \mathcal{Q}_e \wedge \mathcal{R} \wedge \textit{wonBy}(G, Z, e)[\mathbf{X}]$
    $V_{>p} \leftarrow \bigwedge_{i=p+1}^{d} \mathcal{F}_i$
    if $Z' \equiv \bot$ **then**
        $C_p \leftarrow C_p \wedge \neg V_{>p}$
        $p \leftarrow p - 1$
    else
        $W \leftarrow \textit{wonBy}(G, Z, a)$
        $C_p \leftarrow C_p \vee V_{>p} \wedge \bigwedge_{i=0}^{d} \neg C_i \wedge V \wedge \neg W$
        $Z \leftarrow \neg W \wedge Z$
        $p \leftarrow d$
    end if
end while
$W_c \leftarrow \textit{wonBy}(G, Z, c)$
$W_e \leftarrow \textit{wonBy}(G, Z, e)$
$S_c \leftarrow W_c \wedge V_c \wedge S$
$S_e \leftarrow W_e \wedge V_e \wedge S$
return W_c, W_e, S_c, S_e

the input atomic propositions, and outputs the next state and the output atomic propositions. Symbolically, this means if we have an assignment of current state variables and input atomic propositions (i.e. the input assignment) then the conjunction of that with the strategy should yield a BDD which has exactly one satisfying assignment for the next state variables and the output atomic propositions (i.e. the output assignment). We can obtain such a determinized strategy as follows:

Suppose we have a game $G = (\mathcal{Q}, \mathcal{Q}_c, AP_c, AP_e, \mathcal{I}, \mathcal{R}, \bigsqcup_{p=0}^{d} \mathcal{F}_p)$ and some strategy S_c that is winning for c as obtained from Algorithm 7. Then we can easily determinize it by collecting for all possible input assignments i, a single satisfying assignment from $i \wedge S_c$. However, doing so nullifies the compression obtained from using a BDD as we explicitly iterate over assignments of state variables and input atomic propositions.

Instead of such an explicit iteration, we need to take advantage of the BDD structure. For that, we recall that every satisfying assignment is a unique path through the BDD. Essentially what we are after is a BDD S'_c such that $S_c \rightarrow S'_c$ and that there is exactly one unique path through S'_c for a single input assignment. If we order the BDD such that input variables precede the output variables, then we can obtain these paths using a pruned depth-first search algorithm. Before presenting this algorithm, let us first introduce some notation:

DEFINITION 6.6 *Given a propositional formula φ, we refer to the root node of the BDD representing φ with variable ordering O as $\langle\varphi\rangle_O$. Furthermore, we use var$(\langle\varphi\rangle_O)$ to refer to the variable associated with that node. Finally, we use high$(\langle\varphi\rangle_O)$ (resp. low$(\langle\varphi\rangle_O)$) to refer to the successor node of $\langle\varphi\rangle_O$ if var$(\langle\varphi\rangle_O)$ is true (resp. false).*

We now present the algorithm in Algorithm 8 (where initially $S = \emptyset$) for which we use any variable ordering O (as a list of variables) such that all the input variables precede the output variables o. We use *firstout*(O) for the first output variable occurring in O. This variable essentially separates the input variables from the output variables in the ordering.

The algorithm is a depth-first search algorithm that keeps track of the variables it encounters on each path. The algorithm explores each path of the input variables, but as soon as it reaches an output variable, it will only take a single path to true. Whenever it reaches the true terminal, it checks all variables that have been encountered on the path. If there are output variables that have not been encountered then that means that for these output variables any assignment is satisfying. We, therefore, take the conjunction of negations of those variables to fix them to false, enforcing a single satisfying assignment. The result is a BDD for which every input assignment identifies a unique satisfying assignment.

6.4 Strategy to Mealy Machine

A deterministic strategy for a parity game intuitively resembles a reactive controller already, but for completeness sake, we have to consider some technicalities to convert the strategy to a Mealy machine. In an implementation of the construction, this step is highly dependent on the desired output format. We informally describe the generic approach here using the mathematical

Algorithm 8 `stratdet`$(\langle\varphi\rangle_O, o, S)$

if $\langle\varphi\rangle_O = \langle\bot\rangle_O$ **then**
    return $\langle\varphi\rangle_O$
end if
if $\langle\varphi\rangle_O = \langle\top\rangle_O$ **then**
    return $\bigwedge_{s\in o\setminus S} \bar{s}$
end if
$x \leftarrow$ *var*$(\langle\varphi\rangle_O)$
$S \leftarrow S \cup \{x\}$
$L \leftarrow$ `stratdet`(*low*$(\langle\varphi\rangle_O, o, S)$)
if (*firstout*$(O) \in S) \wedge (L \neq \langle\bot\rangle_O)$ **then**
    return $\langle\bar{x}\rangle_O \wedge L$
end if
$H \leftarrow$ `stratdet`(*high*$(\langle\varphi\rangle_O, o, S)$)
if *firstout*$(O) \in S$ **then**
    return $\langle x\rangle_O \wedge H$
end if
return $\langle x\rangle_O \wedge H \vee \langle\bar{x}\rangle_O \wedge L$

representation of a Mealy machine and but in the next chapter, we cover more concretely how this step is done in the implementation.

To convert the strategy to a Mealy machine, we interpret the strategy as a (restricted) transition relation that we then explicitly explore breadth-first, starting at the initial state. As we explore the strategy, we replace the intermediate states with transitions that directly go from the state of each incoming transition to the state of each outgoing transition. We then set the environment's variables to the input of the transition and the controller variables to the output. More concretely, suppose we have some input atomic proposition a and an output atomic proposition b. Also, suppose we have some strategy S_c that tells us that if the environment decides to play from some state A to the intermediate state i using atomic proposition a then the system should respond by moving to state B using atomic proposition b. Then the Mealy transition belonging to this strategy is a transition from state A to state B with input a and output b as also graphically represented in Figure 6.3.

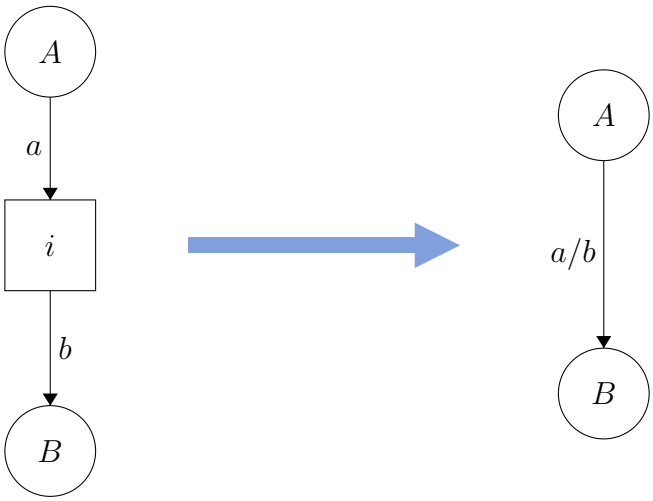


FIGURE 6.3 Transforming intermediate states from a parity game to a Mealy transition

CHAPTER 7

Implementation

In this chapter, we discuss how the construction is realized in the tool Otus. We start with a description of the overall architecture and dependencies of Otus. We then discuss how we use binary decision diagrams. Finally, we provide implementation details on each part of the construction separately.

7.1 Framework

Otus is implemented as part of the LTL and ω-automata library Owl [49]. Owl is used for parsing, simplifying and normalizing the LTL formulas. Furthermore, it implements the explicit automata constructions as discussed in Chapter 4. Owl is written in Java but is compiled to a native executable using GraalVM native image [76] to reduce the startup latency that is inherently present in Java applications.

Otus is developed as a module in Owl's module architecture. The module that implements the construction is `ltl2aig`. It accepts an LTL formula and returns an and-inverter graph, which we discuss later. The benefit of this modular approach is that it is possible to add any pre- and post-processing modules. For example, LTL simplification modules can be executed prior to the construction. Furthermore, if simplification modules for and-inverter graphs are later added to Owl, they can simply be added after the `ltl2aig` module. Currently the recommended configuration is to run the following modules in sequence: `ltl`, `simplify-ltl`, `ltl2aig`, `string`.

The `ltl` module is responsible for parsing the LTL formulas and can accept input from various sources. The input format is a plain LTL formula. Atomic propositions start with a lowercase letter and are optionally quoted. Owl uses "`&`" for conjunction, "`|`" for disjunction and "`!`" for negation as is common in other tools that deal with boolean logic. Furthermore, the $\top$ and $\bot$ literals are represented by "`1`" and "`0`" respectively. Alternative syntax is available but we refer to the grammar that is embedded in the implementation for more information. Furthermore, Owl's readme file contains more information on the configuration parameters for input and output as well as instructions on how to execute and combine the modules.

7.2 Binary Decision Diagrams

Binary decision diagrams are at the core of Otus. The library used for processing binary decision diagrams essentially serves as the engine for the tool. A good performance of the BDD library is critical for Otus to deliver timely results.

Owl includes the BDD library JBDD [64], which is a pure Java library that is not very performant. It is used in the construction of the explicit automata provided by Owl. It was also used during development for its simplicity and because its pure Java nature allows the usage of debuggers that support the Java Debug Interface. However, for performance, we use an alternative library called Sylvan [93].

Sylvan is a highly optimized BDD library that uses the work-stealing framework Lace [96] to parallelize BDD operations. Sylvan is a native library so we dynamically link it to Owl as part of the native image generation. GraalVM native image's C interface is used to communicate with Sylvan. However, we do not communicate with Sylvan directly, but rather through an intermediate wrapper implemented in C for reasons that will become clear later. A downside of using the native image C interface is that Sylvan is only enabled if Owl is compiled to a native image. If it is compiled to normal Java bytecode then Sylvan is disabled and JBDD is used instead.

We configure Sylvan with an 8 GB memory cap, a table ratio of 0 and an initial ratio of 10 and use 6 workers threads with a task deque size of 1000000. For the meaning of these parameters, we refer to Sylvan [93]. The parameters have been chosen such that they are well suited for the machines that run the empirical evaluation (which is discussed in Chapter 8).

Sylvan, like any BDD library, needs a way to know which nodes are still in use and which can be reclaimed. This is commonly achieved by keeping a manual reference count. However, keeping such a reference count has similarities with manual memory management which is rather unorthodox in Java, as it uses garbage collection for memory management instead. We, therefore, use a cleaner solution in which we create wrapper Java objects for each root node that results from a BDD operation and let BDD nodes be garbage collected automatically.

For each wrapper object, we maintain a weak reference (i.e. a reference that does not prevent garbage collection) in some central location. This enables us to ensure that there is always only a single wrapper object for one BDD node. This not only means we efficiently cache these wrapper objects but also that we can check for the equality of two BDDs simply by comparing the memory addresses of the objects. However, the main reason for maintaining these references is to support garbage collection.

Garbage collection is initiated by Sylvan as a response to a BDD operation request if during the operation there is not enough space to hold all the nodes. If this happens, Sylvan calls a callback function of the C wrapper, which in turn wakes up a special thread that then requests the list of nodes from the Java implementation. Unfortunately, we cannot let Sylvan request this information from there directly as it is necessary for the thread to be registered with the native image for it to call into the Java implementation. We, therefore, use a "exchange" thread that is started in Java during the initialization of the application. This thread then repeatedly waits for

a signal from Sylvan to request the list of nodes.

As soon as the signal is received, the wrapper calls back into Java and immediately triggers a Java garbage collection. Next, the list of weak references is pruned such that the list only contains references to objects that have not been garbage collected. We now allocate a block of raw unmanaged memory in Java and write the BDD node identifiers of the list to this block of memory. A pointer to this block of memory is then returned and communicated to Sylvan.

We graphically summarize this process in Figure 7.1.

7.3 Constructing Explicit Automata

The constructions of Chapter 4 are pre-existing in Owl. Otus simply combines these constructions to obtain the DCWs and DBWs. First, the input LTL formula is converted to positive normal form, for which we use an existing implementation that applies Definition 2.3 to obtain a LTL formula in line with Definition 2.1. Then, a list of pairs of Σ_2- and Π_2-formulas is extracted. For this, a pre-existing implementation is used that applies the idea as discussed in Section 4.1, and performs some additional intermediate syntactic simplifications.

In the next step, these formulas are translated to DBWs and DCWs. Again, this step uses pre-existing implementations. The implementation inlines the AWW construction to obtain an on-the-fly DBW and DCW construction. Internally, this uses JBDD to compactly encode the LTL formulas that are associated with each state. For simplicity, we do not replace this with Sylvan. Furthermore, since the sizes of these BDDs are likely considerably smaller, we do not expect a large performance improvement from using Sylvan for this part of the construction.

7.4 Constructing a Symbolic DPW

Each of the DBWs and DCWs is converted to a symbolic representation. The encoding consists of two BDDs for each automaton. A BDD representing the initial state and a BDD for the transition relation. Note that unlike what is discussed in Section 5.2 there are no separate BDDs for the acceptance condition. Instead, we encode the acceptance sets in the transition relation using a BDD variable for each acceptance set. The acceptance condition is then simply constructed from these variables.

The variable ordering chosen for each of the BDDs is based on the kind of variable and is ordered as (1) atomic propositions, (2) current state, (3) acceptance sets and finally (4) the successor state. The orders of atomic proposition and acceptance set variables are simply identical to their order in the explicit representation, i.e. the first atomic proposition is assigned the first atomic proposition variable in the encoding. States are encoded naively by assigning a unique integer to each explicit state and setting the BDD variables according to the binary representation of the integer. Thus, the transition relation of an automaton with N states, a acceptance sets and p atomic propositions is encoded in a BDD consisting of $2\lceil\log_2(N)\rceil + a + p$ variables.

Next, the product DRW is constructed. The transition relation is constructed such that all atomic proposition variables are removed from the individual automata and placed at the top of

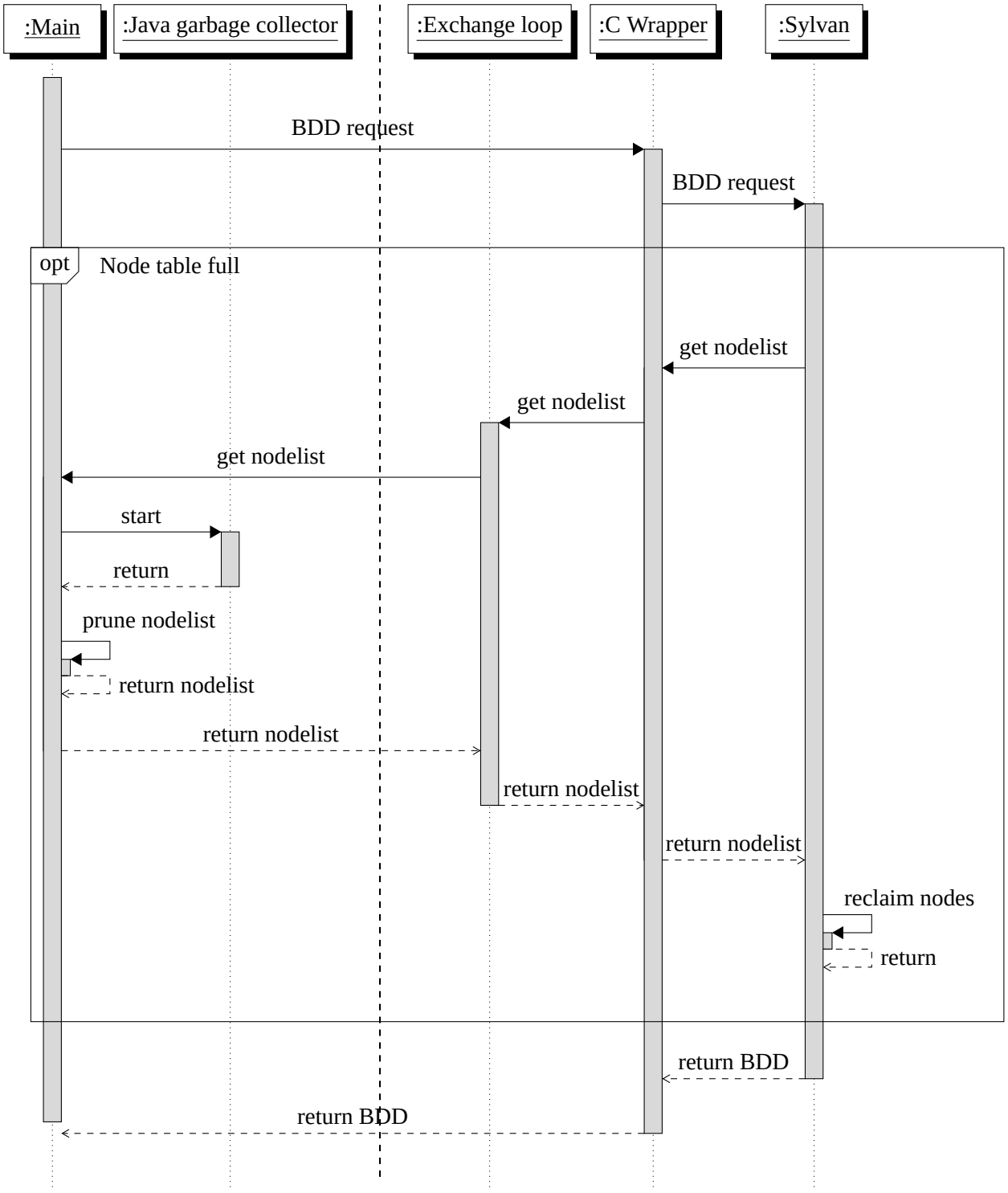


FIGURE 7.1 Sequence diagram of a single BDD operation. The vertical dashed line indicates the boundary between the Java native image (left) and other native components (right)

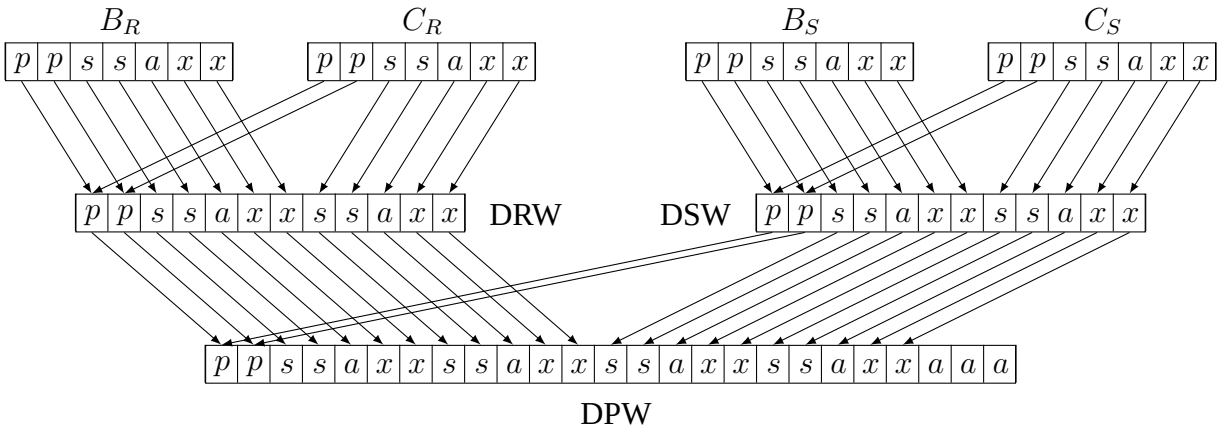


FIGURE 7.2 A visualization of the variable ordering for Example 7.1. Every bar visualizes the BDD ordering where the leftmost variable in the bar maps to the variable at the root of the BDD and the rightmost to that closest to the terminal nodes. The arrows indicate how the variables are reordered in each step of the construction. We use p, s, a and x for variables of the types "atomic proposition", "state", "acceptance set" and "successor state" respectively.

the BDD representing the product transition relation. The remaining variables of the DBWs and DCWs are then concatenated to obtain the product transition relation. The initial state formulas are concatenated similarly.

After the construction of the DRW, the DPW conversion is attempted. If successful, the DSW construction is skipped. Otherwise, the DSW is constructed similarly to the DRW and the DPW conversion is applied on the product of the DSW and DRW which is constructed just as how the DBWs and DCWs are combined. Finally, the acceptance sets of the DPW are appended.

Apart from the variable ordering, the implementation of the DPW construction and the SCC decomposition as discussed in Chapter 5.2 are identical to the theoretical presentation. We, therefore, do not discuss them further and proceed with an example on the variable ordering.

EXAMPLE 7.1 Suppose a DRW and a DSW are constructed for a specification consisting of two atomic propositions p_1 and p_2. Also suppose both the DRW and the DSW are constructed from a DCW and a DBW, each of which consists of 4 states and 1 acceptance set. We label the DBWs as B_R and B_S and the DCWs C_R and C_s where R (resp. S) refers to DRW (resp. DSW). If we use p for an atomic proposition variable, x to denote a successor state variable, s to denote a state variable and a for an acceptance variable, then we give the ordering of the variables during the construction as in Figure 7.2.

Each bar represents a BDD for the transition relation of the automaton. p_1 is at the root of each BDD, followed by p_2. In each DCW and DBW, we then have, in order, the state variables, acceptance set variables and next state variables. Since each automaton has 4 states, we need $\log(4) = 2$ state variables and 2 successor state variables.

For the product DRW and DSW, we concatenate all variables except for p_1 and p_2 which are still at the top of the BDD. We repeat this for the DPW. Finally, we assume the DPW is constructed with 3 acceptance sets, so we append 3 variables and obtain the final result. □

Perceptive readers might note that it is unnecessary to use 3 variables to encode 3 DPW acceptance sets since the acceptance sets of a DPW induce a partition and therefore only require $\lceil \log_2 n \rceil$ variables for n acceptance sets. Although certainly true from a theoretical perspective, it is more convenient to use n variables from an engineering perspective as this ensures no special treatment for DPW acceptance conditions is necessary. We leave this optimization as future work.

7.5 Generating a Mealy Machine

The final step in the construction is to convert the DPW to a game which is then solved and the result is converted to a Mealy machine. There are two important differences in the implementation from the theory presented in Chapter 6. First of all, we do not construct the game explicitly but instead, we run the distraction fix-point iteration algorithm directly on the DPW. In the next section, we will discuss how this is achieved.

The second difference is in the final product. In Chapter 6, we discussed the translation to a mathematical model of a Mealy machine. For Otus, we of course want something closer to reality. For that purpose, we produce a Mealy machine as an *and-inverter graph*, which we discuss in more detail in Section 7.5.2.

7.5.1 Strategy

Rather than constructing a parity game as a separate step, we inline this construction in the distraction fix-point algorithm. We do so by adapting Algorithm 7 to work on a DPW instead of a game. We use idea the as described in Section 6.1 to split Algorithm 7 such that we track system states and environment states separately. This is possible since we know that the sets of system states and environment states induce a bipartition on the game if constructed from a DPW. This means that all transitions in the DPW will go from the set of system states to the set of environment states or vice versa. Just like discussed in Section 6.1, we assume the environment's atomic propositions are part of the state variables as to "remember" the actions of the environment. However, instead of defining new state variables we simply use the existing variables for this. This means that these variables now not only represent an atomic proposition but also a state variable. We start by trivially lifting the definition of *wonBy* to a DPW and then we redefine *onestep*.

DEFINITION 7.2 *Given a symbolic DPW A, a propositional formula Z representing the set of distraction and a player a, the states estimated to be won by a are defined as:*

$$\mathit{wonBy}(A, Z, a) = \begin{cases} V_{\mathit{even}} \wedge \neg Z \vee V_{\mathit{odd}} \wedge Z & a = c \\ V_{\mathit{even}} \wedge Z \vee V_{\mathit{odd}} \wedge \neg Z & a = e \end{cases}$$

where V_{even} (resp. V_{odd}) represents the states of even (resp. odd) priority analogous to the definition for parity games.

DEFINITION 7.3 *Given a symbolic DPW $A = (\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \bigsqcup_{p=0}^{d} \mathcal{F}_p)$, the controlled atomic propositions $AP_c \subseteq AP$, two propositional formulas Z^c and Z^e representing the current set of distractions for player c and e respectively, a player a and a priority p, the new set of distractions for a is defined as:*

$$\mathit{onestep}(A, AP_c, Z^c, Z^e, a, p) = \begin{cases} \neg\exists_{\mathcal{Q}_\mathbf{x} \cup AP_c} \mathit{wonBy}(A, Z^e, c)[\boldsymbol{X}] \wedge \mathcal{R} & (a = c) \wedge \mathit{even}(p) \\ \exists_{\mathcal{Q}_\mathbf{x} \cup AP} \neg Z^c & (a = e) \wedge \mathit{even}(p) \\ \exists_{\mathcal{Q}_\mathbf{x} \cup AP_c} \mathit{wonBy}(A, Z^e, c)[\boldsymbol{X}] \wedge \mathcal{R} & (a = c) \wedge \mathit{odd}(p) \\ \neg\exists_{\mathcal{Q}_\mathbf{x} \cup AP} \neg Z^c & (a = e) \wedge \mathit{odd}(p) \end{cases}$$

We distinguish four cases by considering the parity of the provided priority and the player. The purpose of the priority is to indicate whether we want to compute the one-step-distractions for the even or the odd states and the purpose of the player is to select the owner of the considered states. For each case, we can compute the one-step-distractions separately.

Consider the first case: If the player is c then the one-step-distractions for even states are those states where c is forced to play to a state won by the environment. In other words, those states where there is no transition such that the system wins. Since the environment's atomic propositions are included in the states, this can be computed simply by taking the successor substitution of *wonBy* formula for player c, intersecting that with the transition formula of the automaton, and using an existential quantification such that we take those assignments where there is no satisfying assignment of successor variables and system atomic propositions. If instead the states currently considered are odd, then these states are distractions for e if there *is* a transition such that the system wins, thus we obtain the dual of the first case.

Now consider the second case, where we take the states controlled by e that are even. The one-step-distractions for c are then those states where the environment can make a move to a state won by itself. Recall now that the transitions of the environment are simply transitions that set the environment's atomic propositions in the state variables but leave the other state variables untouched. Thus, the environment's transitions are independent of the transition formula and we can simply compute the one-step-distractions for c as an existential quantification over the atomic propositions for the *wonBy* formula of the environment. Now since we are only considering even states, we can simplify *wonBy* to simply the complement of controller distractions. Again, this argument can be dualized to obtain the last case.

Using this redefinition, we now present the algorithm in Algorithm 9. The algorithm follows the same approach as Algorithm 7 but it keeps track of the environment states and the system states separately. Recall that we assign the lowest priority (and thus highest index) to the intermediate states, i.e. those owned by the system. This priority is, of course, implicit and thus we need to treat the system states specially. For example, lines 22 and 23 correspond with $V_{>p} \wedge \bigwedge_{i=0}^{d} \neg C_i$ in Algorithm 7. Furthermore, we apply similar reasoning as in *onestep* to simplify calls to *wonBy*. Other than that, the algorithm is identical to Algorithm 7 except that we now use the new definitions for *onestep* and *wonBy* to track system states and environment states separately.

Algorithm 9 `dpwdfi`$(A = (\mathcal{Q}, AP, \mathcal{I}, \mathcal{R}, \bigsqcup_{p=0}^{d} \mathcal{F}_p), AP_c)$

$Z^c, Z^e \leftarrow \bot$
$C^c_0, \ldots, C^c_d \leftarrow \bot$
$C^e_0, \ldots, C^e_d \leftarrow \bot$
$S^c, S^e \leftarrow \bot$
$p \leftarrow d$
while $p \geq 0$ **do**
  $a \leftarrow \begin{cases} c & p \mod 2 = 0 \\ e & \textit{otherwise} \end{cases}$
  $X^c \leftarrow \mathcal{F}_p \wedge \neg Z^c \wedge \bigwedge_{i=0}^{d} \neg C^c_i$
  $X^e \leftarrow \mathcal{F}_p \wedge \neg Z^e \wedge \bigwedge_{i=0}^{d} \neg C^e_i$
  $Z'^c \leftarrow X^c \wedge \textit{onestep}(G, Z^c, Z^e, c, p)$
  $Z'^e \leftarrow X^e \wedge \textit{onestep}(G, Z^c, Z^e, e, p)$
  $Z^c \leftarrow Z^c \vee Z'^c$
  $Z^e \leftarrow Z^e \vee Z'^e$
  $S^c \leftarrow S^c \wedge \neg X^c \vee X^c \wedge \mathcal{R} \wedge \textit{wonBy}(A, Z^e, c)[\mathbf{X}]$
  $S^e \leftarrow S^e \wedge \neg X^e \vee \exists_{\mathcal{Q}_\mathbf{x} \cup AP_c} X^e \wedge \neg Z^c$
  $V_{>p} \leftarrow \bigwedge_{i=p+1}^{d} \mathcal{F}_i$
  if $(Z'^c \equiv \bot) \wedge (Z'^e \equiv \bot)$ **then**
    $C^c_p \leftarrow \begin{cases} \bot & p = d \\ C^c_p & \textit{otherwise} \end{cases}$
    $C^e_p \leftarrow C^e_p \wedge \neg V_{>p}$
    $p \leftarrow p - 1$
  else
    $X^c \leftarrow \begin{cases} \bot & p = d \\ \bigwedge_{i=0}^{d} & \textit{otherwise} \end{cases}$
    $X^e \leftarrow V_{>p} \wedge \bigwedge_{i=0}^{d} \neg C_i$
    $W^c \leftarrow X^c \wedge \begin{cases} Z^c & a = c \\ \neg Z^c & \textit{otherwise} \end{cases}$
    $W^e \leftarrow \textit{wonBy}(G, Z, a)$
    $C^c_p \leftarrow C^c_p \vee X^c \wedge \neg W^c$
    $C^e_p \leftarrow C^e_p \vee X^e \wedge \neg W^e$
    $Z^c \leftarrow \neg W^c \wedge Z^c$
    $Z^e \leftarrow \neg W^e \wedge Z^e$
    $p \leftarrow d$
  end if
end while
$W^c \leftarrow \textit{wonBy}(G, Z^e, c)$
$W^e \leftarrow \textit{wonBy}(G, Z^c, e)$
return W_c, W_e, S_c, S_e

7.5.2 And-Inverter Graph

After having obtained the strategy, we apply the strategy determinization as discussed in Section 6.3. The implementation closely reflects the theory so we do not discuss this step further. Instead, we cover the step thereafter, namely how to convert the determinized strategy to a Mealy machine.

The approach for this highly depends on the desired output format. We chose the AIGER format [7] as this is the format used by the SYNTCOMP competition [41]. This format describes how to represent an *and-inverter graph* (AIG) with Mealy semantics. As the name suggests, and-inverter graphs are graphs consisting of logical AND and NOT gates. In particular, they represent circuits consisting of these gates, thus having a set of inputs and outputs. The AIGER format additionally describes latches explicitly such that our circuit does not need a clock circuit. The idea now is that the environment atomic propositions are each associated with an input. and the controller atomic propositions to an output. The latches are used to hold state information of the Mealy machine. The input value of a latch becomes the output value in the next iteration of the implicit clock. Finally, the Mealy machine is encoded in AND and NOT gates using these inputs, outputs and latches.

Like a BDD, an AIG is essentially just another representation of propositional formulas. It is therefore not hard to translate our strategy to an AIG. However, the AIG we want needs to have distinct inputs, outputs. It represents a boolean function that maps inputs and latch outputs to outputs and latch inputs. Our BDD currently encodes a boolean function that maps all the variables to a single truth value.

To convert the strategy BDD S to an AIG in the desired format, we extract a BDD for each of the outputs to the function (i.e. the next state variables and the controller atomic propositions). For each output o, we substitute $\top$ for o in S and project out the remaining input variables. Formally, for each $o \in \mathcal{Q}_{\mathbf{x}} \cup AP_c$, we obtain:

$$S_o = \exists_{(\mathcal{Q}_{\mathbf{x}} \cup AP_c) \setminus \{o\}} (S[o/\top])$$

Each S_o is a boolean function from the inputs to a truth value for output o. We now translate each S_o from a BDD to an AIG by traversing the nodes of the BDD. We associate the root of the BDD of S_o with the output o in the AIG. Then, for each visited node of the BDD, we associate the BDD variables to AIG inputs as follows: if the variable is a state variable, it is associated with a corresponding latch output and if it is an environment atomic proposition, it is associated with an input. Now recall that a BDD node x can be recursively interpreted as $x \wedge \mathit{high}(x) \vee \bar{x} \wedge \mathit{low}(x)$. In terms of AND and NOT gates, this is equivalent to $\neg(\neg(\bar{x} \wedge \mathit{low}(x)) \wedge \neg(x \wedge \mathit{high}(x)))$. Thus we recursively translate each BDD node to an and-inverter graph as shown in Figure 7.3.

To ensure the and-inverter graph is compact, we do not create new gates for each node. After all, if we encounter the same BDD node multiple times during our traversal, we can simply reuse the previously generated gate. We, therefore, maintain a mapping of BDD nodes to gates and share this mapping for all S_o. Furthermore, we also do not apply the full translation of Figure 7.3 if one or both of the recursive steps results in $\top$ or $\bot$, but rather we first simplify to obtain a

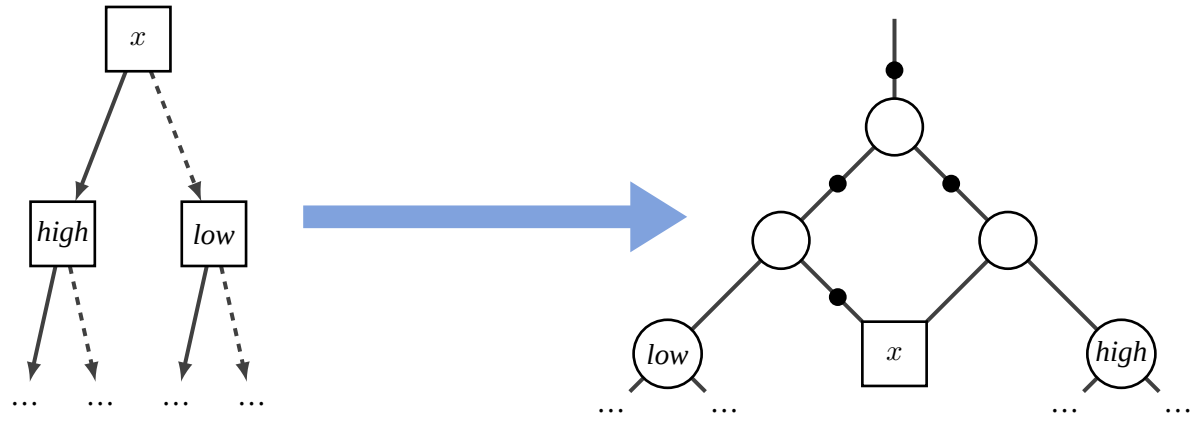


FIGURE 7.3 Conversion from a BDD (left) to an AIG (right) where the nodes labelled *high* and *low* represent the recursive applications of this conversion. Each white circular node in the AIG is an AND gate and each black node is a NOT gate. The rectangular node is the input and the dangling edge at the top is the output.

smaller circuit.

Finally, we finish with a small remark on the initial states. So far, we have only covered how we translate the strategy to an and-inverter graph, which means the and-inverter graph now encodes the transition relation of the Mealy machine but not the initial states. The AIGER format [7] is defined such that each latch is initialized to false, which means that we implicitly assumed that, in the DPW, $\mathcal{I} = \bot$. This is clearly not always the case so we need to "initialize" our Mealy machine correctly.

Recall that our DPW is deterministic and that there is therefore only a single satisfying assignment for $\mathcal{I}$. For every variable that is equal to false in this assignment, we need not do anything as the latch is correctly initialized. For those variables that are set to true, we add extra gates between the latch output and the connections to it. The gate simply copies the output of this "state" latch except in the initial state, in which case the gate always outputs true. To detect the initial state, we use a special latch that is connected to itself such that its output is false in the initial state and true thereafter. Suppose we call the output of this special latch L, and the output of the state latch S, then the gates between S and the rest of the circuit can be represented as $\neg(L \wedge \bar{S})$.

CHAPTER 8

Empirical Evaluation

This chapter presents the empirical evaluation of Otus, for which we have executed several benchmarks. We first cover some common methodology. Then we present and discuss the results of each benchmark. For those diagrams that rely on colours, we provide a tabular alternative in Appendix A.

8.1 Methodology

For the evaluation of Otus, we use a collection of specifications in the high-level TLSF format [40] that have been used in previous years of the SYNTCOMP synthesis competition.[1] We use Syfco [40] to pre-process these high-level specifications into the format as described in Section 7.1.

We execute several benchmarks as part of this evaluation. All benchmarks are run on a cluster of Dell PowerEdge M610 servers with two Xeon E5520 processors and each specification is assigned 8 cores and 56 GB memory. We use two configurations for Otus: One in which Sylvan is used as the primary BDD library, referred to as Otus-Sylvan and one where that is JBDD, referred to as Otus-JBDD. We say *primary* BDD library as JBDD is in any case used for constructing the explicit automata as discussed in Section 7.2. We evaluate Otus using both BDD libraries to draw a comparison between the libraries later.

We compare Otus against the state-of-the-art synthesis tool Strix [59]. Strix applies a construction very similar to ours. However, Strix does not apply the Δ_2-normalization but instead applies a best-effort decomposition like Morgenstern [70] and Sohail et al. [87] have done before.[2] Strix applies the construction explicitly and uses a latest appearance record to construct a DPW. It is an ideal comparison target because of its similar but explicit approach, and because it has won the SYNTCOMP reactive synthesis competitions of 2018, 2019 and 2020 [39, 41].[3]

For Strix, we use the winning configuration[4] of the 2020 SYNTCOMP synthesis competition [41], but we disable postprocessing using ABC [6]. ABC is a toolset for logical circuits that is used by Strix to minimize the circuit. For a fair computation time comparison with Otus, we

[1] Available on `https://github.com/meyerphi/syntcomp-reference`

[2] That is at the time of this evaluation. Strix is still in active development and is likely released with Δ_2-normalization soon.

[3] The results of 2021 are pending at the time of this thesis.

[4] We use `strix -f "$formula" --ins "$ins" --outs "$outs" --no-compress-circuit --auto -e pq -c`

Stage label	Description
DRW_DCW	Normalization and construction of the symbolic DCW automata that will form the DRW.
DRW_PRODUCT	Combining the DCW automata into a single DRW automaton.
DPW1	Attempting to construct the DPW from the DRW directly.
DSW_DCW	Normalization and construction of the symbolic DCW automata that will form the DSW.
DSW_PRODUCT	Combining the DCW automata into a single DSW automaton.
DRW_DSW_PRODUCT	Combining the DRW and DSW automata to obtain a new DRW on which an equivalent DSW acceptance condition exists.
DPW2	Constructing a DPW from the combination of the DRW and DSW.
DFI	Applying distraction fix-point iteration to solve the parity game.
SD	Modifying the strategy such that there always is a single output assignment for each input assignment.
AIG	Splitting the strategy into a BDD for each output variable and constructing an and-inverter graph.

TABLE 8.1 Stage labels and their descriptions.

disable this minimization in Strix. Instead, we run ABC externally (without time measurements) for both Otus and Strix to see how much minimization can be achieved.

Using this approach, we execute a number of benchmarks. We will now discuss each benchmark. For each, we discuss its purpose and some methodological aspects, and we present and discuss the results.

8.2 Exploratory Benchmark

An initial "exploratory" benchmark is performed to discover which specifications can be completed within five minutes. This benchmark is used to select the specifications to be used in the next benchmark. In total 421 realizable specifications and 157 unrealizable specifications are evaluated.

For each specification, we only collect whether the tool was able to successfully terminate within five minutes. The correctness of the result of the tool is not verified in this benchmark. Instead, those specifications that were completable within five minutes are verified in the next benchmark. Finally, for Otus, we additionally collect during which stage a timeout occurred as this can give a first impression on the bottleneck of the construction.

We label the stages of Otus as shown in Table 8.2. Note that not always all stages are visited. For example, for some specifications, the DPW can be constructed directly from the DRW in which case the DSW construction is skipped. Furthermore, specifications that are unrealizable will not enter the SD and AIG stage. The data for this benchmark is given in Appendix B.

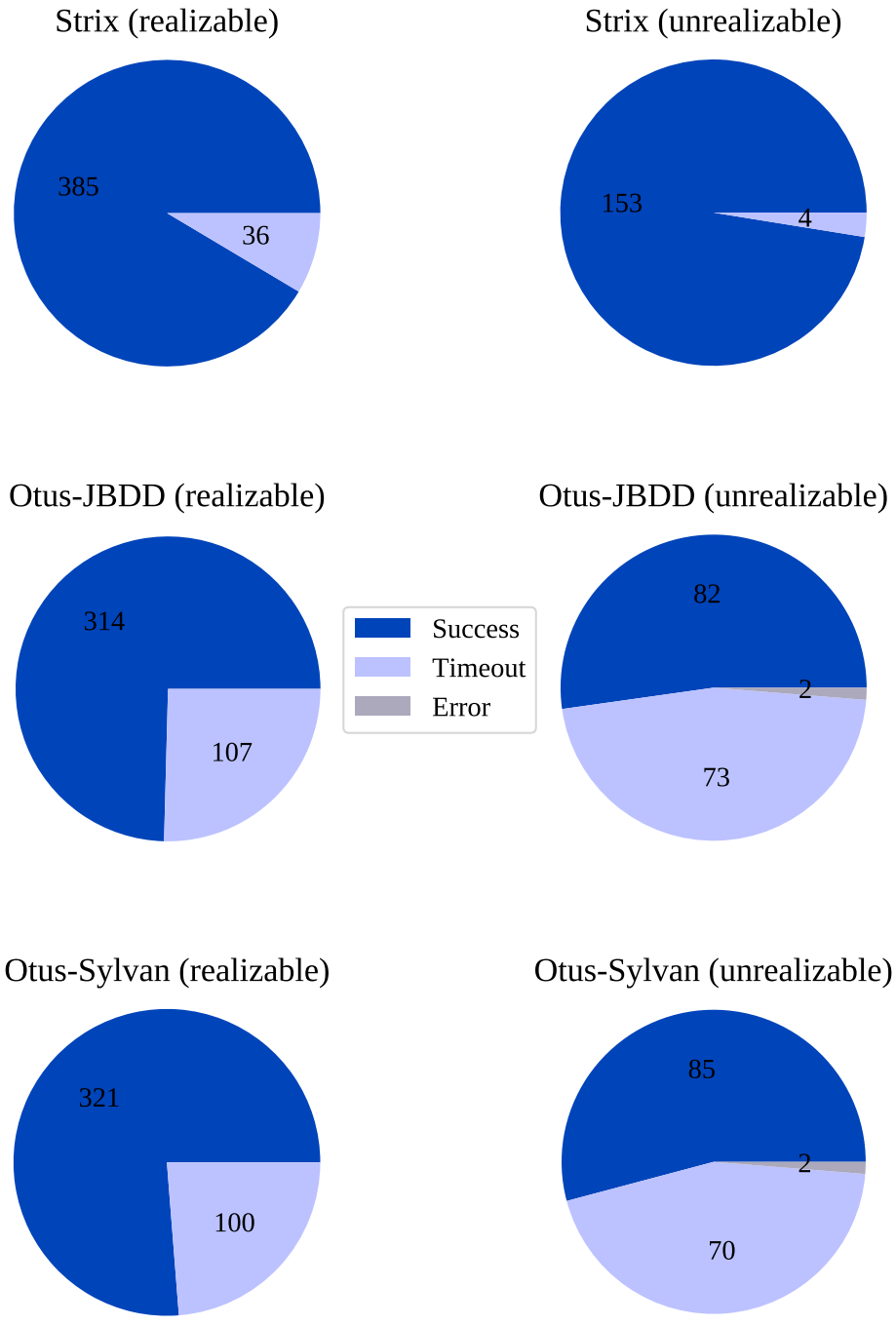


FIGURE 8.1 Number of specifications completed within 5 minutes. "Error" indicates that the specification could not be solved because of an error due to resource exhaustion, such as a full node table. A tabular representation can be found in Table A.1.

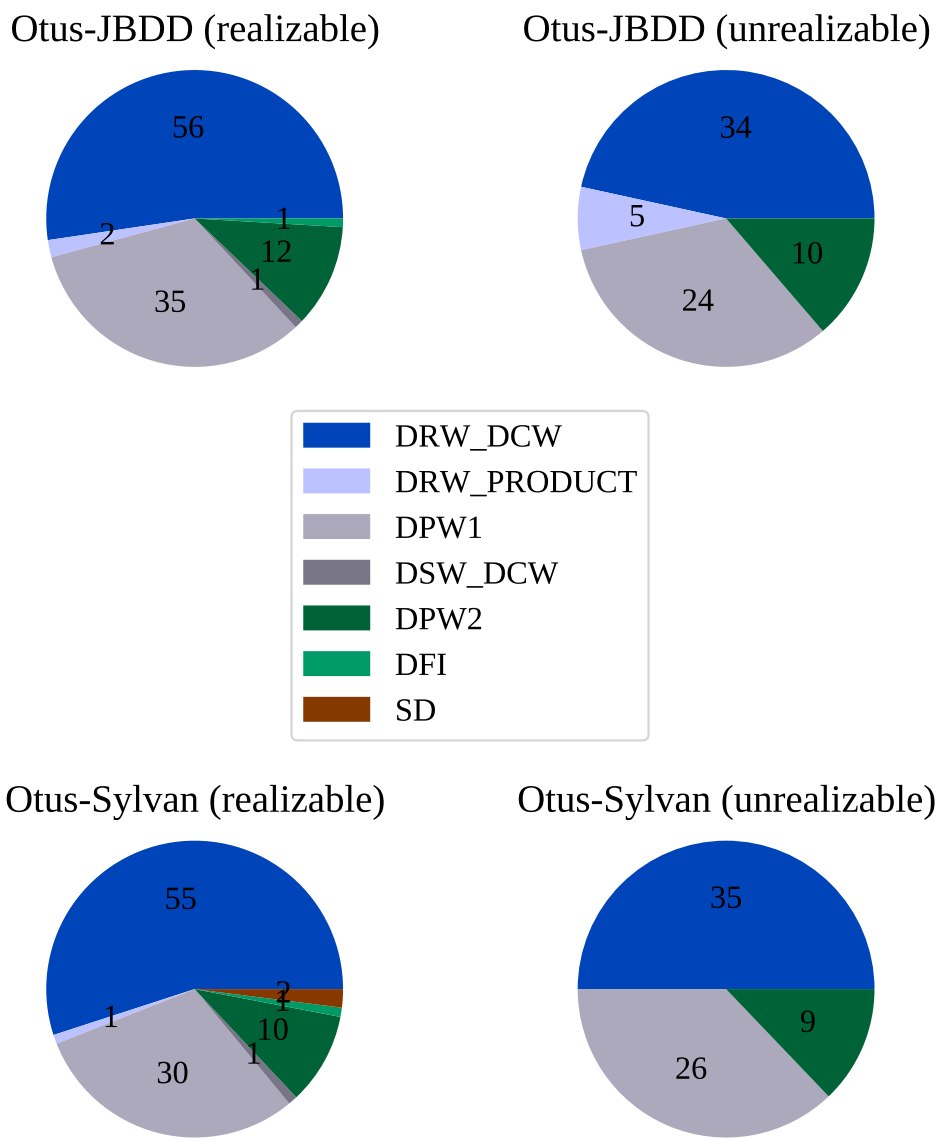


FIGURE 8.2 Stage at five minute timeout. A tabular representation can be found in Table A.2.

Figure 8.1 presents the number of specifications completed. We observe that Strix is able to complete significantly more specifications within 5 minutes than Otus-Sylvan and Otus-JBDD. This is especially true for the unrealizable specifications. We also observe that Otus-Sylvan can complete a total of ten specifications more than Otus-JBDD.

For those specifications that have timed out, we report on the stage during which the timeout occurred. Figure 8.2 presents the results. We observe that the timeout occurs most frequently in the first stage, in which the DCW and DBW automata are generated. We also note that it occurs very frequently in the DPW construction.

However, from these results, we cannot immediately conclude that the first stage of the construction is the bottleneck. It could be that many of the specifications are so difficult to solve that it is unreasonable to expect the first stage to complete within five minutes. Subsequent stages may still take much longer to complete than the first stage, given sufficient time.

Yet, the relatively high number of timeouts occurring in the `DPW1` and `DPW2` stages is remarkable. Although the DPW algorithm is polynomial, the runtime may be relatively high due to

its dependence on the SCC decomposition. Under poor circumstances, the SCC decomposition algorithm could force an explicit enumeration of the state space which nullifies the potential performance improvements from a symbolic encoding.

8.3 Evaluatory Benchmark

The previous benchmark has given an initial idea of which specifications are feasible to solve in 5 minutes. For each tool, we take those specifications solvable in 5 minutes and repeat the experiment 5 times to reduce the influence of external factors on the measurements. For Strix, we collect the total execution time for each specification and the number of used gates for each realizable specification as a measure of the circuit's size. For Otus, we additionally collect the duration of the stages of Table 8.2 for a more detailed execution analysis.

In this benchmark, we additionally verify the correctness of the results. We check whether realizability is correctly identified, and for the realizable specification, we also attempt to verify the generated controllers. For that purpose, we use a script[5] which conveniently combines a collection of tools that together allow for the verification of AIGER circuits against a TLSF specification using the state of the art symbolic model checker nuXmv [19]. We attempt the verification for at most three minutes, after which we move on to synthesizing the next specification.

The data from this benchmark can be found in Appendix C. For each specification, we have compared the average total execution time of Strix, Otus-JBDD and Otus-Sylvan over the 5 runs. We first present the results for these total execution times and then we present the detailed execution times of Otus-JBDD and Otus-Sylvan using the stages of Table 8.2.

8.3.1 Total Execution Time Comparison

Comparisons of the average total execution time over 5 runs for Strix, Otus-JBDD and Otus-Sylvan are presented in Figure 8.3 and Figure 8.4 for realizable and unrealizable specifications respectively. Because many of the execution times are concentrated in the initial second(s) of the five minute execution, we present each plots in three different magnification levels. Note that each plot only shows specifications where *both* tools solved the specification in under five minutes.

We observe that the total execution time when comparing Otus-JBDD and Otus-Sylvan to Strix varies substantially for realizable specifications. Extreme differences exist in favour of Strix as well as Otus-JBDD and Otus-Sylvan. These relative extremes are observed regardless of the observed absolute execution time. This result suggests that using a symbolic encoding can have a positive impact on the total execution time in some cases, yet harm others. However, it is not certain that the performance difference can be attributed to an (in)efficient variable ordering or whether the cause is in other characteristics of the construction. Further research through an explicit implementation of the construction of Otus could provide more insights into

[5] Available on `https://github.com/meyerphi/syntcomp-reference`

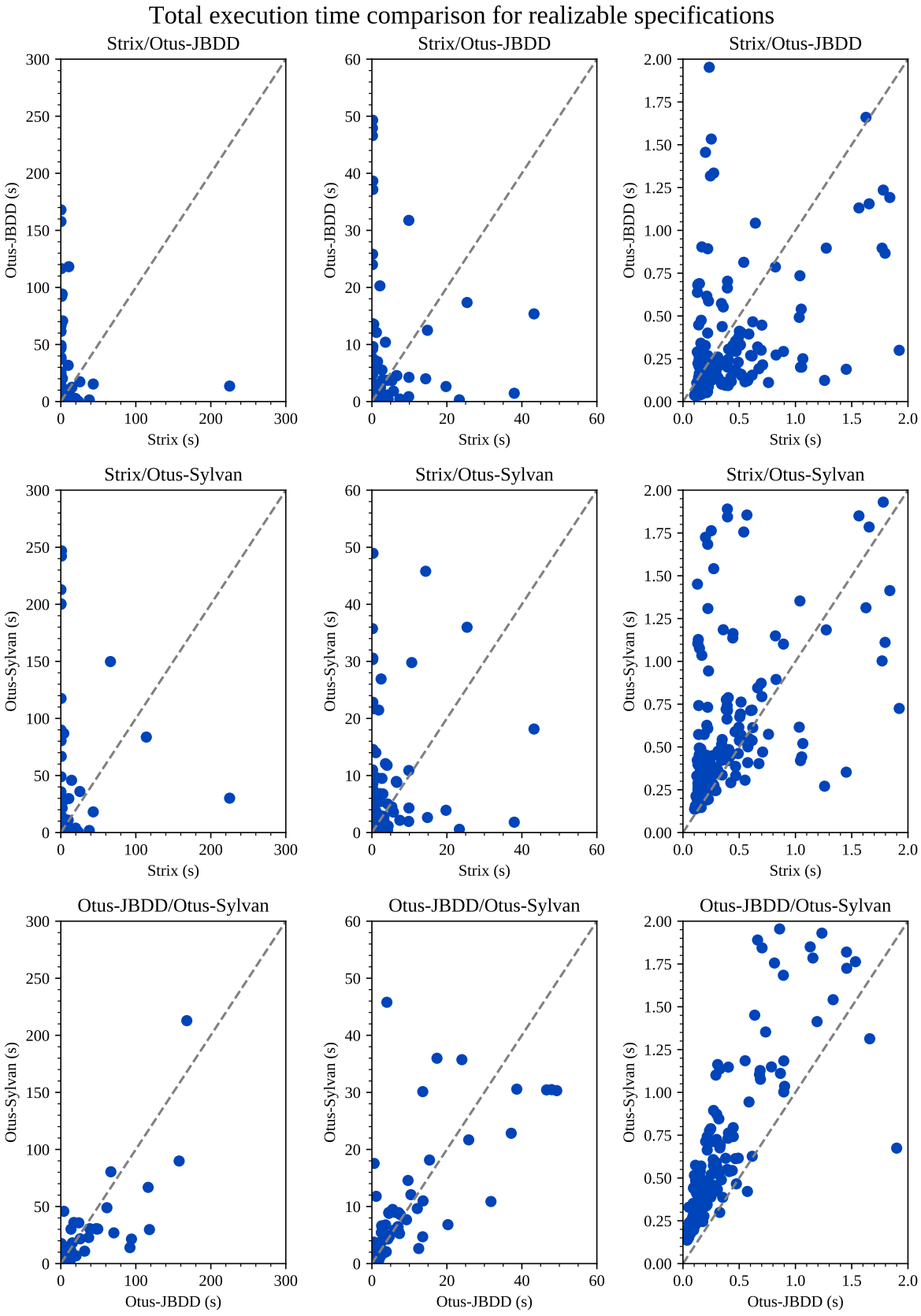


FIGURE 8.3 Comparison of the average total execution times over five runs of all realizable specifications solvable in five minutes.

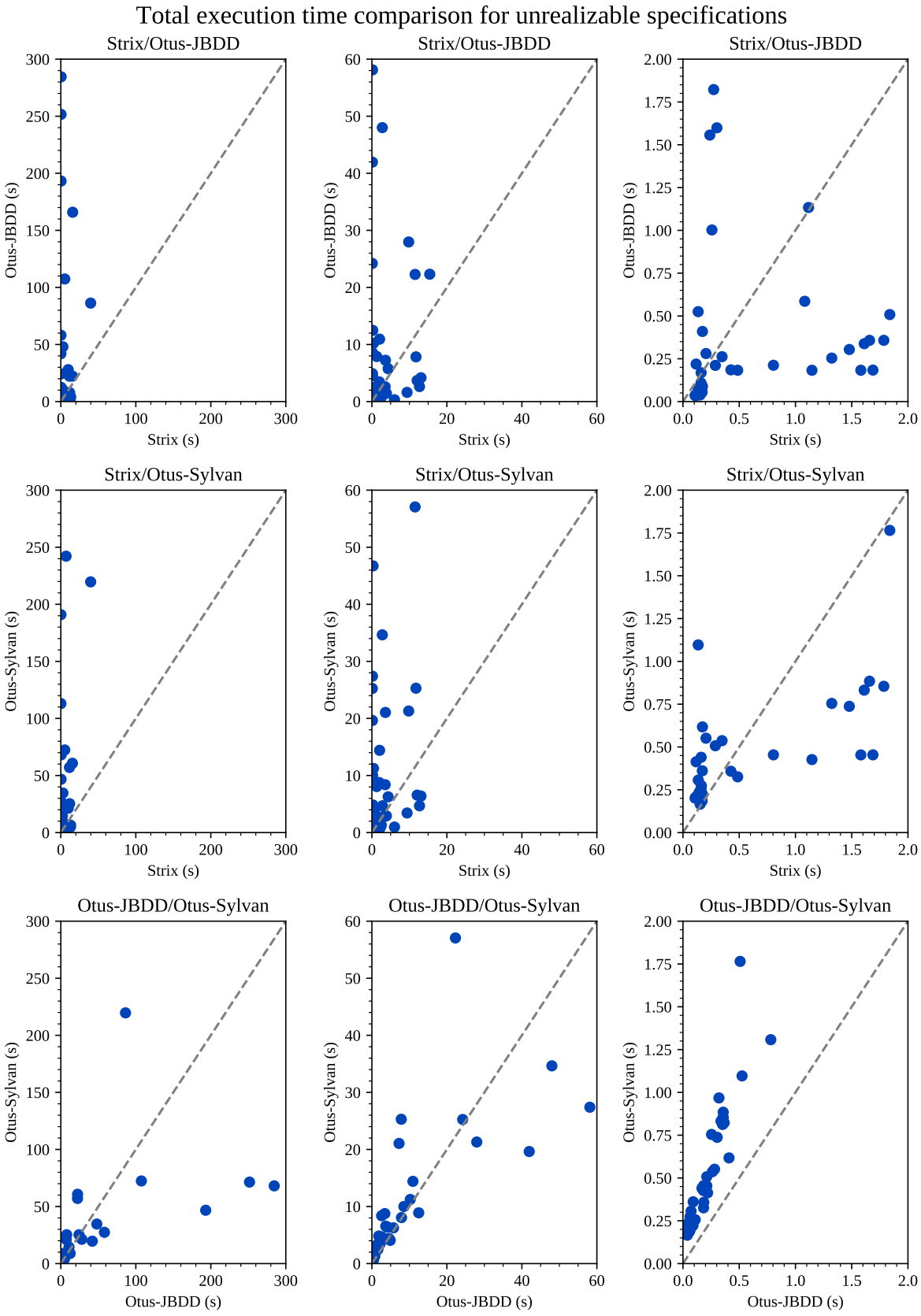


FIGURE 8.4 Comparison of the average total execution times over five runs of all unrealizable specifications solvable in five minutes.

the performance impact that the symbolic encoding has. Nevertheless, it shows that there are cases where our approach outperforms Strix.

We highlight those specifications where the differences in total execution times are the highest as these cases are interesting targets for further research and can act as inspiration for possible variable order improvements. We present the top and bottom ten specifications for the ratio of the total execution time of Otus-Sylvan over that of Strix in Table 8.2. We refrain from presenting a similar table for Otus-JBDD and Strix since the results are similar, and refer to Appendix C instead. We also refer to Appendix C for the equally interesting cases where only one of the tools has been able to solve the specification within 5 minutes.

Interesting to observe is that the lower ratios are much more extreme, indicating that in those cases Strix is in the order of hundred times faster than Otus-Sylvan whereas Otus-Sylvan is "only" at most 41 times faster than Strix. More research is needed into the specifics of these specifications to understand what causes the large differences to occur. It is interesting to see whether these specifications share some property that makes them particularly suited for synthesis using Strix or Otus.

In addition to the comparison against Strix, we also compared the total execution time of Otus-JBDD and Otus-Sylvan and the difference is expectedly more consistent. We observe that Otus-Sylvan outperforms Otus-JBDD for almost all specifications where both need at least ten seconds. This is as expected since Sylvan is a multi-threaded BDD library engineered for high performance. For the simpler specifications (where both need less than two seconds), we observe that Otus-JBDD is consistently faster than Otus-Sylvan. This is also expectable since the concurrent design of Sylvan naturally comes with some overhead which can become notable for the simpler specifications.

Although this is true for most of the specifications, there are some outliers. For example, the specification `TwoCountersDisButA9` takes, on average, 220 seconds to solve using Otus-Sylvan yet only takes 86 seconds to solve using Otus-JBDD. Since the constructions used in Otus-Sylvan and Otus-JBDD are equal, the observed difference can only be attributed to either the BDD libraries themselves or the interface that connects the BDD library to Owl. Part of this interface is the garbage collection technique as discussed in Section 7.2. If Sylvan is frequently collecting garbage, this could cause a bottleneck and explain the observation. However, after rerunning this specification with garbage collection logging enabled, we conclude that this is not the case since the garbage collection via Sylvan is not triggered. Therefore, further research is desirable to understand what causes these differences to occur.

8.3.2 Detailed Execution Time Analysis

To get a better understanding of which parts of the construction are the most time consuming, we have measured the average time spent in all stages of Table 8.2 for the specifications selected for the evaluatory benchmark. We have studied the results for Otus-Sylvan as well as Otus-JBDD but since the results for Otus-JBDD are expectedly very similar, we do not present them. We grouped the specifications by their realizability and by the condition that a DSW automaton is constructed since the stages visited during the construction depend on these factors. We

Specification	Time Otus-Sylvan (s)	Time Strix (s)	Ratio
collector_v1_5.tlsf	0.57	23.33	41.21413
amba_decomposed_lock_10.tlsf	1.82	37.97	20.86229
amba_decomposed_lock_6.tlsf	0.39	4.16	10.64465
LedMatrix.tlsf	30.13	225.08	7.47000
ltl2dba_beta_5.tlsf	2.64	14.82	5.61316
amba_decomposed_lock_8.tlsf	0.52	2.74	5.25470
tictactoe.tlsf	3.91	19.76	5.06053
amba_decomposed_encode_10.tlsf	1.95	9.81	5.02077
lilydemo22.tlsf	0.27	1.26	4.64845
collector_v1_4.tlsf	0.35	1.45	4.11395
…	…	…	…
detector_5.tlsf	30.43	0.15	0.00478
collector_v3_7.tlsf	80.42	0.35	0.00436
ltl2dba_C1_6.tlsf	35.73	0.15	0.00434
escalator_smart.tlsf	246.85	0.93	0.00378
EscalatorSmart.tlsf	242.23	0.81	0.00335
prioritized_arbiter_6.tlsf	117.53	0.32	0.00268
ltl2dpa03.tlsf	89.87	0.17	0.00195
ltl2dba_C2_6.tlsf	200.47	0.16	0.00079
detector_6.tlsf	200.01	0.15	0.00077
ltl2dba_C1_7.tlsf	212.84	0.15	0.00069

TABLE 8.2 The ten specifications with the highest and lowest total execution time ratio of Otus-Sylvan over Strix. Execution times are presented in seconds and are rounded to two decimals. Ratios are computed using the unrounded execution times.

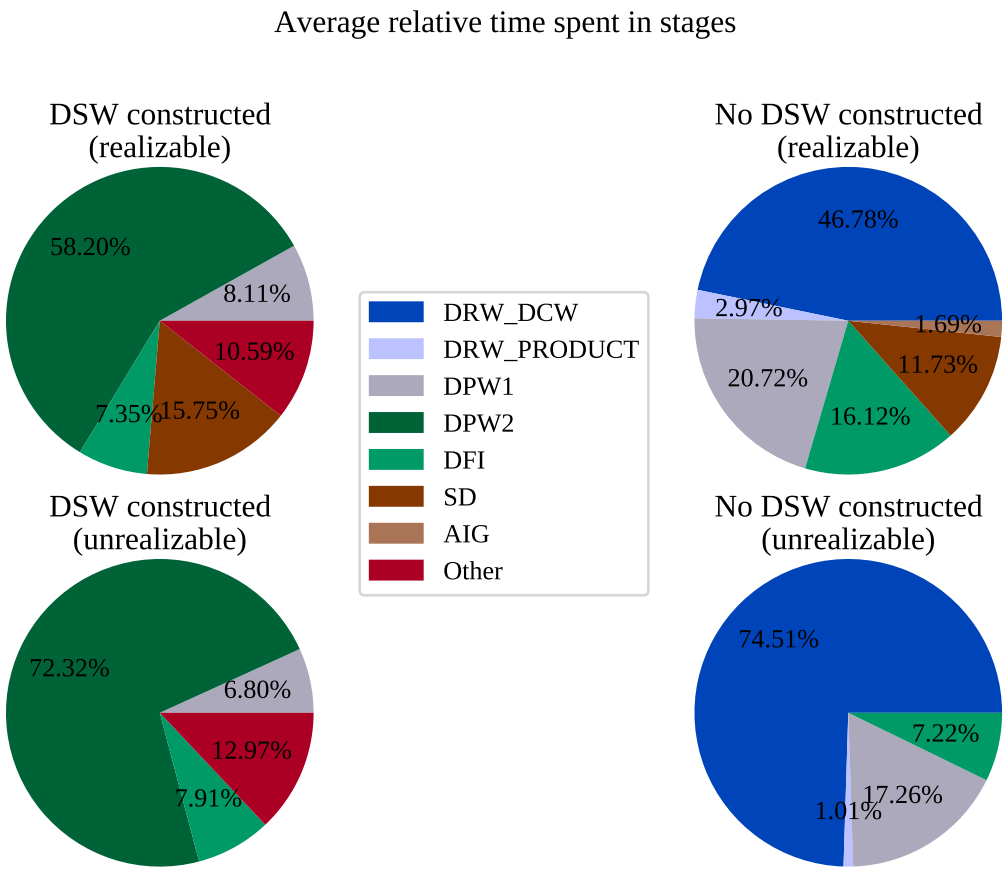


FIGURE 8.5 Average relative time division in Otus-Sylvan over all specifications in the evaluatory benchmark where each slice is computed as the average fraction of time spent in a stage for each specification over 1. A tabular representation can be found in Table A.3.

aggregate the results in two ways.

Figure 8.5 displays the average relative time division where all specifications contribute equally. We compute the average relative time division for each specification independently, resulting in a set of fractions corresponding to the stages that together sum to 1 for each specification. The average of all fractions over one stage forms a slice in the diagram.

Additionally, Figure 8.6 displays the average relative time division weighted according to the total duration of each specification. One slice corresponds to the total duration spent in one stage over all specification.

Immediately noticeable is the large fraction of time spent in the second DPW construction if the first construction was unsuccessful. On average, this construction takes approximately $7\times$ longer than the first DPW construction for realizable specifications and $11\times$ longer for unrealizable specifications. We also notice that in those cases where no DSW is constructed, the majority of the time is spent normalizing the formula and constructing the explicit automata.

The relatively high amount of time spent in the DPW construction reinforces the results of the exploratory benchmark, in which we noticed a remarkably high number of timeouts in the DPW stages. As discussed before, the SCC algorithm in the DPW construction likely causes a (partially) explicit state-space exploration which we can expect to have a significant detrimental

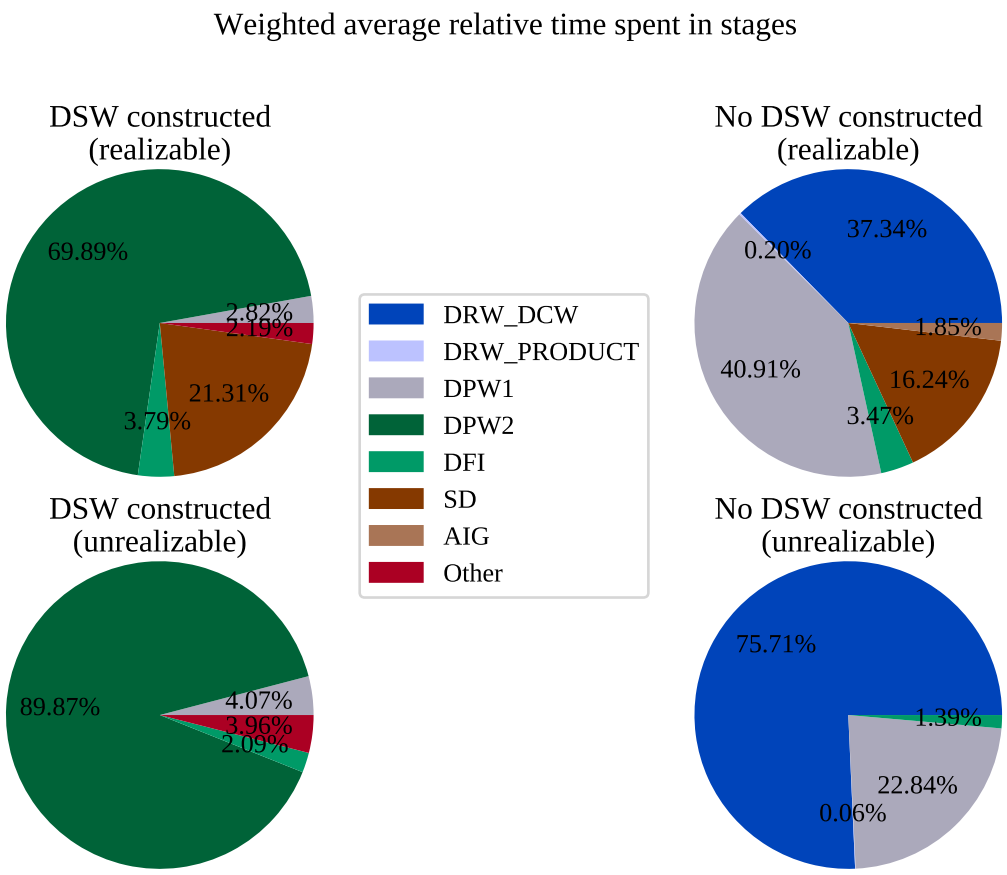


FIGURE 8.6 Weighted average relative time division in Otus-Sylvan over all specifications in the evaluatory benchmark where each slice is computed as a fraction of the combined time spent in one stage for all specifications over the total combined time. A tabular representation can be found in Table A.4.

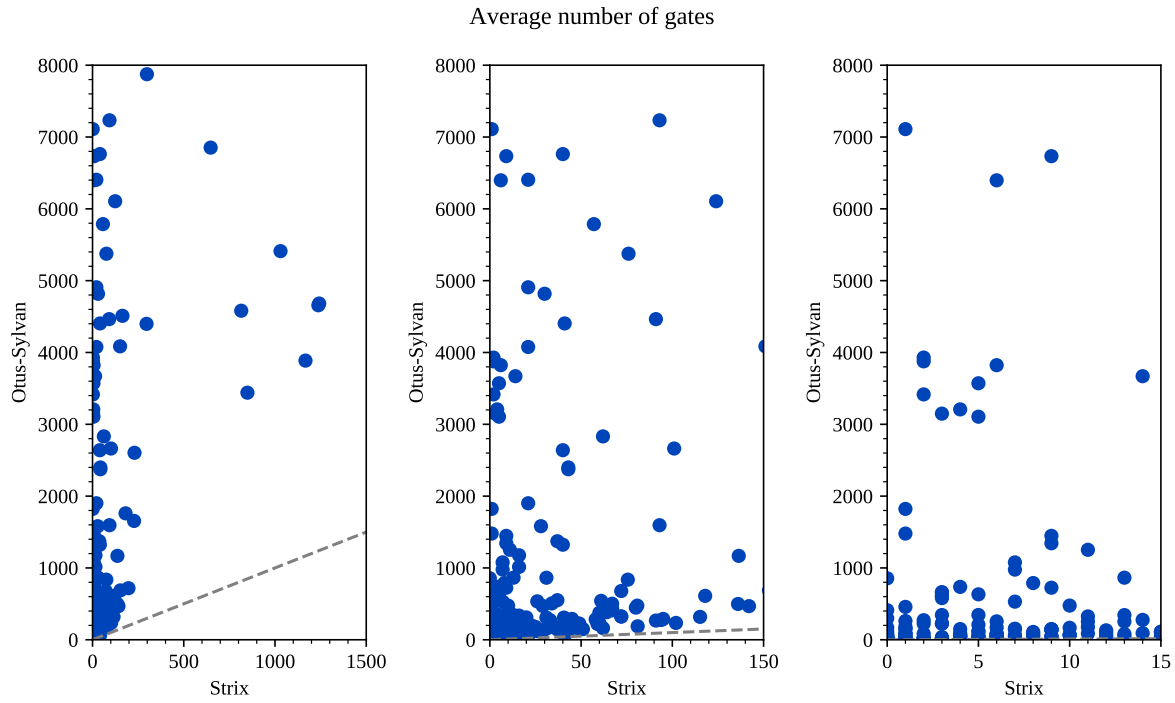


FIGURE 8.7 Average number of gates in the controllers synthesized by Otus-Sylvan and Strix before ABC at three different magnification levels

effect on the performance.

One might now think that this explains the poor execution time ratios observed for some specifications such as the bottom ten specifications of Table 8.2. Indeed, this is likely to play a role, but it cannot explain all observations. For example, among those bottom ten specifications are three specifications that have skipped the `DPW2` stage. Further research is therefore still needed.

8.3.3 Controller Quality

Obtaining controllers of good quality was out of the scope of this research. Nevertheless, we evaluated the quality of the controllers by their circuit size and compared the results for Strix and Otus-Sylvan. In its default configuration, Strix uses ABC [6] to minimize the circuits. This has been disabled since this would lead to an unfair total execution time comparison. Instead, ABC was executed externally for both Strix and Otus. We present the results before ABC in Figure 8.7 and after ABC in Figure 8.8, in which we use three different magnification levels for the x-axis.

It is clear that the quality of the controllers generated by the construction is subpar. All specifications yield a smaller circuit when generated by Strix. Furthermore, circuits generated by Otus-Sylvan are commonly in the order of a thousand times larger than those generated by Strix. On one hand, this can be explained by the fact that the implementation is not engineered to produce controllers of good quality. On the other hand, we believe this is also a result of using symbolic algorithms that operate on the entire state space, versus an explicit algorithm

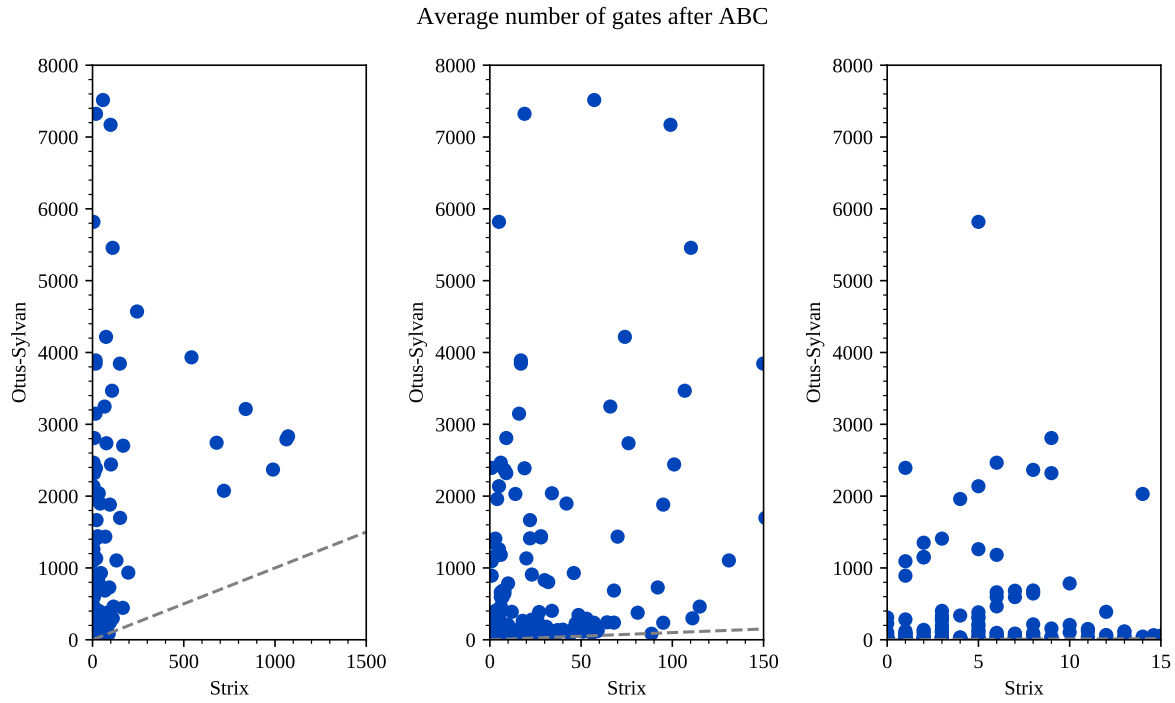


FIGURE 8.8 Average number of gates in the controllers synthesized by Otus-Sylvan and Strix after ABC at three different magnification levels

that often only partially needs to explore the state space.

We observe that, on average, ABC can reduce the circuits of Strix by 17% and of Otus-Sylvan by 47%. Furthermore, we observe that even after using ABC, the circuits generated by Strix are on average 15 times smaller than those generated by Otus-Sylvan. This indicates that the circuits generated by Otus-Sylvan still contain many redundant gates and that the quality can thus still be significantly improved through intermediate minimizations.

8.4 Parameterized Specification Benchmark

Many of the specifications in the benchmark set are parameterized specifications. It is interesting to see whether there are any parameterized specifications for which the total execution time increases differently for Strix and Otus. This could help identify classes of specifications for which one of the tools performs particularly well.

For this purpose, we select several parameterized specifications with a variable amount of parameters. Each parameterized specification is allocated 8 hours of execution time in total. In these 8 hours, we start with the smallest specification (the lowest parameter) and increase the parameter after each is completed until we either complete the specification for all parameters or we reach the 8-hour timeout. Results are not verified since it is expected that the generated circuits are too large to verify in a reasonable amount of time. Furthermore, this experiment is only run once and only for Strix and Otus-Sylvan because of the exorbitant computation power required.

We present the results of the parameterized specification benchmark using bar charts that

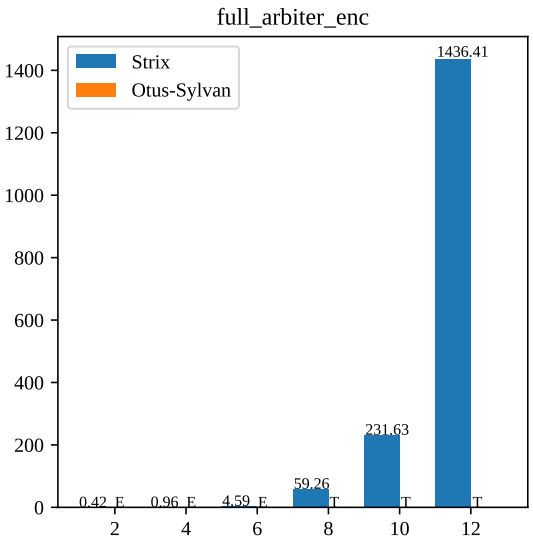


FIGURE 8.9 Execution time comparison for `full_arbiter_enc` with parameters on x-axis, and time in seconds on y-axis.

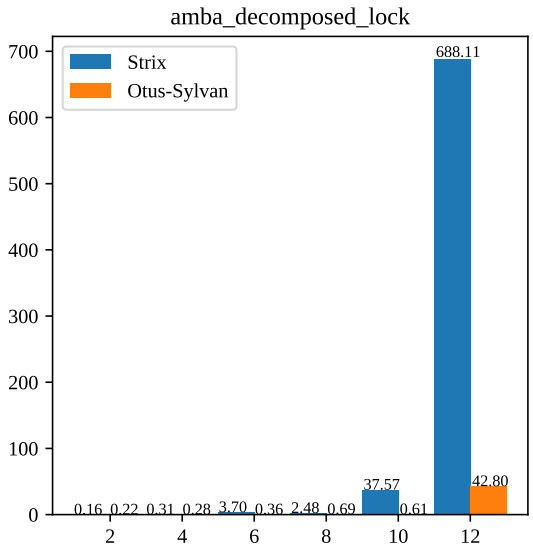


FIGURE 8.10 Execution time comparison for `amba_decomposed_lock` with parameters on x-axis, and time in seconds on y-axis.

visualize the execution time against the used parameter for Otus-Sylvan and Strix. In this section, we highlight the most interesting results. The results for all parameterized specifications can be found in Appendix D.

In all bar charts, we annotate the bars with the execution time. If an execution timed out or was not started within the allocated 8 hours, this is indicated by `T`. If execution failed due to resource exhaustion, such as a lace task queue overflow or a full BDD node table, we use `E`.

In general, we often observe that Strix performs much better than Otus-Sylvan. For example, Figure 8.9 shows a specification where Strix is able to finish all specifications within the 8 hours, whereas Otus-Sylvan cannot even complete the first one. There is an interesting exception, however. Figure 8.10 shows an example where Otus-Sylvan clearly outperforms Strix. Additionally, there are more parameterized specifications (see Appendix D) where Otus-Sylvan outperforms Strix but these are specifications where both tools finish in a few seconds which makes them less interesting.

These results raise the question of whether `amba_decomposed_lock` is an easy specification for Otus-Sylvan or a hard specification for Strix. In any case, a more detailed analysis of this specification can help understand and improve the execution time of both Otus-Sylvan and Strix. Similarly, a detailed execution analysis of Otus for `full_arbiter_enc` can help understand what makes this specification particularly difficult.

8.5 Sylvan Benchmark

It is interesting to see whether any performance gain in the evaluatory benchmark from using Sylvan in favour of JBDD can be attributed to the parallelization introduced by Sylvan. For that purpose, we select the 5 specifications with the greatest performance improvement and

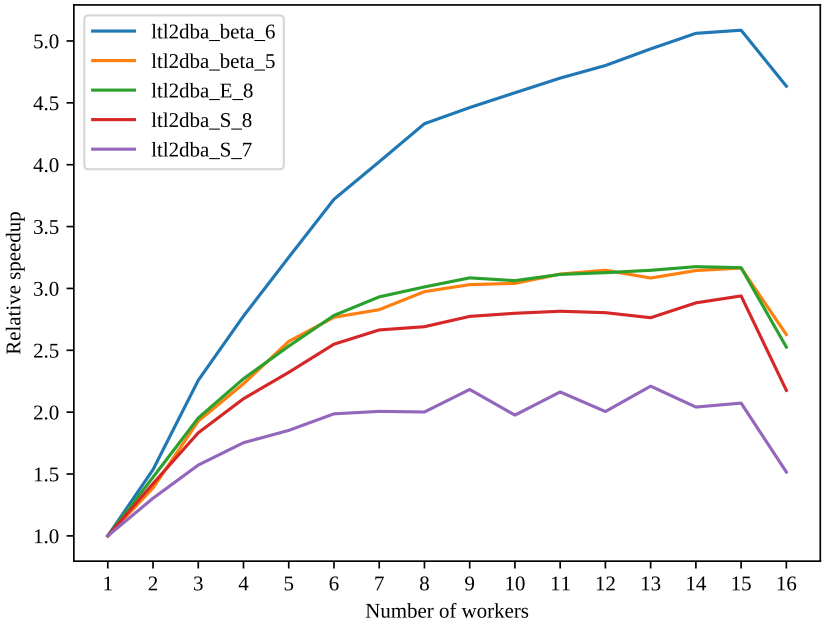


FIGURE 8.11 Speedup of Otus-Sylvan for the top 5 specifications with the largest execution time difference between Otus-JBDD and Otus-Sylvan. A tabular representation can be found in Table A.5.

see if the speedup increases as we increase the number of workers.

We assign all 16 cores to the benchmark and vary the number of workers from 1 to 16. We take the average of the total execution time over 5 runs thus totalling 80 runs for each specification. If the extra parallelization is indeed improving the performance, we expect to see the speedup increase as we increase the number of workers. Since there are also other threads running in the application, we expect that the performance increases less or decreases as we reach the largest number of workers. The benchmark results are given in Figure 8.11.

We can clearly see that the extra parallelization has a significant impact on the performance of the construction. There are limitations, however, as we see that the performance improvements stagnate as we add more workers. Furthermore, we see a slight performance decline when assigning 16 workers. This is not surprising considering that the machine has 16 cores and the application then uses more than 16 threads. These results suggest that the construction could benefit from additional parallelization apart from the BDD operations, such as parallelization of the DCW and DBW constructions. Further research is needed to find the balance between the number of worker threads, which process the BDD operations, and the number of other threads, which schedule the BDD operations.

CHAPTER 9

Discussion

The results shown in the previous chapter are promising, showing examples where Otus outperforms Strix. However, there are still many optimizations possible, some of which we have briefly covered in previous chapters. We discuss these optimizations here together with other suggestions that are worth exploring further.

9.1 Explicit Automata Construction

We observed that a large portion of the execution time is in the explicit automata construction. It is therefore natural to consider optimizations in this stage. The main reason for using explicit automata for this stage is for simplicity as the construction was already implemented in Owl [49]. It is still interesting to see what the effect of a symbolic approach would be. In Section 3.2 we discussed how both Sohail et al. [87] and Morgenstern [70] have investigated a similar approach to synthesis, where Sohail et al. have chosen for an explicit construction and Morgenstern for a symbolic one. Unfortunately, the two approaches have not been compared and thus it is still unknown whether a symbolic construction for the DBWs and DCWs is better.

For this reason, we could extend Otus with a symbolic construction and compare it against our current approach. A translation of Σ_2-formulas to symbolic NCWs is presented by Schneider [85]. Using the symbolic breakpoint construction of Morgenstern et al. [73], we can then symbolically convert these NCWs to DCWs as done by Morgenstern [70].

Unfortunately, the symbolic breakpoint construction requires an explicit state space iteration of the NCW and thus the compression effect of a symbolic encoding is partially nullified. We, therefore, do not expect a large performance improvement from using this symbolic approach, but to our knowledge, there is no symbolic determinization algorithm that has no explicit iteration of the automaton's states. Further research into symbolic determinization is therefore desired.

With the absence of such a determinization algorithm, we can also consider optimizing our current approach. For example, it would be interesting to see what the effect would be of increasing the number of constructed DBW and DCW automata. After normalization, it may be possible to split the LTL formulas even further such that we construct more DBW and DCW automata but each of them will be even smaller.

Finally, we could use Sylvan for this construction instead of JBDD. Currently, we always

use JBDD for this construction. The main reason is that this is easier from an engineering perspective. Nevertheless, it is of course possible to use Sylvan for this construction as well. It is not clear yet whether this will yield an improved performance since the BDDs used in this construction are smaller than those in subsequent steps. We have seen in the previous chapter that JBDD tends to perform better when the specification is easy to synthesize. Still, Sylvan appears to scale much better so further experimentation is needed to see the effect of using Sylvan for this construction.

9.2 DPW Construction

As seen in the previous chapter, the DPW construction constitutes a considerable portion of the total execution time. It is therefore natural to focus our optimizations efforts on this part of the construction. In particular, we suggest two approaches.

First, since we observed that the DPW construction on the DRW and DSW product automaton is considerably more expensive than the DPW construction on the DRW alone, we can consider reducing the number of times we have to construct a DPW on the product automaton. We have already seen that in the majority of the cases, the initial DPW construction using only the DRW is sufficient. If it is not, we immediately proceed with constructing the DRW and DSW product automaton.

Instead of immediately constructing the DPW from the DRW and DSW product, we can also attempt the DPW construction on the DSW. Perhaps many of the cases where the construction fails on the DRW would succeed on the DSW. We could even construct the DPW from the DRW and the DSW in parallel.

Next, instead of reconsidering our application of the DPW algorithm, we can also revise the design of the algorithm itself. As already briefly hinted at in the previous chapter, the SCC decomposition algorithm used in this construction may enforce a partially explicit state space enumeration if the automaton contains very many relatively small SCCs. This is because the SCC decomposition algorithm explicitly enumerates every SCC. It is not clear whether this is actually a problem in practice as we have not measured the sizes of the SCCs.

In addition, the relatively poor performance could also be explained by an imbalance in the size of the parity condition compared to the SCC sizes. Observe that in each recursive step of Algorithm 5, we reduce the size of the parity condition by one and compute the SCC decomposition of the current SCC without the hopeless states. Given a sufficiently large parity condition, the algorithm will run a nested SCC decomposition eventually enumerating every state as its own SCC. This nullifies the benefits of a symbolic encoding. Again, further research is needed to determine if this occurs in practice.

9.3 Amount of BDD Variables

The amount of variables used in a BDD determines its size. Reducing the number of variables decreases the number of nodes in the BDD which can increase the amount of information

each BDD node contains. This effectively compresses the BDD which will make operations on it more efficient. Decreasing the number of variables, in general, is not trivial, but we identify a few options.

A quick way to reduce the number of variables is by revising the encoding of parity sets in the DPW. As discussed in Section 7.4, the number of variables used for the parity sets is currently linear in the number of parity sets whereas it can theoretically be logarithmic. Although this will only slightly reduce the number of variables, it can have a large impact on performance since the number of BDD nodes is at most exponential in the number of variables.

Another approach that could significantly reduce the number of variables is the removal of acceptance set variables that are no longer used in the acceptance condition. Currently, acceptance set variables are never removed. For example, the product automaton of the DRW and the DSW still contains the DSW acceptance set variables. Furthermore, the DPW still contains the acceptance set variables of the DRW and the DSW. Although it would seem like one could simply project the BDDs to remove these variables, this approach does not work. Unfortunately, we lack a sufficiently small counter-example to be able to provide a formal argument, but we have encountered many large ones in practice. We expect that the exact reason is due to the structural changes to the automaton that occur by the removal of these variables and believe that it should be possible to find a small counter-example.

Perhaps the problem can be resolved by using edge-based acceptance sets instead of state-based acceptance sets. Removal of acceptance sets has a different effect on the structure of the automaton if the acceptance sets are on the edges instead of on the states. Although the symbolic encoding of the automata remains the same when using edge-based acceptance sets, the interpretation changes. The algorithms used in the construction will therefore all need to be adapted to use edge-based acceptance sets. However, it is not clear whether this will in fact resolve the before-mentioned problem.

9.4 Variable Ordering

In this implementation, we have chosen for a very simple variable ordering as discussed in Section 7.4. Different variable orderings yield different compression ratios as outlined in Section 5.1. Experimentation using different variable orderings can help optimize the construction.

For example, we can change the order of the variable categories and see which works best. Furthermore, we can choose a completely different strategy that does not assign ranges to categories of variables but instead interleaves variables of different categories. We can also choose to combine the BDDs of automata in the product construction differently. A particularly interesting variable ordering that is worth investigating is a variable ordering that orders input variables before output variables. This prevents the need for a variable reordering as part of determinizing the strategy as discussed in Section 6.3.

In addition, we can also reconsider the strategy used for encoding the explicit automaton. For atomic proposition variables and acceptance set variables, we cannot think of a better strategy than the current one because of its simplicity. However, for the state and next state variables,

we think it is possible to use a different encoding. Currently, we simply assign an integer to every explicit state and use its binary representation as the encoding. This encoding is very generic. Instead of this encoding, a more specific encoding that uses the structural properties of the automata-based constructions such as the AWW translation and the breakpoint construction could be better. Although it is not clear yet how this would work, we believe that this additional information can be used in some way to create a better encoding strategy.

Finally, finding good variable orderings for BDDs is an actively researched field and dedicated algorithms exist to find a good static ordering [65]. Instead of relying on a static order, it is also possible to use a dynamic ordering, i.e. an ordering that changes in-between operations [83]. Applications of these techniques could improve the performance of the construction significantly.

9.5 Parity Game Algorithms

It is known that solving parity games is in the complexity class UP ∩ co-UP [44], which is a class that is contained in NP ∩ co-NP [31]. Since many problems in NP ∩ co-NP have eventually been shown to be in P, it is widely believed that a polynomial-time algorithm for solving parity games must exist. Currently, the best-known solutions are quasi-polynomial [18, 34, 46] but a polynomial algorithm is still not found.

In Otus, we use distraction fix-point iteration [94] which is exponential but also simple and easy to implement symbolically [58]. Since finding a solution for the parity game constitutes only about 3% of the total execution time, we believe distraction fix-point iteration is sufficient for our purpose. However, we want to emphasize that parity games are still actively researched and so we can expect that improved algorithms will appear in the future. Using a different algorithm for solving the parity game is currently not of the highest priority, but this may change if further improvements are made to the other parts of the construction causing the relative time spent on solving the game to increase.

9.6 Controller Quality

Controller quality has not been of much concern in the development of Otus. We have seen that the quality of the controllers created by Otus is currently low. We think the quality can be improved in several ways.

First, we can revise the strategy determinization such that it selects a strategy that will result in a high-quality controller, instead of any arbitrary strategy. The simplest approach would be to consider all strategies and selecting the one that results in the smallest BDD, however, this will hurt the performance. Perhaps we can instead devise a heuristic that selects a strategy that results in a small circuit.

The controller quality can also be improved by finding ways to reduce the sizes of the automata. This can be achieved, for example, by identifying and removing (groups of) states that are *bisimilar*, i.e. that have the same behaviour [66]. Symbolic bisimulation minimization ap-

proaches exist that are interesting for further research [95]. This could be applied at various stages in the construction, such as when the explicit automata are converted to symbolic automata, or after one or several of the product constructions. It is also interesting to see what effect these minimizations have on the execution time.

Finally, improving the variable ordering as discussed in Section 9.4 is another way to reduce the size of the controller. An improved variable ordering ensures that fewer BDD nodes are needed, which then also means that we need fewer gates in the AIG.

9.7 Empirical Evaluation

The empirical evaluation has given rise to many new questions. In particular, it would be interesting to see whether the specifications for which Strix or Otus performs exceptionally good or bad share some property that could explain the observations. This could help identify a class of specifications for which one of the constructions perform very well.

For example, we observed that `amba_decomposed_lock` is solved faster by Otus than by Strix (see Figure 8.10). On the other hand, Strix was able to solve all `full_arbiter_enc` specifications whereas Otus could not complete any of them (see Figure 8.9). This raises the question what makes these specification particularly hard or easy for Strix or Otus. The same can be asked for those specifications presented in Table 8.2.

9.8 Engineering

Finally, we expect that the performance could be further increased through engineering improvements. For example, we currently rely on Sylvan for the parallelization of the construction, but there are other possibilities for parallelization as well. A brief glance at Figure 3.2 shows ample opportunities for additional parallelization and Section 8.5 shows that additional parallelization could actually be beneficial for the performance. More research is needed to determine which level of parallelization is optimal.

Apart from introducing additional parallelization, further engineering improvements for the interface to Sylvan are possible. As discussed in Section 7.2, Sylvan is currently strongly coupled with the Java garbage collector. While we have not encountered any examples where this is problematic, it could hurt performance if the garbage collection is triggered too frequently. Manually managing the references to BDD nodes, while not particularly convenient for the developer, could reduce the amount of garbage collection runs.

Finally, the load on the Java garbage collector can also be reduced by creating fewer intermediate results. This could be achieved by using the builder pattern [37] to chain multiple BDD operations without returning an intermediate BDD. However, we expect that these engineering details will have at most a minor effect on the performance as we have not encountered any congestion on the garbage collector.

CHAPTER 10

Conclusion

We have researched, implemented and evaluated a new LTL reactive synthesis construction which combines the recently discovered normalization technique for LTL formulas [86] and a technique that allows the construction of a DPW from a product of a DRW and a DSW [13] to construct a parity game from an LTL specification in a mostly symbolic manner. We first normalize the LTL formula into simpler fragments and construct explicit DCW and DBW automata for these fragments. We then symbolically compute the product of these automata and attempt to construct a DPW from the resulting DRW. If successful, we convert the DPW to a parity game and solve it. If not successful, we repeat the construction using the negated LTL formula to obtain a DSW that is equivalent to the original formula. The product of the DSW and DRW is then used to construct a DPW, which is then interpreted as a game and solved.

The construction was implemented in a new prototypical synthesis tool called Otus, which is based on the LTL and ω-automata library Owl [49]. It was evaluated in various benchmarks and was compared against the current state-of-the-art synthesis tool Strix [59] which uses a very similar but explicit approach. We have observed mixed but promising result. There are specifications where Otus outperforms Strix regarding execution time by factors in the order of 10. However, there are also specifications where Strix outperforms Otus by factors in the order of 1000.

Nevertheless, these results are very promising and motivate further research into the construction. We have identified abundant future work which could still significantly improve the performance of the construction. Furthermore, the controllers generated by Otus are commonly in the order of $1000\times$ larger than those generated by Strix. Further research in, for example, the ordering of the variables in the BDDs and the use of bisimulation minimizations could both reduce the size of the controllers as well as reduce the execution time significantly.

Bibliography

[1] C. S. Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of büchi automata. *Theoretical Computer Science*, 363(2):224–233, 2006.

[2] T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Kretínský, D. Müller, D. Parker, and J. Strejcek. The hanoi omega-automata format. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 479–486. Springer, 2015.

[3] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[4] M. Bauland, M. Mundhenk, T. Schneider, H. Schnoor, I. Schnoor, and H. Vollmer. The tractability of model-checking for LTL: the good, the bad, and the ugly fragments. *Electronic Notes in Theoretical Computer Science*, 231:277–292, 2009.

[5] M. Benerecetti, D. Dell'Erba, and F. Mogavero. Solving parity games via priority promotion. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 270–290. Springer, 2016.

[6] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[7] A. Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

[8] R. Bloem, K. Chatterjee, and B. Jobstmann. Graph games and reactive synthesis. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 921–962. Springer, 2018.

[9] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006.

[10] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[11] R. Bloem, S. Schewe, and A. Khalimov. CTL* synthesis via LTL synthesis. In D. Fisman and S. Jacobs, editors, *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*, volume 260 of *EPTCS*, pages 4–22, 2017.

[12] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer, 2012.

[13] U. Boker, O. Kupferman, and A. Steinitz. Parityizing Rabin and Streett. In K. Lodaya and M. Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 412–423. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

[14] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[15] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[16] J. R. Buchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:367–378, 1969.

[17] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. International Congress on Logic, Method, and Philosophy of Science*. Stanford University Press, 1960.

[18] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In H. Hatami, P. McKenzie, and V. King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 252–263. ACM, 2017.

[19] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[20] I. Cerná and R. Pelánek. Relating hierarchy of temporal properties to model checking. In B. Rovan and P. Vojtás, editors, *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings*, volume 2747 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2003.

[21] E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 1992.

[22] K. Chatterjee, T. A. Henzinger, and N. Piterman. Generalized parity games. In H. Seidl, editor, *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007, Proceedings*, volume 4423 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2007.

[23] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic*, 1:3–50, 1957.

[24] A. Church. Logic, arithmetic, and automata. *Proceedings of the international congress of mathematicians, 1962, Djursholm, Sweden*, pages 23–35, 1963.

[25] E. M. Clarke, T. A. Henzinger, and H. Veith. Introduction to model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[26] H. Cohen, J. Whaley, J. Wildt, and N. Gorogiannis. Buddy. `https://sourceforge.net/projects/buddy/`.

[27] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[28] R. Ehlers. Unbeast: Symbolic bounded synthesis. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 2011.

[29] R. Ehlers. *Symmetric and efficient synthesis*. PhD thesis, Universität Saarbrücken, 2013.

[30] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 368–377. IEEE Computer Society, 1991.

[31] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258(1-2):491–522, 2001.

[32] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[33] P. Faymonville, B. Finkbeiner, and L. Tentrup. Bosy: An experimentation framework for bounded synthesis. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 325–332. Springer, 2017.

[34] J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak. An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In H. Erdogmus and K. Havelund, editors, *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 112–121. ACM, 2017.

[35] B. Finkbeiner and F. Klein. Bounded cycle synthesis. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 118–135. Springer, 2016.

[36] B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[37] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns CD. Elements of reusable object-oriented software*. Addison-Wesley Professional, 1998.

[38] Y. Godhal, K. Chatterjee, and T. A. Henzinger. Synthesis of AMBA AHB from formal specification: a case study. *International Journal on Software Tools for Technology Transfer*, 15(5):585–601, Oct 2013.

[39] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker. (SYNTCOMP 2018): Benchmarks, participants & results, 2019.

[40] S. Jacobs, F. Klein, and S. Schirmer. A high-level LTL synthesis format: TLSF v1.1. In R. Piskac and R. Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, volume 229 of *EPTCS*, pages 112–132, 2016.

[41] S. Jacobs and G. A. Perez. The reactive synthesis competition. `http://www.syntcomp.org/`. Accessed: 2021-01-17.

[42] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 117–124. IEEE Computer Society, 2006.

[43] B. Jobstmann, S. J. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th*

International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, volume 4590 of *Lecture Notes in Computer Science*, pages 258–262. Springer, 2007.

[44] M. Jurdzinski. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[45] M. Jurdzinski. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.

[46] M. Jurdzinski and R. Lazic. Succinct progress measures for solving parity games. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–9. IEEE Computer Society, 2017.

[47] A. Khalimov, S. Jacobs, and R. Bloem. PARTY parameterized synthesis of token rings. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 928–933. Springer, 2013.

[48] H. Kress-Gazit and H. Torfah. The challenges in specifying and explaining synthesized implementations of reactive systems. In B. Finkbeiner and S. Kleinberg, editors, *Proceedings 3rd Workshop on formal reasoning about Causation, Responsibility, and Explanations in Science and Technology, CREST@ETAPS 2018, Thessaloniki, Greece, 21st April 2018*, volume 286 of *EPTCS*, pages 50–64, 2018.

[49] J. Kretínský, T. Meggendorfer, and S. Sickert. Owl: A library for ω-words, automata, and LTL. In S. K. Lahiri and C. Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 543–550. Springer, 2018.

[50] J. Kretínský, T. Meggendorfer, C. Waldmann, and M. Weininger. Index appearance record for transforming rabin automata into parity automata. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 443–460, 2017.

[51] O. Kupferman. Recent challenges and ideas in temporal synthesis. In M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings*, volume 7147 of *Lecture Notes in Computer Science*, pages 88–98. Springer, 2012.

[52] O. Kupferman. Automata theory and model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 107–151. Springer, 2018.

[53] O. Kupferman, N. Piterman, and M. Y. Vardi. Safraless compositional synthesis. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006.

[54] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 531–542. IEEE Computer Society, 2005.

[55] R. P. Kurshan. Transfer of model checking to industrial practice. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 763–793. Springer, 2018.

[56] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[57] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In R. Parikh, editor, *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[58] O. Lijzenga and T. van Dijk. Symbolic parity game solvers that yield winning strategies. In J. Raskin and D. Bresolin, editors, *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21-22, 2020*, volume 326 of *EPTCS*, pages 18–32, 2020.

[59] M. Luttenberger, P. J. Meyer, and S. Sickert. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica*, 57(1-2):3–36, 2020.

[60] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In F. B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, page 205. ACM, 1987.

[61] N. Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 79:122–128, 2003.

[62] D. A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975.

[63] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[64] T. Meggendorfer. JBDD: A java BDD library. https://github.com/incaseoftrouble/jbdd, 2017.

[65] J. Meijer. *Efficient learning and analysis of system behavior*. PhD thesis, UT, Netherlands, Sept. 2019. DSI Ph.D. thesis series no. 19-015 IPA dissertation series no. 2019-10.

[66] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[67] S. Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

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

[69] E. F. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[70] A. Morgenstern. *Symbolic controller synthesis for LTL specifications*. PhD thesis, University of Kaiserslautern, 2010.

[71] A. Morgenstern and K. Schneider. From LTL to symbolically represented deterministic automata. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, volume 4905 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2008.

[72] A. Morgenstern and K. Schneider. Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. In A. Montanari, M. Napoli, and M. Parente, editors, *Proceedings First Symposium on Games, Automata, Logic, and Formal Verification, GANDALF 2010, Minori (Amalfi Coast), Italy, 17-18th June 2010*, volume 25 of *EPTCS*, pages 89–102, 2010.

[73] A. Morgenstern, K. Schneider, and S. Lamberti. Generating deterministic ω-automata for most LTL formulas by the breakpoint construction. In C. Scholl and S. Disch, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Freiburg, Germany, March 3-5, 2008*, pages 119–128. Shaker, 2008.

[74] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, editor, *Computation Theory - Fifth Symposium, Zaborów, Poland, December 3-8, 1984, Proceedings*, volume 208 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 1984.

[75] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time.

In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 422–427. IEEE Computer Society, 1988.

[76] Oracle. Graalvm. `https://www.graalvm.org/`.

[77] V. A. Pedroni. *Finite State Machines in Hardware: Theory and Design (with VHDL and SystemVerilog)*. The MIT Press, 2013.

[78] N. Piterman. From nondeterministic buchi and streett automata to deterministic parity automata. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 255–264. IEEE Computer Society, 2006.

[79] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 746–757. IEEE Computer Society, 1990.

[80] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[81] M. O. Rabin. Automata on infinite objects and Church's problem. In *Regional Conference Series in Mathematics*, volume 13. American Mathematical Society, 1972.

[82] M. O. Rabin and D. S. Scott. Finite automata and their decision problems. *IBM Journal of Research & Development*, 3(2):114–125, 1959.

[83] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In M. R. Lightner and J. A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 42–47. IEEE Computer Society / ACM, 1993.

[84] S. Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327. IEEE Computer Society, 1988.

[85] K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[86] S. Sickert and J. Esparza. An efficient normalisation procedure for linear temporal logic and very weak alternating automata. In H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 831–844. ACM, 2020.

[87] S. Sohail, F. Somenzi, and K. Ravi. A hybrid algorithm for LTL games. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, volume 4905 of *Lecture Notes in Computer Science*, pages 309–323. Springer, 2008.

[88] F. Somenzi. Efficient manipulation of decision diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):171–181, 2001.

[89] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1/2):121–141, 1982.

[90] T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Pearson Addison Wesley, 2006.

[91] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[92] S. Tasiran, R. Hojati, and R. K. Brayton. Language containment of non-deterministic omega-automata. In P. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95, Frankfurt/Main, Germany, October 2-4, 1995, Proceedings*, volume 987 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 1995.

[93] T. van Dijk. *Sylvan: multi-core decision diagrams*. PhD thesis, University of Twente, July 2016. CTIT Ph.D. thesis series no. 16-398 IPA dissertation series no. 2016-09.

[94] T. van Dijk and B. Rubbens. Simple fixpoint iteration to solve parity games. In J. Leroux and J. Raskin, editors, *Proceedings Tenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2019, Bordeaux, France, 2-3rd September 2019*, volume 305 of *EPTCS*, pages 123–139, 2019.

[95] T. van Dijk and J. van de Pol. Multi-core symbolic bisimulation minimisation. In M. Chechik and J. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 332–348. Springer, 2016.

[96] T. van Dijk and J. C. van de Pol. Lace: Non-blocking split deque for work-stealing. In L. M. B. Lopes, J. Zilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, volume 8806 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 2014.

[97] M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.

[98] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1225–1230, 2000.

[99] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.

Appendices

APPENDIX A

Empirical Evaluation Tabular Results

This chapters presents the results of the empirical evaluation in tabular form for those that have difficulty to distinguish colors or that have a black and white copy.

Tool	Realizable	Success	Timeout	Error
Strix	Yes	385	36	0
Strix	No	153	5	0
Otus-JBDD	Yes	314	107	0
Otus-JBDD	No	82	73	2
Otus-Sylvan	Yes	321	100	0
Otus-Sylvan	No	85	70	2

TABLE A.1 Tabular representation of Figure 8.1

Tool	Realizable	DRW_DCW	DRW_PRODUCT	DPW1	DSW_DCW	DPW2	DFI	SD
Otus-JBDD	Yes	56	2	35	1	12	1	0
Otus-JBDD	No	34	5	24	0	10	0	0
Otus-Sylvan	Yes	55	1	30	1	10	1	2
Otus-Sylvan	No	35	0	26	0	9	0	0

TABLE A.2 Tabular representation of Figure 8.2

Stage	Realizable, DSW Constructed	Realizable, No DSW Constructed	Unrealizable, DSW Constructed	Unrealizable, No DSW Constructed
DRW_DCW	5.29	46.78	3.97	74.51
DRW_PRODUCT	0.41	2.97	1.37	1.01
DPW1	8.11	20.72	6.80	17.26
DSW_DCW	2.92	0.00	5.64	0.00
DSW_PRODUCT	0.31	0.00	0.77	0.00
DRW_DSW_PRODUCT	0.25	0.00	1.22	0.00
DPW2	58.20	0.0	72.32	0.00
DFI	7.35	16.12	7.91	7.22
SD	15.75	11.73	0.00	0.00
AIG	1.41	1.69	0.00	0.00

TABLE A.3 Tabular representation of Figure 8.5

Stage	Realizable, DSW Constructed	Realizable, No DSW Constructed	Unrealizable, DSW Constructed	Unrealizable, No DSW Constructed
DRW_DCW	0.63	37.34	0.72	75.71
DRW_PRODUCT	0.01	0.20	0.01	0.06
DPW1	2.82	40.91	4.07	22.84
DSW_DCW	0.44	0.00	3.17	0.00
DSW_PRODUCT	0.01	0.00	0.04	0.00
DRW_DSW_PRODUCT	0.01	0.00	0.01	0.00
DPW2	69.89	0.0	89.87	0.00
DFI	3.79	3.74	2.09	1.39
SD	21.31	16.24	0.00	0.00
AIG	1.09	1.85	0.00	0.00

TABLE A.4 Tabular representation of Figure 8.6

#workers	ltl2dba_beta_6	ltl2dba_beta_5	ltl2dba_E_8	ltl2dba_S_8	ltl2dba_S_7
1	1.00	1.00	1.00	1.00	1.00
2	1.54	1.39	1.47	1.42	1.30
3	2.26	1.93	1.95	1.83	1.57
4	2.78	2.23	2.27	2.11	1.75
5	3.25	2.57	2.53	2.32	1.85
6	3.72	2.77	2.78	2.55	1.99
7	4.02	2.83	2.93	2.66	2.01
8	4.33	2.97	3.01	2.69	2.00
9	4.46	3.03	3.09	2.77	2.18
10	4.58	3.04	3.06	2.80	1.98
11	4.70	3.12	3.11	2.82	2.16
12	4.80	3.15	3.13	2.80	2.01
13	4.94	3.08	3.15	2.76	2.21
14	5.06	3.14	3.18	2.88	2.04
15	5.09	3.16	3.17	2.94	2.07
16	4.63	2.63	2.52	2.18	1.52

TABLE A.5 Tabular representation of Figure 8.11

APPENDIX B

Exploratory Benchmark Results

This appendix presents the raw results of the exploratory benchmark. The specification columns refers to the filename of the specification (without extension) as available on `https://github.com/meyerphi/syntcomp-reference`.[1] The abbreviations of Table 8.2 are used to indicate in which stage the construction timed out (if applicable).

B.1 Realizable Specifications

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
ActionConverter	Success	Success	-	Success	-
amba_case_study_2	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_case_study_3	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_case_study_4	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_decomposed_arbiter_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_decomposed_arbiter_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_decomposed_arbiter_2	Success	Success	-	Success	-
amba_decomposed_arbiter_3	Success	Timeout	DPW1	Timeout	DPW1
amba_decomposed_arbiter_4	Success	Timeout	DPW1	Timeout	DPW1
amba_decomposed_arbiter_5	Success	Timeout	DPW1	Timeout	DPW1
amba_decomposed_arbiter_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_decomposed_arbiter_7	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_decomposed_arbiter_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_decomposed_decode	Success	Success	-	Success	-
amba_decomposed_encode_10	Success	Success	-	Success	-
amba_decomposed_encode_12	Success	Success	-	Success	-
amba_decomposed_encode_2	Success	Success	-	Success	-
amba_decomposed_encode_4	Success	Success	-	Success	-
amba_decomposed_encode_6	Success	Success	-	Success	-
amba_decomposed_encode_8	Success	Success	-	Success	-
amba_decomposed_lock_10	Success	Success	-	Success	-
amba_decomposed_lock_12	Timeout	Success	-	Success	-
amba_decomposed_lock_2	Success	Success	-	Success	-
amba_decomposed_lock_4	Success	Success	-	Success	-
amba_decomposed_lock_6	Success	Success	-	Success	-
amba_decomposed_lock_8	Success	Success	-	Success	-
amba_decomposed_shift	Success	Success	-	Success	-
amba_decomposed_tburst4	Success	Success	-	Success	-
amba_decomposed_tincr	Success	Success	-	Success	-
amba_decomposed_tsingle	Success	Success	-	Success	-
Automata16S	Success	Success	-	Success	-
Automata32S	Success	Success	-	Success	-
Automata	Success	Success	-	Success	-
button	Success	Success	-	Success	-
Button	Success	Success	-	Success	-
Cockpitboard	Success	Success	-	Success	-

[1] Commit `66ced6d6207d7be919f905546c40701303a46aa3`

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
collector_v1_2	Success	Success	-	Success	-
collector_v1_3	Success	Success	-	Success	-
collector_v1_4	Success	Success	-	Success	-
collector_v1_5	Success	Success	-	Success	-
collector_v1_6	Success	Success	-	Success	-
collector_v1_7	Success	Success	-	Success	-
collector_v2_2	Success	Success	-	Success	-
collector_v2_3	Success	Success	-	Success	-
collector_v2_4	Success	Success	-	Success	-
collector_v2_5	Success	Timeout	DPW2	Timeout	DPW2
collector_v2_6	Success	Timeout	DPW1	Timeout	DPW2
collector_v2_7	Success	Timeout	DPW1	Timeout	DPW1
collector_v3_2	Success	Success	-	Success	-
collector_v3_3	Success	Success	-	Success	-
collector_v3_4	Success	Success	-	Success	-
collector_v3_5	Success	Success	-	Success	-
collector_v3_6	Success	Success	-	Success	-
collector_v3_7	Success	Success	-	Success	-
collector_v4_2	Success	Success	-	Success	-
collector_v4_3	Success	Success	-	Success	-
collector_v4_4	Success	Success	-	Success	-
collector_v4_5	Success	Success	-	Success	-
collector_v4_6	Success	Timeout	DPW2	Timeout	DPW2
collector_v4_7	Success	Timeout	DPW1	Timeout	DPW2
detector_10	Success	Timeout	DPW1	Timeout	DPW1
detector_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
detector_1	Success	Success	-	Success	-
detector_2	Success	Success	-	Success	-
detector_3	Success	Success	-	Success	-
detector_4	Success	Success	-	Success	-
detector_5	Success	Success	-	Success	-
detector_6	Success	Timeout	DPW2	Success	-
detector_7	Success	Timeout	DPW2	Timeout	DPW2
detector_8	Success	Timeout	DPW2	Timeout	DPW2
EnemeyModule	Success	Success	-	Success	-
escalator_bidirectional_init	Success	Success	-	Success	-
EscalatorBidirectionalInit	Success	Success	-	Success	-
escalator_bidirectional	Success	Success	-	Success	-
EscalatorBidirectional	Success	Success	-	Success	-
escalator_counting_init	Success	Success	-	Success	-
EscalatorCountingInit	Success	Success	-	Success	-
escalator_counting	Success	Success	-	Success	-
EscalatorCounting	Success	Success	-	Success	-
escalator_non-counting	Success	Success	-	Success	-
EscalatorNonCounting	Success	Success	-	Success	-
escalator_non-reactive	Success	Success	-	Success	-
EscalatorNonReactive	Success	Success	-	Success	-
escalator_smart	Success	Timeout	DPW2	Success	-
EscalatorSmart	Success	Timeout	DPW2	Success	-
full_arbiter_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_2	Success	Success	-	Success	-
full_arbiter_3	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_4	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_7	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_enc_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_enc_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_enc_2	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_enc_4	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_enc_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_enc_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
Gamelogic	Success	Success	-	Success	-
GamemodeChooser	Success	Success	-	Success	-
Gamemodule	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
genbuf2	Timeout	Timeout	DPW1	Timeout	DPW1
genbuf3	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
genbuf4	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
genbuf5	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
generalized_buffer_2	Timeout	Timeout	DPW1	Timeout	DPW1
generalized_buffer_3	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
generalized_buffer_4	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
generalized_buffer_5	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
generalized_buffer_6	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
increment	Success	Success	-	Success	-
Increment	Success	Success	-	Success	-
KitchenTimerV0	Success	Success	-	Success	-
KitchenTimerV10	Success	Success	-	Success	-
KitchenTimerV1	Success	Success	-	Success	-
KitchenTimerV2	Success	Success	-	Success	-
KitchenTimerV3	Success	Success	-	Success	-
KitchenTimerV4	Success	Success	-	Success	-
KitchenTimerV5	Success	Success	-	Success	-
KitchenTimerV6	Success	Success	-	Success	-
KitchenTimerV7	Success	Success	-	Success	-
KitchenTimerV8	Success	Success	-	Success	-
KitchenTimerV9	Success	Success	-	Success	-
LedMatrix	Success	Success	-	Success	-
lilydemo03	Success	Success	-	Success	-
lilydemo04	Success	Success	-	Success	-
lilydemo05	Success	Success	-	Success	-
lilydemo06	Success	Success	-	Success	-
lilydemo07	Success	Success	-	Success	-
lilydemo08	Success	Success	-	Success	-
lilydemo09	Success	Success	-	Success	-
lilydemo10	Success	Success	-	Success	-
lilydemo12	Success	Success	-	Success	-
lilydemo13	Success	Success	-	Success	-
lilydemo14	Success	Success	-	Success	-
lilydemo17	Success	Success	-	Success	-
lilydemo18	Success	Success	-	Success	-
lilydemo19	Success	Success	-	Success	-
lilydemo20	Success	Success	-	Success	-
lilydemo21	Success	Success	-	Success	-
lilydemo22	Success	Success	-	Success	-
lilydemo23	Success	Success	-	Success	-
lilydemo24	Success	Success	-	Success	-
load_balancer_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_2	Success	Success	-	Success	-
load_balancer_3	Success	Timeout	DPW1	Timeout	DFI
load_balancer_4	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_5	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_7	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_8	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_unreal2_2	Success	Success	-	Success	-
loadcomp2	Success	Success	-	Success	-
loadcomp3	Success	Timeout	DPW1	Timeout	DPW1
loadcomp4	Success	Timeout	DPW1	Timeout	DPW1
loadcomp5	Success	Timeout	DPW1	Timeout	DPW1
loadfull2	Success	Success	-	Success	-
loadfull3	Success	Timeout	DPW1	Timeout	DPW1
loadfull4	Success	Timeout	DPW1	Timeout	DPW1
loadfull5	Success	Timeout	DRW_PRODUCT	Timeout	DPW1
ltl2dba01	Success	Success	-	Success	-
ltl2dba02	Success	Success	-	Success	-
ltl2dba03	Success	Success	-	Success	-
ltl2dba04	Success	Success	-	Success	-
ltl2dba05	Success	Success	-	Success	-
ltl2dba06	Success	Success	-	Success	-
ltl2dba07	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
ltl2dba08	Success	Success	-	Success	-
ltl2dba09	Success	Success	-	Success	-
ltl2dba10	Success	Success	-	Success	-
ltl2dba11	Success	Success	-	Success	-
ltl2dba12	Success	Success	-	Success	-
ltl2dba13	Success	Success	-	Success	-
ltl2dba14	Success	Success	-	Success	-
ltl2dba15	Success	Success	-	Success	-
ltl2dba16	Success	Success	-	Success	-
ltl2dba17	Success	Success	-	Success	-
ltl2dba18	Success	Success	-	Success	-
ltl2dba19	Success	Success	-	Success	-
ltl2dba20	Success	Success	-	Success	-
ltl2dba21	Success	Success	-	Success	-
ltl2dba22	Success	Success	-	Success	-
ltl2dba23	Success	Success	-	Success	-
ltl2dba24	Success	Success	-	Success	-
ltl2dba25	Success	Success	-	Success	-
ltl2dba26	Success	Success	-	Success	-
ltl2dba_alpha_10	Success	Success	-	Success	-
ltl2dba_alpha_12	Success	Success	-	Success	-
ltl2dba_alpha_1	Success	Success	-	Success	-
ltl2dba_alpha_2	Success	Success	-	Success	-
ltl2dba_alpha_3	Success	Success	-	Success	-
ltl2dba_alpha_4	Success	Success	-	Success	-
ltl2dba_alpha_5	Success	Success	-	Success	-
ltl2dba_alpha_6	Success	Success	-	Success	-
ltl2dba_alpha_7	Success	Success	-	Success	-
ltl2dba_alpha_8	Success	Success	-	Success	-
ltl2dba_beta_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_PRODUCT
ltl2dba_beta_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_beta_1	Success	Success	-	Success	-
ltl2dba_beta_2	Success	Success	-	Success	-
ltl2dba_beta_3	Success	Success	-	Success	-
ltl2dba_beta_4	Success	Success	-	Success	-
ltl2dba_beta_5	Success	Success	-	Success	-
ltl2dba_beta_6	Success	Success	-	Success	-
ltl2dba_beta_7	Success	Timeout	DPW1	Success	-
ltl2dba_beta_8	Success	Timeout	DRW_PRODUCT	Timeout	DPW1
ltl2dba_C1_10	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_C1_12	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_C1_1	Success	Success	-	Success	-
ltl2dba_C1_2	Success	Success	-	Success	-
ltl2dba_C1_3	Success	Success	-	Success	-
ltl2dba_C1_4	Success	Success	-	Success	-
ltl2dba_C1_5	Success	Success	-	Success	-
ltl2dba_C1_6	Success	Success	-	Success	-
ltl2dba_C1_7	Success	Success	-	Success	-
ltl2dba_C1_8	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_C2_10	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_C2_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_C2_1	Success	Success	-	Success	-
ltl2dba_C2_2	Success	Success	-	Success	-
ltl2dba_C2_3	Success	Success	-	Success	-
ltl2dba_C2_4	Success	Success	-	Success	-
ltl2dba_C2_5	Success	Success	-	Success	-
ltl2dba_C2_6	Success	Timeout	DPW2	Success	-
ltl2dba_C2_7	Success	Timeout	DPW2	Timeout	DPW2
ltl2dba_C2_8	Success	Timeout	DPW2	Timeout	DPW2
ltl2dba_E_10	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_E_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_E_1	Success	Success	-	Success	-
ltl2dba_E_2	Success	Success	-	Success	-
ltl2dba_E_3	Success	Success	-	Success	-
ltl2dba_E_4	Success	Success	-	Success	-
ltl2dba_E_5	Success	Success	-	Success	-
ltl2dba_E_6	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
ltl2dba_E_7	Success	Success	-	Success	-
ltl2dba_E_8	Success	Success	-	Success	-
ltl2dba_Q_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_Q_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_Q_1	Success	Success	-	Success	-
ltl2dba_Q_2	Success	Success	-	Success	-
ltl2dba_Q_3	Success	Success	-	Success	-
ltl2dba_Q_4	Success	Success	-	Success	-
ltl2dba_Q_5	Success	Success	-	Success	-
ltl2dba_Q_6	Success	Success	-	Success	-
ltl2dba_Q_7	Success	Timeout	DPW1	Success	-
ltl2dba_Q_8	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_R_1	Success	Success	-	Success	-
ltl2dba_S_10	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_S_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_S_1	Success	Success	-	Success	-
ltl2dba_S_2	Success	Success	-	Success	-
ltl2dba_S_3	Success	Success	-	Success	-
ltl2dba_S_4	Success	Success	-	Success	-
ltl2dba_S_5	Success	Success	-	Success	-
ltl2dba_S_6	Success	Success	-	Success	-
ltl2dba_S_7	Success	Success	-	Success	-
ltl2dba_S_8	Success	Success	-	Success	-
ltl2dba_U1_10	Success	Timeout	DFI	Timeout	SD
ltl2dba_U1_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_U1_1	Success	Success	-	Success	-
ltl2dba_U1_2	Success	Success	-	Success	-
ltl2dba_U1_3	Success	Success	-	Success	-
ltl2dba_U1_4	Success	Success	-	Success	-
ltl2dba_U1_5	Success	Success	-	Success	-
ltl2dba_U1_6	Success	Success	-	Success	-
ltl2dba_U1_7	Success	Success	-	Success	-
ltl2dba_U1_8	Success	Success	-	Success	-
ltl2dba_U2_10	Success	Success	-	Success	-
ltl2dba_U2_12	Success	Success	-	Success	-
ltl2dba_U2_1	Success	Success	-	Success	-
ltl2dba_U2_2	Success	Success	-	Success	-
ltl2dba_U2_3	Success	Success	-	Success	-
ltl2dba_U2_4	Success	Success	-	Success	-
ltl2dba_U2_5	Success	Success	-	Success	-
ltl2dba_U2_6	Success	Success	-	Success	-
ltl2dba_U2_7	Success	Success	-	Success	-
ltl2dba_U2_8	Success	Success	-	Success	-
ltl2dpa01	Success	Success	-	Success	-
ltl2dpa02	Success	Success	-	Success	-
ltl2dpa03	Success	Success	-	Success	-
ltl2dpa04	Success	Success	-	Success	-
ltl2dpa05	Success	Success	-	Success	-
ltl2dpa06	Success	Success	-	Success	-
ltl2dpa07	Success	Success	-	Success	-
ltl2dpa08	Success	Success	-	Success	-
ltl2dpa09	Success	Success	-	Success	-
ltl2dpa10	Success	Success	-	Success	-
ltl2dpa11	Success	Success	-	Success	-
ltl2dpa12	Success	Success	-	Success	-
ltl2dpa13	Success	Success	-	Success	-
ltl2dpa14	Success	Success	-	Success	-
ltl2dpa15	Success	Success	-	Success	-
ltl2dpa16	Success	Success	-	Success	-
ltl2dpa17	Success	Success	-	Success	-
ltl2dpa18	Success	Success	-	Success	-
ltl2dpa19	Success	Success	-	Success	-
ltl2dpa20	Success	Success	-	Success	-
ltl2dpa21	Success	Success	-	Success	-
ltl2dpa22	Success	Timeout	DPW1	Timeout	DPW1
ltl2dpa23	Success	Success	-	Success	-
ltl2dpa24	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
ModdifiedLedMatrix5X	Success	Success	-	Success	-
music_app_feedback	Success	Success	-	Success	-
MusicAppFeedback	Success	Success	-	Success	-
music_app_motivating_2	Success	Success	-	Success	-
music_app_motivating	Success	Success	-	Success	-
MusicAppMotivating	Success	Success	-	Success	-
music_app_simple	Success	Success	-	Success	-
MusicAppSimple	Success	Success	-	Success	-
mux_10	Success	Success	-	Success	-
mux_12	Success	Success	-	Success	-
mux_2	Success	Success	-	Success	-
mux_4	Success	Success	-	Success	-
mux_6	Success	Success	-	Success	-
mux_8	Success	Success	-	Success	-
narylatch_10	Timeout	Success	-	Timeout	DRW_DCW
narylatch_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
narylatch_2	Success	Success	-	Success	-
narylatch_4	Success	Success	-	Success	-
narylatch_6	Success	Success	-	Success	-
narylatch_8	Success	Success	-	Success	-
OneCounterGuiA9	Success	Success	-	Success	-
OneCounterInRangeA3	Success	Success	-	Success	-
OneCounter	Success	Success	-	Success	-
prioritized_arbiter_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_1	Success	Success	-	Success	-
prioritized_arbiter_2	Success	Success	-	Success	-
prioritized_arbiter_3	Success	Success	-	Success	-
prioritized_arbiter_4	Success	Success	-	Success	-
prioritized_arbiter_5	Success	Success	-	Success	-
prioritized_arbiter_6	Success	Timeout	DPW1	Success	-
prioritized_arbiter_7	Success	Timeout	DPW1	Timeout	DPW1
prioritized_arbiter_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_enc_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_enc_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_enc_2	Success	Success	-	Success	-
prioritized_arbiter_enc_4	Success	Success	-	Success	-
prioritized_arbiter_enc_6	Success	Timeout	DPW1	Success	-
prioritized_arbiter_enc_8	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
Radarboard	Success	Success	-	Success	-
RegManager	Success	Success	-	Success	-
RotationCalculator	Success	Success	-	Success	-
round_robin_arbiter_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_2	Success	Success	-	Success	-
round_robin_arbiter_3	Success	Timeout	DPW2	Timeout	DPW2
round_robin_arbiter_4	Success	Timeout	DPW2	Timeout	DPW2
round_robin_arbiter_5	Success	Timeout	DSW_DCW	Timeout	DSW_DCW
round_robin_arbiter_6	Success	Timeout	DPW1	Timeout	DPW1
round_robin_arbiter_7	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_8	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
Scoreboard	Success	Success	-	Success	-
SensorInit	Success	Success	-	Success	-
SensorPart	Success	Success	-	Success	-
SensorRegister	Success	Success	-	Success	-
SensorSelector	Success	Success	-	Success	-
SensorSubmodulChooser	Success	Success	-	Success	-
Sensor	Success	Success	-	Success	-
shift_10	Success	Success	-	Success	-
shift_12	Success	Success	-	Success	-
shift_2	Success	Success	-	Success	-
shift_4	Success	Success	-	Success	-
shift_6	Success	Success	-	Success	-
shift_8	Success	Success	-	Success	-
simple_arbiter_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_2	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
simple_arbiter_3	Success	Success	-	Success	-
simple_arbiter_4	Success	Success	-	Success	-
simple_arbiter_5	Success	Success	-	Success	-
simple_arbiter_6	Success	Success	-	Success	-
simple_arbiter_7	Success	Success	-	Success	-
simple_arbiter_8	Success	Timeout	DRW_DCW	Timeout	SD
simple_arbiter_enc_10	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_enc_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_enc_2	Success	Success	-	Success	-
simple_arbiter_enc_4	Success	Success	-	Success	-
simple_arbiter_enc_6	Success	Success	-	Success	-
simple_arbiter_enc_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
slider_default	Success	Success	-	Success	-
SliderDefault	Success	Success	-	Success	-
slider_delayed	Success	Success	-	Success	-
SliderDelayed	Success	Success	-	Success	-
slider_scored	Success	Success	-	Success	-
SliderScored	Success	Success	-	Success	-
SPIPureNext	Success	Success	-	Success	-
SPIReadClk	Success	Success	-	Success	-
SPIReadManag	Success	Success	-	Success	-
SPIReadSdi	Success	Success	-	Success	-
SPI	Success	Success	-	Success	-
SPIWriteClk	Success	Success	-	Success	-
SPIWriteManag	Success	Success	-	Success	-
SPIWriteSdi	Success	Success	-	Success	-
tictactoe	Success	Success	-	Success	-
torcs_accelerating	Success	Success	-	Success	-
TorcsAccelerating	Success	Success	-	Success	-
torcs_gearing	Success	Success	-	Success	-
TorcsGearing	Success	Success	-	Success	-
torcs_simple	Success	Success	-	Success	-
TorcsSimple	Success	Success	-	Success	-
torcs_steering_improved	Success	Success	-	Success	-
TorcsSteeringImproved	Success	Success	-	Success	-
torcs_steering_simple	Success	Success	-	Success	-
TorcsSteeringSimple	Success	Success	-	Success	-
torcs_steering_smart	Success	Success	-	Success	-
TorcsSteeringSmart	Success	Success	-	Success	-
TwoCounters3	Success	Success	-	Success	-
TwoCounters4	Success	Success	-	Success	-
TwoCountersInRangeA6	Success	Success	-	Success	-
TwoCountersRefinedRefined	Success	Success	-	Success	-
UnderapproxStrengthenedDemo	Success	Success	-	Success	-
zoo0	Success	Success	-	Success	-
Zoo0	Success	Success	-	Success	-
zoo10	Success	Success	-	Success	-
Zoo10	Success	Success	-	Success	-
zoo5	Success	Success	-	Success	-
Zoo5	Success	Success	-	Success	-

B.2 Unrealizable Specifications

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
amba_case_study_unreal1_2_2	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_case_study_unreal1_2_2	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
amba_case_study_unreal2_2	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
detector_unreal_10	Success	Error	-	Error	-
detector_unreal_12	Success	Error	-	Error	-
detector_unreal_2	Success	Success	-	Success	-
detector_unreal_4	Success	Success	-	Success	-
detector_unreal_6	Success	Timeout	DPW2	Timeout	DPW2
detector_unreal_8	Success	Timeout	DPW1	Timeout	DPW2
full_arbiter_unreal1_2_12	Success	Timeout	DPW1	Timeout	DPW1

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
full_arbiter_unreal1_2_15	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal1_2_18	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal1_2_3	Success	Success	-	Success	-
full_arbiter_unreal1_2_6	Success	Success	-	Success	-
full_arbiter_unreal1_2_9	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_unreal1_3_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal1_3_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal1_3_1	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_unreal1_3_2	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_unreal1_3_3	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_unreal1_3_4	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_unreal1_3_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal1_3_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal1_3_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal2_2	Success	Success	-	Success	-
full_arbiter_unreal2_3	Success	Timeout	DPW1	Timeout	DPW1
full_arbiter_unreal2_4	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
full_arbiter_unreal2_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
genbuf6	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
generalized_buffer_unreal1_2_2	Success	Timeout	DRW_PRODUCT	Timeout	DPW1
generalized_buffer_unreal2_2	Timeout	Timeout	DRW_PRODUCT	Timeout	DPW1
lilydemo01	Success	Success	-	Success	-
lilydemo02	Success	Success	-	Success	-
lilydemo11	Success	Success	-	Success	-
lilydemo15	Success	Success	-	Success	-
lilydemo16	Success	Success	-	Success	-
load_balancer_unreal1_2_10	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal1_2_12	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal1_2_2	Success	Success	-	Success	-
load_balancer_unreal1_2_4	Success	Success	-	Success	-
load_balancer_unreal1_2_6	Success	Success	-	Success	-
load_balancer_unreal1_2_8	Success	Success	-	Success	-
load_balancer_unreal1_4_1	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal1_4_2	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal1_4_3	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal1_4_4	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal1_4_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_unreal1_4_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
load_balancer_unreal2_3	Success	Timeout	DPW1	Timeout	DPW1
load_balancer_unreal2_4	Success	Timeout	DRW_PRODUCT	Timeout	DPW1
load_balancer_unreal2_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba27	Success	Success	-	Success	-
ltl2dba_psi_10	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_psi_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_psi_1	Success	Success	-	Success	-
ltl2dba_psi_2	Success	Success	-	Success	-
ltl2dba_psi_3	Success	Success	-	Success	-
ltl2dba_psi_4	Success	Success	-	Success	-
ltl2dba_psi_5	Success	Success	-	Success	-
ltl2dba_psi_6	Success	Timeout	DPW2	Success	-
ltl2dba_psi_7	Success	Timeout	DPW2	Timeout	DPW2
ltl2dba_psi_8	Success	Timeout	DPW2	Timeout	DPW2
ltl2dba_R_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_R_12	Timeout	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_R_2	Success	Success	-	Success	-
ltl2dba_R_3	Success	Success	-	Success	-
ltl2dba_R_4	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_R_5	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_R_6	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_R_7	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_R_8	Success	Timeout	DRW_PRODUCT	Timeout	DPW1
ltl2dba_theta_10	Success	Timeout	DPW1	Timeout	DPW1
ltl2dba_theta_12	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
ltl2dba_theta_1	Success	Success	-	Success	-
ltl2dba_theta_2	Success	Success	-	Success	-
ltl2dba_theta_3	Success	Success	-	Success	-
ltl2dba_theta_4	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
ltl2dba_theta_5	Success	Timeout	DPW2	Success	-
ltl2dba_theta_6	Success	Timeout	DPW2	Timeout	DPW2
ltl2dba_theta_7	Success	Timeout	DPW2	Timeout	DPW2
ltl2dba_theta_8	Success	Timeout	DPW1	Timeout	DPW2
ltl2dpa25	Success	Timeout	DPW2	Timeout	DPW2
ModdifiedLedMatrix4X	Success	Success	-	Success	-
OneCounterGuiA0	Success	Success	-	Success	-
OneCounterGuiA1	Success	Success	-	Success	-
OneCounterGuiA2	Success	Success	-	Success	-
OneCounterGuiA3	Success	Success	-	Success	-
OneCounterGuiA4	Success	Success	-	Success	-
OneCounterGuiA5	Success	Success	-	Success	-
OneCounterGuiA6	Success	Success	-	Success	-
OneCounterGuiA7	Success	Success	-	Success	-
OneCounterGuiA8	Success	Success	-	Success	-
OneCounterGui	Success	Success	-	Success	-
OneCounterInRangeA0	Success	Success	-	Success	-
OneCounterInRangeA1	Success	Success	-	Success	-
OneCounterInRangeA2	Success	Success	-	Success	-
OneCounterInRange	Success	Success	-	Success	-
prioritized_arbiter_unreal1_3_10	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_unreal1_3_2	Success	Success	-	Success	-
prioritized_arbiter_unreal1_3_4	Success	Success	-	Success	-
prioritized_arbiter_unreal1_3_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_unreal1_3_8	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_unreal2_2	Success	Success	-	Success	-
prioritized_arbiter_unreal2_3	Success	Success	-	Success	-
prioritized_arbiter_unreal2_4	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
prioritized_arbiter_unreal2_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_unreal1_2_12	Success	Timeout	DPW1	Timeout	DPW1
round_robin_arbiter_unreal1_2_15	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_unreal1_2_18	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_unreal1_2_3	Success	Success	-	Success	-
round_robin_arbiter_unreal1_2_6	Success	Success	-	Success	-
round_robin_arbiter_unreal1_2_9	Success	Timeout	DPW2	Success	-
round_robin_arbiter_unreal2_2	Success	Success	-	Success	-
round_robin_arbiter_unreal2_3	Success	Timeout	DPW2	Timeout	DPW2
round_robin_arbiter_unreal2_4	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
round_robin_arbiter_unreal2_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_unreal1_4_1	Success	Success	-	Success	-
simple_arbiter_unreal1_4_2	Success	Success	-	Success	-
simple_arbiter_unreal1_4_3	Success	Success	-	Success	-
simple_arbiter_unreal1_4_4	Success	Timeout	DPW1	Timeout	DPW1
simple_arbiter_unreal1_4_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_unreal1_4_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_unreal2_2	Success	Success	-	Success	-
simple_arbiter_unreal2_3	Success	Success	-	Success	-
simple_arbiter_unreal2_4	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_unreal2_5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
simple_arbiter_unreal2_6	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
TwoCounters2	Success	Success	-	Success	-
TwoCounters5	Success	Timeout	DRW_DCW	Timeout	DRW_DCW
TwoCountersDisButA0	Success	Success	-	Success	-
TwoCountersDisButA1	Success	Success	-	Success	-
TwoCountersDisButA2	Success	Success	-	Success	-
TwoCountersDisButA3	Success	Success	-	Success	-
TwoCountersDisButA4	Success	Success	-	Success	-
TwoCountersDisButA5	Success	Success	-	Success	-
TwoCountersDisButA6	Success	Success	-	Success	-
TwoCountersDisButA7	Success	Success	-	Success	-
TwoCountersDisButA8	Success	Success	-	Success	-
TwoCountersDisButA9	Success	Success	-	Success	-
TwoCountersDisButAC	Success	Timeout	DRW_PRODUCT	Timeout	DRW_DCW
TwoCountersGui	Success	Success	-	Success	-
TwoCountersInRangeA0	Success	Success	-	Success	-
TwoCountersInRangeA1	Success	Success	-	Success	-
TwoCountersInRangeA2	Success	Success	-	Success	-

Specification	Strix	Otus-JBDD	Timeout stage Otus-JBDD	Otus-Sylvan	Timeout stage Otus-Sylvan
TwoCountersInRangeA3	Success	Success	-	Success	-
TwoCountersInRangeA4	Success	Success	-	Success	-
TwoCountersInRangeA5	Success	Success	-	Success	-
TwoCountersInRangeM0	Success	Success	-	Success	-
TwoCountersInRangeM1	Success	Success	-	Success	-
TwoCountersInRangeM2	Success	Success	-	Success	-
TwoCountersInRangeM3	Success	Success	-	Success	-
TwoCountersInRangeM4	Success	Success	-	Success	-
TwoCountersInRangeM5	Success	Success	-	Success	-
TwoCountersInRange	Success	Success	-	Success	-
TwoCountersRefined	Success	Success	-	Success	-
TwoCounters	Success	Success	-	Success	-
UnderapproxDemo2	Success	Success	-	Success	-
UnderapproxDemo	Success	Success	-	Success	-

APPENDIX C

Evaluatory Benchmark Results

This appendix presents the raw results from the evaluatory benchmark. We present the average total execution time in milliseconds, the average circuit size (i.e. number of gates) before and after ABC and the verification results over 5 runs for Strix, Otus-JBDD and Otus-Sylvan. Additionally, we include the relative standard deviation as an indication of the precision of these averages. The data of the detailed execution time analysis is not included for brevity. This data is available on request.

We use $\mathsf{E}x\mathsf{S}y\mathsf{T}z$ to represent the verification results over the five runs where x is the number of verification errors, y is the number of verification successes and z is the number of verification timeouts such that $x+y+z=5$. Zeros are omitted for conciseness. Note that a verification error is an error that prevented verification from completing successfully. It does *not* indicate that the tool generated an erroneous result.

C.1 Realizable Specifications

Column number	Column description
1	Specification
2	Average total execution time Strix (ms)
3	Relative standard deviation total execution time Strix
4	Average circuit size Strix before ABC
5	Relative standard deviation circuit size Strix before ABC
6	Average circuit size Strix after ABC
7	Relative standard deviation circuit size Strix after ABC
8	Verification result Strix
9	Average total execution time Otus-JBDD (ms)
10	Relative standard deviation total execution time Otus-JBDD
11	Average circuit size Otus-JBDD before ABC

Column number	Column description
12	Relative standard deviation circuit size Otus-JBDD before ABC
13	Average circuit size Otus-JBDD after ABC
14	Relative standard deviation circuit size Otus-JBDD after ABC
15	Verification result Otus-JBDD
16	Average total execution time Otus-Sylvan (ms)
17	Relative standard deviation total execution time Otus-Sylvan
18	Average circuit size Otus-Sylvan before ABC
19	Relative standard deviation circuit size Otus-Sylvan before ABC
20	Average circuit size Otus-Sylvan after ABC
21	Relative standard deviation circuit size Otus-Sylvan after ABC
22	Verification result Otus-Sylvan

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ActionConverter	262.60	0.11	4.00	0.00	4.00	0.00	S5	128.60	0.03	31.80	0.01	10.80	0.04	S5	449.00	0.30	31.40	0.02	10.40	0.05	S5
amba_case_study_2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_case_study_3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_case_study_4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_2	2297.40	0.04	21.00	0.00	16.00	0.00	S5	2516.20	0.18	6415.00	0.10	3151.00	0.12	S5	5442.60	0.21	6405.20	0.14	3147.80	0.14	S5
amba_decomposed_arbiter_3	669.80	0.22	273.00	0.00	196.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_4	1439.00	0.10	795.00	0.00	617.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_5	3732.80	0.07	1490.20	0.03	1124.80	0.05	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_6	17857.20	0.08	2164.60	0.06	1496.00	0.05	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_7	58510.40	0.00	3169.60	0.09	2048.20	0.06	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_arbiter_8	284075.60	0.00	4389.40	0.04	2598.00	0.06	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
amba_decomposed_decode	157.20	0.01	3.00	0.00	3.00	0.00	S5	53.80	0.14	13.80	0.03	7.80	0.05	S5	237.60	0.37	14.00	0.00	8.00	0.00	S5
amba_decomposed_encode_10	9813.60	0.01	118.00	0.00	81.00	0.00	S5	859.20	0.14	617.20	0.04	381.40	0.04	S5	1954.60	0.22	611.40	0.05	376.60	0.05	S5
amba_decomposed_encode_12	5746.20	0.02	197.00	0.00	166.00	0.00	S5	1830.80	0.03	704.40	0.09	439.00	0.10	S5	3587.60	0.15	718.00	0.04	444.40	0.02	S5
amba_decomposed_encode_2	149.40	0.11	7.00	0.00	7.00	0.00	S5	64.20	0.11	61.20	0.22	27.20	0.22	S5	183.40	0.15	53.80	0.31	24.20	0.29	S5
amba_decomposed_encode_4	263.80	0.09	24.00	0.00	23.00	0.00	S5	127.80	0.17	130.00	0.11	69.40	0.10	S5	286.20	0.15	145.80	0.03	78.00	0.07	S5
amba_decomposed_encode_6	491.40	0.03	58.00	0.00	46.00	0.00	S5	229.40	0.16	284.60	0.05	150.40	0.11	S5	461.40	0.13	283.20	0.06	152.40	0.13	S5
amba_decomposed_encode_8	1922.40	0.06	60.00	0.00	57.00	0.00	S5	299.40	0.07	367.60	0.07	225.00	0.12	S5	725.00	0.22	375.60	0.04	237.00	0.07	S5
amba_decomposed_lock_10	37965.20	0.01	80.00	0.00	68.00	0.00	S5	1453.80	0.59	506.80	0.27	281.20	0.25	S5	1819.80	0.82	447.40	0.26	238.60	0.29	S5
amba_decomposed_lock_12	-	-	-	-	-	-	-	597.40	0.10	611.00	0.26	293.60	0.29	S5	17548.00	1.16	618.80	0.16	337.80	0.16	S5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
amba_decomposed_lock_2	166.80	0.10	16.00	0.00	13.00	0.00	S5	66.20	0.08	96.20	0.14	44.80	0.12	S5	201.80	0.22	84.80	0.14	33.00	0.14	S5
amba_decomposed_lock_4	348.40	0.08	32.00	0.00	27.00	0.00	S5	111.40	0.14	174.60	0.15	107.00	0.11	S5	336.40	0.14	166.40	0.15	97.00	0.20	S5
amba_decomposed_lock_6	4157.80	0.13	48.00	0.00	40.00	0.00	S5	165.80	0.15	230.80	0.23	138.00	0.22	S5	390.60	0.13	228.00	0.16	140.40	0.24	S5
amba_decomposed_lock_8	2739.80	0.02	64.00	0.00	54.00	0.00	S5	319.40	0.40	421.80	0.24	251.20	0.25	S5	521.40	0.31	384.00	0.12	208.60	0.10	S5
amba_decomposed_shift	162.00	0.08	3.00	0.00	3.00	0.00	S5	41.60	0.08	21.60	0.16	12.00	0.17	S5	146.80	0.04	23.20	0.21	13.60	0.27	S5
amba_decomposed_tburst4	708.20	0.00	35.00	0.00	34.00	0.00	S5	214.40	0.16	191.80	0.10	72.00	0.07	S5	470.60	0.11	188.60	0.09	72.20	0.11	S5
amba_decomposed_tincr	468.80	0.02	17.00	0.00	13.00	0.00	S5	185.60	0.04	319.20	0.11	127.60	0.08	S5	332.20	0.03	304.00	0.10	110.20	0.05	S5
amba_decomposed_tsingle	551.80	0.04	16.00	0.00	14.00	0.00	S5	115.40	0.15	103.00	0.19	41.60	0.08	S5	306.00	0.08	120.20	0.21	43.40	0.16	S5
Automata16S	1655.00	0.03	94.00	0.00	59.00	0.00	S5	1155.00	0.07	295.80	0.03	139.80	0.09	S5	1784.80	0.06	275.80	0.03	130.60	0.05	S5
Automata32S	4195.20	0.02	142.00	0.00	95.00	0.00	S5	3728.40	0.03	493.80	0.05	248.80	0.06	S5	5006.00	0.03	468.20	0.05	237.00	0.05	S5
Automata	1033.60	0.12	62.00	0.00	44.00	0.00	S5	491.80	0.14	155.40	0.04	77.00	0.07	S5	615.60	0.07	161.80	0.11	78.00	0.07	S5
button	154.00	0.15	0.00	0.00	0.00	0.00	S5	48.80	0.04	11.80	0.03	4.80	0.08	S5	154.80	0.03	11.60	0.04	4.60	0.11	S5
Button	148.40	0.06	0.00	0.00	0.00	0.00	S5	66.60	0.01	11.60	0.04	4.60	0.11	S5	211.00	0.06	11.60	0.04	4.60	0.11	S5
Cockpitboard	620.40	0.02	13.00	0.00	7.00	0.00	S5	465.80	0.00	48.80	0.01	13.80	0.03	S5	613.00	0.02	48.80	0.01	13.80	0.03	S5
collector_v1_2	154.00	0.08	16.00	0.00	13.00	0.00	S5	73.40	0.01	81.20	0.06	47.60	0.07	S5	226.20	0.03	87.40	0.03	52.20	0.06	S5
collector_v1_3	197.80	0.02	45.00	0.00	31.00	0.00	S5	105.40	0.01	260.80	0.17	159.00	0.21	S5	280.00	0.03	290.40	0.14	180.20	0.19	S5
collector_v1_4	1451.40	0.09	136.40	0.39	92.00	0.35	S5	188.60	0.02	1200.00	0.06	746.80	0.06	S5	352.80	0.15	1168.20	0.11	728.20	0.10	S5
collector_v1_5	23327.20	0.02	296.00	0.00	168.00	0.00	S5	301.20	0.01	4292.00	0.17	2642.80	0.18	S5	566.00	0.04	4398.60	0.10	2702.20	0.11	S5
collector_v1_6	1838.80	0.06	438.00	0.00	236.00	0.00	S5	1192.00	0.05	18377.60	0.14	10721.20	0.14	T5	1413.40	0.11	19060.20	0.13	11077.60	0.13	T5
collector_v1_7	9880.20	0.01	523.00	0.00	349.00	0.00	S5	4247.00	0.02	78323.80	0.07	43243.00	0.06	T5	4307.40	0.02	82895.40	0.10	45170.00	0.08	T5
collector_v2_2	164.60	0.18	3.00	0.00	3.00	0.00	S5	191.60	0.03	670.20	0.03	304.20	0.05	S5	373.20	0.02	659.80	0.02	297.20	0.04	S5
collector_v2_3	181.20	0.09	9.00	0.00	8.00	0.00	S5	2291.40	0.05	6614.80	0.02	2292.40	0.02	S5	2754.40	0.04	6733.60	0.04	2365.20	0.05	S5
collector_v2_4	246.00	0.02	29.00	0.00	23.00	0.00	S5	38636.60	0.03	62889.20	0.05	18562.00	0.05	T5	30567.80	0.01	62099.00	0.03	18170.40	0.03	T5
collector_v2_5	1227.40	0.10	42.00	0.00	26.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
collector_v2_6	7233.40	0.02	82.00	0.00	53.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
collector_v2_7	23902.40	0.01	86.00	0.00	63.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
collector_v3_2	167.80	0.15	4.00	0.00	2.00	0.00	S5	65.40	0.02	131.40	0.18	70.80	0.23	S5	186.80	0.03	148.80	0.07	81.40	0.13	S5
collector_v3_3	204.80	0.08	3.00	0.00	3.00	0.00	S5	196.60	0.01	624.00	0.11	384.20	0.12	S5	452.80	0.02	662.60	0.10	402.40	0.10	S5
collector_v3_4	227.20	0.17	4.00	0.00	4.00	0.00	S5	587.20	0.13	2913.00	0.07	1755.40	0.06	S5	944.00	0.05	3207.60	0.06	1959.80	0.06	S5
collector_v3_5	233.80	0.04	5.00	0.00	5.00	0.00	S5	1952.40	0.05	16809.00	0.04	10080.00	0.05	S5	3085.00	0.12	16215.80	0.04	9734.80	0.04	S5
collector_v3_6	269.80	0.02	6.00	0.00	6.00	0.00	S5	9642.60	0.02	87798.60	0.06	50582.20	0.06	T5	14564.80	0.09	85370.80	0.04	49303.20	0.05	T5
collector_v3_7	350.80	0.05	7.00	0.00	7.00	0.00	S5	66712.40	0.01	467241.40	0.04	262383.20	0.04	T5	80419.40	0.01	459520.00	0.04	258004.20	0.04	T5
collector_v4_2	161.20	0.09	13.00	0.00	12.00	0.00	S5	223.80	0.17	897.60	0.04	389.20	0.06	S5	443.00	0.16	864.40	0.05	388.80	0.04	S5
collector_v4_3	273.20	0.03	57.00	0.00	42.00	0.00	S5	1334.80	0.04	5776.20	0.03	1870.40	0.04	S5	1541.40	0.03	5787.40	0.04	1896.60	0.07	S5
collector_v4_4	1200.80	0.02	195.00	0.00	146.00	0.00	S5	12116.80	0.04	36009.20	0.02	9535.40	0.02	S5	9641.80	0.02	35729.20	0.04	9327.80	0.03	S5
collector_v4_5	543.60	0.11	114.00	1.00	77.00	1.01	S5	116468.20	0.04	219502.00	0.02	48125.20	0.01	T5	66779.00	0.01	219776.40	0.05	48419.80	0.04	T5
collector_v4_6	2671.00	0.06	317.60	0.63	213.60	0.59	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
collector_v4_7	17168.80	0.01	89.00	0.00	59.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
detector_10	453.00	0.15	173.00	0.00	68.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
detector_12	1255.40	0.06	191.00	0.00	97.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
detector_1	124.00	0.16	0.00	0.00	0.00	0.00	S5	37.80	0.05	25.00	0.15	12.40	0.16	S5	147.00	0.04	25.40	0.14	12.40	0.16	S5
detector_2	132.80	0.13	3.00	0.00	3.00	0.00	S5	98.80	0.03	231.80	0.03	107.00	0.03	S5	287.80	0.18	223.00	0.08	107.00	0.05	S5
detector_3	131.80	0.04	9.00	0.00	8.00	0.00	S5	680.20	0.01	1381.20	0.06	657.80	0.07	S5	1104.20	0.05	1343.60	0.06	644.20	0.08	S5
detector_4	141.60	0.15	26.00	0.00	17.00	0.00	S5	4676.80	0.03	8283.60	0.02	3811.40	0.03	S5	4873.40	0.01	8414.20	0.03	3889.60	0.05	S5
detector_5	145.60	0.08	34.00	0.00	25.00	0.00	S5	46592.00	0.07	49901.20	0.05	22467.00	0.04	T5	30433.00	0.01	51785.80	0.02	23149.40	0.03	T5
detector_6	153.60	0.06	52.00	0.00	40.00	0.00	S5	-	-	-	-	-	-	-	200009.60	0.01	297880.00	0.03	129234.40	0.03	T5
detector_7	146.80	0.15	83.00	0.00	51.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
detector_8	155.20	0.09	93.00	0.00	54.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
EnemeyModule	166.40	0.11	2.00	0.00	2.00	0.00	S5	94.80	0.14	18.80	0.02	6.80	0.06	S5	196.00	0.06	18.40	0.03	6.40	0.08	S5
escalator_bidirectional_init	510.80	0.01	91.00	0.00	51.00	0.00	S5	331.60	0.02	267.60	0.05	110.00	0.08	S5	692.60	0.01	267.20	0.04	113.20	0.07	S5
EscalatorBidirectionalInit	505.60	0.02	102.00	0.00	56.00	0.00	S5	328.00	0.01	237.00	0.01	91.60	0.08	S5	675.60	0.02	234.40	0.02	91.40	0.08	S5
escalator_bidirectional	463.20	0.01	49.00	0.00	34.00	0.00	S5	294.20	0.01	220.00	0.03	108.00	0.06	S5	589.00	0.02	225.80	0.04	109.40	0.03	S5
EscalatorBidirectional	472.60	0.03	60.00	0.00	38.00	0.00	S5	293.60	0.01	226.80	0.04	106.80	0.05	S5	590.80	0.03	233.20	0.04	110.80	0.06	S5
escalator_counting_init	301.80	0.02	21.00	0.00	17.00	0.00	S5	202.60	0.01	64.40	0.05	34.80	0.07	S5	362.80	0.02	64.60	0.04	31.80	0.06	S5
EscalatorCountingInit	310.20	0.04	21.00	0.00	17.00	0.00	S5	203.20	0.01	55.40	0.03	32.20	0.04	S5	363.60	0.02	54.40	0.03	33.40	0.04	S5
escalator_counting	238.60	0.01	9.00	0.00	8.00	0.00	S5	143.40	0.01	49.00	0.03	25.80	0.07	S5	295.40	0.02	48.80	0.04	24.20	0.09	S5
EscalatorCounting	240.80	0.03	9.00	0.00	8.00	0.00	S5	144.40	0.02	39.00	0.04	20.80	0.10	S5	295.60	0.02	37.80	0.01	20.40	0.02	S5
escalator_non-counting	159.80	0.02	3.00	0.00	3.00	0.00	S5	83.20	0.03	15.60	0.03	9.60	0.05	S5	219.40	0.02	15.80	0.03	9.80	0.04	S5
EscalatorNonCounting	173.80	0.10	3.00	0.00	3.00	0.00	S5	83.60	0.03	15.40	0.03	9.40	0.05	S5	217.40	0.03	15.60	0.03	9.60	0.05	S5
escalator_non-reactive	152.40	0.13	0.00	0.00	0.00	0.00	S5	55.80	0.04	7.60	0.06	2.60	0.19	S5	191.40	0.03	7.60	0.06	2.60	0.19	S5
EscalatorNonReactive	148.40	0.05	0.00	0.00	0.00	0.00	S5	55.80	0.04	7.40	0.07	2.40	0.20	S5	193.20	0.07	7.60	0.06	2.60	0.19	S5
escalator_smart	932.60	0.01	197.00	0.00	165.00	0.00	S5	-	-	-	-	-	-	-	246853.40	0.01	67067.60	0.02	13422.00	0.02	S5
EscalatorSmart	810.40	0.18	174.00	0.01	144.80	0.01	S5	-	-	-	-	-	-	-	242234.40	0.01	67712.80	0.01	13485.80	0.02	S5
full_arbiter_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_2	251.60	0.10	41.00	0.00	34.00	0.00	S5	1533.40	0.05	4326.80	0.04	2033.40	0.03	S5	1763.20	0.06	4404.40	0.03	2040.00	0.02	S5
full_arbiter_3	1787.60	0.02	162.00	0.00	119.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_4	460.80	0.07	308.00	0.00	188.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_5	1970.60	0.08	599.00	0.00	352.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_6	11886.80	0.01	875.00	0.00	490.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_7	109454.40	0.00	1584.00	0.00	1040.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_8	90083.00	0.00	2132.00	0.00	1340.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_enc_10	222543.00	0.01	295596.00	0.00	259508.00	0.00	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_enc_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_enc_2	395.60	0.11	36.00	0.00	25.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_enc_4	1107.20	0.02	475.60	0.03	401.80	0.05	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_enc_6	4975.60	0.04	4333.00	0.00	3710.00	0.00	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
full_arbiter_enc_8	87307.40	0.03	8547.20	0.02	5128.20	0.02	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Gamelogic	1796.80	0.03	51.00	0.00	34.00	0.00	S5	866.40	0.05	145.00	0.04	55.00	0.12	S5	1111.00	0.11	148.00	0.03	59.80	0.14	S5
GamemodeChooser	675.20	0.12	59.00	0.00	38.00	0.00	S5	191.80	0.10	217.40	0.02	112.80	0.03	S5	402.80	0.04	220.00	0.03	114.80	0.03	S5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Gamemodule	425.40	0.14	6.00	0.00	6.00	0.00	S5	142.00	0.01	55.60	0.05	22.00	0.03	S5	291.60	0.03	55.60	0.10	21.40	0.04	S5
genbuf2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
genbuf3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
genbuf4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
genbuf5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
generalized_buffer_2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
generalized_buffer_3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
generalized_buffer_4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
generalized_buffer_5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
generalized_buffer_6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
increment	218.40	0.22	0.00	0.00	0.00	0.00	S5	58.00	0.04	9.40	0.05	4.40	0.11	S5	199.80	0.10	9.80	0.04	4.80	0.08	S5
Increment	215.80	0.08	0.00	0.00	0.00	0.00	S5	57.80	0.03	9.60	0.05	4.60	0.11	S5	191.00	0.04	9.40	0.05	4.40	0.11	S5
KitchenTimerV0	203.20	0.13	1.00	0.00	1.00	0.00	S5	58.20	0.04	13.80	0.03	5.80	0.07	S5	194.20	0.01	13.80	0.03	5.80	0.07	S5
KitchenTimerV10	2078.60	0.02	1167.00	0.00	989.00	0.00	S5	1708.80	0.02	3993.00	0.02	2412.20	0.02	T5	2674.00	0.06	3886.80	0.03	2369.80	0.04	S2T3
KitchenTimerV1	293.20	0.04	41.00	0.00	25.00	0.00	S5	176.00	0.36	56.00	0.09	27.80	0.09	S5	245.60	0.03	58.40	0.07	29.00	0.11	S5
KitchenTimerV2	496.60	0.14	1.00	0.00	1.00	0.00	S5	379.60	0.02	505.60	0.17	311.80	0.14	S5	615.00	0.02	457.80	0.14	282.80	0.13	S5
KitchenTimerV3	516.20	0.02	1.00	0.00	1.00	0.00	S5	404.20	0.23	1568.00	0.04	940.80	0.04	S5	762.40	0.17	1479.00	0.10	892.00	0.09	S5
KitchenTimerV4	701.00	0.11	1.00	0.00	1.00	0.00	S5	446.40	0.02	1764.60	0.05	1063.60	0.05	S5	794.20	0.03	1822.00	0.04	1093.20	0.04	S5
KitchenTimerV5	1039.60	0.01	848.80	0.03	720.00	0.04	S5	734.80	0.18	3425.00	0.06	2041.80	0.05	S5	1352.80	0.05	3440.20	0.03	2074.00	0.03	S5
KitchenTimerV6	1564.40	0.14	1237.60	0.16	1062.60	0.18	S5	1131.00	0.07	4569.60	0.02	2750.40	0.02	S1T4	1849.80	0.04	4657.40	0.02	2791.00	0.02	S1T4
KitchenTimerV7	1780.60	0.04	1242.00	0.17	1071.60	0.18	S5	1235.20	0.04	4583.40	0.03	2759.40	0.03	T5	1930.20	0.05	4681.60	0.02	2832.40	0.02	T5
KitchenTimerV8	2051.40	0.13	815.00	0.03	680.00	0.02	S5	1300.60	0.05	4648.00	0.02	2819.80	0.02	S1T4	2544.80	0.04	4581.40	0.04	2743.00	0.04	S1T4
KitchenTimerV9	2183.80	0.09	1030.20	0.02	839.60	0.00	S5	1693.80	0.03	5615.40	0.03	3397.40	0.03	T5	3008.40	0.03	5411.20	0.02	3213.20	0.04	T5
LedMatrix	225084.60	0.00	134.00	0.00	99.00	0.00	S5	13563.80	0.03	12699.40	0.02	7325.60	0.02	T5	30131.80	0.01	12471.40	0.03	7170.20	0.03	T5
lilydemo03	179.00	0.05	13.00	0.00	10.00	0.00	S5	82.20	0.15	353.20	0.08	210.80	0.08	S5	200.60	0.04	343.40	0.11	206.20	0.13	S5
lilydemo04	181.00	0.05	29.00	0.00	26.00	0.00	S5	129.20	0.02	446.80	0.02	272.40	0.02	S5	319.60	0.04	470.40	0.07	287.20	0.08	S5
lilydemo05	200.60	0.07	5.00	0.00	5.00	0.00	S5	131.60	0.03	364.40	0.05	225.20	0.04	S5	264.60	0.08	345.20	0.10	218.80	0.10	S5
lilydemo06	217.00	0.03	7.00	0.00	5.00	0.00	S5	156.60	0.01	491.20	0.06	304.20	0.06	S5	372.60	0.01	530.60	0.07	323.20	0.06	S5
lilydemo07	195.00	0.04	5.00	0.00	5.00	0.00	S5	103.00	0.14	206.00	0.11	119.40	0.11	S5	233.20	0.06	209.60	0.07	115.20	0.08	S5
lilydemo08	167.00	0.07	0.00	0.00	0.00	0.00	S5	48.20	0.05	9.00	0.00	4.00	0.00	S5	192.60	0.02	9.00	0.00	4.00	0.00	S5
lilydemo09	164.20	0.05	5.00	0.00	5.00	0.00	S5	113.20	0.68	40.40	0.12	18.60	0.14	S5	235.60	0.03	38.40	0.21	19.00	0.21	S5
lilydemo10	150.60	0.12	0.00	0.00	0.00	0.00	S5	56.60	0.04	40.80	0.09	11.40	0.04	S5	205.40	0.03	42.00	0.10	11.60	0.04	S5
lilydemo12	147.40	0.07	0.00	0.00	0.00	0.00	S5	61.40	0.03	75.60	0.15	41.00	0.20	S5	213.60	0.02	80.80	0.15	39.80	0.18	S5
lilydemo13	151.00	0.07	0.00	0.00	0.00	0.00	S5	51.00	0.03	10.60	0.13	4.20	0.23	S5	193.40	0.03	11.80	0.16	4.80	0.38	S5
lilydemo14	136.60	0.07	0.00	0.00	0.00	0.00	S5	120.40	0.48	117.20	0.08	42.20	0.10	S5	267.40	0.04	119.80	0.03	41.00	0.12	S5
lilydemo17	150.40	0.06	3.00	0.00	3.00	0.00	S5	259.40	0.02	592.60	0.04	185.00	0.05	S5	446.60	0.02	613.20	0.08	195.80	0.11	S5
lilydemo18	188.20	0.09	5.00	0.00	5.00	0.00	S5	25824.20	0.03	17340.80	0.02	6041.60	0.03	S5	21679.40	0.01	16978.20	0.04	5818.80	0.03	S5
lilydemo19	176.00	0.12	1.00	0.00	1.00	0.00	S5	82.00	0.03	174.60	0.10	82.40	0.13	S5	220.80	0.03	184.00	0.11	92.80	0.10	S5
lilydemo20	220.40	0.01	8.00	0.00	6.00	0.00	S5	268.80	0.01	698.80	0.36	401.60	0.35	S5	607.40	0.33	789.60	0.21	465.00	0.22	S5
lilydemo21	358.20	0.02	21.00	0.00	20.00	0.00	S5	552.60	0.16	1921.60	0.01	1153.40	0.01	S5	1184.80	0.05	1900.60	0.02	1132.60	0.02	S5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
lilydemo22	1258.80	0.01	26.00	0.00	23.00	0.00	S5	123.60	0.02	561.80	0.06	292.40	0.08	S5	270.80	0.02	534.80	0.04	280.40	0.02	S5
lilydemo23	158.20	0.05	1.00	0.00	1.00	0.00	S5	69.20	0.04	51.60	0.15	24.80	0.17	S5	179.00	0.07	49.40	0.13	22.20	0.15	S5
lilydemo24	212.00	0.01	0.00	0.00	0.00	0.00	S5	176.60	0.12	407.60	0.22	220.80	0.21	E5	383.80	0.06	411.40	0.16	228.00	0.17	E5
load_balancer_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_2	288.80	0.03	30.00	0.00	19.00	0.00	S5	2139.80	0.03	5055.80	0.03	2462.60	0.02	S5	2893.20	0.02	4817.60	0.06	2388.00	0.07	S5
load_balancer_3	1283.80	0.10	78.40	0.02	47.80	0.03	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_4	6913.60	0.02	216.00	0.00	113.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_5	4195.20	0.02	293.00	0.00	174.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_6	36886.60	0.01	441.00	0.00	241.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_7	60324.00	0.01	597.00	0.00	369.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
load_balancer_unreal2_2	286.80	0.14	30.00	0.00	19.00	0.00	S5	7118.20	0.05	15387.00	0.07	7187.20	0.07	S5	8933.60	0.01	15573.60	0.12	7323.20	0.12	S5
loadcomp2	244.00	0.07	3.00	0.00	3.00	0.00	S5	1318.00	0.04	3100.20	0.16	1412.60	0.18	S5	2641.60	0.03	3148.40	0.16	1408.20	0.16	S5
loadcomp3	289.80	0.02	33.00	0.00	20.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loadcomp4	365.00	0.13	64.00	0.00	50.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loadcomp5	604.80	0.06	124.00	0.00	88.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loadfull2	255.40	0.08	27.00	0.00	17.00	0.00	S5	6399.40	0.22	20315.60	0.22	9560.60	0.22	S5	8953.80	0.02	18221.40	0.16	8661.40	0.16	S5
loadfull3	368.60	0.03	88.00	0.09	61.20	0.04	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loadfull4	639.00	0.09	203.00	0.00	125.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
loadfull5	3060.00	0.10	378.00	0.00	256.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba01	136.00	0.04	12.00	0.00	8.00	0.00	S5	44.20	0.03	44.80	0.02	25.00	0.07	S5	150.00	0.03	45.20	0.02	25.40	0.05	S5
ltl2dba02	192.80	0.05	136.00	0.00	111.00	0.00	S5	103.80	0.18	469.60	0.04	276.20	0.06	S5	280.00	0.14	498.20	0.03	298.80	0.03	S5
ltl2dba03	156.00	0.05	8.00	0.00	6.00	0.00	S5	59.80	0.12	107.00	0.10	50.00	0.19	S5	168.00	0.05	108.80	0.10	55.80	0.17	S5
ltl2dba04	140.60	0.06	9.00	0.00	8.00	0.00	S5	65.60	0.11	130.40	0.06	49.80	0.11	S5	214.20	0.09	146.80	0.09	56.00	0.16	S5
ltl2dba05	147.00	0.10	30.00	0.00	26.40	0.02	S5	74.20	0.04	162.40	0.16	66.60	0.17	S5	221.80	0.04	141.40	0.12	61.00	0.14	S5
ltl2dba06	147.80	0.06	23.00	0.00	17.00	0.00	S5	91.00	0.03	156.20	0.23	47.80	0.16	S5	232.40	0.02	152.20	0.10	52.20	0.13	S5
ltl2dba07	342.40	0.05	62.00	0.00	46.00	0.00	S5	572.80	0.01	2654.60	0.06	913.40	0.19	S5	421.40	0.01	2831.60	0.21	929.40	0.33	S5
ltl2dba08	197.60	0.50	34.00	0.00	25.00	0.00	S5	49301.80	0.05	51302.20	0.04	22966.80	0.04	T5	30308.20	0.01	50722.60	0.04	22916.40	0.04	T5
ltl2dba09	150.00	0.19	2.00	0.00	2.00	0.00	S5	109.60	0.08	218.20	0.24	108.60	0.15	S5	349.60	0.01	224.40	0.22	120.60	0.19	S5
ltl2dba10	157.60	0.33	2.00	0.00	2.00	0.00	S5	50.60	0.19	77.80	0.22	33.40	0.24	E5	210.20	0.04	84.40	0.13	34.60	0.14	E5
ltl2dba11	151.80	0.05	0.00	0.00	0.00	0.00	S5	80.80	0.04	150.00	0.09	64.20	0.11	S5	249.20	0.04	169.80	0.04	70.80	0.05	S5
ltl2dba12	149.60	0.11	5.00	0.00	5.00	0.00	S5	67.40	0.05	69.40	0.04	39.40	0.06	S5	214.00	0.03	64.40	0.06	33.60	0.18	S5
ltl2dba13	165.00	0.07	16.00	0.00	11.00	0.00	S5	121.20	0.02	355.40	0.02	159.80	0.02	S5	315.20	0.02	336.00	0.03	150.20	0.05	S5
ltl2dba14	135.00	0.05	9.00	0.00	7.00	0.00	S5	76.80	0.04	129.00	0.11	57.40	0.16	S5	243.00	0.04	145.80	0.06	59.60	0.07	S5
ltl2dba15	136.20	0.10	1.00	0.00	1.00	0.00	S5	79.00	0.04	77.20	0.04	42.00	0.03	S5	258.00	0.04	86.80	0.04	47.80	0.12	S5
ltl2dba16	138.80	0.06	13.00	0.00	11.00	0.00	S5	141.00	0.04	247.60	0.07	115.60	0.04	S5	298.60	0.02	253.60	0.03	118.40	0.04	S5
ltl2dba17	139.00	0.10	28.00	0.00	23.00	0.00	S5	292.00	0.01	1657.60	0.05	965.40	0.05	S5	572.80	0.02	1581.40	0.09	908.00	0.10	S5
ltl2dba18	140.20	0.04	10.00	0.00	8.00	0.00	S5	132.40	0.06	471.00	0.04	217.20	0.03	S5	260.80	0.02	475.80	0.07	214.20	0.08	S5
ltl2dba19	136.20	0.05	14.00	0.00	9.00	0.00	S5	101.00	0.04	278.20	0.05	148.60	0.05	S5	260.20	0.03	276.60	0.04	156.60	0.04	S5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ltl2dba20	144.40	0.07	31.00	0.00	27.00	0.00	S5	268.00	0.01	865.80	0.05	388.40	0.03	S5	453.80	0.02	863.60	0.03	385.80	0.04	S5
ltl2dba21	146.60	0.05	124.00	0.03	106.80	0.02	S5	688.80	0.04	6067.60	0.05	3484.60	0.05	S5	1076.60	0.02	6106.00	0.04	3467.40	0.04	S5
ltl2dba22	146.00	0.17	1.00	0.00	1.00	0.00	S5	38.40	0.05	25.40	0.05	12.40	0.04	S5	157.00	0.06	23.00	0.15	11.40	0.09	S5
ltl2dba23	136.00	0.06	8.00	0.00	7.00	0.00	S5	48.20	0.14	57.80	0.17	28.80	0.20	S5	190.20	0.08	58.40	0.15	32.00	0.17	S5
ltl2dba24	138.00	0.09	1.00	0.00	1.00	0.00	S5	51.40	0.04	23.40	0.09	11.40	0.07	S5	190.60	0.03	22.40	0.10	11.00	0.08	S5
ltl2dba25	135.80	0.06	10.00	0.00	8.00	0.00	S5	89.60	0.03	145.20	0.05	76.80	0.08	S5	265.80	0.03	166.60	0.05	90.80	0.11	S5
ltl2dba26	145.60	0.10	17.80	0.02	14.60	0.05	S5	76.00	0.04	161.40	0.14	60.00	0.17	S5	236.80	0.10	168.60	0.12	64.00	0.05	S5
ltl2dba_alpha_10	205.60	0.06	33.00	0.00	26.00	0.00	S5	190.20	0.02	228.00	0.05	122.20	0.03	S5	354.60	0.04	231.00	0.05	124.40	0.04	S5
ltl2dba_alpha_12	207.60	0.05	33.00	0.00	26.00	0.00	S5	153.60	0.01	270.40	0.01	148.00	0.01	S5	290.80	0.05	272.80	0.02	148.80	0.01	S5
ltl2dba_alpha_1	135.60	0.14	6.00	0.00	6.00	0.00	S5	48.60	0.18	40.20	0.04	20.40	0.07	S5	150.60	0.04	37.40	0.12	18.40	0.12	S5
ltl2dba_alpha_2	136.40	0.06	10.00	0.00	7.00	0.00	S5	61.60	0.19	72.00	0.06	32.40	0.10	S5	184.60	0.07	70.80	0.05	32.60	0.07	S5
ltl2dba_alpha_3	157.40	0.08	11.00	0.00	11.00	0.00	S5	87.40	0.10	85.40	0.07	43.20	0.11	S5	233.00	0.02	87.00	0.04	42.20	0.07	S5
ltl2dba_alpha_4	156.00	0.06	15.00	0.00	15.00	0.00	S5	101.20	0.09	101.00	0.09	52.80	0.09	S5	260.40	0.13	110.00	0.05	56.80	0.04	S5
ltl2dba_alpha_5	161.80	0.04	21.00	0.00	17.00	0.00	S5	113.20	0.08	122.40	0.05	63.20	0.03	S5	243.00	0.09	124.40	0.06	66.00	0.06	S5
ltl2dba_alpha_6	161.40	0.06	20.00	0.00	17.00	0.00	S5	106.80	0.23	153.80	0.07	78.00	0.08	S5	244.40	0.10	157.00	0.01	77.20	0.05	S5
ltl2dba_alpha_7	166.20	0.09	18.00	0.00	17.00	0.00	S5	99.60	0.09	167.00	0.06	86.40	0.08	S5	216.80	0.04	172.40	0.04	93.20	0.04	S5
ltl2dba_alpha_8	161.40	0.21	23.00	0.00	22.00	0.00	S5	108.60	0.09	195.20	0.03	104.40	0.02	S5	224.60	0.03	196.00	0.04	102.00	0.04	S5
ltl2dba_beta_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_beta_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_beta_1	109.80	0.13	6.00	0.00	3.00	0.00	S5	47.20	0.19	69.60	0.07	40.60	0.10	S5	155.20	0.07	66.00	0.06	36.20	0.07	S5
ltl2dba_beta_2	141.60	0.03	34.00	0.00	18.00	0.00	S5	110.00	0.17	496.80	0.03	261.00	0.06	S5	249.00	0.23	506.20	0.07	263.80	0.09	S5
ltl2dba_beta_3	198.60	0.04	40.00	0.00	22.00	0.00	S5	326.80	0.11	2612.80	0.06	1411.60	0.09	S5	298.80	0.02	2639.20	0.03	1412.00	0.05	S5
ltl2dba_beta_4	2245.80	0.06	107.00	0.07	57.20	0.10	S5	1899.80	0.08	13691.40	0.05	7467.80	0.08	T5	674.40	0.01	13812.40	0.06	7515.00	0.05	T5
ltl2dba_beta_5	14821.00	0.05	137.80	0.16	85.40	0.23	S5	12490.40	0.02	63142.80	0.04	33264.20	0.04	T5	2640.40	0.02	65763.40	0.04	36576.80	0.03	T5
ltl2dba_beta_6	1046.80	0.12	219.80	0.14	133.60	0.27	S5	92195.40	0.03	300494.20	0.04	163594.60	0.07	T5	14003.20	0.03	320883.00	0.04	170840.60	0.05	T5
ltl2dba_beta_7	3933.80	0.04	157.00	0.00	102.00	0.00	S5	-	-	-	-	-	-	-	86880.00	0.02	1490836.60	0.02	837493.80	0.01	T5
ltl2dba_beta_8	17504.00	0.02	301.80	0.09	188.60	0.17	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C1_10	155.80	0.11	9.00	0.00	9.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C1_12	193.20	0.12	11.00	0.00	11.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C1_1	131.00	0.07	0.00	0.00	0.00	0.00	S5	40.80	0.09	24.20	0.16	11.60	0.20	S5	154.00	0.04	23.00	0.05	11.20	0.09	S5
ltl2dba_C1_2	131.40	0.06	1.00	0.00	1.00	0.00	S5	74.20	0.04	78.60	0.10	43.80	0.08	S5	197.60	0.13	83.80	0.09	47.40	0.08	S5
ltl2dba_C1_3	145.20	0.10	2.00	0.00	2.00	0.00	S5	162.20	0.05	223.40	0.26	112.60	0.24	S5	494.20	0.02	272.20	0.19	137.20	0.17	S5
ltl2dba_C1_4	128.80	0.01	3.00	0.00	3.00	0.00	S5	637.80	0.06	500.00	0.33	248.20	0.30	S5	1451.20	0.04	580.20	0.31	268.00	0.29	S5
ltl2dba_C1_5	142.00	0.03	4.00	0.00	4.00	0.00	S5	2589.80	0.10	1431.60	0.31	655.20	0.31	S5	6616.00	0.01	736.20	0.32	337.40	0.16	S5
ltl2dba_C1_6	155.00	0.12	5.00	0.00	5.00	0.00	S5	24000.40	0.19	2210.60	0.21	863.80	0.20	S5	35729.20	0.01	3106.00	0.28	1261.20	0.32	S5
ltl2dba_C1_7	146.00	0.06	6.00	0.00	6.00	0.00	S5	167987.00	0.04	5165.60	0.52	1873.80	0.55	S5	212842.40	0.01	6397.40	0.36	2465.20	0.36	S5
ltl2dba_C1_8	139.80	0.02	7.00	0.00	7.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C2_10	534.60	0.01	173.00	0.00	68.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C2_12	1203.80	0.02	191.00	0.00	97.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C2_1	113.20	0.15	0.00	0.00	0.00	0.00	S5	38.60	0.03	25.00	0.15	12.40	0.16	S5	144.80	0.02	25.40	0.14	12.40	0.16	S5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ltl2dba_C2_2	122.60	0.13	3.00	0.00	3.00	0.00	S5	109.20	0.15	233.60	0.04	109.00	0.03	S5	330.80	0.06	228.60	0.02	107.60	0.05	S5
ltl2dba_C2_3	136.20	0.08	9.00	0.00	8.00	0.00	S5	685.80	0.01	1409.00	0.07	652.20	0.10	S5	1127.60	0.04	1446.60	0.03	685.80	0.02	S5
ltl2dba_C2_4	143.20	0.07	26.00	0.00	17.00	0.00	S5	4676.00	0.04	8429.80	0.05	3914.00	0.04	S5	4803.20	0.01	8323.00	0.04	3844.60	0.03	S5
ltl2dba_C2_5	151.20	0.09	34.00	0.00	25.00	0.00	S5	47926.20	0.08	50727.40	0.04	22868.80	0.04	T5	30481.00	0.01	52179.00	0.03	23291.00	0.04	T5
ltl2dba_C2_6	158.60	0.05	52.00	0.00	40.00	0.00	S5	-	-	-	-	-	-	-	200471.00	0.00	300539.60	0.02	130495.40	0.02	T5
ltl2dba_C2_7	138.80	0.22	83.00	0.00	51.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_C2_8	159.40	0.19	93.00	0.00	54.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_E_10	25918.20	0.01	46.00	0.00	19.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_E_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_E_1	125.80	0.18	1.00	0.00	1.00	0.00	S5	40.80	0.08	25.00	0.00	12.00	0.00	S5	143.00	0.03	22.80	0.09	11.20	0.07	S5
ltl2dba_E_2	124.80	0.07	6.00	0.00	3.00	0.00	S5	51.40	0.14	62.20	0.03	35.20	0.04	S5	199.40	0.07	70.00	0.07	41.80	0.10	S5
ltl2dba_E_3	142.40	0.10	11.00	0.00	5.00	0.00	S5	98.00	0.03	234.80	0.04	127.20	0.06	S5	240.60	0.03	253.40	0.11	132.00	0.12	S5
ltl2dba_E_4	142.00	0.08	16.00	0.00	7.00	0.00	S5	206.60	0.01	989.40	0.06	563.20	0.06	S5	336.80	0.03	1014.40	0.06	594.20	0.05	S5
ltl2dba_E_5	162.00	0.11	21.00	0.00	9.00	0.00	S5	474.80	0.03	4223.60	0.04	2392.80	0.04	S5	465.80	0.01	4076.40	0.03	2320.00	0.04	S5
ltl2dba_E_6	221.60	0.06	26.00	0.00	11.00	0.00	S5	2324.80	0.00	17427.80	0.03	10066.60	0.04	T5	1308.60	0.04	17811.40	0.01	10366.60	0.02	T5
ltl2dba_E_7	503.40	0.01	31.00	0.00	13.00	0.00	S5	13546.40	0.02	75989.80	0.02	42429.60	0.02	T5	4696.20	0.01	74395.40	0.02	41471.00	0.03	T5
ltl2dba_E_8	1800.40	0.06	36.00	0.00	15.00	0.00	S5	94209.80	0.04	330236.20	0.02	178622.00	0.03	T5	21471.80	0.01	331651.80	0.01	177679.40	0.01	T5
ltl2dba_Q_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_Q_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_Q_1	143.80	0.24	1.00	0.00	1.00	0.00	S5	39.40	0.03	24.00	0.08	11.60	0.07	S5	152.60	0.09	23.40	0.09	11.40	0.07	S5
ltl2dba_Q_2	133.20	0.13	9.00	0.00	5.00	0.00	S5	78.00	0.02	142.80	0.04	80.00	0.08	S5	220.60	0.07	135.00	0.07	74.20	0.07	S5
ltl2dba_Q_3	147.40	0.06	72.00	0.00	34.00	0.00	S5	164.80	0.02	737.20	0.05	430.80	0.06	S5	340.60	0.01	678.80	0.03	402.80	0.02	S5
ltl2dba_Q_4	348.60	0.02	151.00	0.00	101.00	0.00	S5	438.00	0.01	4198.40	0.04	2447.20	0.04	S5	543.40	0.02	4085.60	0.03	2440.80	0.04	S5
ltl2dba_Q_5	2880.00	0.03	251.00	0.00	151.00	0.00	S5	3834.80	0.03	25852.60	0.03	14979.40	0.03	T5	2078.40	0.06	26026.80	0.02	15285.60	0.03	T5
ltl2dba_Q_6	9852.60	0.00	370.00	0.00	207.00	0.00	S5	31749.80	0.01	161833.60	0.02	89108.40	0.03	T5	10875.80	0.01	163684.60	0.01	91219.60	0.02	T5
ltl2dba_Q_7	114020.20	0.01	496.00	0.00	260.00	0.00	S5	-	-	-	-	-	-	-	83681.00	0.00	969029.60	0.01	544049.00	0.01	T5
ltl2dba_Q_8	6766.40	0.04	639.00	0.00	326.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_R_1	126.00	0.10	0.00	0.00	0.00	0.00	S5	40.20	0.03	25.40	0.14	12.40	0.16	S5	153.20	0.03	25.00	0.15	12.40	0.16	S5
ltl2dba_S_10	21261.40	0.00	46.00	0.00	19.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_S_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_S_1	134.40	0.12	1.00	0.00	1.00	0.00	S5	47.40	0.12	24.40	0.16	12.20	0.10	S5	166.00	0.12	22.20	0.12	11.00	0.06	S5
ltl2dba_S_2	132.20	0.13	6.00	0.00	3.00	0.00	S5	67.80	0.02	81.20	0.14	43.60	0.14	S5	203.60	0.02	84.80	0.08	47.80	0.07	S5
ltl2dba_S_3	132.80	0.10	11.00	0.00	5.00	0.00	S5	104.00	0.01	327.20	0.06	198.20	0.07	S5	253.20	0.02	325.20	0.04	189.80	0.03	S5
ltl2dba_S_4	143.60	0.03	16.00	0.00	7.00	0.00	S5	248.00	0.02	1190.60	0.04	698.00	0.05	S5	388.60	0.02	1176.40	0.06	682.60	0.06	S5
ltl2dba_S_5	212.00	0.04	21.00	0.00	9.00	0.00	S5	615.80	0.01	4882.40	0.03	2746.80	0.03	S5	626.60	0.02	4907.80	0.03	2808.80	0.03	S5
ltl2dba_S_6	568.60	0.01	26.00	0.00	11.00	0.00	S5	3144.80	0.04	19288.00	0.02	10910.20	0.03	T5	1854.80	0.02	19512.00	0.02	11370.20	0.03	T5
ltl2dba_S_7	2141.80	0.05	31.00	0.00	13.00	0.00	S5	20275.60	0.03	82348.40	0.02	45789.20	0.03	T5	6820.80	0.01	82541.40	0.02	47331.60	0.02	T5
ltl2dba_S_8	10624.20	0.01	36.00	0.00	15.00	0.00	S5	118172.00	0.01	355863.60	0.01	192042.20	0.02	T5	29788.60	0.01	354804.00	0.00	191667.20	0.02	T5
ltl2dba_U1_10	5778.60	0.03	12041.20	0.02	9738.80	0.02	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ltl2dba_U1_12	97992.00	0.02	27601.00	0.00	23201.00	0.00	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ltl2dba_U1_1	125.00	0.13	5.00	0.00	4.00	0.00	S5	39.20	0.19	32.80	0.14	16.60	0.11	S5	153.20	0.13	32.80	0.14	16.40	0.11	S5
ltl2dba_U1_2	131.60	0.09	6.00	0.00	5.00	0.00	S5	52.40	0.05	39.80	0.06	19.80	0.08	S5	187.20	0.07	40.00	0.04	19.60	0.05	S5
ltl2dba_U1_3	133.20	0.08	26.00	0.00	24.00	0.00	S5	68.40	0.01	132.20	0.16	67.40	0.12	S5	211.60	0.01	120.60	0.19	57.80	0.22	S5
ltl2dba_U1_4	151.20	0.05	80.80	0.03	64.20	0.01	S5	121.20	0.02	486.80	0.03	255.00	0.07	S5	290.20	0.01	474.80	0.03	242.80	0.02	S5
ltl2dba_U1_5	240.60	0.02	227.60	0.01	196.00	0.02	S5	247.40	0.19	1744.40	0.05	962.40	0.07	S5	408.00	0.02	1654.60	0.04	934.80	0.03	S5
ltl2dba_U1_6	821.80	0.08	647.40	0.01	542.40	0.01	S5	786.40	0.07	6730.80	0.05	3847.20	0.05	T5	1148.80	0.02	6852.40	0.04	3932.40	0.06	T5
ltl2dba_U1_7	5335.20	0.01	1803.20	0.03	1530.00	0.03	T5	3666.80	0.06	27871.00	0.03	15729.20	0.02	T5	4492.20	0.02	26500.40	0.03	15194.80	0.05	T5
ltl2dba_U1_8	43236.40	0.02	3950.00	0.00	3241.00	0.00	T5	15366.20	0.02	114879.00	0.02	66162.60	0.03	T5	18144.60	0.01	113663.40	0.03	64983.20	0.02	T5
ltl2dba_U2_10	1626.80	0.22	164.00	0.01	95.00	0.25	S5	1660.20	0.04	4187.00	0.15	1690.40	0.20	S5	1313.20	0.11	4510.40	0.05	1881.60	0.08	S5
ltl2dba_U2_12	1511.00	0.36	198.60	0.00	149.80	0.12	S5	7004.20	0.02	9558.60	0.05	3744.20	0.02	T5	5908.40	0.00	9533.60	0.10	3844.80	0.10	T5
ltl2dba_U2_1	103.20	0.08	5.00	0.00	4.00	0.00	S5	36.40	0.02	30.80	0.18	15.80	0.15	S5	136.60	0.02	35.00	0.00	17.80	0.07	S5
ltl2dba_U2_2	125.80	0.14	6.00	0.00	5.00	0.00	S5	42.40	0.09	37.40	0.15	20.60	0.13	S5	162.80	0.10	37.20	0.15	19.40	0.13	S5
ltl2dba_U2_3	131.80	0.06	14.00	0.00	9.00	0.00	S5	58.60	0.08	92.00	0.11	39.40	0.09	S5	205.40	0.03	92.00	0.16	42.60	0.20	S5
ltl2dba_U2_4	148.20	0.18	25.00	0.00	19.00	0.00	S5	74.60	0.02	174.20	0.07	70.00	0.11	S5	216.80	0.01	178.20	0.15	65.80	0.15	S5
ltl2dba_U2_5	148.40	0.07	40.60	0.12	33.00	0.12	S5	97.40	0.01	343.40	0.05	138.20	0.15	S5	242.20	0.01	308.80	0.10	107.20	0.14	S5
ltl2dba_U2_6	160.20	0.06	61.00	0.00	47.00	0.00	S5	136.20	0.03	556.80	0.19	245.80	0.16	S5	277.20	0.02	541.20	0.10	225.80	0.15	S5
ltl2dba_U2_7	203.00	0.02	75.60	0.09	48.60	0.22	S5	176.60	0.09	960.40	0.06	408.20	0.06	S5	263.40	0.03	836.80	0.10	344.80	0.15	S5
ltl2dba_U2_8	467.20	0.09	93.00	0.00	68.00	0.00	S5	353.20	0.15	1793.00	0.04	770.60	0.08	S5	386.60	0.01	1595.00	0.07	685.00	0.06	S5
ltl2dpa01	158.80	0.09	0.00	0.00	0.00	0.00	S5	341.00	0.27	317.20	0.05	106.40	0.08	S5	489.00	0.01	299.00	0.05	104.40	0.09	S5
ltl2dpa02	139.00	0.08	4.00	0.00	2.00	0.00	S5	70.60	0.01	83.60	0.12	34.80	0.14	S5	215.60	0.02	89.80	0.28	36.60	0.29	S5
ltl2dpa03	174.80	0.02	11.00	0.00	6.00	0.00	S5	157763.20	0.03	28897.00	0.03	8542.20	0.04	S5	89869.00	0.01	29652.20	0.06	8794.40	0.07	S5
ltl2dpa04	136.40	0.09	4.00	0.00	2.00	0.00	S5	46.00	0.02	33.60	0.08	15.40	0.05	S5	147.80	0.01	36.60	0.06	15.20	0.06	S5
ltl2dpa05	143.80	0.06	1.00	0.00	1.00	0.00	S5	89.60	0.16	140.40	0.10	76.00	0.08	S5	278.40	0.02	148.20	0.09	84.20	0.11	S5
ltl2dpa06	151.40	0.15	3.00	0.00	3.00	0.00	S5	64.60	0.02	37.40	0.04	13.80	0.05	S5	209.20	0.02	37.80	0.05	14.00	0.08	S5
ltl2dpa07	139.80	0.03	5.00	0.00	3.00	0.00	S5	74.40	0.02	119.20	0.19	48.00	0.24	S5	230.40	0.04	116.00	0.22	41.40	0.30	S5
ltl2dpa08	143.60	0.11	8.00	0.00	8.00	0.00	S5	72.00	0.03	64.20	0.08	35.00	0.14	S5	220.60	0.01	57.60	0.06	34.00	0.08	S5
ltl2dpa09	147.20	0.10	5.00	0.00	3.00	0.00	S5	74.00	0.01	121.80	0.18	38.00	0.22	S5	226.60	0.01	117.80	0.13	35.80	0.17	S5
ltl2dpa10	177.60	0.08	2.00	0.00	2.00	0.00	S5	6980.60	0.01	3877.80	0.02	1133.20	0.02	S5	6436.20	0.00	3929.00	0.02	1146.00	0.01	S5
ltl2dpa11	139.60	0.08	9.00	0.00	5.00	0.00	S5	447.80	0.13	739.00	0.05	395.40	0.07	S5	742.00	0.03	724.60	0.02	382.20	0.04	S5
ltl2dpa12	184.20	0.02	2.00	0.00	2.00	0.00	S5	13648.00	0.02	3412.00	0.03	1354.60	0.03	S5	10988.60	0.01	3416.20	0.02	1351.20	0.02	S5
ltl2dpa13	200.40	0.07	6.00	0.00	6.00	0.00	S5	9231.80	0.01	3711.40	0.06	1164.40	0.06	S5	7687.20	0.01	3823.20	0.04	1182.40	0.03	S5
ltl2dpa14	167.60	0.03	0.00	0.00	0.00	0.00	S5	903.40	0.07	865.80	0.03	322.80	0.03	S5	1035.40	0.01	855.40	0.05	310.60	0.06	S5
ltl2dpa15	133.00	0.11	3.00	0.00	3.00	0.00	S5	223.00	0.02	359.80	0.06	154.20	0.07	S5	397.80	0.00	344.40	0.02	145.80	0.03	S5
ltl2dpa16	144.00	0.06	6.00	0.00	3.00	0.00	S5	77.00	0.02	81.80	0.08	44.20	0.07	E5	217.20	0.01	80.80	0.21	41.00	0.23	E5
ltl2dpa17	172.20	0.04	12.00	0.00	12.00	0.00	S5	97.80	0.02	99.40	0.05	55.80	0.06	S5	242.60	0.01	98.40	0.09	51.60	0.09	S5
ltl2dpa18	173.80	0.07	12.00	0.00	12.00	0.00	S5	127.20	0.02	133.00	0.07	64.80	0.12	S5	279.20	0.01	131.40	0.05	64.20	0.05	S5
ltl2dpa19	200.60	0.05	40.00	0.00	22.00	0.00	S5	7368.20	0.02	7039.80	0.06	1767.80	0.05	S5	5272.00	0.01	6762.60	0.06	1665.80	0.06	S5
ltl2dpa20	156.00	0.04	6.00	0.00	3.00	0.00	S5	109.60	0.03	253.80	0.10	118.60	0.06	S5	232.00	0.01	256.00	0.08	115.60	0.08	S5
ltl2dpa21	178.40	0.09	2.00	0.00	2.00	0.00	S5	6932.00	0.02	3892.60	0.02	1157.60	0.02	S5	6449.00	0.02	3876.80	0.03	1153.60	0.03	S5
ltl2dpa22	1075.60	0.39	56.80	0.26	34.20	0.19	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ltl2dpa23	150.20	0.10	1.00	0.00	1.00	0.00	S5	87.80	0.02	150.40	0.09	83.00	0.10	S5	231.40	0.03	153.60	0.14	85.40	0.10	S5
ltl2dpa24	158.20	0.09	1.00	0.00	1.00	0.00	S5	193.00	0.01	254.40	0.03	121.40	0.06	S5	400.40	0.01	261.20	0.09	122.00	0.09	S5
ModdifiedLedMatrix5X	2688.20	0.05	101.00	0.00	70.00	0.00	S5	5522.60	0.04	2771.40	0.04	1501.40	0.04	S5	9491.00	0.01	2663.00	0.04	1435.40	0.07	S5
music_app_feedback	276.00	0.04	41.00	0.00	32.00	0.00	S5	156.40	0.16	183.40	0.14	102.80	0.11	S5	274.60	0.02	193.40	0.09	107.80	0.10	S5
MusicAppFeedback	309.20	0.00	41.00	0.00	32.00	0.00	S5	259.80	0.01	229.60	0.02	130.20	0.04	S5	459.80	0.03	219.00	0.14	123.20	0.14	S5
music_app_motivating_2	1065.60	0.01	67.00	0.00	53.00	0.00	S5	250.20	0.04	460.60	0.08	269.80	0.08	S5	520.00	0.01	438.20	0.03	256.40	0.04	S5
music_app_motivating	1044.80	0.04	67.00	0.00	53.00	0.00	S5	201.00	0.15	449.60	0.09	262.60	0.07	S5	420.40	0.17	410.00	0.09	247.20	0.11	S5
MusicAppMotivating	1055.40	0.01	67.00	0.00	53.00	0.00	S5	201.40	0.14	468.00	0.09	268.20	0.11	S5	442.00	0.12	501.40	0.05	293.00	0.05	S5
music_app_simple	278.40	0.07	4.00	0.00	4.00	0.00	S5	148.80	0.13	64.60	0.09	30.00	0.11	S5	246.80	0.15	59.60	0.07	27.00	0.26	S5
MusicAppSimple	295.20	0.02	15.00	0.00	12.00	0.00	S5	213.40	0.01	121.40	0.07	67.60	0.10	S5	380.80	0.00	111.80	0.11	61.60	0.13	S5
mux_10	405.80	0.04	0.00	0.00	0.00	0.00	S5	305.80	0.01	5.20	0.28	1.20	1.22	E5	433.20	0.00	4.60	0.26	0.60	2.00	E5
mux_12	501.40	0.00	0.00	0.00	0.00	0.00	S5	410.80	0.01	6.40	0.19	2.40	0.50	E5	537.60	0.01	5.20	0.28	1.20	1.22	E5
mux_2	186.80	0.04	0.00	0.00	0.00	0.00	S5	60.80	0.01	5.20	0.28	1.20	1.22	E5	192.20	0.03	5.80	0.25	1.80	0.82	E5
mux_4	192.60	0.11	0.00	0.00	0.00	0.00	S5	95.40	0.01	5.20	0.28	1.20	1.22	E5	221.80	0.01	5.80	0.25	1.80	0.82	E5
mux_6	259.20	0.08	0.00	0.00	0.00	0.00	S5	146.60	0.01	6.40	0.19	2.40	0.50	E5	275.00	0.00	7.00	0.00	3.00	0.00	E5
mux_8	306.80	0.01	0.00	0.00	0.00	0.00	S5	215.20	0.00	5.80	0.25	1.80	0.82	E5	344.60	0.01	5.80	0.25	1.80	0.82	E5
narylatch_10	-	-	-	-	-	-	-	149217.00	0.89	9233.80	0.00	6029.40	0.00	T5	-	-	-	-	-	-	-
narylatch_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
narylatch_2	188.40	0.08	15.00	0.00	15.00	0.00	S5	62.00	0.01	50.20	0.04	31.80	0.11	S5	227.80	0.01	49.60	0.06	32.40	0.05	S5
narylatch_4	311.20	0.01	81.00	0.00	59.00	0.00	S5	140.60	0.17	189.00	0.01	123.20	0.03	S5	475.00	0.03	187.20	0.06	122.00	0.06	S5
narylatch_6	3219.20	0.02	153.00	0.00	115.00	0.00	S5	401.20	0.19	659.00	0.02	447.60	0.02	S5	2816.20	0.02	687.60	0.03	461.60	0.03	S5
narylatch_8	14342.00	0.02	230.00	0.00	151.00	0.00	S5	3995.40	0.02	2586.20	0.02	1694.20	0.01	T5	45789.60	0.02	2603.20	0.01	1696.80	0.01	S3T2
OneCounterGuiA9	6501.80	0.03	43.00	0.00	28.00	0.00	S5	4501.60	0.04	2297.00	0.04	1372.00	0.03	S5	8953.00	0.02	2399.20	0.02	1424.80	0.04	S5
OneCounterInRangeA3	226.00	0.04	12.00	0.00	9.00	0.00	S5	87.00	0.03	70.60	0.08	34.80	0.19	S5	195.00	0.04	69.00	0.08	31.00	0.12	S5
OneCounter	6655.00	0.01	43.00	0.00	28.00	0.00	S5	4541.00	0.02	2343.00	0.04	1412.80	0.04	S5	8798.40	0.10	2374.00	0.04	1441.60	0.03	S5
prioritized_arbiter_10	1248.20	0.04	52.00	0.00	38.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
prioritized_arbiter_12	2641.60	0.08	55.00	0.00	38.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
prioritized_arbiter_1	132.60	0.13	1.00	0.00	1.00	0.00	S5	64.40	0.03	86.60	0.15	37.20	0.16	S5	190.00	0.02	82.60	0.14	35.20	0.12	S5
prioritized_arbiter_2	170.40	0.04	5.00	0.00	5.00	0.00	S5	184.20	0.01	620.60	0.07	310.80	0.08	S5	410.40	0.01	632.60	0.05	299.20	0.08	S5
prioritized_arbiter_3	219.40	0.09	14.00	0.00	14.00	0.00	S5	400.40	0.16	3785.60	0.03	2064.20	0.04	S5	732.00	0.11	3669.80	0.02	2030.00	0.02	S5
prioritized_arbiter_4	272.00	0.04	11.00	0.00	11.00	0.00	S5	2647.40	0.03	23060.20	0.01	12708.00	0.02	T5	2392.40	0.01	23006.20	0.01	12587.20	0.01	T5
prioritized_arbiter_5	247.20	0.07	29.00	0.00	24.00	0.00	S5	37158.40	0.00	134847.60	0.00	75585.80	0.01	T5	22837.80	0.02	134453.20	0.00	75300.00	0.01	T5
prioritized_arbiter_6	315.00	0.11	30.00	0.00	22.00	0.00	S5	-	-	-	-	-	-	-	117532.00	0.01	796797.00	0.00	476244.00	0.00	T5
prioritized_arbiter_7	469.40	0.15	38.00	0.00	29.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
prioritized_arbiter_8	600.60	0.17	21.00	0.00	21.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
prioritized_arbiter_enc_10	183735.80	0.01	37775.20	0.01	27757.60	0.01	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
prioritized_arbiter_enc_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
prioritized_arbiter_enc_2	200.00	0.04	93.00	0.00	74.00	0.00	S5	1455.80	0.04	7329.60	0.04	4320.00	0.05	S5	1724.80	0.02	7233.00	0.03	4217.40	0.01	S5
prioritized_arbiter_enc_4	3608.60	0.04	1003.00	0.08	731.80	0.07	S5	10401.00	0.01	30685.60	0.00	17883.80	0.01	T5	12073.40	0.01	30709.20	0.01	17918.20	0.01	T5
prioritized_arbiter_enc_6	66159.60	0.07	2825.40	0.04	1945.00	0.03	S5	-	-	-	-	-	-	-	149823.40	0.00	316177.60	0.00	175765.80	0.00	T5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
prioritized_arbiter_enc_8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Radarboard	585.60	0.17	9.00	0.00	6.00	0.00	S5	394.20	0.05	50.00	0.00	13.00	0.00	S5	550.00	0.07	49.80	0.01	12.80	0.03	S5
RegManager	243.60	0.06	0.00	0.00	0.00	0.00	S5	158.60	0.01	18.00	0.00	7.00	0.00	S5	296.60	0.01	17.60	0.03	6.60	0.07	S5
RotationCalculator	273.20	0.01	19.00	0.00	18.00	0.00	S5	174.00	0.01	89.60	0.03	43.40	0.13	S5	347.80	0.02	93.60	0.06	50.00	0.12	S5
round_robin_arbiter_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_2	222.20	0.14	1.00	0.00	1.00	0.00	S5	5987.60	0.01	7118.40	0.03	2402.40	0.02	S5	5889.80	0.01	7110.20	0.03	2393.20	0.02	S5
round_robin_arbiter_3	439.00	0.35	25.40	0.33	18.80	0.38	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_4	1974.00	1.49	126.60	0.08	103.00	0.05	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_5	4084.60	0.14	596.20	0.02	509.00	0.02	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_6	38887.40	0.01	1419.60	0.01	1121.80	0.05	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_7	227274.00	0.02	4415.80	0.04	3591.40	0.04	T5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
round_robin_arbiter_8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Scoreboard	245.80	0.07	4.00	0.00	4.00	0.00	S5	186.20	0.16	32.80	0.01	10.80	0.04	S5	277.60	0.01	32.40	0.02	10.40	0.05	S5
SensorInit	1770.20	0.02	49.00	0.00	36.00	0.00	S5	896.00	0.06	164.40	0.09	87.00	0.05	S5	1002.60	0.01	151.20	0.05	77.80	0.09	S5
SensorPart	1273.80	0.06	44.00	0.00	34.00	0.00	S5	896.20	0.03	154.60	0.09	73.00	0.13	S5	1184.00	0.07	148.20	0.06	71.00	0.10	S5
SensorRegister	135.80	0.08	0.00	0.00	0.00	0.00	S5	47.80	0.15	9.40	0.05	4.40	0.11	S5	151.80	0.02	9.60	0.05	4.60	0.11	S5
SensorSelector	4051.00	0.01	16.00	0.00	16.00	0.00	S5	1144.60	0.02	139.20	0.03	78.00	0.03	S5	11786.60	0.01	127.80	0.02	72.00	0.02	S5
SensorSubmodulChooser	540.40	0.05	37.00	0.00	29.00	0.00	S5	813.40	0.02	136.20	0.05	81.60	0.10	S5	1755.60	0.01	144.20	0.07	89.80	0.08	S5
Sensor	2452.20	0.27	155.60	0.76	110.20	0.67	S5	70706.00	0.07	11976.40	0.06	5388.80	0.06	S5	26915.00	0.03	12548.20	0.04	5457.00	0.07	S5
shift_10	760.20	0.12	0.00	0.00	0.00	0.00	S5	111.20	0.01	25.60	0.02	12.60	0.04	S5	574.20	0.06	25.60	0.02	12.60	0.04	S5
shift_12	7452.20	0.04	0.00	0.00	0.00	0.00	S5	441.00	0.00	29.20	0.01	14.20	0.03	S5	2150.00	0.03	29.40	0.02	14.40	0.03	S5
shift_2	146.60	0.07	0.00	0.00	0.00	0.00	S5	49.80	0.03	9.80	0.04	4.80	0.08	S5	188.20	0.02	9.40	0.05	4.40	0.11	S5
shift_4	136.00	0.04	0.00	0.00	0.00	0.00	S5	57.80	0.02	13.40	0.04	6.40	0.08	S5	203.00	0.05	13.80	0.03	6.80	0.06	S5
shift_6	155.00	0.10	0.00	0.00	0.00	0.00	S5	68.80	0.02	18.00	0.00	9.00	0.00	S5	238.60	0.04	17.60	0.03	8.60	0.06	S5
shift_8	216.20	0.04	0.00	0.00	0.00	0.00	S5	90.00	0.03	22.00	0.00	11.00	0.00	S5	351.40	0.02	21.60	0.02	10.60	0.05	S5
simple_arbiter_10	502.20	0.03	31.00	0.00	26.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
simple_arbiter_12	487.80	0.14	35.00	0.00	29.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
simple_arbiter_2	115.80	0.14	0.00	0.00	0.00	0.00	S5	66.40	0.02	69.60	0.16	32.20	0.15	S5	213.00	0.04	66.40	0.24	31.00	0.38	S5
simple_arbiter_3	164.40	0.20	3.00	0.00	3.00	0.00	S5	106.00	0.02	580.40	0.05	305.20	0.09	S5	290.00	0.03	606.20	0.09	331.00	0.06	S5
simple_arbiter_4	188.60	0.06	5.00	0.00	5.00	0.00	S5	270.20	0.01	3642.80	0.03	2169.20	0.04	S5	573.00	0.11	3571.40	0.04	2137.80	0.05	S5
simple_arbiter_5	219.80	0.06	9.00	0.00	9.00	0.00	S5	892.60	0.07	22842.00	0.01	13085.40	0.02	T5	1684.00	0.04	23079.40	0.02	13332.20	0.01	T5
simple_arbiter_6	261.00	0.07	13.00	0.00	13.00	0.00	S5	7544.60	0.03	139255.40	0.00	79049.40	0.01	T5	8592.20	0.01	139299.60	0.01	79138.00	0.01	T5
simple_arbiter_7	292.40	0.02	17.00	0.00	16.00	0.00	S5	61461.20	0.09	834849.20	0.00	497305.40	0.00	T5	48937.00	0.01	836509.40	0.00	501740.20	0.00	T5
simple_arbiter_8	344.20	0.04	15.00	0.00	15.00	0.00	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
simple_arbiter_enc_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
simple_arbiter_enc_12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
simple_arbiter_enc_2	127.00	0.09	11.00	0.00	10.00	0.00	S5	289.60	0.81	1228.40	0.05	753.40	0.05	S5	422.40	0.48	1252.40	0.06	784.20	0.05	S5
simple_arbiter_enc_4	643.60	0.03	298.00	0.00	244.00	0.00	S5	1042.20	0.08	7980.60	0.01	4693.80	0.01	T5	2194.40	0.02	7874.60	0.01	4570.60	0.02	T5
simple_arbiter_enc_6	25382.20	0.04	1734.60	0.02	1225.20	0.03	S5	17358.20	0.01	114991.20	0.00	68118.60	0.01	T5	35978.60	0.01	115035.60	0.00	68255.20	0.01	T5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
simple_arbiter_enc_8	34933.60	0.00	4946.20	0.02	3362.00	0.05	S5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
slider_default	215.60	0.12	7.00	0.00	6.00	0.00	S5	161.00	0.01	943.40	0.07	583.60	0.05	S5	382.60	0.16	1077.20	0.09	661.20	0.08	S5
SliderDefault	230.80	0.01	7.00	0.00	6.00	0.00	S5	207.00	0.17	1091.40	0.06	657.20	0.07	S5	385.80	0.16	976.80	0.11	596.40	0.12	S5
slider_delayed	396.00	0.01	76.00	0.00	66.00	0.00	S5	703.00	0.05	5562.20	0.04	3358.60	0.04	S5	1844.00	0.08	5375.20	0.09	3247.00	0.09	S5
SliderDelayed	394.80	0.01	91.00	0.00	76.00	0.00	S5	663.40	0.05	4406.40	0.09	2687.20	0.09	S5	1890.00	0.07	4464.40	0.23	2736.20	0.21	S5
slider_scored	442.80	0.01	37.00	0.00	30.00	0.00	S5	327.00	0.18	1285.20	0.12	795.40	0.13	S5	1136.60	0.05	1371.80	0.02	829.80	0.02	S5
SliderScored	444.80	0.01	40.00	0.00	32.00	0.00	S5	308.40	0.06	1217.40	0.20	732.60	0.18	S5	1162.60	0.03	1323.40	0.05	803.20	0.05	S5
SPIPureNext	2924.20	0.02	181.00	0.00	131.00	0.00	T5	3654.80	0.02	1755.20	0.01	1105.20	0.01	T5	6778.00	0.07	1760.20	0.02	1104.40	0.02	T5
SPIReadClk	407.60	0.02	2.00	0.00	2.00	0.00	S5	112.40	0.07	19.00	0.00	7.00	0.00	S5	484.80	0.09	18.40	0.03	6.40	0.08	S5
SPIReadManag	3228.40	0.07	37.00	0.00	26.00	0.00	S5	513.60	0.02	590.60	0.22	266.80	0.17	S5	2055.60	0.03	551.00	0.19	251.20	0.17	S5
SPIReadSdi	429.60	0.07	7.00	0.00	3.00	0.00	S5	118.00	0.08	62.80	0.01	23.20	0.06	S5	453.40	0.15	62.60	0.03	22.80	0.12	S5
SPI	1051.20	0.18	95.00	0.00	53.00	0.00	S5	539.80	0.03	285.80	0.06	118.80	0.12	S5	3759.40	0.05	289.80	0.05	126.80	0.08	S5
SPIWriteClk	402.80	0.08	6.00	0.00	4.00	0.00	S5	114.80	0.04	30.80	0.01	8.80	0.05	S5	420.60	0.20	30.80	0.01	8.80	0.05	S5
SPIWriteManag	893.40	0.08	8.00	0.00	6.00	0.00	S5	292.60	0.13	108.40	0.11	42.60	0.21	S5	1101.00	0.08	96.40	0.09	35.80	0.08	S5
SPIWriteSdi	516.20	0.08	13.00	0.00	13.00	0.00	S5	160.00	0.05	78.60	0.03	33.80	0.08	S5	570.00	0.13	80.40	0.05	36.60	0.18	S5
tictactoe	19764.40	0.02	0.00	0.00	0.00	0.00	S5	2634.40	0.01	49.60	0.30	21.20	0.13	S5	3905.60	0.03	57.20	0.40	23.40	0.04	S5
torcs_accelerating	348.60	0.10	0.00	0.00	0.00	0.00	S5	114.60	0.09	15.60	0.03	6.60	0.07	S5	487.40	0.09	15.80	0.03	6.80	0.06	S5
TorcsAccelerating	343.40	0.06	0.00	0.00	0.00	0.00	S5	101.60	0.16	15.80	0.03	6.80	0.06	S5	515.60	0.12	15.60	0.03	6.60	0.07	S5
torcs_gearing	402.00	0.19	2.00	0.00	2.00	0.00	S5	93.80	0.12	26.40	0.04	14.20	0.08	S5	440.00	0.17	26.00	0.02	14.00	0.06	S5
TorcsGearing	377.40	0.15	2.00	0.00	2.00	0.00	S5	95.00	0.11	25.80	0.02	13.80	0.03	S5	443.40	0.13	26.40	0.03	14.40	0.06	S5
torcs_simple	523.20	0.18	4.00	0.00	4.00	0.00	S5	150.40	0.12	56.80	0.05	29.20	0.09	S5	556.00	0.18	61.20	0.03	30.80	0.10	S5
TorcsSimple	614.60	0.02	4.00	0.00	4.00	0.00	S5	154.00	0.28	59.60	0.04	28.80	0.09	S5	537.80	0.19	60.40	0.05	30.80	0.10	S5
torcs_steering_improved	695.20	0.06	17.00	0.00	13.00	0.00	S5	300.20	0.12	196.00	0.07	101.60	0.13	S5	872.20	0.12	219.40	0.08	116.40	0.19	S5
TorcsSteeringImproved	663.60	0.10	20.00	0.00	17.00	0.00	S5	319.20	0.03	283.20	0.11	131.40	0.25	S5	845.20	0.06	310.80	0.09	169.00	0.10	S5
torcs_steering_simple	577.40	0.01	10.00	0.00	7.00	0.00	S5	128.80	0.08	66.20	0.05	32.60	0.24	S5	500.80	0.15	63.80	0.08	26.80	0.28	S5
TorcsSteeringSimple	575.60	0.01	10.00	0.00	7.00	0.00	S5	122.20	0.05	67.80	0.07	27.00	0.16	S5	408.40	0.15	67.60	0.08	32.00	0.23	S5
torcs_steering_smart	614.40	0.01	72.00	0.00	57.00	0.00	S5	266.00	0.01	357.60	0.06	176.80	0.15	S5	715.40	0.12	325.00	0.07	150.00	0.06	S5
TorcsSteeringSmart	601.60	0.06	72.00	0.00	57.00	0.00	S5	269.40	0.02	340.00	0.09	154.80	0.10	S5	712.80	0.10	322.60	0.06	153.60	0.06	S5
TwoCounters3	4333.00	0.23	115.20	0.05	88.60	0.06	S5	403.40	0.03	324.20	0.01	91.60	0.10	S5	1147.40	0.03	318.60	0.01	83.80	0.19	S5
TwoCounters4	2442.80	0.13	45.00	0.00	37.00	0.00	S5	422.60	0.00	252.80	0.03	128.00	0.01	S5	2278.80	0.07	261.60	0.02	133.00	0.05	S5
TwoCountersInRangeA6	2411.00	0.10	45.00	0.00	37.00	0.00	S5	395.80	0.14	252.40	0.04	129.40	0.05	S5	2229.20	0.05	251.60	0.03	128.80	0.04	S5
TwoCountersRefinedRefined	827.40	0.11	31.00	0.00	22.00	0.00	S5	272.00	0.01	310.40	0.03	169.20	0.05	S5	894.40	0.08	308.40	0.02	169.60	0.06	S5
UnderapproxStrengthenedDemo	211.80	0.13	0.00	0.00	0.00	0.00	S5	52.20	0.01	2.00	0.00	0.00	0.00	E5	329.00	0.26	2.00	0.00	0.00	0.00	E5
zoo0	402.60	0.11	11.00	0.00	10.00	0.00	S5	247.00	0.02	168.60	0.04	99.60	0.08	S5	788.00	0.11	176.40	0.08	104.80	0.08	S5
Zoo0	385.60	0.07	9.00	0.00	7.00	0.00	S5	235.80	0.15	160.80	0.11	97.60	0.10	S5	776.60	0.14	137.80	0.07	81.60	0.09	S5
zoo10	391.40	0.10	6.00	0.00	6.00	0.00	S5	215.80	0.02	163.00	0.07	91.80	0.18	S5	663.20	0.06	170.00	0.06	96.00	0.10	S5
Zoo10	395.80	0.15	7.00	0.00	6.00	0.00	S5	201.60	0.12	161.60	0.14	97.00	0.15	S5	713.00	0.16	149.00	0.21	89.00	0.19	S5
zoo5	383.60	0.13	7.00	0.00	7.00	0.00	S5	215.20	0.02	154.20	0.04	84.00	0.13	S5	720.20	0.12	153.60	0.08	84.20	0.14	S5
Zoo5	393.00	0.09	9.00	0.00	8.00	0.00	S5	215.60	0.02	140.00	0.15	84.40	0.14	S5	741.40	0.14	146.00	0.13	85.60	0.11	S5

C.2 Unrealizable Specifications

Specification	Avg. time Strix (ms)	Rel. std. dev. time Strix	Avg. time Otus-JBDD (ms)	Rel. std. dev. time Otus-JBDD	Avg. time Otus-Sylvan (ms)	Rel. std. dev. time Otus-Sylvan
amba_case_study_unreal1_2_2	161190.20	0.00	-	-	-	-
amba_case_study_unreal2_2	-	-	-	-	-	-
detector_unreal_10	417.80	0.03	-	-	-	-
detector_unreal_12	727.40	0.09	-	-	-	-
detector_unreal_2	161.60	0.13	168.60	0.04	439.80	0.37
detector_unreal_4	206.40	0.14	8529.00	0.01	10004.60	0.42
detector_unreal_6	263.60	0.24	-	-	-	-
detector_unreal_8	399.60	0.09	-	-	-	-
full_arbiter_unreal1_2_12	8342.20	0.06	-	-	-	-
full_arbiter_unreal1_2_15	7313.20	0.05	-	-	-	-
full_arbiter_unreal1_2_18	21791.00	0.47	-	-	-	-
full_arbiter_unreal1_2_3	219.40	0.01	2173.00	0.02	3427.00	0.46
full_arbiter_unreal1_2_6	2774.60	0.09	48001.00	0.01	34646.80	0.45
full_arbiter_unreal1_2_9	7380.60	0.05	-	-	-	-
full_arbiter_unreal1_3_10	12480.00	0.06	-	-	-	-
full_arbiter_unreal1_3_12	11507.40	0.04	-	-	-	-
full_arbiter_unreal1_3_1	338.40	0.14	-	-	-	-
full_arbiter_unreal1_3_2	383.60	0.15	-	-	-	-
full_arbiter_unreal1_3_3	519.40	0.13	-	-	-	-
full_arbiter_unreal1_3_4	885.40	0.30	-	-	-	-
full_arbiter_unreal1_3_5	1658.80	0.11	-	-	-	-
full_arbiter_unreal1_3_6	3130.00	0.21	-	-	-	-
full_arbiter_unreal1_3_8	11687.00	0.11	-	-	-	-
full_arbiter_unreal2_2	426.40	0.12	10235.00	0.02	11239.40	0.41
full_arbiter_unreal2_3	2415.60	0.39	-	-	-	-
full_arbiter_unreal2_4	897.60	0.19	-	-	-	-
full_arbiter_unreal2_5	49065.20	0.25	-	-	-	-
genbuf6	-	-	-	-	-	-
generalized_buffer_unreal1_2_2	209129.20	0.06	-	-	-	-
generalized_buffer_unreal2_2	-	-	-	-	-	-
lilydemo01	141.00	0.11	45.20	0.02	204.00	0.51
lilydemo02	163.60	0.21	63.00	0.14	273.60	0.60
lilydemo11	142.00	0.08	41.20	0.01	232.60	0.38
lilydemo15	146.40	0.13	45.60	0.14	228.60	0.24
lilydemo16	172.00	0.16	90.60	0.16	360.80	0.34
load_balancer_unreal1_2_10	4318.20	0.72	-	-	-	-
load_balancer_unreal1_2_12	5556.80	0.71	-	-	-	-
load_balancer_unreal1_2_2	238.80	0.12	1556.60	0.05	2732.20	0.38
load_balancer_unreal1_2_4	799.00	0.16	2509.40	0.02	4092.00	0.46
load_balancer_unreal1_2_6	2050.60	0.61	10935.60	0.10	14411.40	0.43
load_balancer_unreal1_2_8	5234.00	0.66	107402.40	0.01	72398.00	0.43
load_balancer_unreal1_4_1	396.00	0.00	-	-	-	-
load_balancer_unreal1_4_2	537.80	0.07	-	-	-	-
load_balancer_unreal1_4_3	854.00	0.09	-	-	-	-
load_balancer_unreal1_4_4	987.40	0.11	-	-	-	-
load_balancer_unreal1_4_5	1027.60	0.07	-	-	-	-
load_balancer_unreal1_4_6	1329.60	0.07	-	-	-	-
load_balancer_unreal2_3	2428.60	0.73	-	-	-	-
load_balancer_unreal2_4	470.60	0.01	-	-	-	-
load_balancer_unreal2_5	2618.40	0.01	-	-	-	-
ltl2dba27	107.40	0.11	35.60	0.01	201.60	0.61
ltl2dba_psi_10	151.40	0.15	-	-	-	-
ltl2dba_psi_12	225.20	0.16	-	-	-	-
ltl2dba_psi_1	115.60	0.13	34.80	0.02	208.20	0.61
ltl2dba_psi_2	136.00	0.11	72.00	0.01	306.40	0.51
ltl2dba_psi_3	135.20	0.06	525.20	0.14	1096.20	0.23
ltl2dba_psi_4	149.00	0.18	4535.00	0.06	4400.20	0.32
ltl2dba_psi_5	140.20	0.16	41961.20	0.03	19639.60	0.01
ltl2dba_psi_6	151.80	0.15	-	-	113032.20	0.01
ltl2dba_psi_7	147.20	0.02	-	-	-	-
ltl2dba_psi_8	161.80	0.03	-	-	-	-
ltl2dba_R_10	47060.40	0.00	-	-	-	-

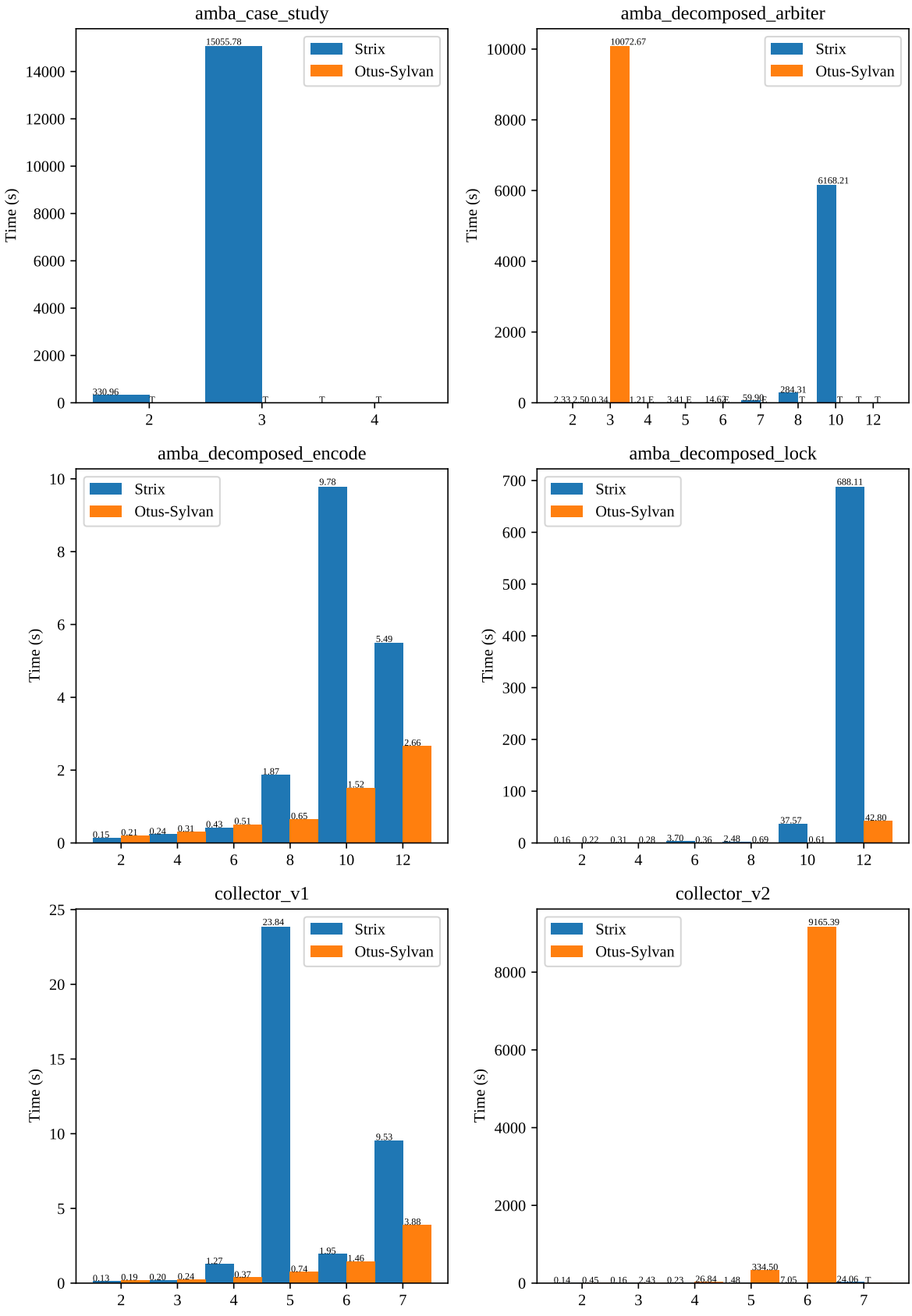
Specification	Avg. time Strix (ms)	Rel. std. dev. time Strix	Avg. time Otus-JBDD (ms)	Rel. std. dev. time Otus-JBDD	Avg. time Otus-Sylvan (ms)	Rel. std. dev. time Otus-Sylvan
ltl2dba_R_12	-	-	-	-	-	-
ltl2dba_R_2	116.80	0.11	219.00	0.09	413.40	0.04
ltl2dba_R_3	107.80	0.05	24205.80	0.02	25253.00	0.01
ltl2dba_R_4	126.20	0.09	-	-	-	-
ltl2dba_R_5	128.00	0.05	-	-	-	-
ltl2dba_R_6	165.60	0.12	-	-	-	-
ltl2dba_R_7	388.00	0.09	-	-	-	-
ltl2dba_R_8	1718.80	0.03	-	-	-	-
ltl2dba_theta_10	248.60	0.01	-	-	-	-
ltl2dba_theta_12	699.80	0.10	-	-	-	-
ltl2dba_theta_1	155.80	0.07	88.20	0.01	221.80	0.02
ltl2dba_theta_2	173.00	0.15	409.40	0.10	617.20	0.05
ltl2dba_theta_3	135.80	0.12	4897.20	0.05	4076.60	0.02
ltl2dba_theta_4	157.60	0.19	58124.60	0.03	27400.00	0.02
ltl2dba_theta_5	165.20	0.17	-	-	190931.60	0.01
ltl2dba_theta_6	182.80	0.04	-	-	-	-
ltl2dba_theta_7	192.00	0.01	-	-	-	-
ltl2dba_theta_8	216.80	0.01	-	-	-	-
ltl2dpa25	206.40	0.10	-	-	-	-
ModdifiedLedMatrix4X	2819.00	0.03	2224.60	0.07	4694.60	0.02
OneCounterGuiA0	1688.60	0.07	183.80	0.01	454.00	0.07
OneCounterGuiA1	6027.00	0.01	320.00	0.09	967.60	0.06
OneCounterGuiA2	2526.00	0.49	780.80	0.04	1307.80	0.08
OneCounterGuiA3	3391.20	0.02	1391.60	0.08	2892.60	0.09
OneCounterGuiA4	3858.60	0.04	1457.40	0.04	2925.60	0.12
OneCounterGuiA5	9381.80	0.01	1610.40	0.01	3433.80	0.09
OneCounterGuiA6	12694.80	0.01	2627.80	0.05	4687.60	0.13
OneCounterGuiA7	13092.40	0.01	4169.60	0.01	6394.40	0.15
OneCounterGuiA8	12067.20	0.01	3688.40	0.06	6561.00	0.12
OneCounterGui	1581.20	0.07	183.00	0.01	453.60	0.07
OneCounterInRangeA0	141.20	0.06	60.60	0.01	197.80	0.11
OneCounterInRangeA1	150.00	0.08	84.60	0.17	238.00	0.08
OneCounterInRangeA2	160.20	0.11	108.60	0.01	256.40	0.04
OneCounterInRange	171.20	0.12	86.60	0.01	226.80	0.01
prioritized_arbiter_unreal1_3_10	607.80	0.04	-	-	-	-
prioritized_arbiter_unreal1_3_2	301.60	0.01	1599.00	0.06	2471.60	0.04
prioritized_arbiter_unreal1_3_4	401.80	0.01	284548.60	0.01	68099.00	0.01
prioritized_arbiter_unreal1_3_6	444.20	0.07	-	-	-	-
prioritized_arbiter_unreal1_3_8	611.20	0.09	-	-	-	-
prioritized_arbiter_unreal2_2	204.80	0.10	280.60	0.03	551.20	0.04
prioritized_arbiter_unreal2_3	294.40	0.02	251630.00	0.00	71412.20	0.01
prioritized_arbiter_unreal2_4	435.40	0.00	-	-	-	-
prioritized_arbiter_unreal2_5	700.60	0.05	-	-	-	-
round_robin_arbiter_unreal1_2_12	7896.40	0.22	-	-	-	-
round_robin_arbiter_unreal1_2_15	29448.40	0.26	-	-	-	-
round_robin_arbiter_unreal1_2_18	69873.80	0.43	-	-	-	-
round_robin_arbiter_unreal1_2_3	4333.40	0.02	5762.80	0.10	6266.40	0.02
round_robin_arbiter_unreal1_2_6	9807.00	0.01	27966.20	0.01	21302.60	0.02
round_robin_arbiter_unreal1_2_9	6976.20	1.67	-	-	242152.60	0.00
round_robin_arbiter_unreal2_2	1286.80	0.10	7893.00	0.01	8063.00	0.01
round_robin_arbiter_unreal2_3	326.20	0.05	-	-	-	-
round_robin_arbiter_unreal2_4	2899.60	0.03	-	-	-	-
round_robin_arbiter_unreal2_5	50725.60	0.05	-	-	-	-
simple_arbiter_unreal1_4_1	257.20	0.15	1002.40	0.05	3118.80	0.04
simple_arbiter_unreal1_4_2	273.00	0.14	1822.40	0.03	4833.80	0.08
simple_arbiter_unreal1_4_3	312.60	0.15	193184.40	0.01	46734.80	0.03
simple_arbiter_unreal1_4_4	366.20	0.00	-	-	-	-
simple_arbiter_unreal1_4_5	379.80	0.01	-	-	-	-
simple_arbiter_unreal1_4_6	389.00	0.01	-	-	-	-
simple_arbiter_unreal2_2	166.80	0.06	61.80	0.01	193.20	0.03
simple_arbiter_unreal2_3	227.40	0.09	12484.60	0.02	8897.20	0.02
simple_arbiter_unreal2_4	307.20	0.01	-	-	-	-
simple_arbiter_unreal2_5	467.20	0.03	-	-	-	-
simple_arbiter_unreal2_6	733.80	0.05	-	-	-	-
TwoCounters2	1146.60	0.01	182.80	0.02	426.40	0.02
TwoCounters5	6269.40	0.02	-	-	-	-

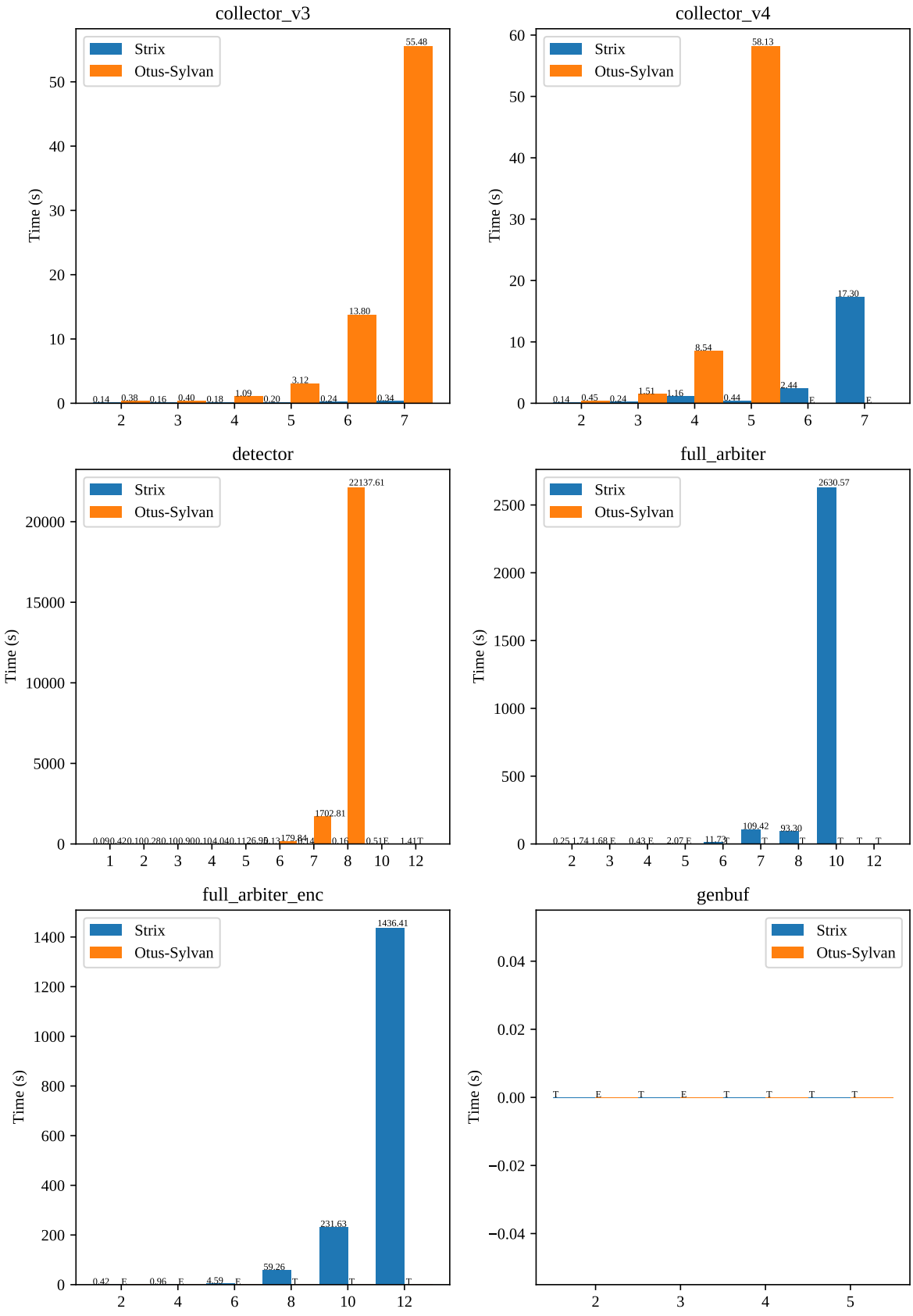
Specification	Avg. time Strix (ms)	Rel. std. dev. time Strix	Avg. time Otus-JBDD (ms)	Rel. std. dev. time Otus-JBDD	Avg. time Otus-Sylvan (ms)	Rel. std. dev. time Otus-Sylvan
TwoCountersDisButA0	1838.60	0.07	508.20	0.14	1765.00	0.04
TwoCountersDisButA1	1116.40	0.09	1133.60	0.05	2978.20	0.04
TwoCountersDisButA2	1985.80	0.06	3451.00	0.01	8759.00	0.07
TwoCountersDisButA3	3536.60	0.03	2591.60	0.08	8389.80	0.02
TwoCountersDisButA4	3607.20	0.06	7240.60	0.04	21053.80	0.09
TwoCountersDisButA5	11755.60	0.02	7831.40	0.02	25292.00	0.05
TwoCountersDisButA6	11522.20	0.02	22280.40	0.00	57060.60	0.10
TwoCountersDisButA7	15423.80	0.01	22313.00	0.02	60727.80	0.11
TwoCountersDisButA8	15784.60	0.02	165876.40	0.01	310493.40	0.09
TwoCountersDisButA9	39786.80	0.01	86277.20	0.01	219662.80	0.01
TwoCountersDisButAC	40027.00	0.01	-	-	-	-
TwoCountersGui	1082.40	0.06	585.80	0.15	2922.00	0.06
TwoCountersInRangeA0	486.20	0.15	182.80	0.02	325.60	0.03
TwoCountersInRangeA1	804.20	0.05	212.00	0.13	454.00	0.12
TwoCountersInRangeA2	1322.60	0.10	254.00	0.14	754.40	0.06
TwoCountersInRangeA3	1478.80	0.08	304.20	0.01	737.40	0.13
TwoCountersInRangeA4	1611.20	0.08	338.00	0.01	832.00	0.17
TwoCountersInRangeA5	2075.20	0.07	351.60	0.00	818.60	0.03
TwoCountersInRangeM0	1658.40	0.03	357.60	0.00	884.20	0.06
TwoCountersInRangeM1	1785.40	0.09	357.60	0.00	854.40	0.02
TwoCountersInRangeM2	2090.00	0.07	352.40	0.00	812.60	0.06
TwoCountersInRangeM3	2073.40	0.07	366.60	0.06	822.80	0.06
TwoCountersInRangeM4	2166.20	0.04	353.40	0.01	846.20	0.04
TwoCountersInRangeM5	2199.60	0.04	352.80	0.01	820.40	0.05
TwoCountersInRange	426.40	0.05	185.20	0.01	358.00	0.02
TwoCountersRefined	347.40	0.11	262.00	0.01	536.80	0.04
TwoCounters	287.80	0.03	211.40	0.01	507.20	0.03
UnderapproxDemo2	169.80	0.09	54.40	0.01	184.00	0.02
UnderapproxDemo	152.40	0.10	38.60	0.10	165.80	0.03

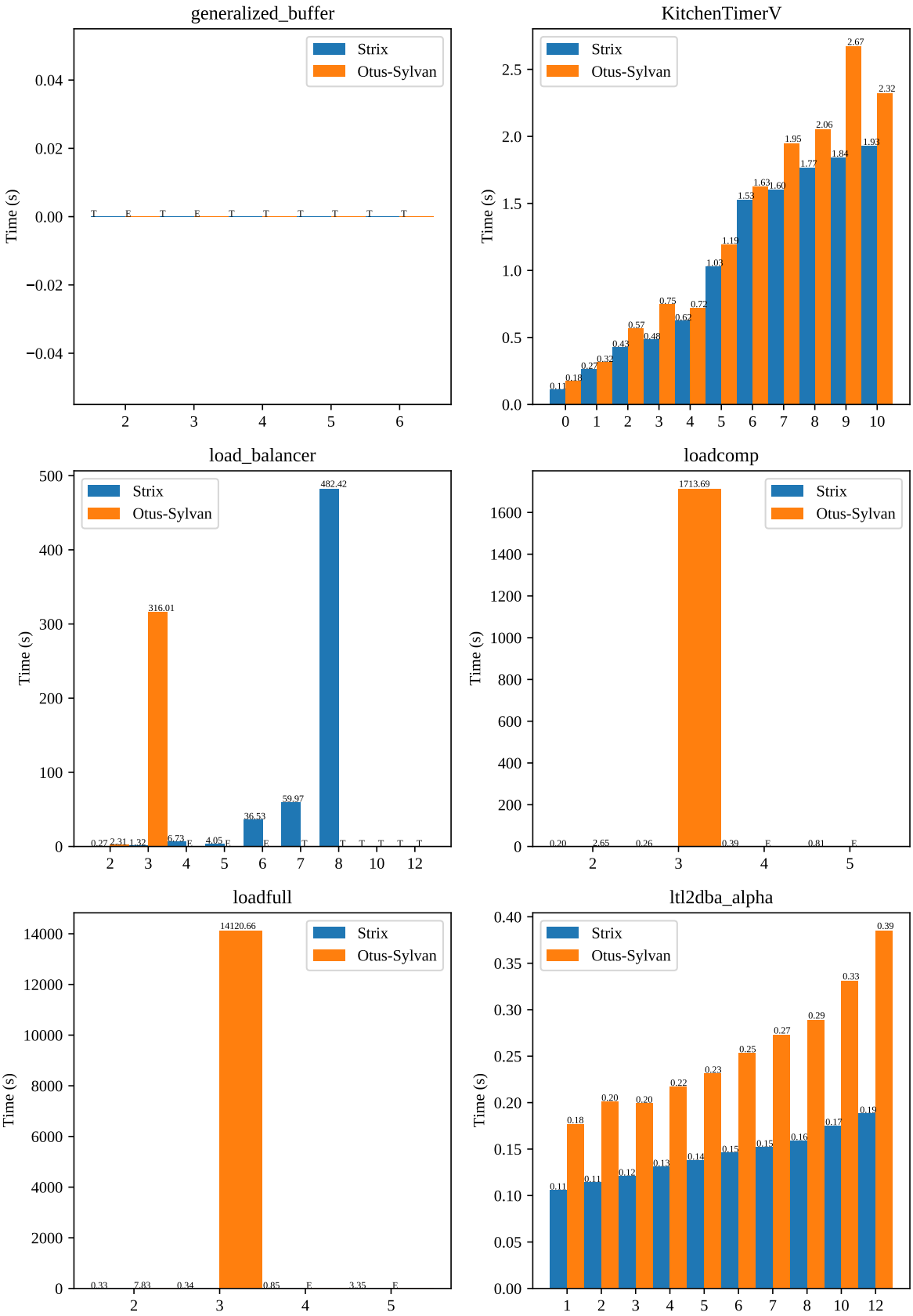
APPENDIX D

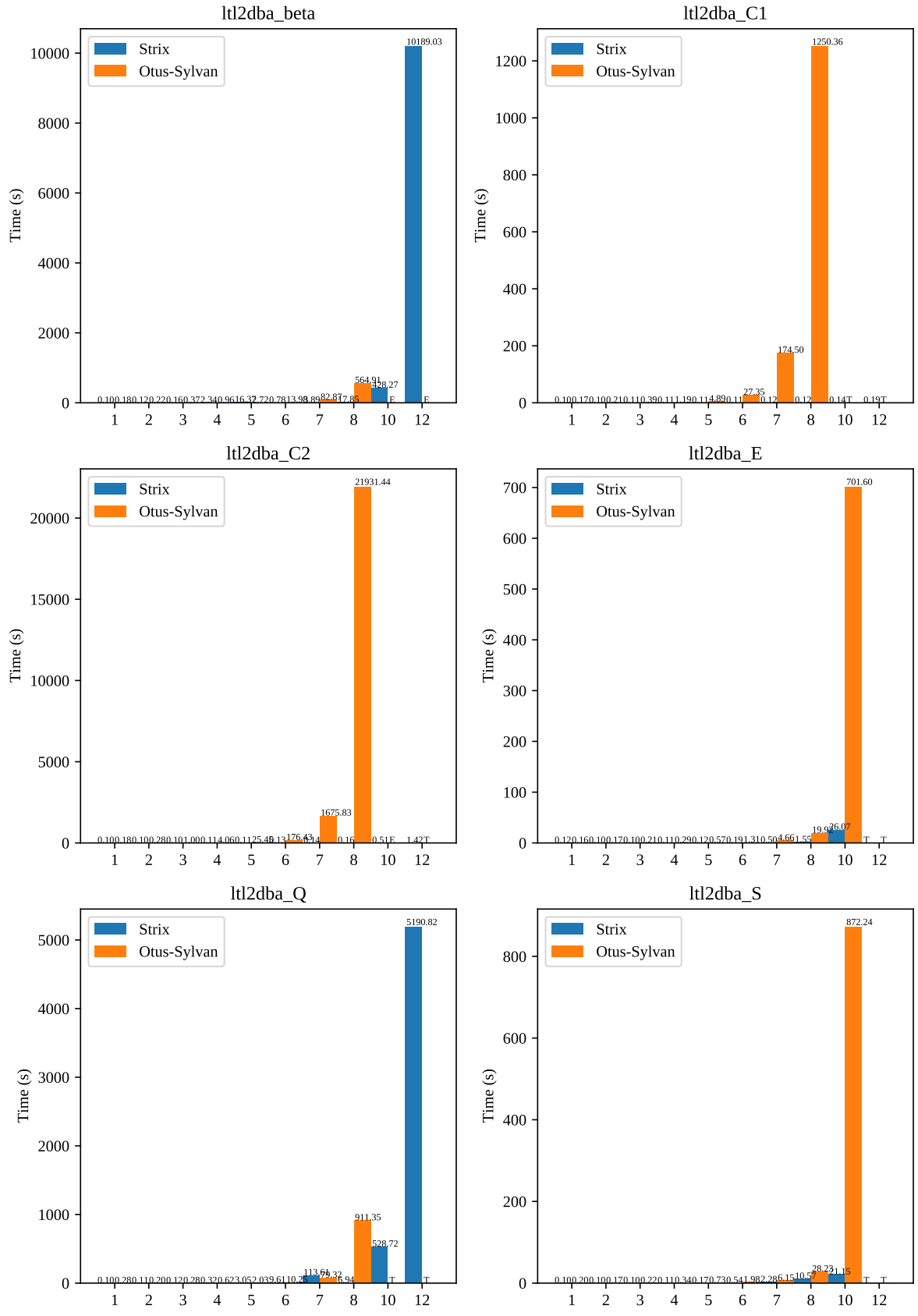
Parameterized Specification Benchmark Results

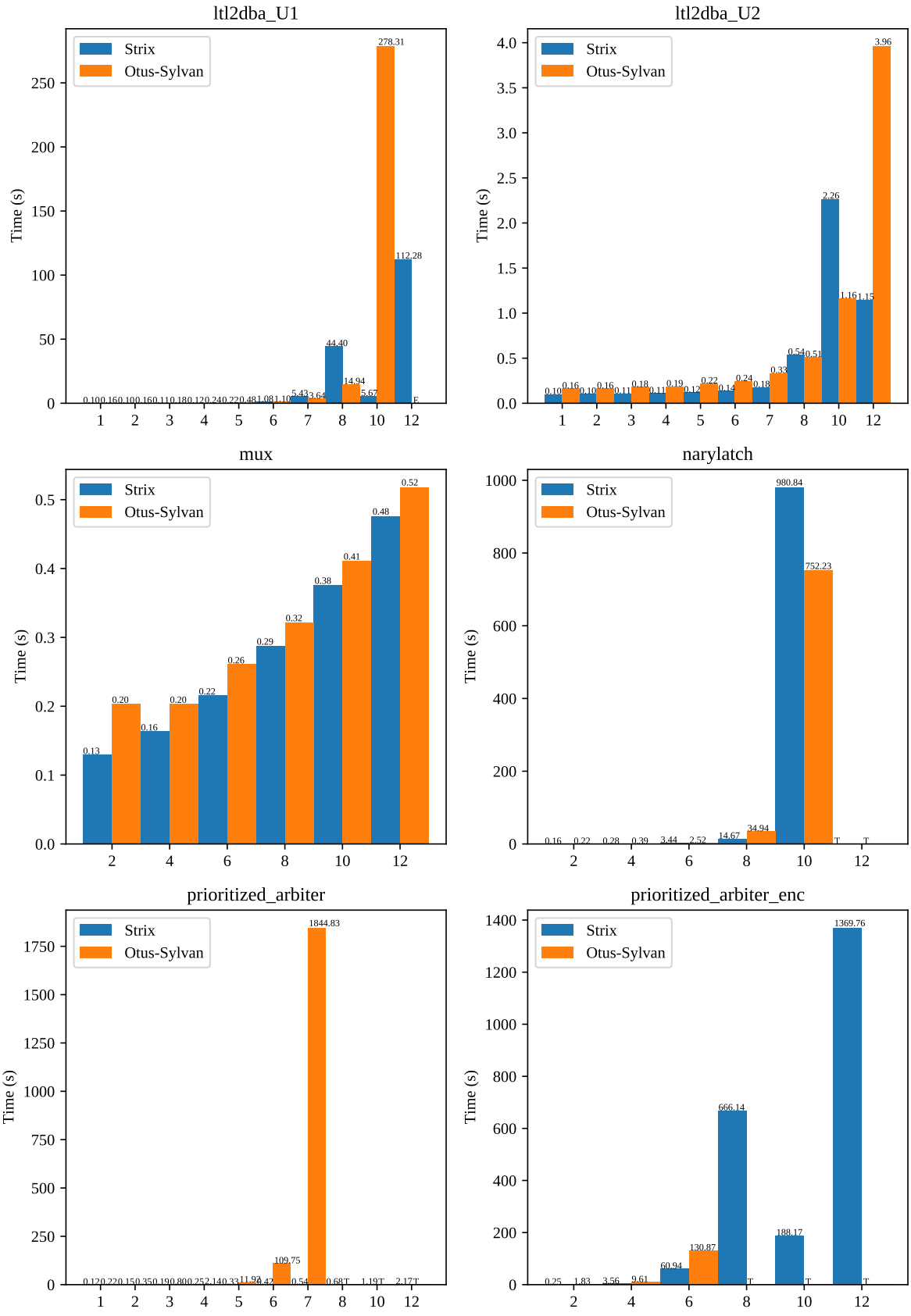
This appendix contains the results of the parameterized specification benchmark. For each parameterized specification, we provide a bar chart that displays the execution time against the parameter of the specification for both Otus-Sylvan and Strix. Each bar is annotated with its execution time. We indicate timeouts and errors by `T` and `E` respectively. Errors that appear are always related to resource exhaustion, such as a full bdd node table.

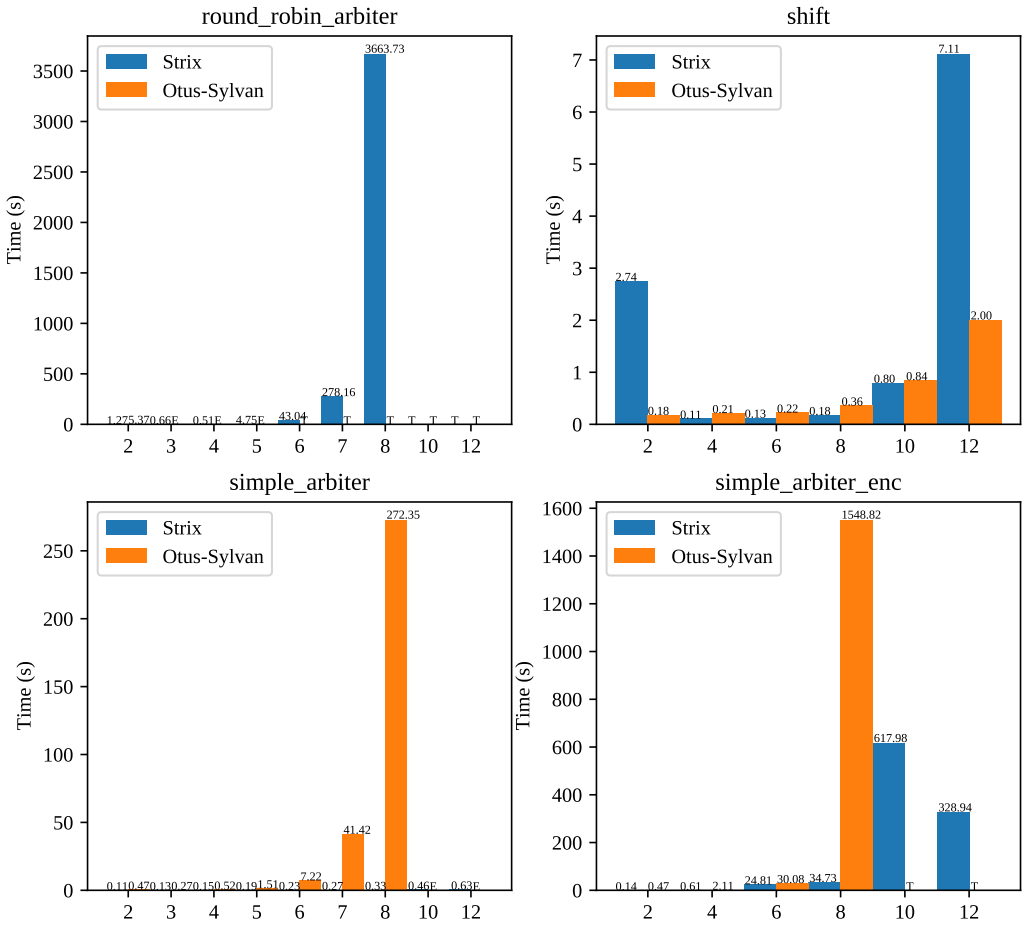

amba_case_study
amba_decomposed_arbiter
amba_decomposed_encode
amba_decomposed_lock
collector_v1
collector_v2
Strix
Otus-Sylvan
Time (s)

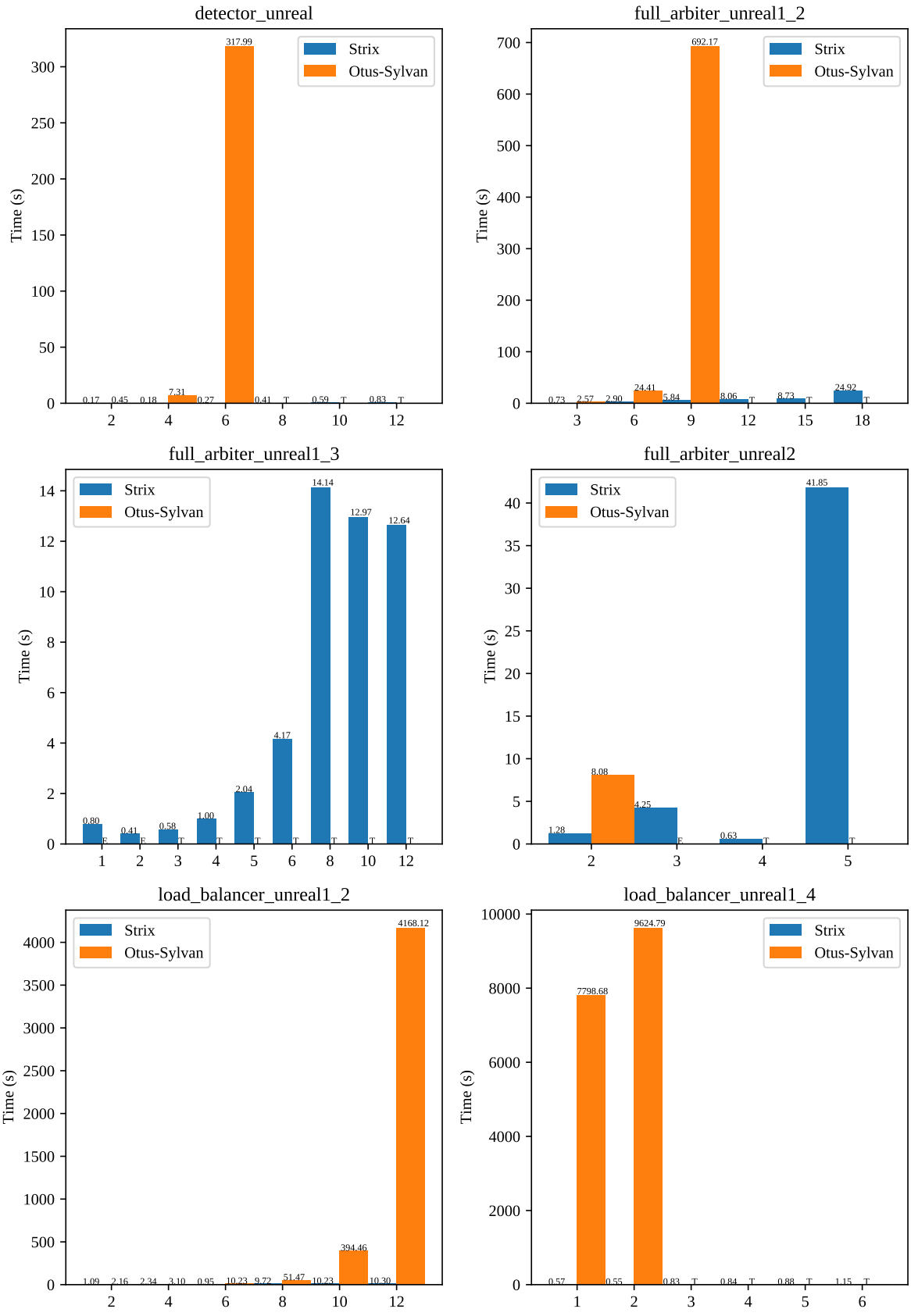

collector_v3
Strix
Otus-Sylvan
Time (s)
0.14
0.38
0.16
0.40
0.18
1.09
0.20
3.12
0.24
13.80
0.34
55.48
collector_v4
Strix
Otus-Sylvan
Time (s)
0.14
0.45
0.24
1.51
1.16
8.54
0.44
58.13
2.44
F
17.30
F
detector
Strix
Otus-Sylvan
Time (s)
1702.81
22137.61
full_arbiter
Strix
Otus-Sylvan
Time (s)
109.42
93.30
2630.57
full_arbiter_enc
Strix
Otus-Sylvan
Time (s)
231.63
59.26
1436.41
genbuf
Strix
Otus-Sylvan
Time (s)

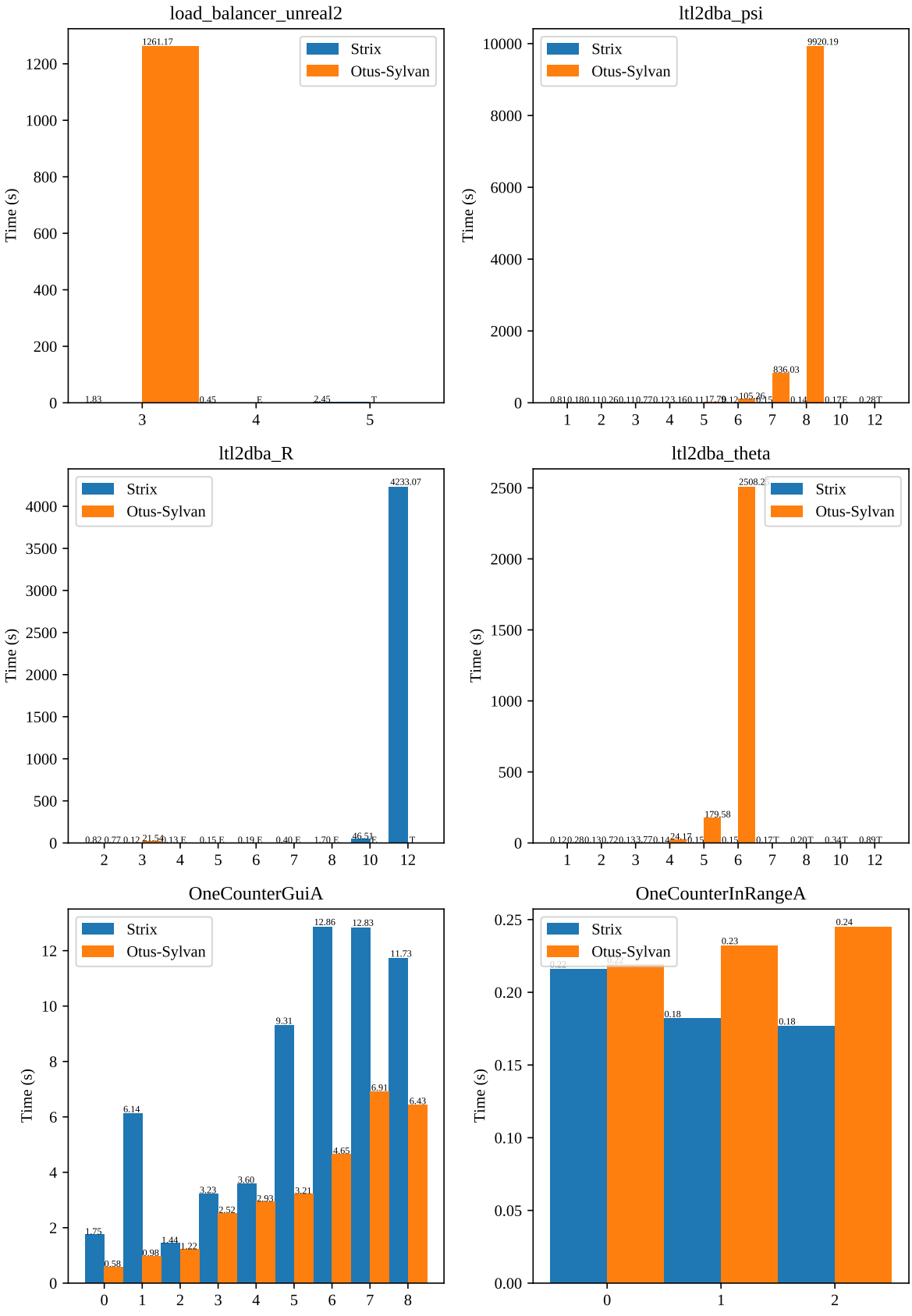

generalized_buffer
Strix
Otus-Sylvan
Time (s)
KitchenTimerV
load_balancer
316.01
482.42
59.97
36.53
loadcomp
1713.69
loadfull
14120.66
ltl2dba_alpha

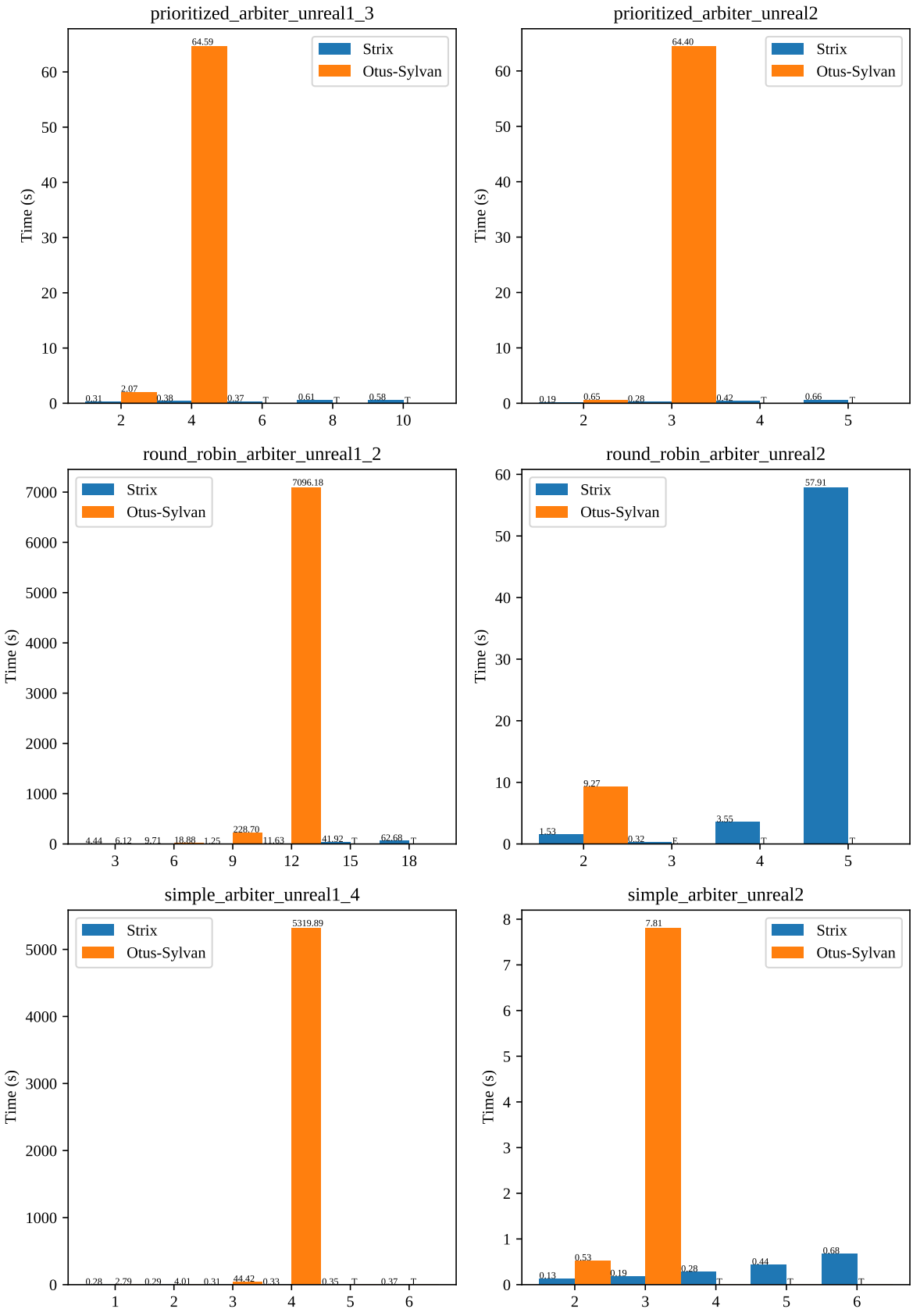

ltl2dba_beta
Strix
Otus-Sylvan
Time (s)
10189.03
564.91
428.27
ltl2dba_C1
1250.36
174.50
27.35
ltl2dba_C2
21931.44
1675.83
ltl2dba_E
701.60
26.07
ltl2dba_Q
5190.82
911.35
528.72
ltl2dba_S
872.24

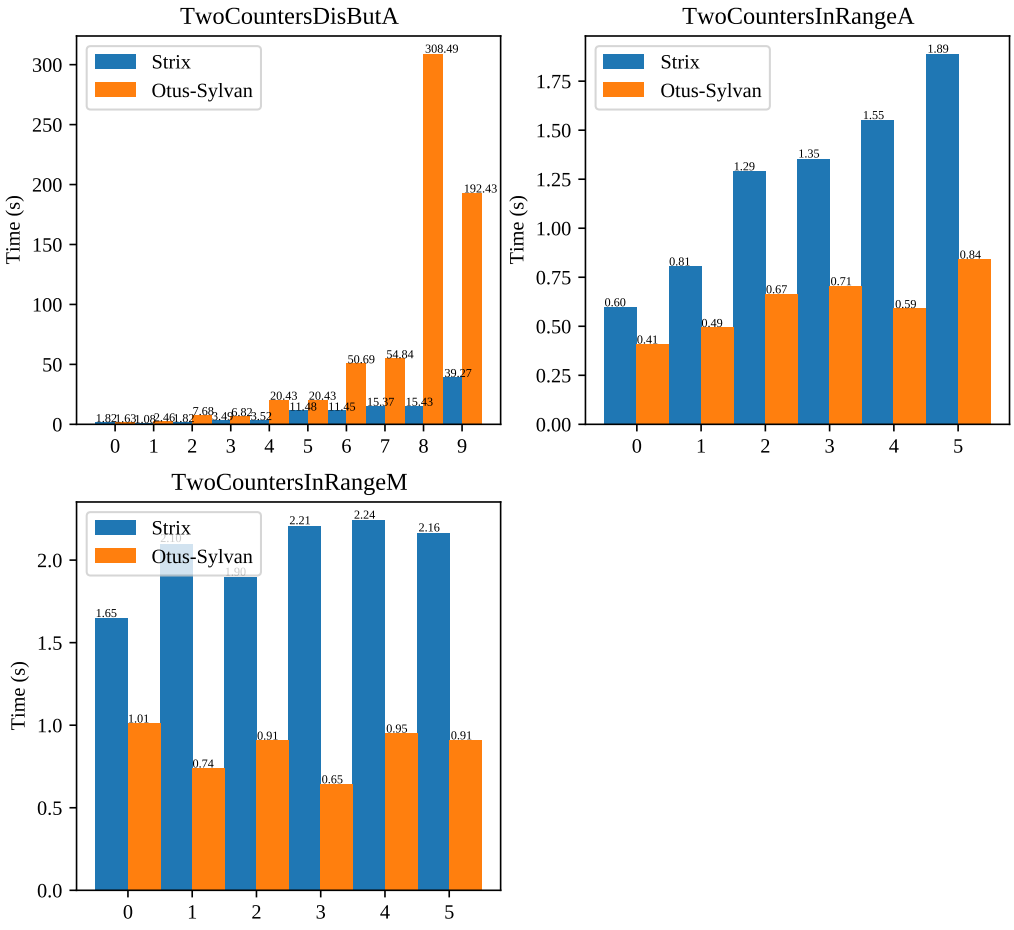

ltl2dba_U1
ltl2dba_U2
mux
narylatch
prioritized_arbiter
prioritized_arbiter_enc
Strix
Otus-Sylvan
Time (s)


round_robin_arbiter
shift
simple_arbiter
simple_arbiter_enc
Strix
Otus-Sylvan
Time (s)
3663.73
278.16
43.04
1.27
5.37
0.66F
0.51F
4.75F
2.74
7.11
2.00
0.80
0.84
0.36
0.22
0.21
0.18
0.13
0.11
272.35
41.42
7.22
1548.82
617.98
328.94
24.81
30.08
34.73
0.14
0.47
0.61
2.11
0.46F
0.63F
T


detector_unreal
full_arbiter_unreal1_2
full_arbiter_unreal1_3
full_arbiter_unreal2
load_balancer_unreal1_2
load_balancer_unreal1_4
Strix
Otus-Sylvan
Time (s)


load_balancer_unreal2
Time (s)
Strix
Otus-Sylvan
1261.17
1.83
0.45
F
2.45
T
ltl2dba_psi
9920.19
836.03
0.17F
0.28T
ltl2dba_R
4233.07
46.51
ltl2dba_theta
2508.2
179.58
24.17
OneCounterGuiA
12.86
12.83
11.73
9.31
6.14
6.91
6.43
4.65
3.60
3.23
2.93
2.52
3.21
1.75
0.58
0.98
1.44
1.22
OneCounterInRangeA
0.24
0.23
0.22
0.18
0.18


prioritized_arbiter_unreal1_3
Strix
Otus-Sylvan
Time (s)
0.31
2.07
0.38
64.59
0.37
T
0.61
T
0.58
T
prioritized_arbiter_unreal2
Strix
Otus-Sylvan
Time (s)
0.19
0.65
0.28
64.40
0.42
T
0.66
T
round_robin_arbiter_unreal1_2
Strix
Otus-Sylvan
Time (s)
4.44
6.12
9.71
18.88
1.25
228.70
11.63
7096.18
41.92
T
62.68
T
round_robin_arbiter_unreal2
Strix
Otus-Sylvan
Time (s)
1.53
9.27
0.32
F
3.55
T
57.91
T
simple_arbiter_unreal1_4
Strix
Otus-Sylvan
Time (s)
0.28
2.79
0.29
4.01
0.31
44.42
0.33
5319.89
0.35
T
0.37
T
simple_arbiter_unreal2
Strix
Otus-Sylvan
Time (s)
0.13
0.53
0.19
7.81
0.28
T
0.44
T
0.68
T


TwoCountersDisBut A
Strix
Otus-Sylvan
Time (s)
TwoCountersInRange A
TwoCountersInRange M