Reactive Synthesis: Towards Output-Sensitive Algorithms

Bernd Finkbeiner and Felix Klein
Universität des Saarlandes

Abstract. Reactive synthesis is a technology for the automatic construction of reactive systems from logical specifications. In these lecture notes, we study different algorithms for the reactive synthesis problem of linear-time temporal logic (LTL). The classic game-based synthesis algorithm is input-sensitive in the sense that its performance is asymptotically optimal in the size of the specification, but it produces implementations that may be larger than necessary. We contrast this algorithm with output-sensitive algorithms for reactive synthesis, i.e., algorithms that are optimized towards the size or structural complexity of the synthesized system. We study the bounded synthesis algorithm, which produces an implementation with a minimal number of states, and the bounded cycle synthesis algorithm, which additionally guarantees that the number of cycles of the implementation is minimal.

Keywords. reactive systems, synthesis, temporal logic, output-sensitive algorithms

1. Introduction

Hardware circuits, communication protocols, and embedded controllers are typical examples of *reactive systems* [13], i.e., computer systems that maintain a continuous interaction with their environment. Reactive systems play a crucial role in many applications in transport systems, building technology, energy management, health care, infrastructure, and environmental protection. Designing reactive systems is difficult, because one needs to anticipate every possible behavior of the environment and prepare an appropriate response.

Synthesis is a technology that constructs reactive systems *automatically* from a logical specification: that is, after the specification of the system is complete, no further manual implementation steps are necessary. The developer focuses on "what" the system should do instead of "how" it should be done. Because synthesis analyzes objectives, not implementations, it can be applied at an early design stage, long before the system has been implemented. The vision is that a designer analyzes the design objectives with a synthesis tool, automatically identifies competing or contradictory requirements and obtains an error-free prototype implementation. Coding and testing, the most expensive stages of development, are eliminated from the development process.

The automatic synthesis of implementations from specifications is one of the grand challenges of computer science. Its pursuit dates back at least to Alonzo

Church [5] and has ignited research on many fundamental topics, notably on the connection between logics and automata, on algorithmic solutions of infinite games over finite graphs [4], and on the theory of automata over infinite objects [17]. It is only in the last decade, however, that the theoretical ideas have been translated into practical tools (cf. [14,6,3,2,8]). The tools have made it possible to tackle real-world design problems, such as the synthesis of an arbiter for the *AMBA AHB bus*, an open industrial standard for the on-chip communication and management of functional blocks in system-on-a-chip (SoC) designs [1].

A common argument *against* synthesis is its complexity. It is natural to compare synthesis with the verification problem, where the implementation is already given, and one needs to check whether the specification is satisfied. For both synthesis and verification, the most commonly used specification language is linear-time temporal logic (LTL). Measured in the size of an LTL specification, the synthesis of a single-process finite-state machine is 2EXPTIME-complete, while the corresponding verification problem is in PSPACE. But is this comparison between verification and synthesis fair? The high complexity of synthesis is due to the fact that there exist small LTL formulas that can only be realized by very large implementations. As a result, synthesis "looks" much more expensive than verification, because the size of the implementation is an explicit parameter in the complexity of verification, and left implicit in the complexity of synthesis.

This paper gives an introduction to a new class of synthesis algorithms, whose performance is measured not only in the size of the specification, i.e., the input to the synthesis algorithm, but also in the size and complexity of the implementation, i.e., the output of the synthesis algorithm. Such algorithms are called *output sensitive*. The prototypical output-sensitive synthesis approach is *bounded synthesis*. In bounded synthesis, we look for an implementation where the number of states is limited by a given bound. By incrementally increasing the bound, bounded synthesis can be used to find a minimal implementation.

We first describe the classic game-theoretic approach to synthesis in Section 4, and then the bounded synthesis approach in Section 5. The two approaches differ fundamentally. The game-based approach is to translate the given LTL formula into an equivalent deterministic automaton, and then use the state space of the deterministic automaton to define a two-player game. In this game, the "output player" sets the outputs of the system and attempts to satisfy the specification, i.e., ensures that the resulting play is accepted by the automaton, and the "input player" sets the inputs and attempts to ensure that the play violates the specification, i.e., is rejected by the automaton. This game can be solved automatically, and a winning strategy for the output player can, if it exits, be translated into an implementation that is guaranteed to satisfy the specification. Unfortunately, the translation from LTL to deterministic automata is doubly exponential, which results in the 2EXPTIME complexity. In bounded synthesis, the LTL formula is not translated to a deterministic automaton; instead, its negation is translated to a nondeterministic automaton. This translation is single, rather than double exponential. The nondeterministic automaton suffices to check if a given implementation is correct: the implementation is correct if its product with the automaton does not contain an accepting path. In bounded synthesis, we "guess" an implementation of bounded size and make sure it is correct. This is done via

propositional constraint solving: we build a constraint system that is satisfiable if and only if an implementation that is correct with respect to the automaton.

The reduction of the synthesis problem to a constraint solving problem opens the possibility to add further constraints in order to focus the search towards the most desirable solutions. In Section 6, we describe such an extension: *bounded cycle synthesis*. In addition to the number of states, bounded cycle synthesis also bounds the number of cycles in the implementation. This leads to implementations that are not only small but also structurally simple.

2. The Synthesis Problem

In reactive synthesis, we transform a temporal specification into an implementation that is guaranteed to satisfy the specification for all possible inputs of the environment. In the following, we consider formulas of linear-time temporal logic (LTL) over a set of atomic propositions $AP = I \dot{\cup} O$ that is partitioned into a set of *inputs* I and a set of *outputs* O. A *trace* t is an infinite sequence over subsets of the atomic propositions. We define the set of traces $TR \coloneqq (2^{AP})^\omega$. An LTL formula describes a subset of TR. The idea is that in each step of a computation, the inputs are chosen by the environment, and the outputs are chosen by the system under construction. In a correctly synthesized system, all possible sequences satisfy the LTL formula.

Linear-time temporal logic (LTL). Linear-time temporal logic (LTL) [16] combines the usual Boolean connectives with temporal modalities such as the *Next* operator $\bigcirc$ and the *Until* operator $\mathcal{U}$. The syntax of LTL is given by the following grammar:

$$\varphi \ ::= \ p \ \mid \ \neg\varphi \ \mid \ \varphi \vee \varphi \ \mid \ \bigcirc\varphi \ \mid \ \varphi \,\mathcal{U}\, \varphi$$

where $p \in AP$ is an atomic proposition. $\bigcirc\varphi$ means that φ holds in the *next* position of a trace; $\varphi_1 \,\mathcal{U}\, \varphi_2$ means that φ_1 holds *until* φ_2 holds. There are several derived operators, such as $\diamondsuit\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, $\square\varphi \equiv \neg\diamondsuit\neg\varphi$, and $\varphi_1 \,\mathcal{W}\, \varphi_2 \equiv (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee \square\varphi_1$. $\diamondsuit\varphi$ states that φ will *eventually* hold in the future and $\square\varphi$ states that φ holds *globally*; $\mathcal{W}$ is the *weak* version of the *until* operator.

We use the following notation to manipulate traces: let $t \in TR$ be a trace and $i \in \mathbb{N}$ be a natural number. $t[i]$ denotes the i-th element of t. Therefore, $t[0]$ represents the starting element of the trace. Let $j \in \mathbb{N}$ and $j \geq i$, then $t[i,j]$ denotes the sequence $t[i]\ t[i+1] \ldots t[j-1]\ t[j]$, and $t[i,\infty]$ denotes the infinite suffix of t starting at position i. Let $p \in AP$ and $t \in TR$. The semantics of an LTL formula is defined as the smallest relation $\models$ that satisfies the following conditions:

$$
\begin{array}{lll}
t \models p & \text{iff} & p \in t[0] \\
t \models \neg\psi & \text{iff} & t \not\models \psi \\
t \models \psi_1 \vee \psi_2 & \text{iff} & t \models \psi_1 \text{ or } t \models \psi_2 \\
t \models \bigcirc\psi & \text{iff} & t[1,\infty] \models \psi \\
t \models \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \text{there exists } i \geq 0 : t[i,\infty] \models \psi_2 \\
 & & \text{and for all } 0 \leq j < i \text{ we have } t[j,\infty] \models \psi_1
\end{array}
$$

Example 1 *Suppose, for example, we are interested in constructing an arbiter circuit. Arbiters are used when more than one client needs access to some shared resource, such as a communication bus. To access the resource, the client sends a* request *signal R and waits until it receives a* grant *signal G from the arbiter. The task of the arbiter is to answer each request with a grant without giving grants to the two clients at the same time. In LTL, an arbiter with two clients can be specified as a conjunction of three properties:*

$$
\begin{array}{ll}
\square(\neg G_1 \vee \neg G_2) & \textit{(mutual exclusion)} \\
\square(R_1 \rightarrow \diamondsuit G_1) & \textit{(response 1)} \\
\square(R_2 \rightarrow \diamondsuit G_2) & \textit{(response 2)}
\end{array}
$$

The mutual exclusion *property states that at every point in time x, at most one grant signal can be set; the* response *properties state that if a request is made at some point in time, then there must exist a point in time, either immediately or later, where the corresponding grant signal is set.*

Implementations. We represent the result of the synthesis process as a finite-state machine. Let the set $AP = I \dot{\cup} O$ of atomic propositions be, as before, partitioned into the inputs I and the outputs O. A *Mealy machine* over I and O has the form $\mathcal{M} = (S, s_0, \delta, \gamma)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $\delta\colon S \times 2^I \to S$ is the transition function, and $\gamma\colon S \times 2^I \to 2^O$ is the output function. The output of the Mealy machine thus depends on the current state and the last input letter. A *path* of a Mealy machine is an infinite sequence $p = (s_0,\sigma_0)(s_1,\sigma_1)(s_2,\sigma_2)\ldots \in (S \times 2^{AP})^\omega$ of states and sets of atomic propositions that starts with the initial state s_0 and where $\delta(s_n, I \cap \sigma_n) = s_{n+1}$ and $\gamma(t_n, I \cap \sigma_n) = O \cap \sigma_n$ for all $n \in \mathbb{N}$. We refer to the projection of a path p to its second component $\pi = \sigma_0\sigma_1\sigma_2\ldots \in \Sigma^\omega$, as a *computation* of the Mealy machine. The Mealy machine *satisfies* the LTL formula φ, denoted by $M \models \varphi$, if all its computations satisfy φ.

Example 2 *Figure 1 shows two Mealy machines that implement the arbiter specification from Example 1. The Mealy machine shown on the left carefully answers every request and only issues a grant if there is an open request. The machine on the right always issues the grant to the same client, initially to the first client, and switches to the other client as soon as there is a request from the other client. Both machines satisfy the specification from Example 1.*

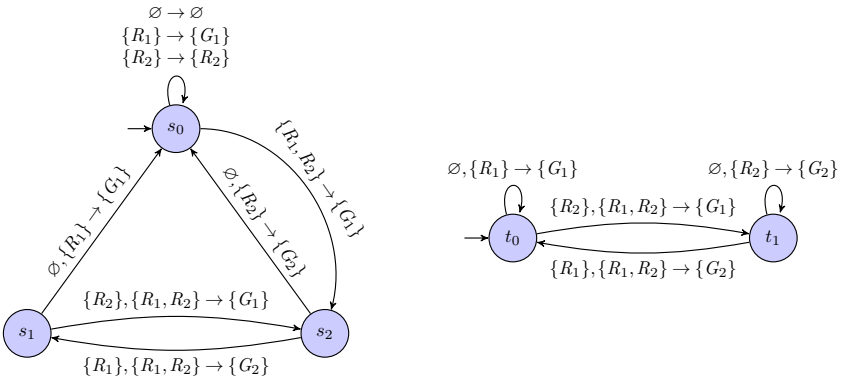

Figure 1. Two Mealy machines implementing the arbiter specification.

Realizability and Synthesis. We say that an LTL formula φ is *realizable* if there exists a Mealy machine $\mathcal{M}$ over the same inputs I and outputs O as φ such that $\mathcal{M} \models \varphi$. The *synthesis problem* of an LTL formula φ is to determine whether φ is realizable and, if the answer is yes, to construct a Mealy machine $\mathcal{M}$ such that $\mathcal{M} \models \varphi$.

3. Model checking

Before we address the synthesis problem, we take a quick detour into model checking. In model checking, the implementation is already given and we are interested in determining whether the implementation is correct.

Given a Mealy machine $\mathcal{M}$ and an LTL formula φ, model checking determines whether $\mathcal{M}$ satisfies φ. In case of a negative answer, model checking produces a *counterexample*, i.e., a trace $t \in (2^{AP})^\omega$ that is a computation of $\mathcal{M}$ that does not satisfy φ.

To model check a given Mealy machine, we translate the *negation* of the specification into an equivalent automaton, and then check the intersection of the Mealy machine with that automaton for language emptiness. LTL specifications can be represented as Büchi automata.

A *nondeterministic Büchi automaton* over the alphabet Σ is a tuple $\mathcal{A} = (Q, q_0, \Delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is an initial state, $\Delta \subseteq Q \times \Sigma \times Q$ a set of transitions, and $F \subseteq Q$ a subset of accepting states. A nondeterministic Büchi automaton accepts an infinite word $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega$ iff there exists a run r of $\mathcal{A}$ on w, i.e., an infinite sequence $r_0 r_1 r_2 \ldots \in Q^\omega$ of states such that $r_0 = q_0$ and $(r_i, w_i, r_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$, such that $r_j \in F$ for infinitely many $j \in \mathbb{N}$. The set of sequences accepted by $\mathcal{A}$ is called the *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$.

Example 3 *Consider the negation of the arbiter specification from Example 1, i.e., the LTL formula*

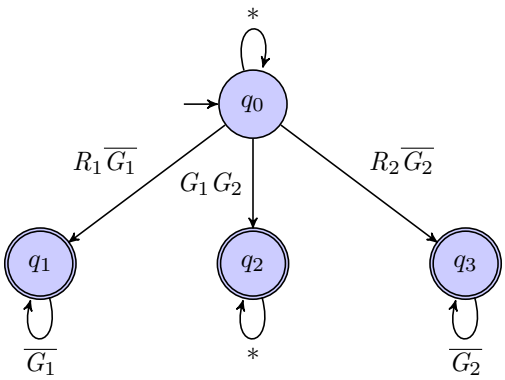

Figure 2. Nondeterministic Büchi automaton corresponding to the negation of the arbiter specification. The states depicted as double circles (q_1, q_2, and q_3) are the accepting states in F. The abbreviations $R_1\overline{G_1}$, $G_1\,G_2$, $R_2\overline{G_2}$, $\overline{G_1}$, $\overline{G_2}$ are used to indicate, in Boolean notation, letters of the alphabet 2^{AP}. E.g., $R_1\overline{G_1}$ represents the letters $\{R_1, R_2, G_2\}$, $\{R_1, R_2\}$, $\{R_1, G_2\}$, and $\{R_1\}$. The symbol $*$ represents all letters of the alphabet, i.e., all subsets of $\{R_1, R_2, G_1, G_2\}$.

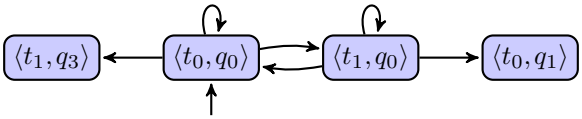

Figure 3. Product of the simple Mealy machine shown on the right in Fig. 1 with the Büchi automaton from Fig. 2.

$$\begin{aligned} &\Diamond(G_1 \wedge G_2) \\ \vee\, &\Diamond(R_1 \wedge \Box\neg G_1) \\ \vee\, &\Diamond(R_2 \wedge \Box\neg G_2)\ . \end{aligned}$$

A nondeterminstic Büchi automaton that accepts exactly the traces that satisfy this formula, i.e., all traces that violate the arbiter specification, is shown in Fig. 2.

Let $\mathcal{A}_{\neg\varphi} = (Q_{\neg\varphi}, q^0_{\neg\varphi}, \Delta_{\neg\varphi}, F_{\neg\varphi})$ be a Büchi automaton that accepts all sequences in $(2^{AP})^\omega$ that satisfy $\neg\varphi$, and therefore violate φ.

In model checking, we verify the Mealy machine $\mathcal{M}$ against a specification φ by building the product $\mathcal{M} \times \mathcal{A}_{\neg\phi}$ of the Mealy machine $\mathcal{M} = (S, s_0, \delta, \gamma)$ over inputs I and outputs O, and the Büchi automaton $\mathcal{A}_{\neg\phi} = (Q_{\neg\varphi}, q^0_{\neg\varphi}, \Delta_{\neg\varphi}, F_{\neg\varphi})$ with alphabet $2^{I\cup O}$. The product is a directed graph (V, E) with vertices $V = T \times Q$ and edges $E \subseteq V \times V$, where $(\langle s, q\rangle, \langle s', q'\rangle) \in E$ iff there is an input $\vec{i} \in 2^I$ such that $\delta(s, \vec{i}) = s'$ and $q' \in \Delta_{\neg\varphi}(q, \vec{i} \cup \gamma(s, \vec{i}))$. The Mealy machine satisfies φ iff there is no path in $\mathcal{M} \times \mathcal{A}_{\neg\phi}$ that visits an accepting state of $\mathcal{A}_{\neg\phi}$ infinitely often.

Example 4 *Figure 3 shows the product $\mathcal{M} \times \mathcal{A}_{\neg\varphi}$ of the small Mealy machine $\mathcal{M}$ shown on the right in Fig. 1 with the Büchi automaton $\mathcal{A}_{\neg\varphi}$ from Fig. 2. The only infinite paths are the self-loops from $\langle t_0, q_0\rangle$ and $\langle t_1, q_0\rangle$ and the path that oscillates forever between $\langle t_0, q_0\rangle$ and $\langle t_1, q_0\rangle$. These paths do not visit any accepting states. $\mathcal{M}$ thus satisfies φ.*

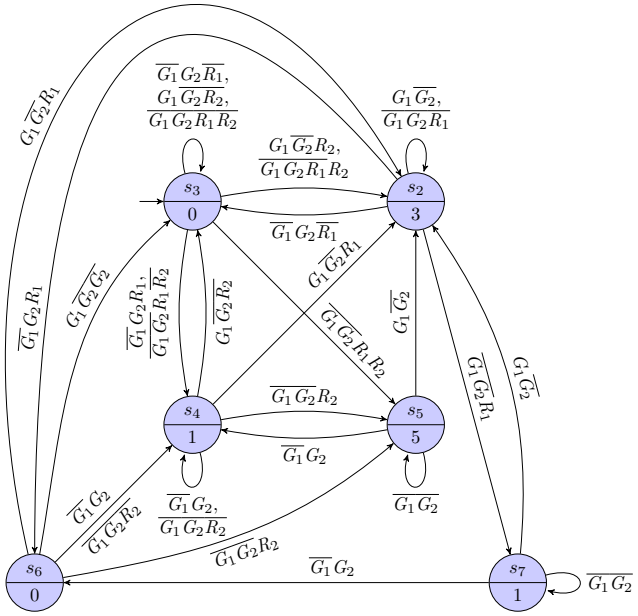

Figure 4. Deterministic parity automaton corresponding to the arbiter specification. The colors of the states are shown in the lower part of the state labels.

4. Game-based Synthesis

In the classic game-based approach to synthesis [17], the problem is analyzed in terms of a two-player game. The game is played between two players: the input player *Player I* determines the inputs to the system with the goal of *violating* the specification. The output player *Player O* controls the outputs of the system with the goal of *satisfying* the specification. A winning strategy for Player I can be translated into an implementation that is guaranteed to satisfy the specification. To solve the synthesis problem, we must therefore check whether Player I has a winning strategy.

In order to turn the specification into a game, we translate the LTL formula into a deterministic automaton that accepts all traces that satisfy the formula. An automaton is *deterministic* if each state and input has unique successor state, i.e., the set of transitions Δ is a total function from $Q \times \Sigma$ to Q. Since deterministic Büchi automata are not expressive enough to represent every possible LTL specification, we must use a more expressive acceptance condition such as the parity condition. Whereas a Büchi acceptance condition identifies a set $F \subseteq S$ of accepting states, which have to be visited infinitely often, a *parity condition* $c : S \to \mathbb{N}$ labels every state with a natural number. We call such a number the *color* of the state. A run of a parity automaton is accepting if the smallest color that appears infinitely often as a label of the states of the run is even. This introduces a hierarchy in the acceptance condition, as from some point on, every odd color has to be answered by a smaller even color. The Büchi acceptance condition

is a special case of the parity condition, where the accepting states are colored with 0 and the remaining states are colored with 1.

Example 5 *Figure 4 shows a deterministic parity automaton, whose language consists of all traces that satisfy the arbiter specification from Example 1. The colors of the states are shown in the lower part of the state labels.*

The deterministic automaton is then translated into an infinite game over a finite graph. A *game graph* is a directed graph (V, E) with vertices V and edges E. The vertices $V = V_I \dot\cup V_O$ are partitioned into the vertices V_I controlled by Player I and the vertices V_O controlled by Player O. A *parity game* (V, E, c) consists of a game graph (V, E) and a parity condition $c : V \to \mathbb{N}$. To play the game, a token is placed on some initial vertex v, which is then moved by the player owning the vertex to one of its successors v', i.e., such that $(v, v') \in E$. This is repeated ad infinitum, resulting in an infinite sequence of vertices, called a *play* of the game. If the underlying color sequence, i.e., the sequence resulting by the reduction of the vertices to their labels, satisfies the parity condition, Player O wins the game, otherwise Player I wins the game.

The game for the synthesis problem is obtained from the deterministic automaton by separating the moves of Player I, namely the choice of the inputs I to the system, and the moves of Player O, i.e., the choice of the outputs O.

We are interested in finding a winning strategy for Player O, i.e., an appropriate choice of output after every possible prefix of a play. We call such a prefix a *history* of the play. A useful property of parity games is that they are *memoryless determined*, which means that if one of the players has a winning strategy, then there also exists a winning strategy that only depends on the last vertex of the history, ignoring the previously visited vertices. For parity games, it is possible to automatically compute the set of vertices from which Player O has a winning strategy. This set of vertices is called the *winning region*. If the vertex corresponding to the initial state of the automaton is in the winning region, then there exists a solution to the synthesis problem.

Example 6 *Figure 5 shows the parity game for the synthesis problem of the arbiter specification from Example 1. The game was constructed by first translating the LTL formula into the deterministic automaton shown in Fig. 4, and then separating the moves of the input and output players. In Fig. 5, vertices controlled by Player I are depicted as rectangles, vertices controlled by Player O as circles. The winning region of Player O is marked by the highlighting. Since the initial vertex is in the winning region, the specification can be realized. The (memoryless) winning strategy is indicated by the thick edges.*

Game-based synthesis is asymptotically optimal in the size of the input. However, the synthesized implementations are often much larger than necessary. Compare, for example, the size of the winning strategy in Fig. 5 with the small Mealy machine on the right in Fig. 1.

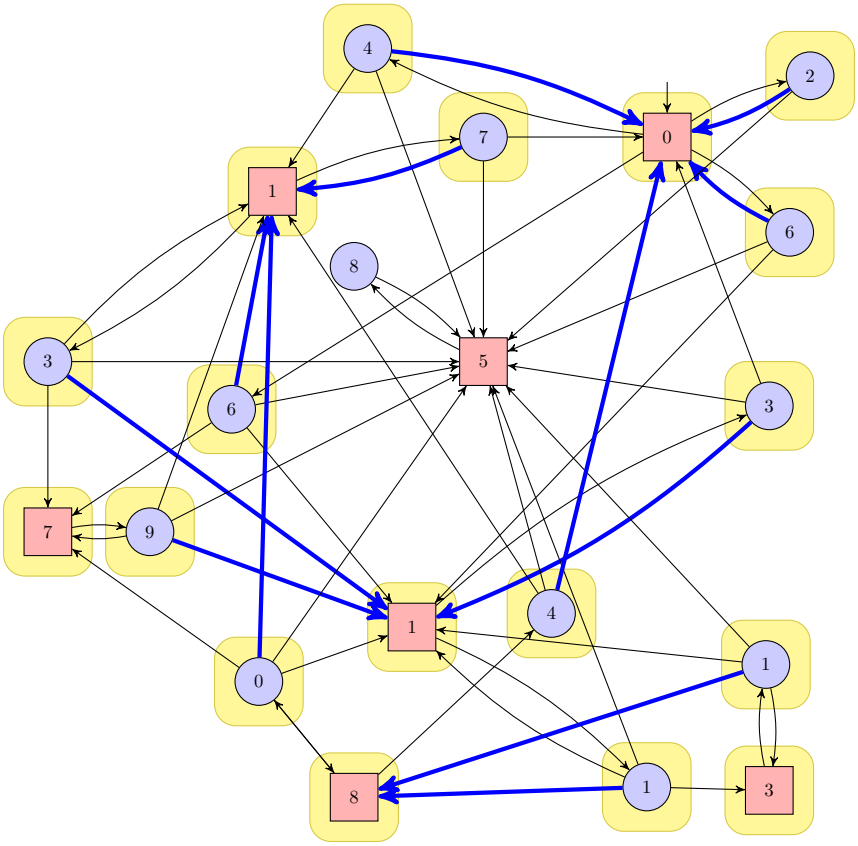

Figure 5. Parity game resulting from the deterministic parity automaton depicted in Fig. 4. Vertices controlled by Player I are depicted as rectangles, vertices controlled by Player O as circles. The highlighted states mark the winning region of Player O. The winning strategy is indicated by the thick edges.

5. Bounded Synthesis

In bounded synthesis [12], we set a bound on the number of states of the synthesized Mealy machine. By incrementally increasing the bound, we can use bounded synthesis to find a Mealy machine with a minimal number of states. The Mealy machine is found as a solution of a constraint system. To ensure that all solutions of the constrain system satisfy the specification, we encode not only the states, transitions, and outputs of the Mealy machine, but, additionally, an annotation of the states of the Mealy machine that ensures that the given LTL specification is satisfied. This annotation essentially ensures that the model checking of the Mealy machine succeeds, i.e., that the language of the product with the Büchi automaton corresponding to the negation of the specification is empty.

Let $\langle V, E \rangle$ be the product of a Mealy machine $\mathcal{M}$ and a Büchi automaton $\mathcal{A}_{\neg\varphi}$ for the negation of the specification. An *annotation* $\lambda : S \times Q \to \{\bot\} \cup \mathbb{N}$ is a function that maps nodes from the run graph to either unreachable $\bot$ or a natural number k. An annotation is valid if it satisfies the following conditions:

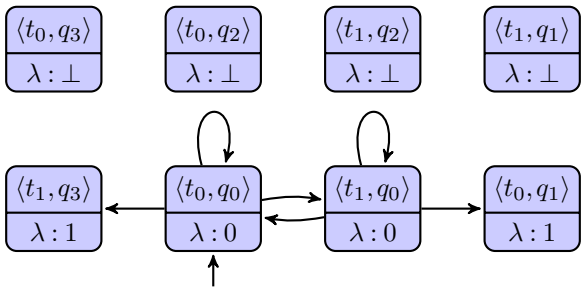

Figure 6. Annotated product of the simple Mealy machine shown on the right in Fig. 1 with the Büchi automaton from Fig. 2.

- the initial vertex $\langle s_0, q_0 \rangle$ is labeled by a natural number: $\lambda(t_0, q_0) \neq \bot$, and
- if a vertex $\langle s, q \rangle$ is annotated with a natural number, i.e., $\lambda(t, q) = k \neq \bot$, then for every $\vec{i} \in 2^I$ and $q' \in \Delta_{\neg\varphi}(q, \vec{i} \cup \gamma(s, \vec{i}), q')$, the successor pair $\langle \tau(s, \vec{i}), q' \rangle\rangle$ is annotated with a *greater or equal* number, which needs to be *strictly* greater if q' is a rejecting state. That is, $\lambda(t', q') > k$ if $q' \in F$ and $\geq \lambda(t', q') \geq k$ otherwise.

Example 7 *Figure 6 shows the annotated product of the simple Mealy machine from the right in Fig. 1 with the Büchi automaton from Fig. 2. One can verify that the annotation is correct by checking every edge individually. For example, the annotation has to increase from* $\langle t_0, q_0 \rangle \to \langle t_1, q_3 \rangle$ *and from* $\langle t_1, q_0 \rangle \to \langle t_0, q_1 \rangle$ *as* q_1 *and* q_3 *are rejecting.*

The existence of a Mealy machine with a corresponding annotation of the product graph can be expressed as a propositional constraint. For this purpose, we encode the Mealy machine and the annotation with Boolean variables.

- $\text{TRANS}(t, \nu, t')$ for all $t, t' \in S$ and $\nu \in 2^{\mathcal{I}}$, for the transition function $\delta \colon S \times 2^I \to S$ of the Mealy machine $\mathcal{M} = (S, s_0, \delta, \gamma)$.
- $\text{OUTPUT}(t, \nu, x)$ for all $t \in S$, $\nu \in 2^{\mathcal{I}}$ and $x \in \mathcal{O}$, for the output function $\delta \colon S \times 2^I \to S$.
- $\text{RGSTATE}(t, q)$ for all $t \in T$ and $q \in Q$, to encode the reachable states of the product graph G of $\mathcal{M}$ and $\mathcal{A}_{\neg\varphi}$, i.e., those state pairs $\langle t, q \rangle$ where $\lambda(t, q) \neq \bot$.
- $\text{ANNOTATION}(t, q, i)$ for all $t \in T$, $q \in Q$ and $0 < i \leq \log(n \cdot k)$, where n is the bound on the size of the Mealy machine and k is the number of states of the Büchi automaton. The variables encode the numerical annotation of a state pair (t, q) of G. We use a logarithmic number of bits to encode the annotated value in binary.

Given an LTL formula φ and a bound n on the states of the Mealy machine, we solve the bounded synthesis problem by checking the satisfiability of the propositional formula $\mathcal{F}_{BS}(\varphi, n)$, consisting of the following constraints:

- The pair of initial states $\langle s_0, q_0 \rangle$ for some arbitrary, but fixed, s_0 is reachable and annotated with 1.

$$\textsc{rgstate}(s_0, q_0) \wedge \textsc{annotation}(1,1) = 1$$

- Each annotation of a vertex of the product graph bounds the number of visited accepting states, not counting the current vertex itself:

$$\bigwedge_{t \in T,\, q \in Q} \textsc{rgstate}(t,q) \rightarrow \bigwedge_{\sigma \in 2^{\Sigma}} \mathit{output}(t,\sigma) \rightarrow \bigwedge_{t' \in T} \textsc{trans}(t, \mathcal{I} \cap \sigma, t') \rightarrow$$

$$\bigwedge_{q' \in \Delta(q,\sigma)} \textsc{rgstate}(t',q') \wedge \textsc{annotation}(t,q) \prec_q \textsc{annotation}(t',q')$$

  where $\prec_q$ equals $<$ if $q \in R$ and equals $\leq$ otherwise. The formula $\mathit{output}(t,\sigma)$ ensures that the output corresponds to the output function of the Mealy machine, i.e.,

$$\mathit{output}(t,\sigma) = \bigwedge_{x \in \mathcal{O} \cap \sigma} \textsc{output}(t, \mathcal{I} \cap \sigma, x) \wedge \bigwedge_{x \in \mathcal{O} \setminus \sigma} \neg \textsc{output}(t, \mathcal{I} \cap \sigma, x).$$

Theorem 1 (Bounded Synthesis [12]) *For an LTL formula φ and a bound $n \in \mathbb{N}$, the propositional formula $\mathcal{F}_{BS}(\varphi, n)$ is satisfiable if and only if there is a Mealy machine $\mathcal{M}$ with $|\mathcal{M}| = n$ that satisfies φ.*

The propositional constraint can be solved by a standard SAT solver. In addition to the encoding as a propositional constraint, the bounded synthesis problem has also been reduced to the satisfiability of quantified Boolean formulas (QBF) and dependency quantified Boolean formulas (DQBF) [7], as well as to satisfiability modulo theories (SMT) [11]. Such encodings are more concise than the encoding as a Boolean formula. Even though the satisfiability problems of these logics are more expensive than propositional satisfiability, in particular the QBF encoding has proven advantageous in experiments (cf. [8]).

Another powerful optimization is *lazy synthesis* [9], which avoids the full construction of the constraint system. Lazy synthesis alternates between constraint solving, where a model is constructed for an incomplete constraint system, and verification, where errors in the previously constructed model are identified and used to extend the constraint system.

6. Bounded Cycle Synthesis

Bounded cycle synthesis [10] extends bounded synthesis by bounding not only the number of states, but also the number of cycles of the Mealy machine. Bounded cycle synthesis allows us to find implementations that are not only small but also structurally simple. A cycle is a path of a Mealy machine that ends in the same state it started in. Even Mealy machines with a small number of states can have many cycles: the number of cycles can be exponential in the number of states.

The explosion of the number of circles is in fact worse than the explosion of the number of states: while a realizable LTL formula has an implementation with at most doubly exponentially many states, there exist LTL formulas where the number of cycles in the Mealy machine is triply exponential [10]. This makes the number of cycles a particularly interesting metric for output-sensitive synthesis algorithms.

Let $G = (V, E)$ be a directed graph. A *(simple) cycle* c of G is a a tuple (C, η), consisting of a non-empty set $C \subseteq V$ and a bijection $\eta \colon C \mapsto C$ such that

- $\forall v \in C.\ (v, \eta(v)) \in E$ and
- $\forall v \in C.\ n \in \mathbb{N}.\ \eta^n(v) = v \ \Leftrightarrow\ n \mod |C| = 0$,

where η^n denotes n times the application of η. In other words, a cycle of G is a path through G that starts and ends at the same vertex and visits every vertex of V at most once. We say that a cycle $c = (C, \eta)$ has length n iff $|C| = n$.

We extend the notion of a cycle of a graph G to Mealy machines $\mathcal{M} = (T, t_I, \delta, \lambda)$, such that c is a cycle of $\mathcal{M}$ iff c is a cycle of the graph (T, E) for $E = \{(t, t') \mid \exists \nu \in 2^{\mathcal{I}}.\ \delta(t, \nu) = t'\}$. Thus, we ignore the input labels of the edges of $\mathcal{M}$. The set of all cycles of a Mealy machine $\mathcal{M}$ is denoted by $\mathcal{C}(\mathcal{M})$.

6.1. Counting Cycles

A classical algorithm for counting the number of cycles of a directed graph is due to Tiernan [18]. We review this algorithm here as a preparation for the bounded cycle synthesis encoding.

Algorithm 1. Given a directed graph $G = (V, E)$, we count the cycles of G using the following algorithm:

(1) Initialize the cycle counter c to $c := 0$ and some set P to $P := \varnothing$.
(2) Pick some arbitrary vertex v_r of G, set $v := v_r$ and $P := \{v_r\}$.
(3) For all edges $(v, v') \in E$, with $v' \notin P \setminus \{v_r\}$:
  (3a) If $v' = v_r$, increase c by one.
  (3b) Oherwise, set $v := v'$, add v' to P and recursively execute (3). Afterwards, reset P to its value before the recursive call.
(4) Obtain the sub-graph G', by removing v_r from G:
  (4a) If G' is empty, return c.
  (4b) Otherwise, continue from (2) with G'.

The algorithm starts by counting all cycles that contain the first picked vertex v_r. This is done by an unfolding of the graph into a tree, rooted in v_r, such that there is no repetition of a vertex on any path from the root to a leaf. The number of vertices that are connected to the root by an edge of E then represents the corresponding number of cycles through v_r. The remaining cycles of G do not contain v_r and, thus, are cycles of the sub-graph G' without v_r, as well. Hence, we count the remaining cycles by recursively counting the cycles of G'. The algorithm terminates as soon as G' becomes empty.

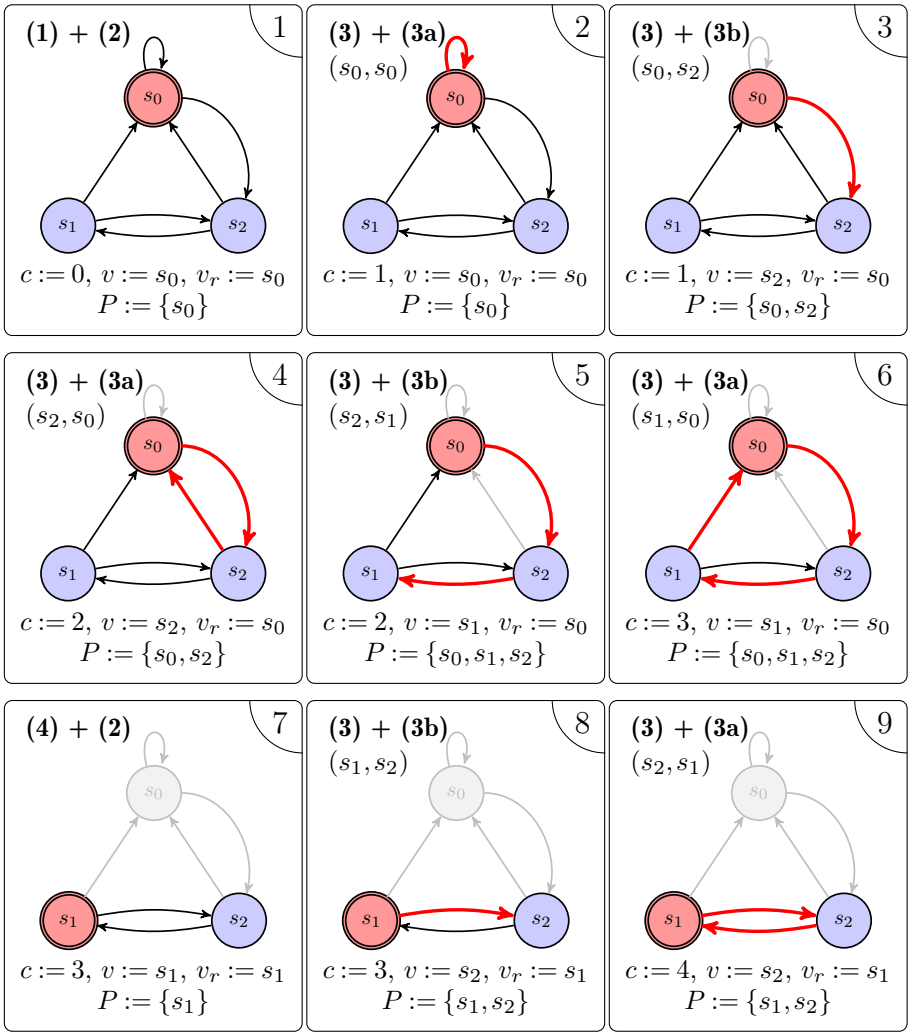

Figure 7. Execution of Tiernan's algorithm for the larger Mealy machine on the left in Fig. 1.

The algorithm, as described so far, has the disadvantage that the number of unfolded trees is exponential in the size of the graph, even if none of their vertices is connected to the root, i.e., even if there is no cycle to be counted. This drawback can be avoided by first reducing the graph to all its strongly connected components (SCCs) and then counting the cycles of each SCC separately [19,15]. This reduction is sound, as a cycle never leaves an SCC of the graph.

The improved algorithm is exponential in the size of G, and linear in the number of cycles m. Furthermore, the time between two detections of a cycle, during the execution, is bounded linear in the size of G.

Example 8 *To see Tiernan's algorithm in action, we count the number of simple cycles of the larger Mealy machine on the left in Fig. 1. The execution is shown in Fig. 7. In this example, we do not need to apply the reduction to individual*

SCCs, because the Mealy machine consists of a single SCC. As result we obtain that the Mealy machine has four simple cycles.

6.2. The Bounded Cycle Synthesis Encoding

Like in the bounded synthesis approach, we solve the bounded cycle synthesis problem via a reduction to propositional satisfiability. We extend the constraint system from bounded synthesis with additional constraints that ensure that the number of cycles, as determined by Tiernan's algorithm, does not exceed the given bound.

We call a tree that witnesses m cycles in G, all containing the root r of the tree, a witness-tree $\mathcal{T}_{r,m}$ of G. Formally, a *witness-tree* $\mathcal{T}_{r,m}$ of $G = (V, E)$ is a labeled graph $\mathcal{T}_{r,m} = ((W, B \cup R), \tau)$, consisting of a graph $(W, B \cup R)$ with $m = |R|$ and a labeling function $\tau \colon W \to V$, such that:

1. The edges are partitioned into blue edges B and red edges R.
2. All red edges lead back to the root:

   $$R \subseteq W \times \{r\}$$
3. No blue edges lead back to the root:

   $$B \cap W \times \{r\} = \varnothing$$
4. Each non-root has at least one blue incoming edge:

   $$\forall w' \in W \setminus \{r\}.\ \exists w \in W.\ (w, w') \in B$$
5. Each vertex has at most one blue incoming edge:

   $$\forall w_1, w_2, w \in W.\ (w_1, w) \in B \wedge (w_2, w) \in B \Rightarrow w_1 = w_2$$
6. The graph is labeled by an unfolding of G:

   $$\forall w, w' \in B \cup R.\ (\tau(w), \tau(w')) \in E,$$
7. The unfolding is complete:

   $$\forall w \in W.\ \forall v' \in V.\ (\tau(w), v') \in E \Rightarrow \exists w' \in W.\ (w, w') \in B \cup R \wedge \tau(w') = v'$$
8. Let $w_i, w_j \in W$ be two different vertices that appear on a path from the root to a leaf in the r-rooted tree (W, B)[1]. Then the labeling of w_i and w_j differs, i.e., $\tau(v_i) \neq \tau(v_j)$.
9. The root of the tree is the same as the corresponding vertex of G, i.e., $\tau(r) = r$.

Lemma 1 ([10]) *Let $G = (V, E)$ be a graph consisting of a single SCC, $r \in V$ be some vertex of G and m be the number of cycles of G containing r. Then there is a witness-tree $\mathcal{T}_{r,m} = ((W, B \cup R), \tau)$ of G with $|W| \leq m \cdot |V|$.*

Lemma 2 ([10]) *Let $G = (V, E)$ be a graph consisting of a single SCC and let $\mathcal{T}_{r,m}$ be a witness-tree of G. Then there are at most m cycles in G that contain r.*

[1] Note that the tree property is enforced by Conditions 3 – 5.

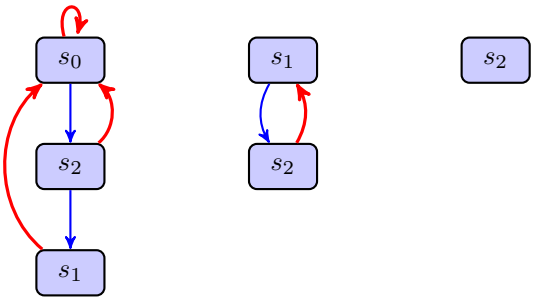

Figure 8. The forest of witness trees proving prove the overall number of four cycles in the larger Mealy machine of Fig. 1.

From Lemma 1 and 2 we derive that $\mathcal{T}_{r,m}$ is a suitable witness to bound the number of cycles of an implementation $\mathcal{M}$. Furthermore, from Lemma 1, we also obtain an upper bound on the size of $\mathcal{T}_{r,m}$.

Example 9 *Figure 8 shows the witness trees for the larger Mealy machine on the left of Fig. 1. Each red edge, leading back to s_0 and s_1 on the first tree level, captures one cycle of the machine. Thereby, the properties of the tree enforce that all cycles are captured by these trees.*

We now encode the bound on the number of cycles as a propositional constraint. First, we construct a simple directed graph G out of the implementation $\mathcal{M}$. Then, we guess all the sub-graphs, obtained from G via iteratively removing vertices, and split them into their corresponding SCCs. Finally, we guess the witness-tree for each such SCC.

In order to keep the encoding compact, we introduce some further optimizations. First, we do not need to introduce a fresh copy for each SCC, since the SCC of a vertex is always unique. Thus, it suffices to guess an annotation for each vertex. Second, we have to guess n trees $\mathcal{T}_{r_i,m_i}, i = 1 \ldots n$, each consisting of at most $m_i \cdot n$ vertices, such that the sum of all m_i is equal to the overall number of cycles m. One possible solution would be to overestimate each m_i by m. Another possibility would be to guess the exact distribution of the cycles over the different witness-trees $\mathcal{T}_{r_i,m_i}$. In our encoding, we guess all trees together in a single graph bounded by $m \cdot n$. We annotate each vertex with its corresponding witness-tree $\mathcal{T}_{r_i,m_i}$. Instead of bounding the number of red edges separately for each $\mathcal{T}_{r_i,m_i}$ by m_i, we just bound the number of all red edges in the whole forest by m. In this way, we not only reduce the size of the encoding, but also avoid additional constrains that would be needed to sum up the different witness-tree bounds i to m.

Let T be some ordered set with $|T| = n$ and $S = T \times \{1, 2, \ldots, m\}$. We use T to denote the vertices of G and S to denote the vertices of the forest of $\mathcal{T}_{r_i,m_i}$ s. Further, we use $M = T \times \{1\}$ to denote the roots and $N = S \setminus M$ to denote the non-roots of the corresponding trees. We introduce the following Boolean variables:

- $\text{EDGE}(t, t')$ for all $t, t' \in T$, denoting the edges of the abstraction of $\mathcal{M}$ to G.

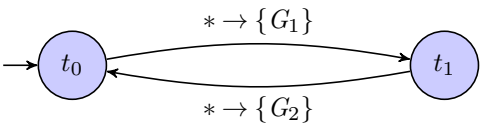

Figure 9. The implementation of the arbiter specification with the smallest number of states and cycles.

- $\text{BEDGE}(s, s')$ for all $s \in S$ and $s' \in N$, denoting a blue edge.
- $\text{REDGE}(s, s')$ for all $s \in S$ and $s' \in M$, denoting a red edge.
- $\text{WTREE}(s, i)$ for all $s \in S$, $0 < i \leq \log(n)$, denoting the witness-tree for each s. Thereby, each tree is referenced by a unique number encoded in binary using a logarithmic number of bits.
- $\text{VISITED}(s, t)$ for all $s \in S$ and $t \in T$, denoting the set of all vertices t, already visited at s, since leaving the root of the corresponding witness-tree.
- $\text{RBOUND}(c, i)$ for all $0 < c \leq m$, $0 < i \leq \log(n \cdot m)$, denoting an ordered list of all red edges, bounding the red edges of the forest.
- $\text{SCC}(k, t, i)$ for all $0 < k \leq n$, $t \in T$, and $0 \leq i < \log n$, denoting the SCC of t in the k-th sub-graph of G. The sub-graphs are obtained by iteratively removing vertices of T, according to the pre-defined order. This way, each sub-graph contains exactly all vertices that are larger than the root.

Note that, by the definition of S, we introduce m explicit copies for each vertex of G. This is sufficient, since each cycle contains each vertex at most once. Thus, the labeling τ of a vertex s can be directly derived from the first component of s.

Given the respective bounded synthesis encoding for the specification φ and a bound n on the states of the resulting implementation $\mathcal{M}$, and a bound m on the number of cycles of $\mathcal{M}$, we encode the bounded cycle synthesis problem as the propositional formula

$$\mathcal{F} = \mathcal{F}_{BS}(\varphi, n) \ \wedge \ \mathcal{F}_{CS}(n, m) \ \wedge \ \mathcal{F}_{\mathcal{M} \to G}(\varphi, n) \ \wedge \ \mathcal{F}_{SCC}(n)$$

The constraints of $\mathcal{F}_{BS}(\varphi, n)$ represent the bounded synthesis encoding. The constraints of $\mathcal{F}_{\mathcal{M} \to G}(\varphi, n)$ simplify the representation of the Mealy machine $\mathcal{M}$ to G. The constraints of $\mathcal{F}_{CS}(\mathcal{A}, n, m)$ bound the cycles of the system and are presented in Table 1. The constraints of $\mathcal{F}_{SCC}(n)$ enforce that each vertex is labeled by a unique SCC [10].

Theorem 2 (Bounded Cycle Synthesis [10]) *For an LTL formula φ and a pair of bounds $n, m \in \mathbb{N}$, the propositional formula $\mathcal{F}$ is satisfiable if and only if there is a Mealy machine $\mathcal{M}$ with $|\mathcal{M}| = n$ and $|\mathcal{C}(\mathcal{M})| = m$ that satisfies φ.*

Example 10 *Using our encoding, we can now search for the implementation of the arbiter specification from Example 1 with the smallest number of states and, additionally, smallest number of cycles. It turns out that neither Mealy machine from Fig. 1 is the minimal solution. The smallest implementation for the arbiter specification, with respect to the number of states and cycles is shown in Fig. 9.*

Table 1. Constraints of the SAT formula $\mathcal{F}_{CS}(\mathcal{A}, n, m)$.

$\bigwedge_{r \in T}$	$\textsc{wtree}((r,1)) = r$	Roots indicate the witness-tree.
$\bigwedge_{s \in S,\,(r,1) \in M}$	$\textsc{redge}(s,(r,1)) \rightarrow \textsc{wtree}(s) = r$	Red edges only connect vertices of the current $\mathcal{T}_{r_i,m_i}$.
$\bigwedge_{s \in S,\,s' \in N}$	$\textsc{bedge}(s,s') \rightarrow \textsc{wtree}(s) = \textsc{wtree}(s')$	Blue edges only connect vertices of the current $\mathcal{T}_{r_i,m_i}$.
$\bigwedge_{s' \in N}$	$\mathit{exactlyOne}(\{\textsc{bedge}(s,s') \mid s \in S\})$	Every non-root has exactly one blue incoming edge.
$\bigwedge_{(t,c) \in S,\,r \in T,}$	$\textsc{redge}((t,c),(r,1)) \rightarrow \textsc{edge}(t,r)$	Red edges are related to the edges of the graph G.
$\bigwedge_{(t,c) \in S,\,(t',c') \in N}$	$\textsc{bedge}((t,c),(t',c')) \rightarrow \textsc{edge}(t,t')$	Blue edges are related to the edges of the graph G.
$\bigwedge_{(t,c) \in S,\,r \in T,\,t \geq r}$	$\textsc{edge}(t,r) \wedge \textsc{scc}(r,t) = \textsc{scc}(r,r) \wedge \textsc{wtree}((t,c)) = r \rightarrow \textsc{redge}((t,c),(r,1))$	Every possible red edge must be taken.
$\bigwedge_{(t,c) \in S,\,r,t' \in T,\,t \geq t'}$	$\textsc{edge}(t,t') \wedge \textsc{scc}(r,t) = \textsc{scc}(r,t') \wedge \textsc{wtree}((t,c)) = r \wedge \textsc{visited}((t,c),t') \rightarrow \bigvee_{0 < c' \leq m} \textsc{bedge}((t,c),(t',c'))$	Every possible blue edge must be taken.
$\bigwedge_{r \in T}$	$\bigwedge_{t \leq r} \neg\textsc{visited}((r,1),t) \wedge \bigwedge_{t > r} \textsc{visited}((r,1),t)$	Only non-roots of the corresponding sub-graph can be successors of a root.
$\bigwedge_{(t,c) \in S,\,s \in N}$	$\textsc{bedge}((t,c),s) \rightarrow \neg\textsc{visited}(s,t) \wedge (\textsc{visited}(s,t') \leftrightarrow \textsc{visited}((t,c),t'))$	Every vertex appears at most once on a path from the root to a leaf.
$\bigwedge_{s \in S,\,s' \in M}$	$\textsc{redge}(s,s') \rightarrow \bigvee_{0 < c \leq m} \textsc{rbound}(c) = f(s)$	The list of red edges is complete. ($f(s)$ maps each state of S to a unique number in $\{1,\ldots,n \cdot m\}$)
$\bigwedge_{0 < c \leq m}$	$\textsc{rbound}(c) < \textsc{rbound}(c+1)$	Red edges are strictly ordered.

The minimal implementation switches the grant at every time step, completely ignoring the requests. This solution only requires two states and a single cycle. The solution may not be the best choice with respect to a possible target application, but it is definitely the smallest one.

In general, it is not always possible to minimize the two parameters simultaneously. There are specifications for which the smallest possible number of states and the smallest possible number of cycles cannot be realized within a single solution [10]. In such situations it may be helpful to have an explicit optimization function specified by the user that resolves the trade-off.

7. Conclusions

We have studied three different algorithms for the reactive synthesis problem. The classic game-based synthesis algorithm is input-sensitive in the sense that its performance is asymptotically optimal in the size of the specification, but it produces implementations that may be larger than necessary. Bounded synthesis produces implementations with a minimal number of states. Bounded cycle synthesis ad-

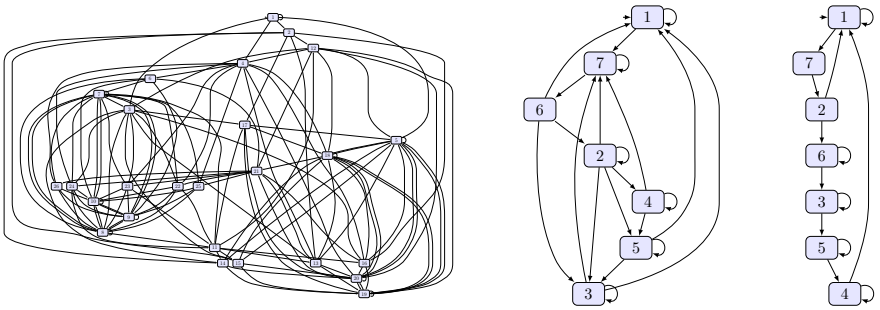

Figure 10. Three implementations of the TBURST4 component of the AMBA bus controller [10]. Game-based synthesis produces a Mealy machine with the shape shown on the left with 14 states and 61 cycles. Bounded synthesis produces the implementation in the middle with 7 states and 19 cycles. The implementation on the right, produced by bounded cycle synthesis, has 7 states and 7 cycles, which is the minimum.

ditionally minimizes the number of cycles. Bounded synthesis and bounded cycle synthesis belong to the new class of output-sensitive synthesis algorithms.

A direct comparison of the three algorithms is shown in Fig. 10. The figure depicts the shape of the synthesized implementations from the AMBA TBURST4 specification [10] using, from left to right, game-based synthesis, bounded synthesis, and bounded cycle synthesis. Even just based on a superficial visual comparison, it is immediately clear that the output-sensitive algorithms produce dramatically simpler implementations.

Acknowledgement. This work was partially supported by the European Research Council (ERC) Grant OSARES (No. 683300).

References

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

[2] R. P. Bloem, H.-J. Gamauf, G. Hofferek, B. Könighofer, and R. Könighofer. Synthesizing robust systems with RATSY. In *Proceedings of the Workshop on Synthesis (SYNT)*, volume 84, pages 47 – 53. Electronic Proceedings in Theoretical Computer Science, 2012.

[3] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer, 2012.

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

[5] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, volume 1, pages 3–50. Cornell Univ., Ithaca, NY, 1957.

[6] R. Ehlers. Unbeast: Symbolic bounded synthesis. In P. A. Abdulla and K. R. M. Leino, editors, *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 2011.

[7] P. Faymonville, B. Finkbeiner, M. N. Rabe, and L. Tentrup. Encodings of bounded synthesis. 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, pages 354–370, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[8] P. Faymonville, B. Finkbeiner, and L. Tentrup. Bosy: An experimentation framework for bounded synthesis. In *29th International Conference on Computer Aided Verification (CAV 2017)*, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[9] B. Finkbeiner and S. Jacobs. Lazy synthesis. In *13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, pages 219–234. Springer Verlag, 2012.

[10] 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.

[11] B. Finkbeiner and S. Schewe. SMT-based synthesis of distributed systems. In *Proceedings of the Second Workshop on Automated Formal Methods*, AFM '07, pages 69–76, New York, NY, USA, 2007. ACM.

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

[13] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[14] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV)*, pages 258–262, 2007.

[15] D. B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[16] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[17] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Boston, MA, USA, 1972.

[18] J. C. Tiernan. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Commun. ACM*, 13(12):722–726, 1970.

[19] H. Weinblatt. A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph. *J. ACM*, 19(1):43–56, 1972.