Acta Informatica (2020) 57:3–36
https://doi.org/10.1007/s00236-019-00349-3

ORIGINAL ARTICLE

Practical synthesis of reactive systems from LTL specifications via parity games

You *can* teach an old dog new tricks: making a classic approach structured, forward-explorative, and incremental

Michael Luttenberger[1] · Philipp J. Meyer[1] · Salomon Sickert[1]

Received: 16 January 2019 / Accepted: 2 November 2019 / Published online: 21 November 2019

Abstract
The synthesis of reactive systems from linear temporal logic (LTL) specifications is an important aspect in the design of reliable software and hardware. We present our adaption of the classic automata-theoretic approach to LTL synthesis, implemented in the tool Strix which has won the two last synthesis competitions (Syntcomp2018/2019). The presented approach is (1) *structured*, meaning that the states used in the construction have a semantic structure that is exploited in several ways, it performs a (2) *forward exploration* such that it often constructs only a small subset of the reachable states, and it is (3) *incremental* in the sense that it reuses results from previous inconclusive solution attempts. Further, we present and study different guiding heuristics that determine where to expand the on-demand constructed arena. Moreover, we show several techniques for extracting an implementation (Mealy machine or circuit) from the witness of the tree-automaton emptiness check. Lastly, the chosen constructions use a symbolic representation of the transition functions to reduce runtime and memory consumption. We evaluate the proposed techniques on the Syntcomp2019 benchmark set and show in more detail how the proposed techniques compare to the techniques implemented in other leading LTL synthesis tools.

1 Introduction

Synthesis refers to the problem of finding for a formal specification of an input–output relation a matching implementation [37], e.g. an (I/O)-transducer, a Mealy machine, a Moore machine or a circuit. In our case we focus on *linear temporal logic (LTL)* as the specification logic.

All the authors contributed equally to this work and are listed in alphabetical order.

This work was partially funded and supported by the German Research Foundation (DFG) projects 'Game-based Synthesis for Industrial Automation' (253384115) and 'Verified Model Checkers' (317422601) and the ERC Advanced Grant No. 787367 (PaVeS).

✉ Salomon Sickert
s.sickert@tum.de

Extended author information available on the last page of the article

While an asymptotically optimal synthesis algorithm has been given in [37], this approach and other algorithms solving this task[1] have not yet been successfully put into industrial practice. Tools able to deal with large specifications have been elusive and those that are available often produce subpar results compared to straight-forward manual implementations when successfully applied. [29] identifies four challenges that hinder the practical impact of these synthesis algorithms: "algorithmic, methodological, scope, and qualitative" [29]. The first challenge is to find efficient synthesis algorithms. The second challenge is the discrepancy between the assumed synthesis setting and reality: a finished and complete specification is the exception and not the rule. Users often iterate specifications and thus this brings up the task of reusing and composing intermediate results. The third challenge is expressiveness and succinctness of input and output formats. Finally, the fourth challenge is not only to compute any valid solution, but to find implementations that have *good quality*. In this paper we primarily address the algorithmic and qualitative side of the synthesis problem, but also sketch ideas for the two other areas.

The classic automata-theoretic synthesis procedure using deterministic automata suffers from the "messy state space" [29] of Safra's determinisation, which hinders efficient implementations that need to work on top of it. Moreover this automata-theoretic approach to synthesis requires the construction of a potentially double exponentially sized automaton (in the length of the specification) [22]. These two issues gave rise to "Safraless" approaches [5,9,14,27,30] to avoid the complicated state structure and to alleviate the state space explosion problem. Further, bounded synthesis adds to the synthesis problem an additional size constraint on the matching implementation. This effectively turns the synthesis problem into a search problem.

We, on the other hand, address the "messy state space" issue by employing a collection of "Safraless" LTL to *deterministic parity automaton (DPA)* translations [12,13,38] in combination with a special product automaton construction that includes a *latest appearance record (LAR)* construction and a formula decomposition in the spirit of [10,15,34,35]. Our construction recovers the Boolean structure present in the input specification. The state explosion problem is tackled by exploring the on-demand constructed parity game using a forward search resembling the optimisation described in [17]. This enables our customised strategy iteration [31] to leave most of the arena (and thus of the automaton) unexplored. Further, the decomposition allows us to split-off formulas and use them to prune the search-space which is a generalisation of a central insight from [39].

Further, we propose two heuristics guiding the construction of the arena in directions of probably decisive regions and thus focussing on important states, while skipping irrelevant parts. One approach is agnostic about the internal structure of the parity game, while the other one extracts information from the special parity automaton construction. Lastly, since we use strategy iteration to compute winning strategies, we can reuse so-far constructed solution attempts after expanding the arena, thereby reducing the amount of iterations until stabilisation is reached. We believe that our approach could be adapted to cache intermediate results (constructed automata, partial strategies) when using LTL synthesis interactively to speed up synthesis, which addresses the second area.

Regarding the quality of the synthesised structures "there is no emphasize [sic]" on constructing optimal or well-structured systems [29]. While in this paper we do not look at general methods for producing qualitatively *good* solutions and do not support synthesis under a specific quality measure, we provide a set of *best-effort* heuristics to produce *good* solutions: we make use of a range of post-processing steps to ensure that the solution is

[1] See [3] for an introduction to reactive synthesis and related graph games.

as small as possible. We also provide a modular encoding of the product automata into circuits that retains the Boolean structure of the specification. This approach surprisingly yields on some of the specifications used in the experimental evaluation smaller circuits compared to extracting a circuit out of a minimised Mealy machine. This data suggests that minimisation of the implementation represented as a Mealy machine might be in some cases diametral to generating small circuits. It seems that this area has not been studied enough and we think enriching specifications with additional, explicit optimisation goals is worthwhile, but currently these ideas have not manifested in specifications such as the SYNTCOMP2019 benchmarks.

We implement and test the outlined ideas within STRIX[2] [33], which relies on [28] for automata translations and [32] for parity game solving. An older version of STRIX won in the TLSF/LTL track in all six categories of SYNTCOMP2018 against other mature tools such as LTLSYNT [23], which also implements synthesis using parity games, and BOSY [14], which implements several bounded synthesis approaches. We further improve the prototype by representing the transition relation symbolically to address scalability issues for large alphabets and replace external tooling such as SPECULOOS with an internal implementation able to cope with larger systems. This newer version again won in all TLSF/LTL tracks of SYNTCOMP2019 against LTLSYNT.

The rest of the paper is structured as follows: after introducing preliminaries, we give a high-level overview of the synthesis procedure and detail it in the following subsections. We then put our improvements to the test by evaluating them on the SYNTCOMP2019 benchmarks and comparing them with the old version, LTLSYNT and BOSY. Each section, if appropriate, contains its specific discussion of related work.

Editorial Note. This paper is an extended version of the preliminary report published in [33]. The synthesis approach is the same, but we give a much more detailed explanation of the techniques used, e.g., the decomposition and the product automaton construction. Further, we describe new extensions for different exploration strategies in Sect. 3.2.1 and different encoding strategies in Sect. 3.3.2. We also give an updated experimental evaluation on a larger set of benchmarks and a comprehensive comparison with the old version and other tools.

2 Preliminaries

2.1 ω-languages and ω-automata

Let Σ be a finite alphabet. An ω-word w over Σ is an infinite sequence of letters $a_0a_1a_2\ldots$ with $a_i \in \Sigma$ for all $i \geq 0$ and an ω-language is a set of ω-words. The set of all ω-words is denoted Σ^ω. We denote the i-th letter of an ω-word w (starting at 0) by $w(i)$ and the infinite suffix $w(i)w(i+1)\ldots$ by w_i.

In this paper we focus on deterministic ω-automata with accepting conditions defined on transitions which is nowadays the preferred acceptance condition in implementations due to the succinctness and in-line with other recent papers and tools [2,8,20,28]. The discussed constructions can also be transferred to automata with acceptance defined on states with the folklore translation from transition acceptance to state acceptance.

[2] https://strix.model.in.tum.de/.

A *deterministic pre-automaton (DA)* over Σ is a tuple (Q, δ, q^0) where Q is a finite set of states, $\delta\colon Q \times \Sigma \to Q$ is a transition function, and q^0 is an initial state. A transition is a triple (q, a, q') such that $q' = \delta(q, a)$.

A *deterministic Parity automaton (DPA)* is a deterministic pre-automaton automaton $A = (Q, \delta, q^0, \chi, d, p)$ with the addition of the *transition colouring* $\chi\colon Q \times \Sigma \to \{0, 1, \ldots, d\}$, $d \geq 1$ the *maximal colour* and $p \in \{0, 1\}$ the *parity* that determines whether a run is accepting or not (as defined below). A *run* of A on an ω-word $w\colon \mathbb{N}_0 \to \Sigma$ is an ω-sequence of states $\rho\colon \mathbb{N}_0 \to Q$ such that $\rho(0) = q^0$ and for all positions $i \in \mathbb{N}_0$, we have that $(\rho(i), w(i), \rho(i+1)) \in \delta$. Given a run ρ over a word w, the infinite sequence of colours traversed by the run ρ is denoted by $\chi(\rho) := \big(\chi(\rho(i), w(i))\big)_{i \in \mathbb{N}_0}$. The minimal colour appearing infinitely often along a run ρ is $\liminf \chi(\rho)$. A run ρ is *accepting* if $\liminf \chi(\rho) \equiv_2 p$ (with $x \equiv_2 y :\Leftrightarrow (x - y) \bmod 2 = 0$). An ω-word w is in the *language* of A, denoted $w \in \mathcal{L}(A)$, iff the run for w on A is an accepting run.

For a parity $p \in \{0, 1\}$, define $\overline{p} := 1 - p$ as the *switched parity*. Note that by changing the parity p to $\overline{p}$, we obtain a *complement automaton* $\overline{A} := (Q, \delta, q^0, \chi, d, \overline{p})$ for which we have $\mathcal{L}(\overline{A}) = \Sigma^\omega \backslash \mathcal{L}(A)$.

To change the parity p to $\overline{p}$ while preserving the language, one can use $A' := (Q, \delta, q^0, \chi', d+1, \overline{p})$ with $\chi'(q, a) := \chi(q, a)+1$, which has one more colour and satisfies $\mathcal{L}(A') = \mathcal{L}(A)$.

A *deterministic Büchi automaton (DBA)* with the set of accepting transitions α is a DPA with colours $\{0, 1\}$, parity 0 and χ defined as:

$$\chi(q, a) := \begin{cases} 0 & \text{if } (q, a, \delta(q, a)) \in \alpha \\ 1 & \text{otherwise} \end{cases}$$

A *deterministic co-Büchi automaton (DCA)* with the set of rejecting transitions β is a DPA with colours $\{0, 1\}$, parity 1 and χ defined as:

$$\chi(q, a) := \begin{cases} 0 & \text{if } (q, a, \delta(q, a)) \in \beta \\ 1 & \text{otherwise} \end{cases}$$

A *deterministic weak automaton (DWA)* with the set of accepting states γ is a DPA with colours $\{0, 1\}$ and parity 0 or 1, where for each strongly connected component $S \subseteq Q$, either $S \subseteq \gamma$ or $S \cap \gamma = \emptyset$. Then χ is defined as:

$$\chi(q, a) := \begin{cases} p & \text{if } \delta(q, a) \in \gamma \\ \overline{p} & \text{otherwise} \end{cases}$$

Note that for weak automata, we can switch between parity 0 and 1 while preserving the language without increasing the number of colours.

A *bottom state* of a DPA is a special state $\bot \in Q$ such that $\delta(\bot, a) = \bot$ and $\chi(\bot, a) \equiv_2 \overline{p}$ for each $a \in \Sigma$. A *top state* of a DPA is a special state $\top \in Q$ such that $\delta(\top, a) = \top$ and $\chi(\top, a) \equiv_2 p$ for each $a \in \Sigma$.

2.2 Linear temporal logic

We present LTL [36] with a larger than usual set of modal operators and Boolean connectives, instead of a minimalistic syntax often found in other publications. While a minimalistic syntax reduces the amount of cases, e.g., in an induction, we want to keep as much structure of the given formula as possible and thus add redundancy. In particular, we are going to present customised constructions in order to deal with the Boolean connective $\leftrightarrow$ that effectively

reduces the size of the automaton that is constructed. We work with a syntax for LTL in which formulas *within* the scope of modal operators ($\mathbf{X}$, $\mathbf{U}$) are written in negation-normal form, i.e., negations only occur in front of atomic propositions. Thus we need to introduce $\mathbf{ff}$, $\neg a$, $\vee$, and the temporal operator $\mathbf{R}$ (release) in order to remove $\neg$ from the syntax. It is easy to see that LTL formulas with the usual syntax can be translated to equivalent LTL formulas of the same size in our syntax. A formula of LTL over a set of atomic propositions (Ap) is given by the following syntax:

Definition 1 (*Syntax of LTL*)

$$\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \leftrightarrow \varphi \mid \psi$$
$$\psi ::= \mathbf{tt} \mid \mathbf{ff} \mid a \mid \neg a \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \psi \mathbf{U} \psi \mid \psi \mathbf{R} \psi \qquad \text{with } a \in \mathsf{Ap}$$

We also define the usual abbreviations $\mathbf{F}\varphi := \mathbf{tt}\mathbf{U}\varphi$ (eventually) and $\mathbf{G}\varphi := \mathbf{ff}\mathbf{R}\varphi$ (always). The satisfaction relation $\models$ between ω-words over the alphabet $\Sigma := 2^{\mathsf{Ap}}$ and formulas is inductively defined as follows:

Definition 2 (*Semantics of LTL*)

$$\begin{array}{lll}
w \models \mathbf{tt} & & \\
w \not\models \mathbf{ff} & & \\
w \models a & \text{iff} & a \in w(0) \\
w \models \neg a & \text{iff} & a \notin w(0) \\
w \models \varphi \wedge \psi & \text{iff} & w \models \varphi \text{ and } w \models \psi \\
w \models \varphi \vee \psi & \text{iff} & w \models \varphi \text{ or } w \models \psi \\
w \models \varphi \leftrightarrow \psi & \text{iff} & w \models \varphi \text{ if and only if } w \models \psi \\
w \models \mathbf{X}\varphi & \text{iff} & w_1 \models \varphi \\
w \models \varphi \mathbf{U} \psi & \text{iff} & \exists k.\, w_k \models \psi \text{ and } \forall j < k.\, w_j \models \varphi \\
w \models \varphi \mathbf{R} \psi & \text{iff} & \forall k.\, w_k \models \psi \text{ or } \exists k.\, w_k \models \varphi \text{ and } \forall j \leq k.\, w_j \models \psi
\end{array}$$

We denote by $\mathcal{L}(\varphi)$ the language of φ defined as $\mathcal{L}(\varphi) := \{w \in \Sigma^\omega \mid w \models \varphi\}$.

2.3 Notable fragments of LTL

In the latter section we are going to consider the following four fragments of LTL:

- μLTL and νLTL:
  μLTL is the fragment of LTL restricted to the temporal operator $\mathbf{U}$, the Boolean connectives ($\wedge$, $\vee$), the literals (a, $\neg a$), and the next operator ($\mathbf{X}$). νLTL is defined analogously, but with the operator $\mathbf{R}$ instead of $\mathbf{U}$. In the literature μLTL is also called syntactic co-safety and νLTL syntactic safety.
- $\mathbf{G}(\mu LTL)$ and $\mathbf{F}(\nu LTL)$:
  These fragments contain the formulas of the form $\mathbf{G}\varphi$, where $\varphi \in \mu LTL$, and $\mathbf{F}\varphi$, where $\varphi \in \nu LTL$.

The reason for the names μLTL and νLTL is that $\mathbf{U}$ is a least-fixed-point operator, in the sense that its semantic is naturally formulated by a least fixed point, e.g., in the μ-calculus, while the semantics of $\mathbf{R}$ is naturally formulated by a greatest fixed point.

For all these fragments several translations to deterministic automata are known and we are going to use the constructions for μLTL, νLTL, $\mathbf{GF}(\mu LTL)$, and $\mathbf{FG}(\nu LTL)$ described in [13], for $\mathbf{G}(\mu LTL)$ and $\mathbf{F}(\nu LTL)$ the construction described in [38], and for arbitrary LTL formulas the construction described in [12]. It should be noted that of course

these constructions can be swapped with other constructions, but some of the implemented heuristics rely on the specific state structure these constructions yield.

2.4 Synthesis problem

Let φ be a specification given as an LTL formula and let the atomic propositions $\mathsf{Ap} = \mathsf{Ap}_{\mathsf{in}} \uplus \mathsf{Ap}_{\mathsf{out}}$ be partitioned into *input symbols* $\mathsf{Ap}_{\mathsf{in}}$ and *output symbols* $\mathsf{Ap}_{\mathsf{out}}$. We then define $\Sigma := 2^{\mathsf{Ap}}$, $\Sigma_{\mathsf{in}} := 2^{\mathsf{Ap}_{\mathsf{in}}}$, and $\Sigma_{\mathsf{out}} := 2^{\mathsf{Ap}_{\mathsf{out}}}$.

Then the *synthesis problem* is to decide if a function $\sigma : \Sigma_{\mathsf{in}}^* \to \Sigma_{\mathsf{out}}$ exists such that for every ω-word $v \in \Sigma_{\mathsf{in}}^\omega$, the ω-word $w \in \Sigma^\omega$ defined by $w(i) := v(i) \cup \sigma(v(0)v(1)\dots v(i))$ satisfies $w \in \mathcal{L}(\varphi)$. In the positive case, also a finite and executable representation of σ in the form of a controller should be produced, e.g., a Mealy machine or a circuit.

3 Synthesis procedure

We start with an overview on how Strix constructs parity games from specification formulas and solves them. For the controller extraction we refer the reader to Sect. 3.3. We illustrate the intuition of Algorithm 1 using a simple arbiter example and refer to functionality explained in subsequent sections via *oracles*: First, the formula is analysed with $\mathcal{O}_{\mathcal{T}}$ and a DPA is constructed on-the-fly via $\mathcal{O}_{q_0}$, $\mathcal{O}_p$, and $\mathcal{O}_\delta$. Second, the DPA is interpreted as a parity game and the game is solved via $\mathcal{O}_{\mathsf{win}}$ computing the winning regions. Further an exploration heuristic ($\mathcal{O}_{\mathsf{expl}}$) guides which parts of the parity games are extended. To be more precise, we use the following oracles:

- $\mathcal{O}_{\mathcal{T}}(\varphi)$: Given the formula φ, the oracle returns an annotated formula α that labels syntax nodes with a recommended automaton type to be used for this subformula for translation and how to compose a (product) DPA from these components.
- $\mathcal{O}_{q_0}(\alpha)$: Given the annotated formula α, the oracle returns the initial state q_0 of the (product) DPA recognising φ.
- $\mathcal{O}_p(\alpha)$: Given the annotated formula α, the oracle returns the parity p of the (product) DPA recognising φ.
- $\mathcal{O}_\delta(\alpha, q)$: Given the annotated formula α and a (product) state q, the oracle returns a set of outgoing transitions from q. The elements of the set are tuples (I, O, c, q'), where $I \subseteq \Sigma_{\mathsf{in}}$ are the input letters, $O \subseteq \Sigma_{\mathsf{out}}$ are the output letters, c is the colour of the transition and q' is the successor state. Formally for each such tuple (I, O, c, q') we have $\delta(q, i \cup o) = q'$ and $\chi(q, i \cup o) = c$ for all $i \in I$ and $o \in O$. The oracle will return $q' = \bot$ for states that are trivially losing for the system and $q' = \top$ for states that are trivially winning.
- $\mathcal{O}_{\mathsf{win}}$: Given $(V_\bigcirc, V_\Box, E, \chi, B, q, \mathsf{P}, p, \kappa)$, compute whether the state q is won by player $\mathsf{P} \in \{\bigcirc, \Box\}$, where $V_\bigcirc$ is the set of nodes from which player $\bigcirc$ moves, $V_\Box$ is the set of nodes from which player $\Box$ moves, $E \subseteq (V_\bigcirc \cup V_\Box) \times 2^\Sigma \times (V_\bigcirc \cup V_\Box)$ is the labeled edge relation, $\chi : E \to \mathbb{N}_0$ is an edge colouring, $B \subseteq V_\Box$ is the set of boundary nodes (i.e. nodes whose successors have yet to be constructed), p is the parity for player P, i.e. player P wins if the minimal colour occurring infinitely often along the edges of a play has parity p, and κ is an initial (partial, nondeterministic)[3] strategy for player P. $\mathcal{O}_{\mathsf{win}}$

[3] We will later see the advantage of using nondeterministic strategies (multiple actions allowed) compared to deterministic strategies (only one action allowed).

Algorithm 1 Forward-explorative, incremental synthesis algorithm.

Require: LTL formula φ, input letters Σ_{in}, output letters Σ_{out}
Ensure: (P, κ), where P is the winner of the game and κ is a corresponding (non-deterministic) strategy.
1: $\alpha \leftarrow \mathcal{O}_{\mathcal{T}}(\varphi)$
2: $q_0 \leftarrow \mathcal{O}_{q_0}(\alpha)$
3: $(V_\bigcirc, V_\Box) \leftarrow (\emptyset, \{q_0, \bot, \top\})$
4: $p \leftarrow \mathcal{O}_p(\alpha)$
5: $(E, \chi) \leftarrow (\{(\bot, \Sigma, \bot), (\top, \Sigma, \top)\}, \{((\bot, \Sigma, \bot), \overline{p}), ((\top, \Sigma, \top), p)\})$
6: $\sigma, \tau \leftarrow (\emptyset, \emptyset)$
7: $B \leftarrow \{q_0\}$
8: **while** $B \neq \emptyset$ **do**
9: $\quad X \leftarrow \mathcal{O}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, B, q_0, p, \sigma, \tau)$
10: $\quad B \leftarrow B \setminus X$
11: $\quad$ **for all** $q \in X$ **do**
12: $\quad\quad$ **for all** $(I, O, c, q') \in \mathcal{O}_\delta(\alpha, q)$ **do**
13: $\quad\quad\quad$ **if** $q' \notin V_\Box$ **then**
14: $\quad\quad\quad\quad B \leftarrow B \cup \{q'\}$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad\quad V_\bigcirc \leftarrow V_\bigcirc \cup \{(q, I)\}$
17: $\quad\quad\quad V_\Box \leftarrow V_\Box \cup \{q'\}$
18: $\quad\quad\quad E \leftarrow E \cup \{(q, I, (q, I)), ((q, I), O, q')\}$
19: $\quad\quad\quad \chi \leftarrow \chi \cup \{((q, I, (q, I)), \infty), (((q, I), O, q'), c)\}$
20: $\quad\quad$ **end for**
21: $\quad$ **end for**
22: $\quad (\text{won}_\bigcirc, \sigma) \leftarrow \mathcal{O}_{\mathsf{win}}(V_\bigcirc, V_\Box, E, \chi, B, q_0, \bigcirc, p, \sigma)$
23: $\quad$ **if** $\text{won}_\bigcirc$ **then**
24: $\quad\quad$ **return** $(\bigcirc, \sigma)$
25: $\quad$ **end if**
26: $\quad (\text{won}_\Box, \tau) \leftarrow \mathcal{O}_{\mathsf{win}}(V_\bigcirc, V_\Box, E, \chi, B, q_0, \Box, \overline{p}, \tau)$
27: $\quad$ **if** $\text{won}_\Box$ **then**
28: $\quad\quad$ **return** $(\Box, \tau)$
29: $\quad$ **end if**
30: **end while**

also returns an updated strategy κ', which is winning from q for P if q is won by player P. Depending on the player P boundary nodes are declared as winning for the opponent in order to correctly under-approximate the parity game on the completely constructed arena.

– $\mathcal{O}_{\mathsf{expl}}$: Given a set B of boundary nodes, the so far constructed arena and the intermediate strategies, the oracle returns a nonempty subset of B of nodes that should be further explored.

We use the specification of a simple arbiter as an example. In this setting two processes ($i \in \{1, 2\}$) request access to the critical section by raising the flag r_i and the arbiter eventually grants access to process i by raising g_i. Thus we have $\mathsf{Ap}_{\mathsf{in}} = \{r_1, r_2\}$ and $\mathsf{Ap}_{\mathsf{out}} = \{g_1, g_2\}$ and the following specification:

$$\phi = \underbrace{\mathbf{G}(\neg g_1 \vee \neg g_2)}_{\psi_0} \wedge \underbrace{\mathbf{G}(r_1 \rightarrow \mathbf{F} g_1)}_{\psi_1} \wedge \underbrace{\mathbf{G}(r_2 \rightarrow \mathbf{F} g_2)}_{\psi_2}$$

Applying the annotation oracle, we obtain from the specification ϕ an annotated syntax tree α shown in Fig. 1 that represents a decomposition of ϕ into subformulas w.r.t. the weakest class of deterministic automata needed for their translation and how to combine the automata in order to obtain the automaton for ϕ itself. In our example, ψ_0 is a simple mutex requirement which is classified as recognisable by a DWA (denoted by $\mathcal{W}$), and ψ_1 and ψ_2 are fairness

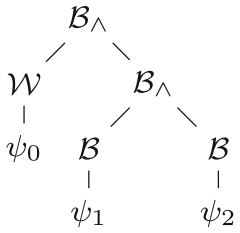
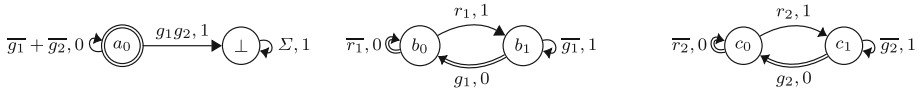


Fig. 1 Annotated syntax tree α for ϕ



(a) $\mathcal{A}_{\psi_0}$: DWA, $p = 0$ **(b)** $\mathcal{A}_{\psi_1}$: DBA, $p = 0$ **(c)** $\mathcal{A}_{\psi_2}$: DBA, $p = 0$

Fig. 2 DAs for ψ_0, ψ_1, and ψ_2. Note that ψ_1 and ψ_2 are isomorphic up to alphabet renaming

requirements that are classified as recognisable by a DBA (denoted by $\mathcal{B}$). The conjunctions $\psi_1 \wedge \psi_2$ and ϕ itself are then also recognisable by a DBA (denoted by $\mathcal{B}_\wedge$). The corresponding automata are displayed in Fig. 2.

We query $\mathcal{O}_{q_0}(\alpha)$ and obtain $q_0 := (a_0, ((b_0, c_0), 0))$, which matches the tree structure of α. Here, we have add a round-robin counter $r \in \{0, 1\}$ for the intersection of the two Büchi automata representing ψ_1 and ψ_2: This round-robin counter remembers which of the two Büchi automata is due to take an accepting transition. To ease notation for our example and the corresponding figures we flatten $(a_0, ((b_0, c_0), 0))$ to $(a_0, \underline{b_0}, c_0)$ with the underlined state representing the round-robin counter. Further, we query $\mathcal{O}_p(\alpha)$ for the parity associated with the controller, i.e. player ○. In our example it is 0 and ○ wins a play if the minimal colour encountered infinitely often is even.

As we are using Mealy semantics, we let the environment □ move from the initial node $(a_0, \underline{b_0}, c_0)$. Our parity game also includes two nodes $\bot$ and $\top$ where by our construction $\bot$ is always won by the environment □, while $\top$ is always won by the controller ○.

We now start the on-the-fly forward exploration of parity game arena. In every iteration of the while-loop in Algorithm 1 we extend the *boundary* B. The boundary always consists of nodes belonging to the environment □ whose successors have yet to be explored. Initially the boundary is just the initial node of the parity game, in our example $B = \{(a_0, \underline{b_0}, c_0)\}$.

As initially B is a singleton set, $\mathcal{O}_{\mathsf{expl}}$ tells us to explore all direct successors of $q_0 = (a_0, \underline{b_0}, c_0)$. $\mathcal{O}_\delta(\alpha, q_0)$ groups the outgoing transitions $\delta(q_0, \star)$ using Σ_{in} and Σ_{out} as previously mentioned as a set of tuples of the shape (I, O, q', c). Due to the Mealy semantics each such tuple (I, O, q', c) is broken up in two steps: starting in q_0, first the environment issues a signal $i \in I$, which leads the game into the intermediate state (q_0, I);[4] as this is only an intermediate step the corresponding edge is assigned the (w.r.t. min-parity) "neutral" colour ∞; in (q_0, I) the controller ○ then issues a signal $o \in O$ leading to the next state of the DPA. We illustrate this using our example: we have 4 choices for the inputs at $(a_0, \underline{b_0}, c_0)$:

1. $\overline{r_1}\,\overline{r_2} = \{\emptyset\}$, i.e. no process requests access;
2. $r_1\,\overline{r_2} = \{\{r_1\}\}$, i.e. only process 1 requests access;
3. $\overline{r_1}\,r_2 = \{\{r_2\}\}$, i.e. only process 2 requests access; and

[4] We represent the intermediate states simply by circular shaped nodes in the figures.

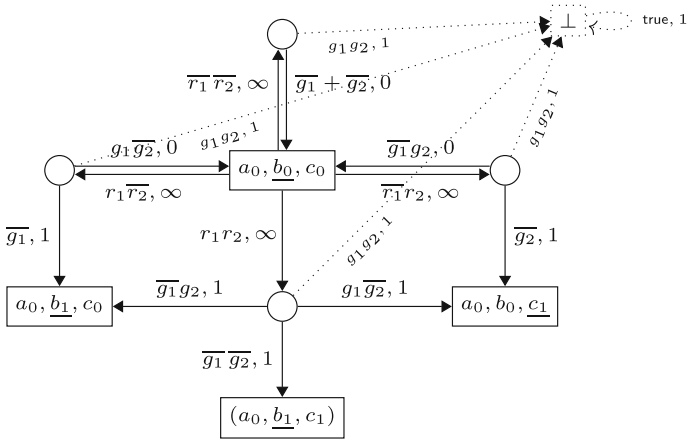


Fig. 3 Parity game arena after one iteration of the main loop

4. $r_1\,r_2 = \{\{r_1, r_2\}\}$, i.e. both processes want to access the critical section.

Consider the case that the environment chooses the input $I = r_1\,\overline{r_2}$. $\mathcal{O}_\delta$ groups the outputs available to $\bigcirc$ into the three groups $g_1\,g_2$ (grant access to both processes), $g_1\,\overline{g_2}$ (grant access to only process 1), and $\overline{g_1}$ (do not grant access to process 1). In case of $O = \overline{g_1} = \{\{\}, \{g_2\}\}$, the DBA $\mathcal{A}_{\psi_1}$ takes a non-accepting transition, hence, the round-robin counter stays unchanged, while the other two automata take a loop, arriving at state $(a_0, \underline{b_1}, c_0)$ in the product; $\mathcal{O}_\delta$ determines by analysing α that we only need the colours $\{0, 1\}$ for the parity game under construction; as $\mathcal{A}_{\psi_1}$ takes a non-accepting transition, $\mathcal{O}_\delta$ gives the input–output pair $(I, O) = (r_1\,\overline{r_2}, \overline{g_1})$ a colour of parity $\overline{p}$ in order to prevent $\bigcirc$ from replying to $r_1\,\overline{r_2}$ by $\overline{g_1}$ infinitely often. Thus $(r_1\,\overline{r_2}, \overline{g_1}, 1, (a_0, \underline{b_1}, c_0)) \in \mathcal{O}_\delta(\alpha, q_0)$. For $O = g_1\,\overline{g_2}$ we obtain analogously the entry $(r_1\,\overline{r_2}, g_1\,\overline{g_2}, 0, (a_0, \underline{b_0}, c_0))$: as all automata take an accepting loop in this case the input–output pair is given the colour 0 and, as the round-robin counter is incremented twice, we are back in the initial node of the parity game. Finally, for $O = g_1\,g_2$ the oracle $\mathcal{O}_\delta$ determines that $\bigcirc$ has no chance of winning anymore as the DWA $\mathcal{A}_{\psi_0}$ representing the mutex requirements cannot reach an accepting transition anymore; for this reason, $\mathcal{O}_\delta$ simplifies the successor state to $\bot$ which by construction is always won by the environment $\square$ and includes the tuple $(r_1\,\overline{r_2}, g_1 g_2, 1, \bot)$ into its output.[5] Analogously, $\mathcal{O}_\delta$ handles the other three possible inputs by $\square$ which eventually leads to the arena shown in Fig. 3 with border $B = \{(a_0, \underline{b_0}, c_1), (a_0, b_0, \underline{c_1}), (a_0, \underline{b_1}, c_1)\}$.

We now run the parity game solver, i.e. query $\mathcal{O}_{\mathsf{win}}$ for the so-far constructed arena. We first mark the boundary nodes as losing for the controller and ask $\mathcal{O}_{\mathsf{win}}$ for an optimal winning strategy for the controller. In this case, this leads to the nondeterministic strategy σ depicted in Fig. 4 where the thick edges in dark green belong to σ, while edges disabled by σ are drawn as dashed lines. We will describe our instantiation of $\mathcal{O}_{\mathsf{win}}$ in Sect. 3.2 in more detail, but intuitively $\mathcal{O}_{\mathsf{win}}$ tries to maximise the distance from the trivial losing states $\bot$ and the boundary B with each colour interpreted as a distance; hence, in the intermediate states corresponding to $I \in \{\overline{r_1}\,r_2, \overline{r_1}\,\overline{r_2}, r_1\,\overline{r_2}\}$, $\mathcal{O}_{\mathsf{win}}$ chooses to let the controller play back to the initial state so that ideally never the losing state is reached and thus the distance to it is maximised. One particular feature of our instantiation of $\mathcal{O}_{\mathsf{win}}$ is that it outputs nondeterministic strategies,

[5] Should $\mathcal{O}_\delta$ determine that all automata will accept any possible input–output pairs from now on, it simplifies the successor state to $\top$ which, again by construction, is always won by $\bigcirc$.

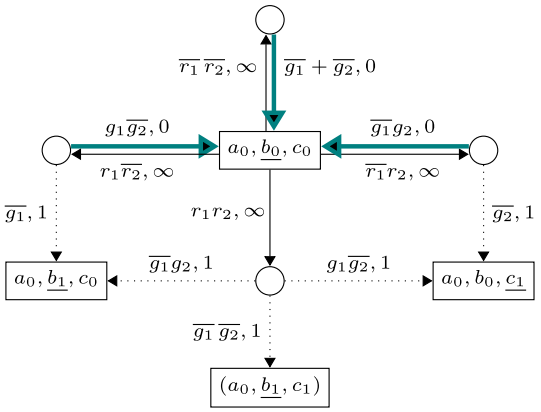


Fig. 4 Optimal strategy for the controller in the parity game arena of Fig. 3

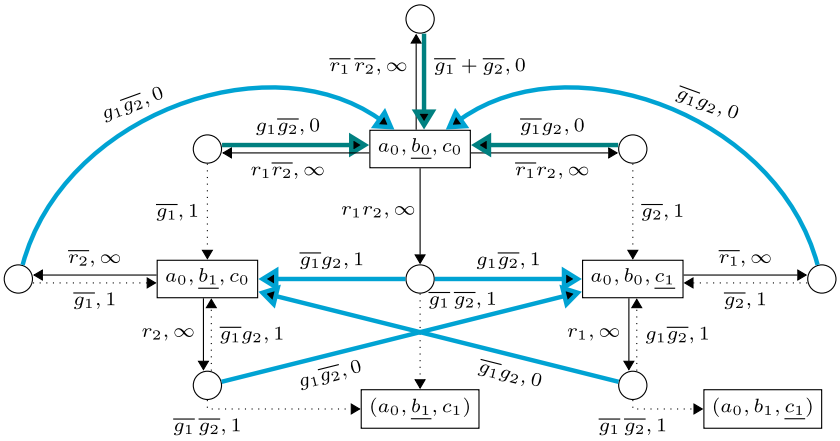


Fig. 5 Parity game arena after two iterations of the main loop and optimal winning strategy for the controller. For the sake of succinctness the colour $\chi(((a_0, \underline{b_1}, c_0), r_2), g_1\overline{g_2})$ was changed from 1 to 0 to reduce the number of iterations

i.e. a strategy σ for $\Box$ is still required to respect the edge relation of the parity game but $\sigma(q)$ is only required to be some subset of the successors of q in the parity game; in particular $\sigma(q) = \emptyset$ is allowed and has to be interpreted as $\bigcirc$ giving up at node q which is the case of the strategy shown in Fig. 4 at the intermediate node $(q_0, r_1\, r_2)$. Note that we do not draw $\bot$ and the corresponding edges in Figs. 4 and 5 as $\mathcal{O}_{\mathsf{win}}$ will always prefer to tell the controller $\bigcirc$ to give up instead of playing to $\bot$ (analogously for $\Box$ and $\top$). More importantly, if a strategy tells a player to give up at a specific node this means that the player loses any play reaching this specific node. For this reason, the computed strategy σ is not winning for the controller so far, hence, we also ask $\mathcal{O}_{\mathsf{win}}$ if the environment $\Box$ might win the initial state—now with the boundary marked as winning for the controller. As $\mathcal{O}_{\mathsf{win}}$ also fails to find a winning strategy for the environment ($\bigcirc$ can easily force $\Box$ directly into the boundary), we proceed to further explore and construct the arena.

In this iteration $\mathcal{O}_{\mathsf{expl}}$ now uses σ and τ: as neither the controller nor the environment wins the initial node so far, starting in $q_0 = (a_0, \underline{b_0}, c_0)$ each player can force his opponent into

the boundary; hence, σ and τ give us some information where to further explore the arena; further, from α we obtain scores that tell us how far from acceptance resp. rejection a given boundary state is w.r.t. to the product of the underlying automata; in our example, using α the oracle $\mathcal{O}_{\mathsf{expl}}$ tells us to first explore only the two states $(a_0, \underline{b_1}, c_0)$ and $(a_0, b_0, \underline{c_0})$ as in both cases only one of the two processes is waiting for being granted access. Proceeding as before, we further extend the parity game under construction leading to the parity game shown in Fig. 5 (which also shows the strategy obtained for $\bigcirc$) and again ask if either $\bigcirc$ or $\square$ can now win the initial node q_0. As the extended parity game coincides with that of the previous iteration in all but the boundary nodes, we pass the so-far computed strategies also to $\mathcal{O}_{\mathsf{win}}$ in order to re-use the information stored in them.

Fig. 5 shows the strategy that $\mathcal{O}_{\mathsf{win}}$ now computes: the updated σ coincides with the previous σ on the nodes where $\bigcirc$ did not give up (edges coloured in dark green); it only adds the edges coloured in light blue. The so updated σ now wins the initial node for the controller. In particular, σ keeps the nondeterminism at $(q_0, r_1\, r_2)$ where it only tells the controller to grant access to exactly one process but it does not tell $\bigcirc$ which one of the processes should be preferred. This ambiguity can be used when translating the strategy into a circuit or a program to reduce the description size. Finally note that as we mark the nodes on the boundary as losing for the respective "main" player when calling $\mathcal{O}_{\mathsf{win}}$, if the "main" player can win the so-far constructed parity game, then his strategy has to avoid the boundary, i.e. by construction we always find winning strategies that try to enclose all plays starting in the initial node in a "minimal" winning region. We remark that this bears some similarity to the local strategy iteration schemes by Friedmann [17]; but there the parity game is assumed to be explicitly given, and the goal is simply to speed-up strategy iteration itself; in our case the goal is to construct as little as possible from the actual parity game, while the actual choice of the oracle $\mathcal{O}_{\mathsf{win}}$ is unimportant at this point.

This brings us to the end of our walk-through of the main algorithm. In the following sections we describe in more detail how we choose to instantiate the oracles.

3.1 DPA construction

3.1.1 Formula analysis and decomposition

Before constructing a DPA the formula is analysed and its syntax-tree is annotated with automata acceptance conditions based on syntactic criteria. Such a formula decomposition focussed on conjunctions has been previously used in other work such as [10,15,34]. However, we will also consider disjunctions and bi-implications. In Sect. 2.1 we introduced the following three sub-classes of DPAs: DWAs, DBAs, and DCAs. Accordingly we annotate the LTL formula with "acceptance-typing" information:

Definition 3 [*Acceptance-Type Annotated LTL*]

$$\begin{array}{rllll} \alpha ::= & \mathcal{B}_\wedge(\alpha,\alpha) & \mid \mathcal{C}_\wedge(\alpha,\alpha) & \mid \mathcal{P}_\wedge(\alpha,\alpha) & \mid \mathcal{W}_\wedge(\alpha,\alpha) \\ & \mid \mathcal{B}_\vee(\alpha,\alpha) & \mid \mathcal{C}_\vee(\alpha,\alpha) & \mid \mathcal{P}_\vee(\alpha,\alpha) & \mid \mathcal{W}_\vee(\alpha,\alpha) \\ & \mid \mathcal{B}_\leftrightarrow(\alpha,\alpha) & \mid \mathcal{C}_\leftrightarrow(\alpha,\alpha) & \mid \mathcal{P}_\leftrightarrow(\alpha,\alpha) & \mid \mathcal{W}_\leftrightarrow(\alpha,\alpha) \\ & \mid \mathcal{B}(\varphi) & \mid \mathcal{C}(\varphi) & \mid \mathcal{P}(\varphi) & \mid \mathcal{W}(\varphi) \qquad \text{with } \varphi \in LTL \end{array}$$

We obtain an acceptance-typed LTL formula α from an LTL formula φ using the following heuristic approach: First, we determine syntactically the "simplest" acceptance type, denoted τ_φ, such that we can build a deterministic automaton with acceptance τ_φ for φ efficiently. Second, we annotate φ with this information and obtain α as the result of $\mathcal{T}_\varphi$. Formally:

Definition 4 Let φ be a formula. Then τ is recursively defined as:

$$\tau_\varphi = \begin{cases} \tau_{\psi_1} \sqcup \tau_{\psi_2} & \text{if } \varphi = \psi_1 \; op \; \psi_2 \text{ with } op \in \{\wedge, \vee\} \\ \mathcal{W} & \text{if } \varphi \in \mu LTL \cup \nu LTL \\ & \quad \text{or if } \varphi = \psi_1 \leftrightarrow \psi_2 \text{ and } \{\tau_{\psi_1}, \tau_{\psi_2}\} = \{\mathcal{W}\} \\ \mathcal{B} & \text{if } \varphi \in \mathbf{G}(\mu LTL) \\ \mathcal{C} & \text{if } \varphi \in \mathbf{F}(\nu LTL) \\ \mathcal{P} & \text{otherwise} \end{cases}$$

where $\sqcup$ denotes the least upper bound relative to the partial order $\preceq$ defined by $\mathcal{W} \prec \mathcal{B}$, $\mathcal{W} \prec \mathcal{C}$, $\mathcal{B} \prec \mathcal{P}$, $\mathcal{C} \prec \mathcal{P}$, $\mathcal{B} \not\preceq \mathcal{C}$ and $\mathcal{C} \not\preceq \mathcal{B}$. The acceptance-type annotated formula $\mathcal{T}_\varphi$ is then recursively defined as:

$$\mathcal{T}_\varphi = \begin{cases} (\tau_\varphi)_{op}(\mathcal{T}_{\psi_1}, \mathcal{T}_{\psi_2}) & \text{if } \varphi = \psi_1 \; op \; \psi_2 \text{ with } op \in \{\wedge, \vee, \leftrightarrow\} \\ & \quad \text{and } \{\tau_{\psi_1}, \tau_{\psi_2}\} \neq \{\mathcal{P}\} \\ \tau_\varphi(\varphi) & \text{otherwise} \end{cases}$$

In a specific implementation this decomposition and annotation might be fine-tuned to allow better translation performance, e.g. safety properties classified as separate weak subformulas might be grouped for performance reasons.

Further, let $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ be acceptance-typed LTL formulas. We make use of the following simple pattern matching notation:

- $\mathcal{X} = \mathcal{B}_\wedge(\mathcal{X}_1, \mathcal{X}_2)$: Here $\mathcal{X}_1$ and $\mathcal{X}_2$ are fresh variables binding to the left and right subtree of $\mathcal{X}$, which is constrained to be a conjunction typed as Büchi acceptance.
- $\mathcal{Y} = \mathcal{Y}_\vee(\mathcal{C}_1, \mathcal{Y}_2)$: Here $\mathcal{C}_1$ and $\mathcal{Y}_2$ are fresh variables binding to the left and right subtree of $\mathcal{Y}$. Further, $\mathcal{C}_1$ has to be typed as co-Büchi acceptance and $\mathcal{Y}$ can be typed with any acceptance condition, but needs to be a disjunction.
- $\mathcal{Z} = \mathcal{W}_\leftrightarrow(\mathcal{W}_1, \mathcal{W}_2)$: Here $\mathcal{W}_1$ and $\mathcal{W}_2$ are fresh variables binding to the left and right subtree of $\mathcal{Z}$ and are both typed with weak acceptance. Moreover, $\mathcal{Z}$ is a bi-implication with weak acceptance.

Moreover, instead of only binary conjunctives, we use Büchi conjunction and co-Büchi disjunction of sets, i.e. we add

$$\alpha ::= \mathcal{B}_\wedge(\mathcal{B}_1, \ldots, \mathcal{B}_n) \mid \mathcal{C}_\vee(\mathcal{C}_1, \ldots, \mathcal{C}_n)$$

for any $n \geq 2$ to the syntax. We restrict this rule to applications where all children are in the Büchi class for $\mathcal{B}_\wedge$ (resp. co-Büchi class for $\mathcal{C}_\vee$). After computing $\mathcal{T}_\varphi$, successive conjunctions are directly grouped together with the rule $\mathcal{B}_\wedge(\mathcal{B}_\wedge(\mathcal{B}_1, \mathcal{B}_2), \mathcal{B}_3) = \mathcal{B}_\wedge(\mathcal{B}_1, \mathcal{B}_\wedge(\mathcal{B}_2, \mathcal{B}_3)) = \mathcal{B}_\wedge(\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3)$, and the respective rule for $\mathcal{C}_\vee$.

3.1.2 Product construction with LAR

Given an acceptance-type annotated syntax tree $\mathcal{T}_\varphi$ for a formula φ, we now describe a recursive procedure to construct a (transition-based) DPA $A(\mathcal{T}_\varphi)$ with $\mathcal{L}(A(\mathcal{T}_\varphi)) = \mathcal{L}(\varphi)$. Observe that not all patterns that are syntactically possible are covered, but all patterns generated by $\mathcal{T}_\varphi$.

Base Case. In the case $\mathcal{T}_\varphi \in \{\mathcal{W}(\varphi), \mathcal{B}(\varphi), \mathcal{C}(\varphi), \mathcal{P}(\varphi)\}$ we use one of the direct automata constructions described in Sect. 2.3.

Conjunction. Now consider the case for the conjunction $\mathcal{T}_\varphi = \mathcal{X}_\wedge(\mathcal{T}_{\psi_1}, \mathcal{T}_{\psi_2})$, or $\mathcal{T}_\varphi = \mathcal{B}_\wedge(\mathcal{T}_{\psi_1}, \ldots, \mathcal{T}_{\psi_n})$. We start the construction by recursively constructing DPAs $A(\mathcal{T}_{\psi_i}) = (Q_i, \delta_i, q_i^0, \chi_i, d_i, p_i)$ for each child $\mathcal{T}_{\psi_i}$. Then we apply a case distinction based on $\mathcal{T}_\varphi$ and each $\mathcal{T}_{\psi_i}$.

– In the cases where we have $\mathcal{X}_\wedge(\mathcal{W}_1, \mathcal{X}_2)$ (and symmetrically $\mathcal{X}_\wedge(\mathcal{X}_1, \mathcal{W}_2)$), we can use a simple product construction. Then define the DPA

$$A(\mathcal{X}_\wedge(\mathcal{W}_1, \mathcal{X}_2)) := (Q_1 \times Q_2, \delta, (q_1^0, q_2^0), \chi, d_2, p_2)$$

with:

$$\delta(q, a) := (\delta_1(q_1, a), \delta_2(q_2, a)) \qquad \chi(q, a) := \begin{cases} \chi_2(q_2, a) & \text{if } \chi_1(q_1, a) = p_1 \\ \overline{p_2} & \text{if } \chi_1(q_1, a) \neq p_1 \end{cases}$$

– In the cases where we have $\mathcal{X}_\wedge(\mathcal{C}_1, \mathcal{X}_2)$ (and symmetrically $\mathcal{X}_\wedge(\mathcal{X}_1, \mathcal{C}_2)$), we can also use a product construction, with possibly one extra colour. W.l.o.g. assume $p_2 = 1$. This can be achieved by switching the parity of $A(\mathcal{X}_2)$ if necessary. Then define the DPA

$$A(\mathcal{X}_\wedge(\mathcal{C}_1, \mathcal{X}_2)) := (Q_1 \times Q_2, \delta, (q_1^0, q_2^0), \chi, d_2, 1)$$

with:

$$\delta(q, a) := (\delta_1(q_1, a), \delta_2(q_2, a)) \qquad \chi(q, a) := \begin{cases} 0 & \text{if } \chi_1(q_1, a) = 0 \\ \chi_2(q_2, a) & \text{if } \chi_1(q_1, a) = 1 \end{cases}$$

– Next, we consider the case $\mathcal{B}_\wedge(\mathcal{B}_1, \ldots, \mathcal{B}_n)$ with two or more Büchi children, and only Büchi children. Here, on top of a product construction, we need an additional *round-robin-counter* to track of successive satisfaction of the Büchi acceptance of the children. We define the DBA

$$A(\mathcal{B}_\wedge(\mathcal{B}_1, \ldots, \mathcal{B}_n)) := (Q \times \{0, 1, \ldots, n-1\}, \delta, (q^0, 0), \chi, 1, 0)$$

with $Q := (Q_1 \times \ldots \times Q_n)$, $q^0 := (q_1^0, \ldots, q_n^0)$ and

$$\delta((q, r), a) := ((\delta_1(q_1, a), \ldots, \delta_n(q_n, a)), r' \bmod n)$$

$$\chi((q, r), a) := \begin{cases} 0 & \text{if } r' = n \\ 1 & \text{if } r' < n \end{cases}$$

where $r' := \max\{s \in \{r, r+1, \ldots, n\} \mid \forall r < j \leq s : \chi_j(q_j, a) = 0\}$.

– Last, we consider the case $\mathcal{P}_\wedge(\mathcal{B}_1, \mathcal{P}_2)$ (and symmetrically $\mathcal{P}_\wedge(\mathcal{P}_1, \mathcal{B}_2)$). W.l.o.g. we may assume $p_2 = 1$ by switching parity if necessary. Here, we need additional memory to remember the *minimal colour* of $\mathcal{P}_2$ between acceptances of $\mathcal{B}_1$. We define the DPA

$$A(\mathcal{P}_\wedge(\mathcal{B}_1, \mathcal{P}_2)) := (Q_1 \times Q_2 \times \{0, 1, \ldots, d_2\}, \delta, ((q_1^0, q_2^0), d_2), \chi, d, 1)$$

with $d := \min\{d \in \{d_2, d_2 + 1\} \mid d \equiv_2 0\}$ and

$$\delta((q, c), a) := \begin{cases} ((\delta_1(q_1, a), \delta_2(q_2, a)), d_2) & \text{if } \chi_1(q_1, a) = 0 \\ ((\delta_1(q_1, a), \delta_2(q_2, a)), c') & \text{otherwise} \end{cases}$$

$$\chi((q, c), a) := \begin{cases} c' & \text{if } \chi_1(q_1, a) = 0 \\ d & \text{otherwise} \end{cases}$$

where $c' := \min(c, \chi_2(q_2, a))$.

Note that if some child in a conjunction reaches a non-accepting sink, then we also know that the conjunction can never accept again, and we can simplify the product state. A similar argument holds if all children reach a accepting sink. Formally, we replace δ by δ' and χ by χ' defined by:

$$\delta'(q,a) := \begin{cases} q & \text{if } q \in \{\bot, \top\} \\ \top & \text{if for all } i \text{ we have } \delta_i(q_i,a) = \top \\ \bot & \text{if for some } i \text{ we have } \delta_i(q_i,a) = \bot \\ \delta(q,a) & \text{otherwise} \end{cases}$$

$$\chi'(q,a) := \begin{cases} p & \text{if } \delta'(q,a) = \top \\ \overline{p} & \text{if } \delta'(q,a) = \bot \\ \chi(q,a) & \text{otherwise} \end{cases}$$

Disjunction. The construction of the DPA $A(\mathcal{X}_\vee(\mathcal{X}_1, \mathcal{X}_2))$ or $A(\mathcal{C}_\vee(\mathcal{C}_1, \ldots, \mathcal{C}_n))$ for the disjunctive $\mathcal{X}_\vee$ is dual to the conjunction case.

Bi-implication. Finally, we consider the bi-implication $\mathcal{X}_\leftrightarrow(\mathcal{X}_1, \mathcal{X}_2)$. This can be expressed through $\mathcal{X}_\wedge$ and $\mathcal{X}_\vee$ by the logical equivalence $\varphi \leftrightarrow \psi \equiv (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$. However, this construction would increase the state space and number of colours in some cases, since four automata $(\varphi, \neg\varphi, \psi, \neg\psi)$ instead of two (φ, ψ) need to be constructed. Therefore we have a special construction for the DPA $A(\mathcal{X}_\leftrightarrow(\mathcal{X}_1, \mathcal{X}_2))$. As before, we start by constructing the DPAs for the children $\mathcal{X}_1, \mathcal{X}_2$. Let $A(\mathcal{X}_i) = (Q_i, \delta_i, q_i^0, \chi_i, d_i, p_i)$ for $i \in \{1, 2\}$.

– In the case $\mathcal{W}_\leftrightarrow(\mathcal{W}_1, \mathcal{W}_2)$, we can apply a simple product construction. W.l.o.g. assume $p_1 = p_2$ by switching parity if necessary. We define the DWA

$$A(\mathcal{W}_\leftrightarrow(\mathcal{W}_1, \mathcal{W}_2)) := (Q_1 \times Q_2, \delta, (q_1^0, q_2^0), \chi, 1, 0)$$

with:

$$\delta(q,a) := (\delta_1(q_1,a), \delta_2(q_2,a)) \qquad \chi(q,a) := (\chi_1(q_1,a) + \chi_2(q_2,a)) \bmod 2$$

– Now we consider the general case for $\mathcal{P}_\leftrightarrow(\mathcal{X}_1, \mathcal{X}_2)$ (note that $\mathcal{B}_\leftrightarrow$ and $\mathcal{C}_\leftrightarrow$ never occur). W.l.o.g. assume that $\mathcal{X}_1 \neq \mathcal{P}$ and thus $d_1 = 1$. We consider $\mathcal{X}_2$ to be of class $\mathcal{P}$ and need to store its *minimal colour* between acceptances of $\mathcal{X}_1$, as for conjunction. However, whenever $\mathcal{X}_1$ does not accept, we emit the colour of A_2, shifted by one. If $\mathcal{X}_1 = \mathcal{W}$, we can actually omit the memory to store the colour of $\mathcal{X}_2$. We define the DPA

$$A(\mathcal{P}_\leftrightarrow(\mathcal{X}_1, \mathcal{X}_2)) := ((Q_1 \times Q_2) \times \{0, 1, \ldots, d_2\}, \delta, ((q_1^0, q_2^0), d_2), \chi, d, p)$$

with $d := d_2 + 1$, $p := (p_1 + p_2) \bmod 2$ and

$$\delta((q,c),a) := \begin{cases} ((\delta_1(q_1,a), \delta_2(q_2,a)), d_2) & \text{if } \chi_1(q_1,a) = p_1 \text{ or } \mathcal{X}_1 = \mathcal{W} \\ ((\delta_1(q_1,a), \delta_2(q_2,a)), c') & \text{if } \chi_1(q_1,a) \neq p_1 \end{cases}$$

$$\chi(q,a) := \begin{cases} c' & \text{if } \chi_1(q_1,a) = p_1 \\ \chi_2(q_2,a) + 1 & \text{if } \chi_1(q_1,a) \neq p_1 \end{cases}$$

where $c' := \min(c, \chi_2(q_2,a))$.

– We note that the construction of the previous case can be generalized to the product of two arbitrary DPAs by remembering for each colour c of $A(\mathcal{X}_1)$ the minimal colour of $A(\mathcal{X}_2)$ between minimal occurrences of c in $A(\mathcal{X}_1)$.

As with conjunction, we apply the simplification to $A(\mathcal{X}_{\leftrightarrow}(\mathcal{X}_1, \mathcal{X}_2))$, that if both children reach a state in $\{\bot, \top\}$, also the product state is either $\bot$ or $\top$. Replace δ by δ' and χ by χ' defined by:

$$\delta'(q,a) := \begin{cases} q & \text{if } q \in \{\bot, \top\} \\ \top & \text{if } \delta_1(q_1,a), \delta_2(q_2,a) \in \{\bot, \top\} \text{ and } \delta_1(q_1,a) = \delta_2(q_2,a) \\ \bot & \text{if } \delta_1(q_1,a), \delta_2(q_2,a) \in \{\bot, \top\} \text{ and } \delta_1(q_1,a) \neq \delta_2(q_2,a) \\ \delta(q,a) & \text{otherwise} \end{cases}$$

$$\chi'(q,a) := \begin{cases} p & \text{if } \delta'(q,a) = \top \\ \overline{p} & \text{if } \delta'(q,a) = \bot \\ \chi(q,a) & \text{otherwise} \end{cases}$$

Let φ be a formula and let $\alpha = \mathcal{T}_\varphi$ be the acceptance-type annotated formula. We then implement the oracles $\mathcal{O}_p$, $\mathcal{O}_{q_0}$, and $\mathcal{O}_\delta$ by the DPA $A(\alpha) = (Q, \delta, q^0, \chi, d, p)$ in the following way: $\mathcal{O}_p(\alpha) := p$, $\mathcal{O}_{q_0}(\alpha) := q^0$, and $\mathcal{O}_\delta(\alpha, q) := \{(\{i\}, \{o\}, \chi(q, i \cup o), \delta(q, i \cup o)) \mid i \in \Sigma_{\mathsf{in}}, o \in \Sigma_{\mathsf{out}}\}$.

3.1.3 Implementation details

On-the-fly construction. For this construction, the functions δ and χ only need to query the local state, and never need a global state. They might call the functions δ and χ for their children, but those also only depend on the local state and their respective children. Therefore it is possible to implement the construction of the DPA $A(\alpha)$ for a decomposition α *on-the-fly*: Starting with the initial state, successor states are only generated when necessary, and the DPA is not fully constructed until all states have been queried. This holds both for the root automaton and for any child automata constructed by the decomposition.

Memoization. When querying the root DPA for successors, the successors of the same state in a child DPA may be needed several times. Instead of recomputing them each time, the successors of the state are *cached* or *memoized* for direct access.

Formula Isomorphism. Building up on the memoization feature the construction only constructs one automaton for a pair of formulas isomorphic under renaming atomic propositions und remaps letters in the query stage, effectively reducing the automata states needed to be constructed for parametric formulas, where the same pattern is repeated with different atomic propositions.

Symbolic Construction and Representation of Transition Relations. The description of the oracle interface only specifies that successor function represented by a set needs to be returned. An implementation can choose to represent such a transition relation explicitly (e.g. by a list of length 2^n) or symbolically (e.g. by an MTBDD). In fact Strix 19.07 relies on a symbolic construction and representation of the transition relation of the arena and the automata. This then allows efficient (symbolic) grouping of inputs and outputs into equivalence classes by $\mathcal{O}_\delta$.

3.2 Parity game solver

We instantiate $\mathcal{O}_{\mathsf{win}}$ with a variant of the strategy iteration algorithm in [31].

We first give a brief description of what strategy iteration is and why we deem it particularly useful when combined with a demand-driven construction of the arena. We then exemplify these ideas in a bit more detail using our preceding example from Sec. 3.

In brief, strategy iteration consists of improving the current strategy (resp. controller) by iterating the two steps: (i) compute the "worst case" the environment can inflict on the system w.r.t. the current strategy; (ii) state-wise redefine the current strategy by selecting any successor(s) which lead to a better "worst case". The iteration stops as soon as the strategy cannot be improved anymore i.e. if the currently selected successors led to the best "worst case". Strategy iteration nicely combines with our approach of constructing the actual arena in a demand driven way: we only construct the arena to that extent that allows the controller to stay within the constructed subarena; all nodes resp. edges which are outside of this subarena are simply flagged to be lost to the controller, when checking for realisability, resp. the system, when checking for unrealisability; in addition, if we need to further explore the arena as we could neither prove realisability nor unrealisability using the subarena constructed so far, we can re-use the already computed strategies as initial strategies for computing the optimal strategies for the extended subarena.

We exemplify these ideas now. Let $\mathcal{A} = (V_\bigcirc, V_\Box, E, \chi, B)$ be a parity game arena with nodes $V = V_\bigcirc \cup V_\Box$ split between the two players $\bigcirc$ and $\Box$, edges E, edge colouring $\chi : E \to \mathbb{N}_0$, and boundary nodes $B \subseteq V$. We assume that $\mathcal{A}$ includes two special nodes $\bot$ and $\top$ where $\bot$ is always won by the environment $\Box$ and $\top$ is always won by the controller $\bigcirc$. We further require that all nodes in $V \setminus B$ have at least one successor w.r.t. E.[6] To simplify notation, we forget about the inputs and outputs that also label the edges (as shown e.g. in Fig. 3) s.t. we can simply write vE for the set of successors of the node v.

Besides $\mathcal{A}$, the parity game solver takes as additional input the "main player" $\mathsf{P} \in \{\bigcirc, \Box\}$, the parity $p \in \{0, 1\}$ with which P wins a play, a node $q \in V$ of the arena whose winner we want to determine, and an initial strategy κ for the main player P. We will write $\overline{\mathsf{P}}$ for the opponent of P s.t. $\{\mathsf{P}, \overline{\mathsf{P}}\} = \{\bigcirc, \Box\}$ with the parity of the opponent $\overline{\mathsf{P}}$ being $\overline{p}$ accordingly. All nodes in B are considered to be losing for the main player P, i.e. $\overline{\mathsf{P}}$ can win be forcing P into B.

In order to solve the parity game, we reformulate the winning condition into a sup-inf-distance problem: To this end, we first introduce an (implicit) auxiliary node $\bullet$ to which only the main player P can move to in order to *give up*. This node is in addition to the two nodes $\top$ and $\bot$ and only serves to simplify presentation. All edges leading to $\bullet$ are defined to be coloured by ∞ ("don't care"). We denote such a modification of an arena $\mathcal{A}$ by $\mathcal{A}^\bullet$.

We interpret the edge colours $\chi(v, w)$ as weights $\gamma(v, w)$ that measure how close P comes to winning resp. losing when taking the corresponding edge. Let $C \subseteq \mathbb{N}_0$ be the set of all colours occurring in $\mathcal{A}$ except for ∞. (Recall that we used the colour "∞" for edges that are unimportant w.r.t. the winning condition.) A colour $c \in C$ is identified with the multiset $\{c\}$ which we represent by its characteristic function w.r.t. $\mathbb{N}_0^C$. The colour ∞ is identified with the empty multiset resp. its characteristic function. Addition on $\mathbb{N}_0^C$ is defined point-wise as usual s.t. it coincides with the union of multisets. The weight of a finite play is then simply the sum of the weights of the edges traversed by it, i.e. the multiset of the colours of the edges of a play.

We order $\mathbb{N}_0^C$ from the point of view of P by means of the following order relation $\prec_\mathsf{P}$: Given two distinct functions $g, g' \in \mathbb{N}_0^C$, let $c := \min\{c' \in C \mid g(c') \neq g'(c')\}$ be the least colour in which the two differ; if c has parity p, we set $g \prec_\mathsf{P} g'$ if and only if $g(c) \prec g'(c)$;

[6] Actually, the edges are also labeled by corresponding inputs and outputs coming from the environment and the controller, respectively.

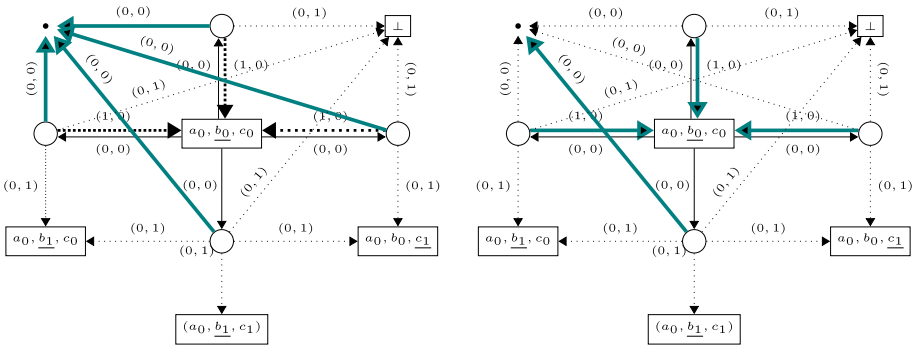


Fig. 6 The arena of Fig. 3 extended by the auxiliary node •. The colours 0, 1 and ∞ have been transformed into the respective edge weights (1, 0), (0, 1) and (0, 0) in $\mathbb{N}_0^C$. We assume that we have to decide whether the controller $\mathsf{P} = \bigcirc$ can win in the initial node (a_0, b_0, c_0). Here, the parity of P is $p = 0$ so the colour 0 resp. the tuple (1, 0) is positive from the point of view of ○, while the colour 1 resp. the tuple (0, 1) is negative. When not given an initial strategy, we start with the strategy that tells P to give up at every node controlled by P, thereby preventing the existence of negative cycles in any case, as shown in the figure on the left. This leads to the sup-inf-distance to be (0, 0) at every node s.t. playing back to the initial node is an improvement as it closes in each case a cycle of positive weight. Choosing all these improvements (thick dotted edges) yields the strategy shown in the figure on the right; after removing the auxiliary node •, this yields the strategy shown already in Fig. 4

else if c has parity $\overline{p}$, we set $g \prec_{\mathsf{P}} g'$ if and only if $g(c) > g'(c)$. For instance, taking a look at the arena of Fig. 3, the colours 0 and 1 are mapped on the functions (represented as tuples) (1, 0) and (0, 1), respectively. From the perspective of the controller ○ and its winning parity 0, the weight (1, 0) (representing the colour 0) is more attractive than the weight (0, 1) (representing the colour 1), i.e. $(1, 0) \prec_{\mathsf{P}} (0, 1)$.

To make the interpretation of the functions in $\mathbb{N}_0^C$ as weights more intuitive, let us remark that one can recover the order $\prec_{\mathsf{P}}$ by reading the function $g \in \mathbb{N}_0^C$ as a numeral w.r.t. the alternating basis $-b$, i.e. g is interpreted as the integer $\sum_{c \in C} g(c) \cdot (-1)^{p+c} \cdot b^{-c+\max C}$ where b is any sufficiently large positive integer, e.g. $b = |V|$. This ensures that in every simple cycle in $\mathcal{A}$ we have that the cycle is won by P if and only if the total weight of the cycle is positive, i.e. staying forever in the cycle leads to gaining infinite distance. For instance, in the arena of Fig. 6 we could choose $b = 2$ s.t. (1, 0) is mapped onto 2, while (0, 1) is mapped on -1.

P's goal thus becomes to maximise the distance to losing, i.e. the minimal total weight accumulated along a play—thus P will only accept infinite plays if these yield an infinite distance to losing, otherwise P will use the option to play to • (or into the boundary) in order to terminate a play and bound the distance to losing to a finite value.

Computation of the sup-inf-distances in $\mathcal{A}^\bullet$ is complicated by the existence of both positive and negative cycles (w.r.t. the interpretation of $\mathbb{N}_0^C$ as numerals). For this reason, we use strategy iteration, i.e. we construct a sequence of strategies for P where each strategy only allows for positive cycles, while negative cycles are prevented by playing to •: Assume P uses a (nondeterministic memoryless) strategy κ (i.e. $\emptyset \neq \kappa(v) \subseteq vE$ for all $v \in V_P$) s.t. in the accordingly restricted arena $\mathcal{A}_\kappa^\bullet$ (i.e. P may only move to a successor in $\kappa(v)$ at every node $v \in V_{\mathsf{P}}$) s.t. no negative cycles exist anymore (e.g. consider $\kappa = V_P \times \{\bullet\}$); then the sup-inf-distance for every node can be easily computed using fixed-point iteration just as in the case of the standard attractor computation using some variant of the Bellmann-Ford algorithm (in order to identify infinite positive sup-inf-distance). Let $d_\kappa(v)$ denote the sup-

inf-distance of node v in $\mathcal{A}^\bullet_\kappa$. For every node v in $B \cup \{\bullet\}$, we define $d_\kappa(v) = 0$; for the nodes $\bot$ and $\top$ we predefine d_κ accordingly; for every v controlled by $\overline{\mathsf{P}}$ we have the unrestricted optimality equation $d_\kappa(v) = \min^{\prec_\mathsf{P}}\{\gamma(v,w)+d_\kappa(w) \mid w \in vE\}$, while for v controlled by P we have the κ-restricted optimality equation $d_\kappa(v) = \max^{\prec_\mathsf{P}}\{\gamma(v,w) + d_\kappa(w) \mid w \in \kappa(v)\}$ (with $\gamma(v,w)$ the weight induced by the colour of the edge from v to w). We call any nondeterministic strategy κ' an improvement of κ if (1) $\kappa' \neq \kappa$, (2) $\emptyset \neq \kappa'(v) \subseteq vE$ for all $v \in V_P$, (3) if $(v,w) \in \kappa' \backslash \kappa$, then $d_\kappa(v) < \gamma(v,w) + d_\kappa(w)$ (i.e. v's sup-inf-distance can be improved by playing to w), and (4) if $(v,w) \in \kappa' \cap \kappa$, then $d_\kappa(v) = \gamma(v,w) + d_\kappa(w)$. [31] shows that any such improvement κ' does not introduce any negative cycles and that $d_{\kappa'} > d_\kappa$, i.e. the sup-inf-distances w.r.t. κ' do not decrease for any node and strictly improves for at least one node. Thus, no strategy can be encountered twice. Finally, note that if $W_\mathsf{P} \uplus W_{\overline{\mathsf{P}}}$ is the winning partition of the nodes V of $\mathcal{A}$ w.r.t. the min-parity-condition, then P can use his (memoryless deterministic) winning strategy from the parity game to ensure infinite sup-inf-distance on W_P also in $\mathcal{A}^\bullet$, while $\overline{\mathsf{P}}$ can use his winning strategy from the parity game to ensure at least finite sup-inf-distance in $\mathcal{A}^\bullet$. Hence, eventually the strategy iteration will terminate in a strategy that guarantees infinite sup-inf-distance on exactly W_P (Fig. 7).

In our implementation, we make use of the fact that the fixed-point iteration used for solving the optimality equations stated above can be easily parallelised in order to make use of modern multi-core CPUs (or even GPUs [32]).

3.2.1 Exploring the boundary

$\mathcal{O}_{\mathsf{expl}}$ selects nodes from the boundary B that should be further explored, i.e., where successors should be computed. After each expansion B is recomputed such that it contains the nodes which successors have not yet been explored. We instantiate $\mathcal{O}_{\mathsf{expl}}$ with two different approaches. One is based on *breadth-first search* exploration, and one on a *priority queue* ordering states in the boundary by some *quality score*. For each method, we then have an additional variant that *filters* the states that are currently needed to determine the winner of the arena. In total, this results in the following four methods.

- $\mathcal{O}^{\mathsf{bfs}}_{\mathsf{expl}}$ (breadth-first search (BFS) exploration): In the initial implementation the algorithm explored and constructed the parity game using a breadth-first-search (BFS). This approach helps to ignore parts of the game that are far away from the initial state and not decisive for winning the game, however it also explores states that are close to the initial state, but are irrelevant with the currently computed strategy.
  Conceptually, we use the exploration function that picks states from B with a minimal distance from the initial state q_0 in each step. Define $\mathcal{O}^{\mathsf{bfs}}_{\mathsf{expl}}$ by:

$$\mathcal{O}^{\mathsf{bfs}}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, B, q_0, p, \sigma, \tau) := \arg\min_{b \in B} \mathsf{dist}(V_\bigcirc \cup V_\Box, E, q_0, b)$$

$$\mathsf{dist}(V, E, q_0, q) := \min\{|\pi| \mid \pi \text{ is a path from } q_0 \text{ to } q \text{ in } (V, E)\}$$

  In the implementation, instead of recomputing the minimal distances in each step, we select the next state from the boundary from a worklist queue.
- $\mathcal{O}^{\mathsf{bfs+}}_{\mathsf{expl}}$ (BFS exploration with strategy-based worklist filtering): This exploration strategy is a variant of $\mathcal{O}^{\mathsf{bfs}}_{\mathsf{expl}}$. The worklist is still populated in a BFS way, but we only keep states that we know are needed to determine if the initial state is winning or losing. These are states which are reachable through some path not blocked by an already winning or

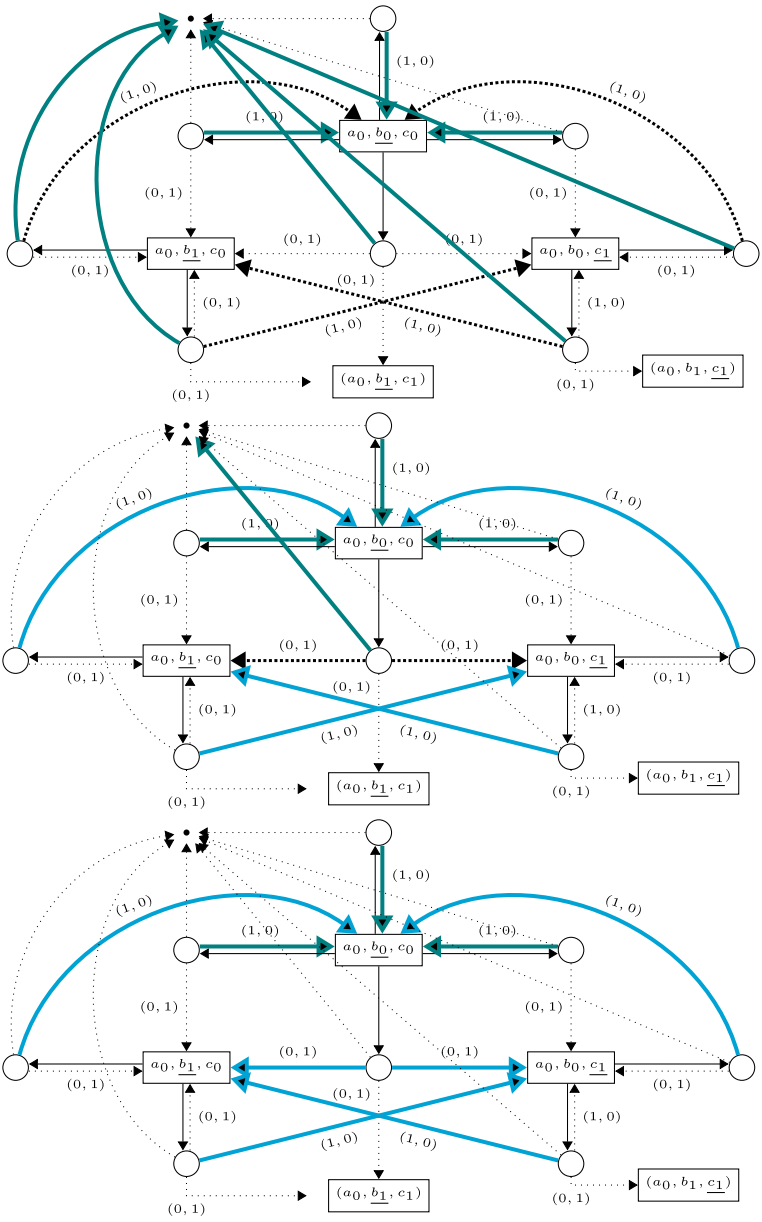


Fig. 7 The arena after one exploration step as show in Fig. 5 but extended with the node •. To simplify presentation, we have removed the node ⊥ as ○ will always rather play to •; we also have dropped the neutral edge weights i.e. every unlabelled edge is implicitly labeled by $(0, 0)$. Top figure: Here, we are given the strategy we computed in Fig. 6 as initial strategy for ○; for every node added by the exploration step, we extend the strategy by letting ○ play to • again. We identify as improvements the thick dotted edges. This gives us the strategy shown in the figure in the middle where the edges in green have been inherited from the initial strategy, while the edges in blue are the improvement identified in the preceding step of the strategy iteration. W.r.t. this strategy we only identify one further improvement which leads to the final strategy shown in the lower figure that wins the node $(a_0, \underline{b_0}, c_0)$

losing state. We define the exploration function $\mathcal{O}^{\mathsf{bfs+}}_{\mathsf{expl}}$ as follows:

$$\begin{aligned}
&\mathcal{O}^{\mathsf{bfs+}}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, B, q_0, p, \sigma, \tau)\\
&\quad := \mathcal{O}^{\mathsf{bfs}}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, \mathsf{filter}(B), q_0, p, \sigma, \tau)\\
&\mathsf{filter}(B) := \{b \in B \mid \exists \text{ path } \pi \text{ from } q_0 \text{to } b \text{ s.t. } \forall q \in \pi : \mathsf{winner}(q) = ?\}\\
&\mathsf{winner}(q) := \begin{cases} \bigcirc & \text{if } \mathcal{O}_{\mathsf{win}}(V_\bigcirc, V_\Box, E, \chi, B, q, \bigcirc, p, \sigma) \text{ returns true}\\ \Box & \text{if } \mathcal{O}_{\mathsf{win}}(V_\bigcirc, V_\Box, E, \chi, B, q, \Box, \overline{p}, \tau) \text{ returns true}\\ ? & \text{otherwise}\end{cases}
\end{aligned}$$

We reuse the results of $\mathcal{O}_{\mathsf{win}}$ already computed in the main algorithm. However, in contrast to $\mathcal{O}^{\mathsf{bfs}}_{\mathsf{expl}}$, we need to check for existence of a path from q_0 to all states $b \in B$ without already won states, which we do using a single linear-time search in each iteration.

– $\mathcal{O}^{\mathsf{pq}}_{\mathsf{expl}}$ (priority queue exploration based on scores): This exploration method is based on *quality scores* (to be defined in the next section) assigned to each state in the boundary. The idea is that a high score means this state is a promising state for $\bigcirc$ to win the game from the initial state, while a low score is a promising state for $\Box$ to win the game. As we initially do not know if the specification is realisable or unrealisable, and thus do not know for which player the initial state is winning, this method explores both states with high and low scores simultaneously. We assign scores not to states, but to edges, to incorporate information from colours and updates in the LAR. Let $\mathcal{O}_{\mathsf{score}}(\alpha, q, a, q')$ be a function assigning a score to a given edge (q, a, q') of the parity automaton $A(\alpha)$. With an intermediate function $s(b)$ assigning to a state $b \in B$ all scores of incoming edges, we define the exploration method $\mathcal{O}^{\mathsf{pq}}_{\mathsf{expl}}$ by:

$$\begin{aligned}
&\mathcal{O}^{\mathsf{pq}}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, B, q_0, p, \sigma, \tau)\\
&\quad := \left(\arg\min_{b\in B}(\min s(b))\right) \cup \left(\arg\max_{b\in B}(\max s(b))\right)\\
&s(b) := \{\mathcal{O}_{\mathsf{score}}(q, i \cup o, b) \mid (q, I, q'), (q', O, b) \in E,\ i \in I,\ o \in O\}
\end{aligned}$$

We implement this method with a double-ended priority queue, in which states are inserted with their respective minimal and maximal score of an incoming edge upon discovery.

– $\mathcal{O}^{\mathsf{pq+}}_{\mathsf{expl}}$ (priority queue exploration with strategy-based worklist filtering): Finally, we can combine the exploration using scores with the filtering function filter from $\mathcal{O}^{\mathsf{bfs+}}_{\mathsf{expl}}$, yielding the method $\mathcal{O}^{\mathsf{pq+}}_{\mathsf{expl}}$ defined by:

$$\begin{aligned}
&\mathcal{O}^{\mathsf{pq+}}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, B, q_0, p, \sigma, \tau)\\
&\quad := \mathcal{O}^{\mathsf{pq}}_{\mathsf{expl}}(\alpha, V_\bigcirc, V_\Box, E, \chi, \mathsf{filter}(B), q_0, p, \sigma, \tau)
\end{aligned}$$

3.2.2 Quality scores

We now describe how to compute scores by $\mathcal{O}_{\mathsf{score}}(\mathcal{T}_\varphi, q, a, q')$ for an edge (q, a, q') of the DPA $A(\mathcal{T}_\varphi)$ leading to a boundary state q'. We first define a function score, which takes the acceptance-type annotated formula $\mathcal{T}_\varphi$ and an edge (q, a, q') of $A(\mathcal{T}_\varphi)$, and returns a *weighted score* $(w, s) = \mathsf{score}(\mathcal{T}_\varphi, q, a, q')$. We always assign the score value s such that $s = 0$ if $q' = \bot$, $s = 1$ if $q' = \top$ and $0 < s < 1$ otherwise. The function $\mathsf{score}(\mathcal{T}_\varphi, q, a, q')$

computes (w, s) recursively on $\mathcal{T}_\varphi$, similarly to the LAR-product construction, and we define it by case distinction on $\mathcal{T}_\varphi$ and q.

- If $q = \bot$, return $w = 1$ and $s = 0$.
- If $q = \top$, return $w = 1$ and $s = 1$.
- In the base case for $\mathcal{W}(\varphi)$, $\mathcal{B}(\varphi)$, $\mathcal{C}(\varphi)$ and $\mathcal{P}(\varphi)$, return $w = 1$ and compute s depending on the type of automaton.
  - If we encounter $\mathcal{W}(\varphi)$, then $\varphi \in \mu LTL \cup \nu LTL$. By using the construction of [13] the state q' is in fact a Boolean formula over modal operators treated as variables. Let $\mathcal{V}$ be the set of variables and let $\mathcal{M}$ be the set of satisfying assignments of q'. We then set $s = |\mathcal{M}|/2^{|\mathcal{V}|}$. Using this definition, we assign to the state **tt** (which corresponds to $\top$) $s = 1$ and to the state **ff** (which corresponds to $\bot$) $s = 0$.
  - If we encounter $\mathcal{B}(\varphi)$, then we apply the construction of [38] and the state $q' = (p, p')$ is a tuple of two Boolean formulas p and p'. We then construct $p \wedge p'$ and compute the scoring for $\mathcal{W}(p \wedge p')$.
  - If we encounter $\mathcal{C}(\varphi)$, then we apply the same approach as in $B(\varphi)$, since these automata are obtained by complementing Büchi automata and have an identical structure besides the acceptance condition.
  - If we encounter $\mathcal{P}(\varphi)$, we bail and return $s = \frac{1}{2}$.
- In the case $\mathcal{T}_\varphi = \mathcal{X}_\wedge(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ and $\mathcal{T}_\varphi = \mathcal{X}_\vee(\mathcal{X}_1, \ldots, \mathcal{X}_n)$, we first compute the child scores $(w_i, s_i) = \mathsf{score}(\mathcal{X}_i, q, a, q')$. Then, for each $\mathcal{X}_i$, we update w_i and s_i as follows:
  - If $\mathcal{T}_\varphi = \mathcal{X}_\wedge$, set $w_i \leftarrow w_i \cdot \log_{1/2} s_i$. This gives score values close to 0 an increased weight, which could make the whole conjunction false.
  - If $\mathcal{T}_\varphi = \mathcal{X}_\vee$, set $w_i \leftarrow w_i \cdot \log_{1/2}(1 - s_i)$. This gives score values close to 1 an increased weight, which could make the whole disjunction true.
  - If $\mathcal{T}_\varphi = \mathcal{B}_\wedge(\mathcal{B}_1, \ldots, \mathcal{B}_n)$, or $\mathcal{T}_\varphi = \mathcal{C}_\vee(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, then $\mathcal{X}_i = \mathcal{B}$ or $\mathcal{X}_i = \mathcal{C}$ and $q = (q'', r)$ and $q' = (q''', r')$ for some round-robin counters r, r'. Now if the child caused the round-robin counter to increase, i.e. we have $r \leq i < r'$, then we set $w_i \leftarrow 2 \cdot w_i$ and update the score. If $\mathcal{X}_i = \mathcal{B}$, set $s_i \leftarrow \frac{3+s_i}{4}$ to increase the score, and if $\mathcal{X}_i = \mathcal{C}$, set $s_i \leftarrow \frac{s_i}{4}$ to decrease the score.
  - If either $\mathcal{T}_\varphi = \mathcal{P}_\wedge(\mathcal{X}_1, \mathcal{X}_2)$, $\mathcal{X}_i = \mathcal{P}$, and $\mathcal{X}_{3-i} = \mathcal{B}$, or $\mathcal{T}_\varphi = \mathcal{P}_\vee(\mathcal{X}_1, \mathcal{X}_2)$, $\mathcal{X}_i = \mathcal{P}$, and $\mathcal{X}_{3-i} = \mathcal{C}$, then $q = (q'', c)$ and $q' = (q''', c')$ for some minimal colour memory values c, c'. Let p be the parity of the DPA $A(\mathcal{T}_\varphi)$. Now if $A(\mathcal{X}_i)$ caused the memory to decrease to a value with parity p, we increase the score, and if it decreased it to a value with different parity, we decrease the score. Therefore if $c' < c$ and $c' \equiv_2 p$, then set $w_i \leftarrow 2 \cdot w_i$ and $s_i \leftarrow \frac{3+s_i}{4}$, and if $c' < c$ and $c' \equiv_2 \overline{p}$, then set $w_i \leftarrow 2 \cdot w_i$ and $s_i \leftarrow \frac{s_i}{4}$.

  Finally, we return $w := \sum_{i=1}^{n} w_i$ and $s := \left(\sum_{i=1}^{n} w_i \cdot s_i\right) / w$.
- In the case $\mathcal{X}_\leftrightarrow(\mathcal{X}_1, \mathcal{X}_2)$, we also first compute the child scores $(w_i, s_i) = \mathsf{score}(\mathcal{X}_i, q, a, q')$ for $i \in \{1, 2\}$, and then update each w_i and s_i similarly to conjunction and disjunction:
  - If $0 < s_i < 1$, set $w_i \leftarrow w_i \cdot \max(\log_{1/2} s_i, \log_{1/2}(1 - s_i))$.
  - If $\mathcal{T}_\varphi = \mathcal{P}_\leftrightarrow(\mathcal{X}_1, \mathcal{X}_2)$ and $A(\mathcal{X}_i)$ is the child for which we store the colours in memory, then $q = (q'', c)$ and $q' = (q''', c')$ for some minimal colour memory values c, c' of $A(\mathcal{X}_i)$. Let p be the parity of the DPA $A(\mathcal{T}_\varphi)$. As for $\mathcal{P}_\wedge$ and $\mathcal{P}_\vee$, if $c' < c$ and $c' \equiv_2 p$, then set $w_i \leftarrow 2 \cdot w_i$ and $s_i \leftarrow \frac{3+s_i}{4}$, and if $c' < c$ and $c' \equiv_2 \overline{p}$, then set $w_i \leftarrow 2 \cdot w_i$ and $s_i \leftarrow \frac{s_i}{4}$.

Then we return $w := \sum_{i=1}^{n} w_i$ and $s := \left(\sum_{i=1}^{n} w_i \cdot s_i\right) / w$.

We then define $\mathcal{O}_{\mathsf{score}}(\mathcal{T}_\varphi, q, a, q') := s$ where $(w, s) = \mathsf{score}(\mathcal{T}_\varphi, q, a, q')$. The value w is only necessary to normalise intermediate scores due to the recursive definition of score.

3.3 Controller extraction

3.3.1 Mealy machine

When we determine that $\bigcirc$ has a winning strategy σ from q_0, we can extract a controller from σ that ensures realisation of the specification.

We use an *incompletely specified* Mealy machine, where some outputs might not be specified and could be instantiated either way. This allows further minimisation and more compact representations by a circuit. Given an input/output partition $\mathsf{Ap} = \mathsf{Ap}_{\mathsf{in}} \uplus \mathsf{Ap}_{\mathsf{out}}$, a Mealy machine is a tuple $M = (Q, q_0, \delta, \lambda)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times 2^{\mathsf{Ap}_{\mathsf{in}}} \to Q$ is the transition function and $\lambda : Q \times 2^{\mathsf{Ap}_{\mathsf{out}}} \to \{0, 1, ?\}^{\mathsf{Ap}_{\mathsf{out}}}$ is the output function, where ? stands for an unspecified output. The output can be given by a Boolean product term, where missing variables are unspecified.

Let $(V_\bigcirc, V_\Box, E)$ be the parity game arena where $\bigcirc$ wins from q_0 with the strategy σ. We use $Q := \{q \in V_\Box \mid \mathcal{O}_{\mathsf{win}}(V_\bigcirc, V_\Box, E, \chi, B, q, \bigcirc, p, \sigma)\}$ as the set of states. For the transition function, we define $\delta(q, i) := q'$ by choosing some q' where $((q, I), O, q') \in \sigma$ for some $I \subseteq \Sigma_{\mathsf{in}}$ with $i \in I$ and any $O \subseteq \Sigma_{\mathsf{out}}$. By construction, and as σ is a winning strategy for all $q \in Q$, such a q' always exists. However, there may be multiple applicable q'. We construct Q and δ iteratively, starting from $Q \leftarrow \{q_0\}$, and try to choose for every $q \in Q$ and $i \in 2^{\mathsf{Ap}_{\mathsf{in}}}$ a $q' = \delta(q, i)$ such that $q' \in Q$, if possible, and otherwise extend $Q \leftarrow Q \cup \{q'\}$. As a secondary heuristic, we try to choose a successor that gives the most flexibility in choosing the output, i.e. a q' such that $\sum\{|O| \mid \exists I : i \in I \wedge ((q, I), O, q') \in \sigma\}$ is maximal.

For the output function, let $\delta(q, i) = q'$ be an edge of the Mealy machine; then take a minimal prime implicant o' of the Boolean formula over $\mathsf{Ap}_{\mathsf{out}}$ that encodes the set $\{o \mid \exists I, O : i \in I \wedge o \in O \wedge (q, I), O, q') \in \sigma\}$, and define $\lambda(q, i) := o'$. This again exploits non-determinism of the strategy σ.

Example 1 Using the winning strategy for the simple arbiter specification in Fig. 5, we obtain $Q = \{(a_0, \underline{b_0}, c_0), (a_0, \underline{b_1}, c_0), (a_0, b_0, \underline{c_1})\}$ as states of the Mealy machine. For $\delta((a_0, \underline{b_0}, c_0), r_1 r_2)$, we may choose any of the other two states as a successor. If we choose $(a_0, \underline{b_1}, c_0)$, we get the output $o = \lambda((a_0, \underline{b_0}, c_0), r_1 r_2) = \overline{g_1} g_2$, specifying $o(g_1) = 0$ and $o(g_2) = 1$. For the output $\lambda((a_0, \underline{b_0}, c_0), \overline{r_1}\,\overline{r_2})$, we may choose one of the two min-terms $\overline{g_1}$ or $\overline{g_2}$ as an output. If we choose $o = \overline{g_1}$, then $o(g_1) = 0$, but the output $o(g_2) = ?$ is unspecified. An implementation is then free to choose $o(g_2) = 0$ or $o(g_2) = 1$.

This incompletely specified Mealy machine can optionally be *minimised*. We use the exact minimisation algorithm from [1], which in turn uses a SAT solver. While this problem is harder than minimisation of fully specified Mealy machines, it can also result in smaller machines.

3.3.2 Controller as BDD or AIG

While we can directly output the controller as a Mealy machine, we can also encode it as a *binary decision diagram (BDD)* or *and-inverter graph (AIG)*, representing a circuit. For

this, we need to encode the transition function δ and output function λ of the Mealy machine as a BDD or AIG. Both of these representations use decisions over binary variables. The inputs $i \in 2^{\mathsf{Ap}_{\mathsf{in}}}$ and outputs $o \in \{0, 1, ?\}^{\mathsf{Ap}_{\mathsf{out}}}$ are already binary vectors after one resolves the unspecified ? outputs. However, one needs to choose a binary encoding of the state space Q. Here we offer two options:

First, we can use an *unstructured encoding* l_{unstr}. We simply enumerate the states $Q = \{q_0, \ldots, q_n\}$ and use the encoding function $l_{\mathsf{unstr}} : Q \to 2^{\lceil \log_2 |Q| \rceil}$ with $l_{\mathsf{unstr}}(q_i) = i$ which maps each state to the binary encoding of its number.

Second, we can use the shape of the states for a *structured encoding* l_{struct}. As a state q is a vector $(q_1, \ldots, q_n)$, we can encode each component separately into a binary vector. Assume that for each component $1 \leq i \leq n$, the states q_i are numbered from 0 to $|Q_i| - 1$. Then we use the encoding function $l_{\mathsf{struct}} : Q \to 2^{\sum_{i=1}^{n} \lceil \log_2 |Q_i| \rceil}$, with

$$l(q_1, \ldots, q_n) := \sum_{i=1}^{n} q_i \cdot 2^{\sum_{j=1}^{i-1} \lceil \log_2 |Q_i| \rceil}$$

which concatenates the binary encoding of the state number in each component. Note that this also includes additional memory information such as the round-robin counter or minimal colour memory.

Example 2 For the Mealy machine obtained from the strategy in Fig. 5, we have as states $Q = \{(a_0, \underline{b_0}, c_0), (a_0, \underline{b_1}, c_0), (a_0, b_0, \underline{c_1})\}$. With the unstructured encoding for $Q = \{q_0, q_1, q_2\}$, we get the binary state encodings $l_{\mathsf{unstr}}(Q) = \{00_2, 01_2, 10_2\}$. If we represent the product states as numbers with $q_i = i$ for $q \in \{a, b, c\}$ and the round-robin counter by $r \in \{0, 1\}$, then after applying the structured encoding function, we get the state encodings $l_{\mathsf{struct}}(Q) = \{0000_2, 0100_2, 0011_2\}$.

We also minimise the controller for these output formats. When constructing a BDD, we minimise it using the CUDD library [40] by reordering the variables. When constructing an AIG, we first construct a BDD and minimise it, and then construct the AIG from the BDD. Afterwards, we minimise the AIG using functionality from the ABC library [6].

When using the encoding function, it can sometimes also be more effective to *not* minimise the Mealy machine before, as this can destroy the structure from the product state. Structured encoding can also sometimes increase the size of a circuit. We offer an option to construct a circuit using the three following combinations in parallel and then return the smallest circuit: Mealy minimisation and unstructured encoding, no minimisation and structured encoding, and no minimisation and unstructured encoding.

4 Experimental evaluation

The three main research questions we want to answer in this section are:

- *RQ1* How does Strix compare to existing tools? Specifically, we analyse:
  - Number of instances correctly identified to be realisable.
  - Number of instances for which a correct controller was synthesised.
  - Circuit size of the constructed controller.
  - Performance with increasing alphabet size.

– *RQ2* What is the difference in performance of the proposed exploration strategies (bfs and pq, with and without filtering)? Specifically, we analyse:
  – Number of constructed states.
  – Runtime.
– *RQ3* What is the difference in size of the proposed circuit encoding strategies (structured and unstructured, with and without minimisation)? Specifically, we analyse:
  – Circuit size.

Experimental Design We approach all three research questions by evaluating the different tools and configurations on the specifications from the TLFS/LTL-track of the SYNTCOMP2019 competition,[7] which subsumes all benchmarks of SYNTCOMP2018 [25]. To the best of our knowledge this dataset is the most complete set of LTL specifications for synthesis stemming from a wide range of different applications. These include industrial examples such as the AMBA AHB arbiter [4,21,26], and case studies for hardware controller synthesis [16,18]. For more details, see previous SYNTCOMP competition reports [23–25]. The SYNTCOMP2019 set of specifications contains in total 434 LTL synthesis specifications, of which 337 are realisable and 97 are unrealisable. All experiments were run on a server with an Intel E5-2630 v4 clocked at 2.2 GHz (boost disabled, 40 cores). We imposed a memory limit of 100 GB (as in SYNTCOMP2019) and a wall-clock time limit of 1 h for each specification.

In Sect. 4.1 we address *RQ1* and evaluate overall performance on the benchmark set and compare against LTLSYNT, BOSY and a previous version of STRIX. In Sect. 4.2 (*RQ2*), we compare the different exploration strategies, and in Sect. 4.3 (*RQ3*) different construction approaches for the circuit.

Independent Evaluations Since the first release of STRIX [33] independent researchers used, evaluated, and compared it to other tools. At SYNTCOMP2019, STRIX in its submitted version (19.07) again made first place in all categories in the LTL synthesis track. In two case studies [16,18] by Finkbeiner et al. STRIX was used to synthesise controllers for small hardware devices; in the second case study [18], STRIX was also compared to BOSY and the not publicly available BOWSER where STRIX was the only tool to solve all synthesis problems, and also performed best w.r.t. time and size of the obtained controller in almost all of the synthesis problems; interestingly, BOWSER clearly outperformed STRIX in the specification `SensorSelector` where BOWSER was roughly seven times faster than STRIX and also succeeds in finding a trivial controller with zero gates within 38 s while STRIX needed 280 s for returning a controller using 17 gates.

Finally, [11] compares STRIX to GUISYNTH, a tool for synthesising code for graphical user interfaces. Here, GUISYNTH clearly outperforms STRIX in most benchmarks. As Ehlers and Adabala already noted this probably has to be contributed to the fact that GUISYNTH is specifically designed for the task at hand and the embedding into a standard LTL synthesis problem causes an exponential blow-up in the alphabet size.

4.1 *RQ1*: comparison with previous version, with LTLSYNT, and with BOSY

Experimental Design We run STRIX using the default exploration strategy $\mathcal{O}^{\text{bfs}}_{\text{expl}}$, AIG output and portfolio minimisation to find the smallest AIG. We compare against STRIX in the version submitted to CAV 2018 [33] in April 2018. This version also used the exploration strategy $\mathcal{O}^{\text{bfs}}_{\text{expl}}$, but did not have many of the improvements such as formula isomorphism detection,

[7] http://www.syntcomp.org/.

Table 1 Overall results for STRIX compared with its previous version, LTLSYNT and BOSY

	Max	STRIX (19.07)	STRIX (CAV'18)	LTLSYNT (2.8.1)	BOSY (July'19)
Realisability	434	**415** (29)	374 (0)	353 (0)	344 (0)
Synthesis	337	**304** (28)	264 (0)	254 (0)	234 (2)
Total quality	674	**571.66**	425.17	242.82	416.94
Avg. quality	2.00	**1.88**	1.61	0.96	1.78

Bold values indicate the best result for each quantity. For **Realisability** and **Synthesis**, we give the number of solved instances (unique instances) and for **Quality** the total accumulated and the average points over solved synthesis instances

memoization, symbolic representation of edges, or structured encoding. Further, we compare our implementation with BOSY and LTLSYNT which achieved second[8] and third place in SYNTCOMP2018 in the "synthesis quantity"-ranking. We use LTLSYNT from the Spot library [8,23] in the version 2.8.1 from July 2019, with parameter `--algo=ds`, and against BOSY [14] in the newest version available as of July 2019, with parameter `--optimize`.

We run each tool on each specification twice: once to check only realisability and in the realisable case once more to synthesise a controller in the AIGER format. Table 1 gives the overall results.

The category **Realisability** counts the number of specifications for which realisability is correctly decided within the time limit, and the category **Synthesis** counts the number of realisable specifications for which additionally a successfully verified controller is produced. For this we verified the circuits with an additional time limit of 1 h using the NUXMV model checker [7] in version 1.1.1 with the `check_ltlspec_klive` routine.

The **Quality** rating compares the size of the solutions according to the SYNTCOMP2019 formula, where a tool gets $\max(0, 2 - \log_{10} \frac{n+1}{r+1})$ quality points for each verified solution of size n for a specification with reference size r. The size of a solution is given by the number of and gates plus number of latches, and as reference size we chose the smallest size of a verified solution produced by any of the four tools during our experiments.

Further, we list notable outliers in pairwise comparisons with other approaches. We compute for each pair of tools and each specification successfully solved by both tools the difference in order of magnitudes ($|\log_{10} \frac{x}{y}|$)[9] and select for each pair of tools the eight largest differences.

In order to study the impact of growing alphabet sizes we look at 4 parametrised benchmarks from SYNTCOMP2019 and measure the execution time for growing parameters. These parameters scale up the number of components and the size of the alphabet. Namely, we pick `arbiter (AMBA)`, `full_arbiter_enc`, `ltl2dba_Q`, and `ltl2dba_beta`.

The results for these comparisons are given in table Table 2 for time to decide realisability, in Table 3 for size of the solutions, and Table 4 gives a cross-comparison.

Analysis. Compared to the previous version of STRIX we solve at least 40 additional specifications in the **Realisability** and **Synthesis** categories and see a considerable improvement in the **Quality** ratings. We believe that this is partly due to the symbolic construction and representation of the transition relation which applies to the arena and the automata for LTL fragments, e.g. safety. Only on smaller instances of the `full_arbiter` specification the previous version is faster. This might be related to the overhead of using symbolic data-

[8] STRIX was ranked on the first place.

[9] ...or $|\log_{10} \frac{x+1}{y+1}|$ to compensate for circuits of size 0.

Table 2 Time (s) to decide **Realisability**, comparing STRIX with its previous version, LTLSYNT and BOSY, both pairwise on the specifications with the 8 largest differences and on selected parameterised instances

Specification	$\lvert Ap_{in}\rvert$	$\lvert Ap_{out}\rvert$	STRIX (19.07)	STRIX (CAV'18)	LTLSYNT (2.8.1)	BOSY (July'19)
STRIX (19.07) *vs* STRIX (CAV'18)						
`simple_arbiter_10`	10	10	**2.4**	1603.2	818.0	TIME
`detector_10`	10	1	**0.8**	486.4	MEM	506.0
`prioritized_arbiter_8`	9	9	**2.1**	1048.6	2450.7	TIME
`ltl2dba_C2_10`	10	1	**1.1**	457.2	MEM	474.0
`detector_unreal_10`	10	1	**1.7**	465.1	MEM	13.1
`loadfull5`	6	5	5.0	1253.9	**3.5**	10.9
`amba_decomposed_lock_8`	17	1	**1.8**	301.0	18.4	2.5
`ltl2dba_theta_8`	10	1	**1.4**	223.9	MEM	96.5
STRIX (19.07) *vs* LTLSYNT (2.8.1)						
`ltl2dba_C2_8`	8	1	**1.2**	17.8	3443.6	7.1
`detector_unreal_8`	8	1	**1.4**	19.7	3320.2	1.5
`ltl2dpa22`	6	3	**1.2**	TIME	2581.1	12.2
`prioritized_arbiter_8`	9	9	**2.1**	1048.6	2450.7	TIME
`prioritized_..._unreal1_3_8`	4	4	**1.7**	TIME	1686.7	TIME
`TorcsSteeringSmart`	4	6	51.2	17.7	**0.1**	0.7
`torcs_steering_smart`	4	6	49.4	16.9	**0.1**	0.6
`simple_arbiter_10`	10	10	**2.4**	1603.2	818.0	TIME
STRIX (19.07) *vs* BOSY (July'19)						
`round_robin_arbiter_4`	4	4	1.3	4.1	**1.2**	3178.3
`ltl2dba_Q_6`	6	1	**0.9**	4.9	1.1	1539.9
`detector_10`	10	1	**0.8**	486.4	MEM	506.0
`ltl2dba_C2_10`	10	1	**1.1**	457.2	MEM	474.0
`SPIWriteManag`	5	15	**3.3**	3.6	16.3	1076.3
`SPI`	4	16	**5.1**	6.6	10.0	1341.1
`ltl2dba_R_6`	6	1	**4.8**	TIME	MEM	1167.0
`OneCounterGuiA6`	9	9	**2.9**	7.3	79.0	464.7
selected parameterised instances						
`amba_..._arbiter_2`	3	4	1.3	0.9	**0.1**	0.1
`amba_..._arbiter_4`	5	6	1.7	1.4	**1.0**	4.8
`amba_..._arbiter_6`	7	8	**9.4**	40.1	38.3	393.1
`amba_..._arbiter_8`	9	10	**177.7**	1629.1	TIME	TIME
`full_arbiter_enc_6`	3	3	11.7	**2.0**	17.2	200.0
`full_arbiter_enc_8`	3	4	63.0	**21.4**	TIME	TIME
`full_arbiter_enc_10`	4	4	337.8	**303.7**	TIME	TIME
`full_arbiter_enc_12`	4	4	**1274.5**	TIME	TIME	TIME
`ltl2dba_beta_4`	8	1	**1.1**	6.1	1.1	2.4
`ltl2dba_beta_6`	12	1	**1.5**	237.0	72.5	152.5
`ltl2dba_beta_8`	16	1	**19.0**	TIME	ERR	MEM
`ltl2dba_beta_10`	20	1	**352.2**	TIME	ERR	MEM

Table 2 continued

Specification	$\lvert Ap_{in} \rvert$	$\lvert Ap_{out} \rvert$	Strix (19.07)	Strix (CAV'18)	ltlsynt (2.8.1)	BoSy (July'19)
ltl2dba_Q_6	6	1	**0.9**	4.9	1.1	1539.9
ltl2dba_Q_8	8	1	**4.2**	145.3	324.9	TIME
ltl2dba_Q_10	10	1	**47.5**	TIME	TIME	TIME
ltl2dba_Q_12	12	1	**1042.4**	TIME	TIME	TIME

Bold values indicate the shortest runtimes. We mark timeouts by TIME, memouts by MEM, and errors by ERR

Table 3 Size of AIG for **Synthesis** and **Quality**, comparing Strix with its previous version, ltlsynt and BoSy, pairwise on the specifications with the 8 largest differences

Specification	$\lvert Ap_{in} \rvert$	$\lvert Ap_{out} \rvert$	Strix (19.07)	Strix (CAV'18)	ltlsynt (2.8.1)	BoSy (July'19)
Strix (19.07) *vs* Strix (CAV'18)						
tictactoe	9	9	**0**✓	153✓	TIME	TIME
collector_v2_4	4	1	20✓	2060◷	8292✓	**10**✓
ltl2dba_Q_6	6	1	**114**✓	9761✓	16563✓	48591◷
collector_v4_7	7	1	**57**✓	3547◷	TIME	TIME
collector_v2_5	5	1	**24**✓	1238✓	TIME	TIME
OneCounter	9	9	**22**✓	870✓	2534✓	MEM
ltl2dba_C2_8	8	1	**58**✓	2133◷	TIME	MEM
detector_8	8	1	**58**✓	2133◷	TIME	440✓
Strix (19.07) *vs* ltlsynt (2.8.1)						
narylatch_10	11	10	**186**◷	TIME	6349541◷	TIME
simple_arbiter_10	10	10	**29**✓	MEM	996921◷	TIME
prioritized_arbiter_8	9	9	**43**✓	MEM	747766◷	MEM
ltl2dba_C2_6	6	1	**31**✓	1071✓	422872◷	468✓
detector_6	6	1	**31**✓	1071✓	422872◷	449✓
ltl2dba_E_10	10	1	**29**✓	TIME	321799◷	TIME
simple_arbiter_8	8	8	**18**✓	MEM	87834◷	21✓
narylatch_8	9	8	**144**✓	MEM	376661◷	TIME
Strix (19.07) *vs* BoSy (July'19)						
simple_arbiter_enc_6	3	3	**751**✓	2699◷	5276◷	611824◷
prioritized_arbiter_enc_6	3	3	**1445**◷	7938◷	66443◷	897716◷
ltl2dba_Q_6	6	1	**114**✓	9761✓	16563✓	48591◷
full_arbiter_7	7	7	**911**◷	MEM	112488◷	388240◷
ltl2dba_E_6	6	1	**17**✓	**17**✓	3630✓	5813✓
full_arbiter_6	6	6	**373**✓	634✓	28264◷	117163◷
round_robin_arbiter_4	4	4	**94**✓	239✓	1928✓	19738✓
full_arbiter_5	5	5	**284**✓	351✓	7039✓	47570◷

Bold values indicate the smallest circuit for each specification. We mark timeouts by TIME, memouts by MEM, and errors by ERR. The symbol ✓ denotes successful verification and ◷ denotes a timeout during verification of the AIG

Table 4 Cross-comparison of different tool

	Time for realisability				Size of AIG			
	Strix (19.07)	Strix (CAV'18)	ltlsynt (2.8.1)	BoSy (July'19)	Strix (19.07)	Strix (CAV'18)	ltlsynt (2.8.1)	BoSy (July'19)
Strix (19.07)	–	105	124	145	–	212	293	190
Strix (CAV'18)	25	–	80	98	31	–	242	125
ltlsynt (2.8.1)	22	34	–	77	0	23	–	82
BoSy (July'19)	14	33	32	–	101	139	202	–

Each cell counts the number of instances where the result of the row tool is strictly better than the result of the column tool, comparing time for realisability and size of the AIG as the number of and gates plus number of latches. A time is only considered better if it is at least 5 s less than the other time

Table 5 Cross-comparison of different exploration strategies

$\lvert V_\Box \rvert$	$\mathcal{O}^{\text{bfs}}_{\text{expl}}$	$\mathcal{O}^{\text{bfs+}}_{\text{expl}}$	$\mathcal{O}^{\text{pq}}_{\text{expl}}$	$\mathcal{O}^{\text{pq+}}_{\text{expl}}$	$\mathcal{O}^{\text{best}}_{\text{expl}}$	Time	$\mathcal{O}^{\text{bfs}}_{\text{expl}}$	$\mathcal{O}^{\text{bfs+}}_{\text{expl}}$	$\mathcal{O}^{\text{pq}}_{\text{expl}}$	$\mathcal{O}^{\text{pq+}}_{\text{expl}}$	$\mathcal{O}^{\text{best}}_{\text{expl}}$
$\mathcal{O}^{\text{bfs}}_{\text{expl}}$	–	10	45	36	5	$\mathcal{O}^{\text{bfs}}_{\text{expl}}$	–	14	57	30	1
$\mathcal{O}^{\text{bfs+}}_{\text{expl}}$	30	–	50	37	11	$\mathcal{O}^{\text{bfs+}}_{\text{expl}}$	43	–	62	45	31
$\mathcal{O}^{\text{pq}}_{\text{expl}}$	31	28	–	6	5	$\mathcal{O}^{\text{pq}}_{\text{expl}}$	13	9	–	2	1
$\mathcal{O}^{\text{pq+}}_{\text{expl}}$	45	45	48	–	30	$\mathcal{O}^{\text{pq+}}_{\text{expl}}$	32	22	47	–	13

Each cell counts the number of instances where the result of the row strategy is strictly better than the result of the column strategy. $\mathcal{O}^{\text{best}}_{\text{expl}}$ is the best of the three other exploration strategies. A time is only considered better if it is at least 5 s less than the other time

structures for small alphabets. Further, the revised AIG encoding strategy yields smaller controllers in comparison to the previous version.

Compared to BoSy, our approach can scale better on larger and complex specifications. One can observe this on parameterised specifications that mainly increase the number of input propositions. Even though BoSy employs an input-symbolic QBF encoding it could not deal with the large specifications. We hypothesise that this is caused by an explicit representation somewhere in the synthesis chain. For synthesis, while BoSy produces a smaller solution in 101 cases, Strix produces a smaller solution in 129 cases, and often by a much larger factor. We believe this is due to our structured encoding, which is hard to recover from a solution given by the constraint solver that bounded synthesis employs.

4.2 *RQ2*: comparison of different exploration strategies

Experimental Design. We compare the four different exploration strategies $\mathcal{O}^{\text{bfs}}_{\text{expl}}$, $\mathcal{O}^{\text{bfs+}}_{\text{expl}}$, $\mathcal{O}^{\text{pq}}_{\text{expl}}$ and $\mathcal{O}^{\text{pq+}}_{\text{expl}}$ and measure the number of explored states and time needed to check realisability. Table 5 gives a cross-comparison in order to identify a dominant approach. Further, Table 6 lists runtimes and number of explored states where there is a significant difference in the number of explored states.

Analysis. Table 5 suggests that $\mathcal{O}^{\text{pq+}}_{\text{expl}}$ can avoid the exploration of states not relevant for deciding realisability. However, this efficiency seems to be costly, since in the runtime comparison it falls behind the $\mathcal{O}^{\text{bfs+}}_{\text{expl}}$ configuration.

Table 6 Comparison of different exploration strategies for checking realisability

Specification	Explored states $\lvert V_\square \rvert$				Time for realisability (s)			
	$\mathcal{O}^{\text{bfs}}_{\text{expl}}$	$\mathcal{O}^{\text{bfs+}}_{\text{expl}}$	$\mathcal{O}^{\text{pq}}_{\text{expl}}$	$\mathcal{O}^{\text{pq+}}_{\text{expl}}$	$\mathcal{O}^{\text{bfs}}_{\text{expl}}$	$\mathcal{O}^{\text{bfs+}}_{\text{expl}}$	$\mathcal{O}^{\text{pq}}_{\text{expl}}$	$\mathcal{O}^{\text{pq+}}_{\text{expl}}$
amba_case_study_2	8187	3852	5598	**2428**	1008.9	451.4	707.3	**306.8**
collector_v3_5	78	73	148	**54**	**2.3**	**2.3**	4.3	2.6
collector_v3_6	148	147	206	**32**	6.5	6.0	13.0	**4.6**
collector_v3_7	306	305	386	**22**	42.2	39.9	94.0	**18.2**
detector_unreal_12	28	31	48	**22**	**1.7**	1.8	2.9	2.1
full_arbiter_unreal1_3_6	65536	65536	7475	**6645**	23.1	**12.4**	30.7	14.1
full_arbiter_unreal1_3_8	1685603	1678429	6576	**6321**	369.9	328.4	34.4	**17.3**
full_arbiter_unreal1_3_10	–	–	6812	**6249**	TIME	TIME	38.0	**16.7**
full_arbiter_unreal1_3_12	–	–	6899	**6418**	TIME	TIME	53.1	**19.4**
genbuf2	77479	61878	83025	**35196**	80.1	**56.9**	175.9	77.5
KitchenTimerV2	23	**12**	24	24	2.7	**2.1**	3.2	3.1
load_balancer_unreal1_2_10	2523	**572**	701	700	12.2	2.6	2.3	**1.5**
load_balancer_unreal1_2_12	10106	**638**	725	713	22.4	2.6	2.3	**1.9**
ltl2dba_R_6	162	80	29	**14**	4.7	4.0	3.8	**3.0**
ltl2dba_R_8	138	138	4	**3**	603.1	613.2	544.7	**437.6**
round_robin_[...]_unreal1_2_12	11703	16384	**4096**	**4096**	**8.1**	15.0	22.9	13.0
round_robin_[...]_unreal1_2_15	93623	93623	22509	**21077**	**21.7**	40.2	66.6	25.2
round_robin_[...]_unreal1_2_18	673158	669697	262144	**45152**	140.5	117.8	243.9	**57.7**
SliderDelayed	81	**48**	81	81	1.5	**1.0**	2.0	1.7
slider_delayed	**50**	81	81	81	**1.3**	1.4	2.2	1.4
TwoCounters3	**11**	**11**	99	99	9.1	**8.2**	44.5	40.4
TwoCountersInRangeM0	**7**	**7**	20	19	2.9	**2.3**	7.8	6.7
TwoCountersInRangeM1	**7**	**7**	24	23	2.7	**2.4**	9.1	7.9

Bold values indicate the shortest runtime and the smallest number of explored states, respectively

Table 7 Cross-comparison of different minimisation strategies

|AIG|	l_{unstr}	$l^{\text{min}}_{\text{unstr}}$	l_{struct}	$l^{\text{min}}_{\text{struct}}$	l_{best}
l_{unstr}	–	27	131	120	18
$l^{\text{min}}_{\text{unstr}}$	157	–	172	200	138
l_{struct}	80	61	–	71	29
$l^{\text{min}}_{\text{struct}}$	106	28	117	–	3

Each cell counts the number of instances where the size of the AIG from the row strategy is strictly smaller than the result of the column strategy, and l_{best} gives the minimum over all three other strategies

We see that filtering the queue in $\mathcal{O}^{\text{bfs+}}_{\text{expl}}$ and $\mathcal{O}^{\text{pq+}}_{\text{expl}}$ in comparison to $\mathcal{O}^{\text{bfs}}_{\text{expl}}$ and $\mathcal{O}^{\text{pq}}_{\text{expl}}$, respectively, often reduces the amount of explored states. However, filtering the queue generally incurs overhead, as the reachable state space needs to be explored and the queue rebuilt. For instance, on amba_case_study_2, we half the number of states explored when filtering is used. Further, comparing $\mathcal{O}^{\text{pq}}_{\text{expl}}$ with $\mathcal{O}^{\text{bfs}}_{\text{expl}}$, we see that using the scoring-based exploration

Table 8 Comparison for different minimisation strategies, giving the size of the AIG as the number of and gates plus number of latches

Specification	$\lvert Ap_{in}\rvert$	$\lvert Ap_{out}\rvert$	$\mathcal{O}^{bfs}_{expl}$	Size of AIG			
			$\lvert V_\square\rvert$	l_{unstr}	l^{min}_{unstr}	l_{struct}	l^{min}_{struct}
collector_v2_5	5	1	144	446✓	**24**✓	73✓	35✓
collector_v2_6	6	1	336	1610✓	**34**✓	101✓	51✓
collector_v2_7	7	1	607	3052◷	**65**✓	113✓	77✓
collector_v4_6	6	1	184	772✓	1216✓	**209**✓	2020✓
collector_v4_7	7	1	375	1451✓	**57**✓	141✓	173✓
full_arbiter_6	6	6	645	**373**✓	3865◷	758✓	8469◷
full_arbiter_7	7	7	1422	**911**◷	10722◷	1080✓	20349◷
full_arbiter_enc_8	3	4	15985	18533◷	11623◷	**2904**◷	31195◷
full_arbiter_enc_10	4	4	70662	TIME	**147730**◷	TIME	TIME
KitchenTimerV3	4	6	69	48✓	**1**✓	64✓	14✓
KitchenTimerV4	4	6	78	48✓	**1**✓	64✓	14✓
load_balancer_8	9	8	2250	12903◷	3751✓	**512**✓	4705✓
ltl2dba_beta_6	12	1	49	1026✓	1026✓	**90**✓	**90**✓
ltl2dba_beta_8	16	1	81	1819✓	1819✓	**151**✓	**151**✓
ltl2dba_beta_10	20	1	121	2445✓	2445✓	**156**✓	**156**✓
ltl2dba_Q_6	6	1	170	3504✓	3504✓	**114**✓	**114**✓
ltl2dba_Q_8	8	1	986	31821◷	31821◷	**187**✓	**187**✓
ltl2dba_Q_10	10	1	5742	TIME	199301◷	**243**✓	**243**✓
ltl2dba_Q_12	12	1	33462	TIME	TIME	**309**✓	TIME
prioritized_arbiter_enc_8	4	4	23554	57136◷	33762◷	**3487**◷	85101◷
prioritized_arbiter_enc_10	4	4	142337	TIME	TIME	**6906**◷	TIME
prioritized_arbiter_enc_12	4	4	593921	TIME	MEM	**12256**◷	MEM
round_robin_arbiter_7	7	7	9189	12119◷	**843**✓	2535✓	1183✓
simple_arbiter_enc_8	3	4	10240	15906◷	11308◷	**1317**◷	24468◷
simple_arbiter_enc_10	4	4	65024	112422◷	62369◷	**3356**◷	TIME
TorcsSteeringSmart	4	6	421	4963◷	**67**✓	5131◷	104✓
torcs_steering_smart	4	6	421	5006◷	**67**✓	5210◷	104✓
Total solved and verified (unique)				292 (0)	299 (2)	300 (3)	298 (0)

Bold values indicate the smallest circuit for each specification. The symbol ✓ denotes successful verification and ◷ denotes a timeout during verification of the AIG

can significantly reduce the amount of explored states. This especially seems to hold for unrealisable specifications. We assume this is because the environment player only needs to find a path to a state that forces the controller player to violate the specification, while the remaining states only contribute to satisfying the specification. However, $\mathcal{O}^{pq}_{expl}$ and $\mathcal{O}^{pq+}_{expl}$ can also drive the exploration of the state-space in the wrong direction as seen for example in `TwoCountersInRangeM0` and `TwoCounters3`.

An artefact of the in-parallel running parity game construction and parity game solution can be observed for the `collector_v3_[5-7]` and the `ltl2dba_R_[6-8]` specifications. Here the construction of the states is faster than the algorithm solving the game and when the translation slows down with the increasing size of the alphabet the solver catches up.

4.3 *RQ3*: Comparison of different minimisation strategies

Experimental design We compare the effect of Mealy machine minimisation and structured encoding on the size of the resulting AIG. For this experiment we use the exploration strategy $\mathcal{O}^{\text{bfs}}_{\text{expl}}$. We compare the four possible combinations: the unstructured and structured encoding functions applied to the unminimised Mealy machine (l_{unstr} and l_{struct}) and to the minimised Mealy machine ($l^{\text{min}}_{\text{unstr}}$ and $l^{\text{min}}_{\text{struct}}$). Table 7 gives a cross-comparison over all realisable specifications and Table 8 contains outliers where the different techniques have the largest effect and at the bottom cumulative results for the whole set.

Analysis The cross-comparison of Table 7 suggests that $l^{\text{min}}_{\text{unstr}}$ is a good all-round strategy, since it yields smaller circuits compared to the three other approaches. However, Table 8 shows that the structured encoding l_{struct} can sometimes give a significant reduction in size, e.g. for `ltl2dba_Q_[6-12]`. But sometimes there is also an increase in size, e.g. for `round_robin_arbiter_7`. This usually happens when minimisation is very effective. The combination $l^{\text{min}}_{\text{struct}}$ does usually not give an improvement over both l_{struct} and $l^{\text{min}}_{\text{unstr}}$ on their own. For `ltl2dba_beta_[6-10]` we have that it can be implemented by a controller using a vector of bits to remember which combinations of inputs have been encountered. The unstructured encoding does not exploit this fact, but this natural structure is restored by l_{struct}. For `full_arbiter_7`, l_{struct} gives a larger AIG than l_{unstr}, however it is easier to verify for our model checker, possibly due to the structure kept by l_{struct}. Due to these different characteristics we think that a portfolio approach is a sensible default configuration for Strix.

5 Conclusion and future work

The success of the described approach implemented in Strix relies on several key factors: (1) a demand-driven construction of the automata and the corresponding arena; (2) LTL translations that produce small deterministic automata on-the-fly; (3) a strategy iteration algorithm for solving parity games; especially the fact that the computed optimal strategy in an exploration step serves as a good initial strategy when computing the strategy for the next step; (4) semantic information that can be used for exploration guidance and controller extraction.

While the experimental evaluation places Strix ahead of other competing tools, specifications with large alphabets are still a challenge and need to be addressed. We think restricting the automata constructions of a decomposed formula to letters not violating other parts of the formula, e.g., safety conditions, could be beneficial, as already shown in [39]. Further, we think we only have scratched the surface of what can be done using the available semantic information and we want to explore potential applications for guiding the exploration of the arena and for extracting implementations, such as circuits but also reactive programs. There are already attempts to extract reactive programs using bounded synthesis [19], but unfortunately this only works for toy examples. Work in this area could help addressing the third challenge mentioned in the introduction (representation of synthesised implementation).

Acknowledgements We want to thank the anonymous reviewers for their helpful comments and remarks on this manuscript. We also want to thank Swen Jacobs and Guillermo A. Pérez for valuable feedback and testing of Strix during Syntcomp2019.

References

1. Abel, A., Reineke, J.: MeMin: SAT-based exact minimization of incompletely specified Mealy machines. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2–6, 2015, pp. 94–101 (2015). https://doi.org/10.1109/ICCAD.2015.7372555
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi omega-automata format. In: Computer Aided Verification—27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I, pp. 479–486 (2015). https://doi.org/10.1007/978-3-319-21690-4_31
3. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 921–962. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8_27
4. Bloem, R., Jacobs, S., Khalimov, A.: Parameterized synthesis case study: AMBA AHB. In: Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23–24, 2014., pp. 68–83 (2014). https://doi.org/10.4204/EPTCS.157.9
5. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: Computer Aided Verification—24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings, pp. 652–657 (2012). https://doi.org/10.1007/978-3-642-31424-7_45
6. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, pp. 24–40 (2010). https://doi.org/10.1007/978-3-642-14295-6_5
7. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: 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, pp. 334–342 (2014). https://doi.org/10.1007/978-3-319-08867-9_22
8. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: Automated Technology for Verification and Analysis—14th International Symposium, ATVA 2016, Chiba, Japan, October 17–20, 2016, Proceedings, pp. 122–129 (2016). https://doi.org/10.1007/978-3-319-46520-3_8
9. Ehlers, R.: Unbeast: symbolic bounded synthesis. In: 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, pp. 272–275 (2011). https://doi.org/10.1007/978-3-642-19835-9_25
10. Ehlers, R.: Symbolic bounded synthesis. Form. Methods Syst. Des. **40**(2), 232–262 (2012). https://doi.org/10.1007/s10703-011-0137-x
11. Ehlers, R., Adabala, K.: Reactive synthesis of graphical user interface glue code. In: Y. Chen, C. Cheng, J. Esparza (eds.) Automated Technology for Verification and Analysis—17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings, *Lecture Notes in Computer Science*, vol. 11781, pp. 387–403. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_23
12. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: 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, pp. 426–442 (2017). https://doi.org/10.1007/978-3-662-54577-5_25
13. Esparza, J., Kretínský, J., Sickert, S.: One theorem to rule them all: A unified translation of LTL into ω-automata. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09–12, 2018, pp. 384–393 (2018). https://doi.org/10.1145/3209108.3209161
14. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An experimentation framework for bounded synthesis. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II, pp. 325–332 (2017). https://doi.org/10.1007/978-3-319-63390-9_17
15. Filiot, E., Jin, N., Raskin, J.: Antichains and compositional algorithms for LTL synthesis. Form. Methods Syst. Des. **39**(3), 261–296 (2011). https://doi.org/10.1007/s10703-011-0115-3
16. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Synthesizing functional reactive programs. In: Eisenberg, R.A. (ed.) Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18–23, 2019, pp. 162–175. ACM (2019). https://doi.org/10.1145/3331545.3342601
17. Friedmann, O., Lange, M.: Two local strategy iteration schemes for parity game solving. Int. J. Found. Comput. Sci. **23**(3), 669–685 (2012). https://doi.org/10.1142/S0129054112400333

18. Geier, G., Heim, P., Klein, F., Finkbeiner, B.: Synthroids: Synthesizing a game for fpgas using temporal logic specifications. In: FMCAD, pp. 1–5. IEEE (2019)
19. Gerstacker, C., Klein, F., Finkbeiner, B.: Bounded synthesis of reactive programs. In: Automated Technology for Verification and Analysis—16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7–10, 2018, Proceedings, pp. 441–457 (2018). https://doi.org/10.1007/978-3-030-01090-4_26
20. Giannakopoulou, D., Lerda, F.: From states to transitions: Improving translation of LTL formulae to Büchi automata. In: Formal Techniques for Networked and Distributed Systems—FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11–14, 2002, Proceedings, pp. 308–326 (2002). https://doi.org/10.1007/3-540-36135-9_20
21. Godhal, Y., Chatterjee, K., Henzinger, T.A.: Synthesis of AMBA AHB from formal specification: a case study. STTT (Int. J. Softw. Tools. Technol. Trans.) **15**(5–6), 585–601 (2013). https://doi.org/10.1007/s10009-011-0207-9
22. Grädel, E., Thomas, W., Wilke, T.: Automata, Logics, and Infinite Games: A Guide to Current Research, *Lecture Notes in Computer Science*, vol. 2500. Springer (2002). https://doi.org/10.1007/3-540-36387-4
23. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results (2017). arxiv:1711.11439
24. Jacobs, S., Bloem, R., Brenguier, R., Khalimov, A., Klein, F., Könighofer, R., Kreber, J., Legg, A., Narodytska, N., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The 3rd reactive synthesis competition (SYNTCOMP 2016): Benchmarks, participants & results (2016). arxiv:1609.00507
25. Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P.J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., Walker, A.: The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results (2019). arxiv:1904.07736
26. Jobstmann, B.: Applications and optimizations for LTL synthesis. Ph.D. thesis, Graz University of Technology (2007)
27. Khalimov, A., Jacobs, S., Bloem, R.: PARTY parameterized synthesis of token rings. In: Computer Aided Verification—25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings, pp. 928–933 (2013). https://doi.org/10.1007/978-3-642-39799-8_66
28. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Automated Technology for Verification and Analysis—16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7–10, 2018, Proceedings, pp. 543–550 (2018). https://doi.org/10.1007/978-3-030-01090-4_34
29. Kupferman, O.: Recent challenges and ideas in temporal synthesis. In: 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, pp. 88–98 (2012). https://doi.org/10.1007/978-3-642-27660-6_8
30. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17–20, 2006, Proceedings, pp. 31–44 (2006). https://doi.org/10.1007/11817963_6
31. Luttenberger, M.: Strategy iteration using non-deterministic strategies for solving parity games (2008). arxiv:0806.2923
32. Meyer, P.J., Luttenberger, M.: Solving mean-payoff games on the GPU. In: Automated Technology for Verification and Analysis—14th International Symposium, ATVA 2016, Chiba, Japan, October 17–20, 2016, Proceedings, pp. 262–267 (2016). https://doi.org/10.1007/978-3-319-46520-3_17
33. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings, Part I, pp. 578–586 (2018). https://doi.org/10.1007/978-3-319-96145-3_31
34. Morgenstern, A., Schneider, K.: Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. In: Proceedings First Symposium on Games, Automata, Logic, and Formal Verification, GANDALF 2010, Minori (Amalfi Coast), Italy, 17–18th June 2010., pp. 89–102 (2010). https://doi.org/10.4204/EPTCS.25.11
35. Müller, D., Sickert, S.: LTL to deterministic Emerson-Lei automata. In: Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Roma, Italy, 20–22 September 2017., pp. 180–194 (2017). https://doi.org/10.4204/EPTCS.256.13
36. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32

37. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, pp. 179–190. ACM, New York, NY, USA (1989). https://doi.org/10.1145/75277.75293
38. Sickert, S., Esparza, J., Jaax, S., Kretínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Computer Aided Verification—28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part II, pp. 312–332 (2016). https://doi.org/10.1007/978-3-319-41540-6_17
39. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for the synthesis of reactive systems. STTT (Int. J. Softw. Tools Technol. Trans.) **15**(5–6), 433–454 (2013). https://doi.org/10.1007/s10009-012-0224-3
40. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0 (2015)

Affiliations

Michael Luttenberger[1] · Philipp J. Meyer[1] · Salomon Sickert[1]

Michael Luttenberger
luttenbe@in.tum.de

Philipp J. Meyer
meyerphi@in.tum.de

[1] Technical University of Munich, Munich, Germany