Int J Softw Tools Technol Transfer (2013) 15:519–539
DOI 10.1007/s10009-012-0228-z

SYNTHESIS

Bounded synthesis

Bernd Finkbeiner · **Sven Schewe**

Published online: 7 April 2012
© Springer-Verlag 2012

Abstract A fundamental challenge in the synthesis of reactive systems is the size of the search space: the number of candidate implementations of a temporal specification is typically superexponential or even, for distributed system architectures, infinite. In this article, we introduce the bounded synthesis approach, which makes it possible to traverse this immense search space in a structured manner. We fix a bound on a system parameter, such as the number of states, and limit the search to those implementations that fall below the bound. By incrementally expanding the search to larger bounds, we maintain completeness, while orienting the search towards the simplest (and often most useful) solutions. The technical backbone of this solution is a novel translation from formulas of linear-time temporal logic to sequences of safety tree automata, which are guaranteed to underapproximate the specification and to eventually become emptiness-equivalent. Bounded synthesis is applicable to the entire range of synthesis problems, from individual processes to synchronous and asynchronous distributed systems, to systems with additional design constraints, such as symmetry. We include experimental results from a SMT-based implementation, which demonstrate that bounded synthesis solves many synthesis problems that were previously considered intractable.

Keywords LTL synthesis · Reactive systems · Infinite games · Co-Büchi automata · SMT-based synthesis · Synthesis of distributed systems

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by the Engineering and Physical Science Research Council (EPSRC) through grant EP/H046623/1 "Synthesis and Verification in Markov Game Structures". It extends our work on bounded synthesis previously presented in [9,25,27].

B. Finkbeiner (✉)
Universität des Saarlandes, Saarbrücken, Germany
e-mail: finkbeiner@cs.uni-sb.de

S. Schewe
University of Liverpool, Liverpool, UK
e-mail: sven.schewe@liverpool.ac.uk

1 Introduction

Verification and synthesis both provide a formal guarantee that a system is implemented correctly. The difference between the two approaches is that while verification proves that a *given* implementation satisfies the specification, synthesis automatically derives one such implementation. Synthesis, thus, has the obvious advantage that it completely eliminates the need for manually writing and debugging code.

The common argument *against* synthesis is its complexity. Measured in the size of the system specification, given as a formula of linear-time temporal logic (LTL), e.g., the synthesis problem is 2EXPTIME-complete, while the model checking problem is in PSPACE. The comparison is even less favorable when one is interested in distributed systems. In verification, one can separately measure the complexity in the size of the specification and in the size of the system, and in either case the problem remains in PSPACE. The synthesis problem, however, is already undecidable for simple distributed architectures. Consider, e.g., the typical 2-process arbiter architecture shown in Fig. 1b: the environment (*env*) sends requests (r_1, r_2) for access to a critical resource to two processes p_1 and p_2, which react by sending out grants (g_1, g_2). As shown by Pnueli and Rosner [21], the synthesis problem is undecidable for this architecture, because both p_1 and p_2 have access to information (r_1 and r_2, respectively) that is hidden from the other process. For system architec-

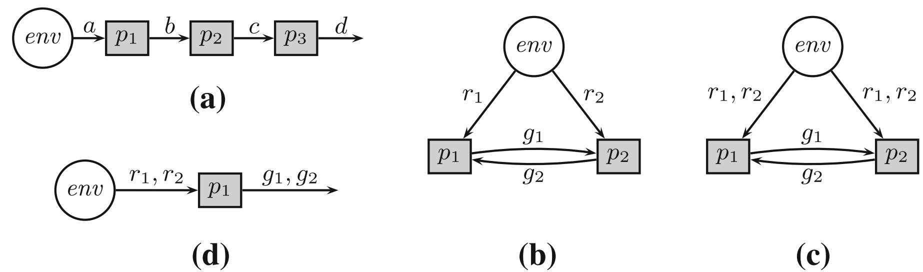


Fig. 1 Distributed architectures: **a** pipeline architecture, **b** 2-process arbiter architecture, **c** 2-process arbiter architecture with complete information, **d** single-process architecture

tures without such *information forks* [8], like pipeline architectures (Fig. 1a shows a pipeline of length 3), the synthesis problem is decidable, but has non-elementary complexity.

But is this comparison between verification and synthesis fair? The high complexity of synthesis is explained by the fact that, as pointed out by Rosner [22], a small LTL formula of size n which refers to m different processes already suffices to specify a system that cannot be implemented with less than m-exp(n) states. It is questionable, however, if this worst case situation should be used as an argument that synthesis, unlike verification, is an intractable problem. From a technical point of view, synthesis looks worse, because the size of the implementation is an explicit parameter in the complexity of verification, and left implicit in the complexity of synthesis; from a practical point of view, it is questionable whether or not huge implementations should even be considered by the synthesis algorithm, because they are likely to violate other design considerations (such as the available memory).

In this article, we introduce the bounded approach to synthesis, where we focus the search towards simple implementations, by setting an upper limit on selected system parameters—like the size of the implementation—in advance. Our starting point is the representation of the LTL specification as a universal co-Büchi tree automaton. We show that the acceptance of a finite-state transition system by a universal co-Büchi automaton can be characterized by the existence of an annotation that maps each pair of a state of the automaton and a state of the transition system to a natural number. The advantage of this characterization is that the acceptance condition can be simplified to a simple *safety condition*: we show that the universal co-Büchi automaton can be translated to an (emptiness-equivalent) deterministic *safety automaton* that implicitly builds a valid annotation.

We focus the search towards simple implementations by bounding the number of rejecting states that appear in a run of the automaton. For a universal co-Büchi tree automaton with n states and a bound b, we obtain a universal safety tree automaton with $b \cdot n$ states that can be determined to a deterministic safety tree automaton with b^n states. The language of the resulting safety automaton is a sub-language of the language of the initial universal co-Büchi automaton, and becomes emptiness-equivalent to the universal co-Büchi automaton for sufficiently high bounds. The emptiness of the safety automaton can then be determined in a simple two-player game, where player *accept* represents the system implementation and wins the game if the (strengthened) specification is satisfied, while the opponent, player *reject*, wins the game if the specification is violated.

The argument that the safety automata are, for a sufficiently large bound, emptiness-equivalent to the universal co-Büchi automaton relies on the existence of an upper bound on the maximal size of a minimal implementation [15,18,23,24]. A natural variation of the problem is to limit the number of states directly, by setting an a-priori bound, or by incrementally increasing such a bound. Inspired by the success of bounded model checking [1,2], we show that the bounded synthesis problem of finding an implementation whose size is bounded by b can be effectively reduced to a SMT problem. We define a constraint system that describes the existence of a valid annotation.

The reduction to SMT has two major advantages: The first advantage is that, in practice, there are often small solutions, and by first searching for small solutions we can greatly accelerate the synthesis process. The second advantage is the flexibility of the approach, which becomes apparent when we turn to extensions of the synthesis problem. We show that bounded synthesis can easily be extended to the synthesis of distributed systems [8,11,21,28], partial designs [8,28], asynchronous systems [20,26,29], distributed architectures/partial designs with asynchronous composition [26], and synthesis from component libraries [13].

The reason for this flexibility is in the difference in the way the incomplete information is represented. If the architecture consists of more than one process, as in the arbiter architecture of Fig. 1b, then a victory for player *accept* only means that the specification can be implemented in the slightly modified architecture (shown for the arbiter example in Fig. 1c), where all processes have the same information. An implementation for the architecture with incompletely informed processes must additionally satisfy a consistency requirement: if a process cannot distinguish between two different computation paths, it must react in the same way.

Since the consistency requirement is not a regular property, it is hard (or even impossible for most architectures [8,21,26,28]) to encode the requirement in the standard automata-based approach [8,11,21,26]. In the

SMT-based approach, on the other hand, this is a simple task: we add constraints which require that the resulting implementation is consistent with the limited information available to the distributed processes. For this purpose, we introduce a mapping that decomposes the states of the safety game into the states of the individual processes: because the reaction of a process only depends on its local state, the process is forced to give the same reaction whenever it cannot distinguish between two paths in the safety game. The satisfiability of the constraint system can be checked using standard SMT solvers [10,14].

As a result, we obtain an effective algorithm for the synthesis of size-bounded implementations from LTL specifications in arbitrary distributed architectures. By iteratively increasing the bound, our construction can also be used as a semi-decision procedure for the standard (unbounded) synthesis problem.

Related work The *synthesis of distributed reactive systems* was pioneered by Pnueli and Rosner [21], who showed that the synthesis problem is undecidable in general and has non-elementary complexity for pipeline architectures. An automata-based synthesis algorithm for pipeline and ring architectures is due to Kupferman and Vardi [11]; Walukiewicz and Mohalik provided an alternative game-based construction [30]. We showed that the synthesis problem is decidable if, and only if, the architecture does not contain an information fork [8]. Madhusudan and Thiagarajan [16] consider the special case of *local* specifications (each property refers only to the variables of a single process). Among the class of acyclic architectures (without broadcast) this synthesis problem is decidable for exactly the doubly flanked pipelines. Castellani et al. [3] consider *transition systems* as the specification language: an implementation is correct if the product of the processes is bisimilar to the specification. In this case, the synthesis problem is decidable independently of the architecture.

The extension to asynchronous distributed systems has been discussed in [17,26]. The general result is that synthesis in asynchronous systems becomes undecidable as soon as we try to synthesize two or more components [26], the automated construction of distributed asynchronous controllers remains decidable if severe restrictions are imposed on the languages [17].

Our translation of LTL formulas to safety tree automata is based on Kupferman and Vardi's *Safraless decision procedures* [12]. We use their idea of avoiding Safra's determinization using universal co-Büchi automata. Our construction improves on [12] in that it produces deterministic safety automata instead of non-deterministic Büchi automata. The resulting safety automata are also smaller than the Büchi automata from their construction. Since the original publication of the bounded synthesis approach in the preliminary version of this article [27], there have been several successful implementations based on bounded synthesis, notably the antichain-based *Acacia* [7] and the BDD-based *Unbeast* [5,6].

2 The synthesis problem

The synthesis problem is to decide whether or not there exists an implementation that satisfies a given specification. In the following, we formalize this problem statement using LTL as the specification language, and labeled transition systems as the representation of implementations.

2.1 Specifications

We use LTL [19] as the specification logic.

Syntax Let Π be a set of atomic propositions. The *syntax* of LTL over a finite set V of atomic propositions defined by the following grammar:

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

where $p \in \Pi$ is an atomic proposition.

As additional abbreviations, we will also use the *Eventually* operator $\diamondsuit$ and the *Globally* operator $\square$, which are defined as follows:

$$\diamondsuit\,\varphi \equiv true\,\mathcal{U}\,\varphi,$$
$$\square\,\varphi \equiv \neg(\diamondsuit\,\neg\varphi).$$

Semantics LTL formulas are interpreted over infinite words. For an infinite word $\sigma \in \omega \to 2^{\Pi}$ and a natural number $i \in \omega$, the semantics of an LTL formula is defined as follows:

- for atomic propositions $p \in \Pi$,
  - $\sigma, i \models p :\Leftrightarrow p \in \sigma(i)$
- for the boolean connectives, where φ and ψ are LTL formulas,
  - $\sigma, i \models \neg\varphi :\Leftrightarrow \sigma, i \not\models \varphi$, and
  - $\sigma, i \models \varphi \vee \psi :\Leftrightarrow \sigma, i \models \varphi$ or $\sigma, i \models \psi$;
- for the temporal path operators, where φ and ψ are LTL formulas,
  - $\sigma, i \models \bigcirc\varphi :\Leftrightarrow \sigma, i+1 \models \varphi$, and
  - $\sigma, i \models \varphi\,\mathcal{U}\,\psi :\Leftrightarrow \exists n \geq i.\ \sigma, n \models \psi$ and $\forall m \in \{i, \ldots, n-1\}.\ \sigma, m \models \varphi$.

A sequence $\sigma \in \omega \to 2^{\Pi}$ is a *model* of an LTL formula φ, denoted by $\sigma \models \varphi$, if, and only if, $\sigma, 0 \models \varphi$.

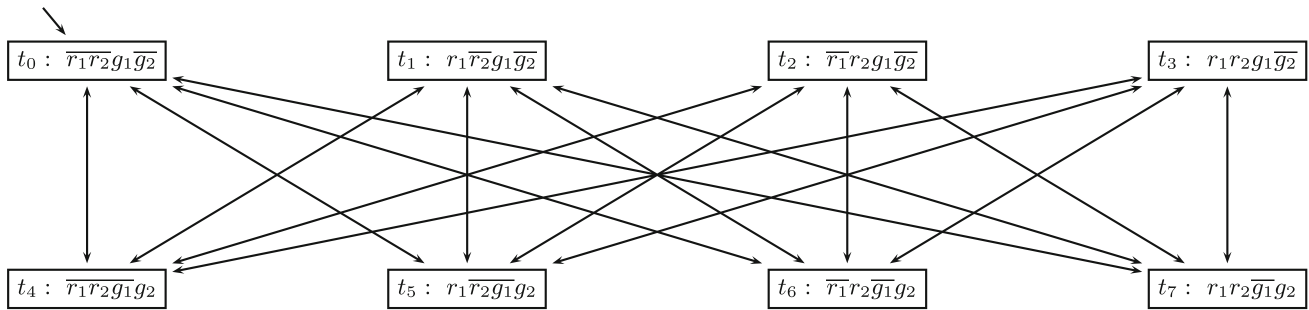


Fig. 2 Example of a labeled transition system with directions $\Upsilon = 2^{\{r_1,r_2\}}$ and labels $\Sigma = 2^{\{r_1,r_2,g_1,g_2\}}$. Each state t is depicted with its label $o(t)$. The transition function is chosen so that the transition system is input-enabled, i.e., $\tau(t_0, \overline{r_1 r_2}) = t_4$, $\tau(t_0, r_1\overline{r_2}) = t_5$, etc. The transition system satisfies the specification $\psi = \Box(r_1 \rightarrow \Diamond g_1) \wedge \Box(r_2 \rightarrow \Diamond g_2) \wedge \Box\neg(g_1 \wedge g_2)$ in the arbiter synthesis example $(\{r_1, r_2\}, \{g_1, g_2\}, \emptyset, \psi)$

2.2 Implementations

For a given finite set Υ of directions and a finite set Σ of labels, a Σ-labeled Υ-*transition system* is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a set of states T, an initial state $t_0 \in T$, a transition function $\tau : T \times \Upsilon \rightarrow T$, and a labeling function $o : T \rightarrow \Sigma$. $\mathcal{T}$ is a *finite-state* transition system if, and only if, T is finite.

We are interested in 2^Π-labeled 2^I-transition systems, which describe implementations for a given set Π of atomic propositions that is partitioned into a set I of boolean input variables and a set O of boolean output variables.

A 2^Π-labeled 2^I-transition system (T, t_0, τ, o) *satisfies* an LTL formula φ if, for all sequences $\mu : \omega \rightarrow T$ that start in the initial state $\mu(0) = t_0$ and adhere to the successor relation, i.e., $\forall i \in \omega \, \exists \upsilon \in \Upsilon.\ \mu(i+1) = \tau(\mu(i), \upsilon)$, the sequence $\sigma_\mu : i \mapsto o(\mu(i))$ is a model of φ.

We assume that the initial input is some fixed set $i_0 \subseteq I$. We call a 2^Π-labeled 2^I-transition system *input preserving* if the label of each state accurately reflects the last input, i.e., if, for all states $t \in T$ and inputs $i \subseteq I$, it holds that $o(\tau(t, i)) \cap I = i$, and for the initial state t_0, we additionally have that $o(t_0) \cap I = i_0$.

2.3 The synthesis problem

The synthesis problem is given as a tuple (I, O, i_0, φ), where I is a set of boolean input variables, O is a set of boolean output variables, $i_0 \subseteq I$ is the fixed initial input, and φ is an LTL formula over the set $\Pi = I \dot\cup O$ of atomic propositions.

We say that the specification φ is (finite-state) *realizable* if there exists an input preserving 2^Π-labeled 2^I-transition system that satisfies φ. The synthesis problem is to decide whether or not φ is realizable.

Example We consider the synthesis of a simple arbiter, which receives requests from two clients, represented by two input variables $I = \{r_1, r_2\}$, and responds by assigning grants, represented by two output variables $O = \{g_1, g_2\}$. We specify that each request should eventually be followed by a grant, and that the two grants should never be assigned simultaneously:

$$\psi = \Box(r_1 \rightarrow \Diamond g_1) \ \wedge\ \Box(r_2 \rightarrow \Diamond g_2) \ \wedge \Box\neg(g_1 \wedge g_2).$$

We assume that initially there are no requests: $i_0 = \emptyset$. In the synthesis problem (I, O, i_0, ψ), the specification ψ is realizable. Figure 2 shows a $\{r_1, r_2, g_1, g_2\}$-labeled $\{r_1, r_2\}$-transition system that satisfies φ by alternating between setting g_1 to true and g_2 to false, denoted by $g_1\overline{g_2}$, and setting g_1 to false and g_2 to true. The transition system has a total of eight states, corresponding to the two output assignments $g_1\overline{g_2}$ and $\overline{g_1}g_2$ for each of the four possible input assignments $\overline{r_1 r_2}$, $r_1\overline{r_2}$, $\overline{r_1}r_2$, and $r_1 r_2$.

3 Bounded synthesis

We now introduce the bounded synthesis approach. After reviewing some basic terminology from automata theory in the following subsection, we define an annotation function for transition systems that maps each state to a bounded domain, such that the existence of a valid annotation implies the satisfaction of the specification. In Sect. 3.3, we show that, for every transition system that satisfies the specification, there exists a bound and a valid annotation function. In Sect. 3.4, we show how annotations can be used to reduce the synthesis problem to a simple emptiness check on safety automata.

3.1 Preliminaries: tree automata

An *alternating parity tree automaton* is a tuple $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$, where Σ denotes a finite set of labels, Υ denotes a finite set of directions, Q denotes a finite set of states, $q_0 \in Q$ denotes a designated initial state, δ denotes a transition function, and $\alpha : Q \rightarrow C \subset \mathbb{N}$ is a coloring

function. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state and an input letter to a positive boolean combination of atoms that are pairs of states and directions (i.e., $\delta(q, \sigma)$ is a formula built from elements of $Q \times \Upsilon$, conjunction $\wedge$, disjunction $\vee$, *true*, and *false*).

In our setting, the automaton runs on Σ-labeled Υ-transition systems. The acceptance mechanism is defined in terms of run graphs.

A *run graph* of an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ on a Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is a minimal directed graph $\mathcal{G} = (G, E)$ that satisfies the following constraints:

- The vertices $G \subseteq Q \times T$ form a subset of the product of Q and T.
- The pair of initial states $(q_0, t_0) \in G$ is a vertex of $\mathcal{G}$.
- For each vertex $(q, t) \in G$, the set $\{(q', \upsilon) \in Q \times \Upsilon \mid ((q, t), (q', \tau(t, \upsilon))) \in E\}$ satisfies $\delta(q, o(t))$.

A run graph is *accepting* if every infinite path $g_0 g_1 g_2 \ldots \in G^\omega$ in the run graph satisfies the *parity condition*, which requires that the highest number occurring infinitely often in the sequence $\alpha_0 \alpha_1 \alpha_2 \in \mathbb{N}$ with $\alpha_i = \alpha(q_i)$ and $g_i = (q_i, t_i)$ is even. A transition system is accepted if it has an accepting run graph.

The set of transition systems accepted by an automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is empty if, and only if, its language is empty.

The acceptance of a transition system can also be viewed as the outcome of a game, where player *accept* chooses, for a pair $(q, t) \in Q \times T$, a set of atoms satisfying $\delta(q, o(t))$, and player *reject* chooses one of these atoms, which is executed. The transition system is accepted if, and only if, player *accept* has a strategy enforcing a path that satisfies the parity condition.

A *non-deterministic* automaton is a special alternating automaton, where the image of δ consists only of such formulas that, when rewritten in disjunctive normal form, contain in every disjunct at most one element of $Q \times \{\upsilon\}$ for each $\upsilon \in \Upsilon$. The emptiness of a non-deterministic automaton can be checked with a variation of the acceptance game called the *emptiness game*, where, in each step, player *accept* additionally chooses the label from Σ. A non-deterministic automaton is empty if the emptiness game is won by player *reject*.

An alternating automaton is called *universal* if, for all states q and input letters σ, $\delta(q, \sigma)$ does not contain disjunctions. In an abuse of notation, we also denote, in this case, by $\delta(q, \sigma)$ the set of conjuncts. A universal and non-deterministic automaton is called *deterministic*.

A parity automaton is called a *Büchi* automaton if the image of α is contained in $\{1, 2\}$, a *co-Büchi* automaton if the image of α is contained in $\{0, 1\}$, and a *safety* automaton if the image of α is $\{0\}$. Büchi and co-Büchi automata are denoted by $(\Sigma, \Upsilon, Q, q_0, \delta, F)$, where $F \subseteq Q$ denotes the states with the higher color. In a Büchi automaton, we call the states in F *accepting*, in a co-Büchi automaton *rejecting*. Safety automata are denoted by $(\Sigma, \Upsilon, Q, q_0, \delta)$. A run graph of a Büchi automaton is, thus, accepting if, on every infinite path, there are infinitely many visits to F; a run graph of a co-Büchi automaton is accepting if, on every path, there are only finitely many visits to F. For safety automata, every run graph is accepting.

3.2 Annotated transition systems

We now introduce an annotation function for labeled transition systems. Our starting point is a representation of the specification as a universal co-Büchi automaton. Since the automaton is universal, every transition system in the language of the automaton has a unique run graph. The annotation assigns to each pair (q, t) of a state q of the automaton and a state t of the transition system either a natural number or a blank sign. The natural number indicates the maximal number of rejecting states that occur on some path to (q, t) in the run graph. Transition systems for which there is an annotation that assigns only natural numbers to the vertices of the run graph, thus, have an upper bound on the number of visits to the rejecting states. We call such annotations *valid*. The transition systems with valid annotations are exactly those that are accepted by the automaton.

Universal co-Büchi automata We translate a given LTL specification φ into an equivalent universal co-Büchi automaton $\mathcal{U}_\varphi$. This can be done with a single exponential blow-up by first negating φ, then translating $\neg\varphi$ into an equivalent non-deterministic Büchi word automaton, and then constructing a universal co-Büchi automaton that simulates the Büchi automaton along each path: if each path is co-Büchi accepting (i.e., it violates the Büchi condition), then the specification φ must hold along every path.

Theorem 1 [12] *Given an LTL formula φ, we can construct a universal co-Büchi automaton $\mathcal{U}_\varphi$ with $2^{O(|\varphi|)}$ states that accepts a transition system $\mathcal{T}$ if, and only if, $\mathcal{T}$ satisfies φ.*

Example Figure 3 shows a universal co-Büchi automaton for the specification $\psi = \Box(r_1 \to \Diamond g_1) \wedge \Box(r_2 \to \Diamond g_2) \wedge \Box\neg(g_1 \wedge g_2)$ of the simple arbiter from Sect. 2.3.

Annotations An *annotation* of a transition system $\mathcal{T} = (T, t_0, \tau, o)$ on a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ is a function $\lambda : Q \times T \to \{_\} \cup \mathbb{N}$. We call an annotation *c-bounded* if its mapping is contained in $\{_\} \cup \{0, \ldots, c\}$, and *bounded* if it is c-bounded for some $c \in \mathbb{N}$. An annotation is *valid* if it satisfies the following conditions:

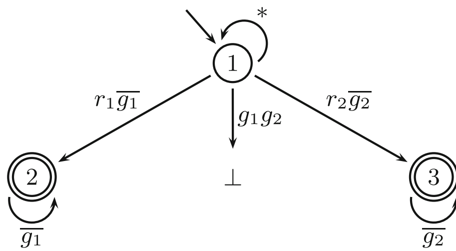


Fig. 3 Specification of a simple arbiter, represented as a universal co-Büchi automaton. The states depicted as *double circles* (2 and 3) are the rejecting states in F

- the pair (q_0, t_0) of initial states is annotated with a natural number ($\lambda(q_0, t_0) \neq _$), and
- if a pair (q, t) is annotated with a natural number ($\lambda(q, t) = n \neq _$) and $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $(q', \tau(t, \upsilon))$ is annotated with a greater number, which needs to be strictly greater if $q' \in F$ is rejecting. That is, $\lambda(q', \tau(t, \upsilon)) \rhd_{q'} n$ where $\rhd_{q'}$ is $>$ for $q' \in F$ and $\geq$ otherwise.

Theorem 2 *A finite-state Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is accepted by a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ if, and only if, it has a valid $(|T| \cdot |F|)$-bounded annotation.*

Proof Since $\mathcal{U}$ is universal, $\mathcal{U}$ has a unique run graph $\mathcal{G} = (G, E)$ on $\mathcal{T}$. Since $\mathcal{T}$ and $\mathcal{U}$ are finite, $\mathcal{G}$ is finite, too.

If $\mathcal{G}$ contains a lasso with a rejecting state in its loop, i.e., a path $(q_0, t_0)(q_1, t_1)\ldots(q_n, t_n) = (q'_0, t'_0)$ and a path $(q'_0, t'_0)(q'_1, t'_1)\ldots(q'_m, t'_m) = (q'_0, t'_0)$ such that q'_i is rejecting for some $i \in \{1, \ldots, m\}$, then, by induction, any valid annotation λ satisfies $\lambda(q_j, t_j) \in \mathbb{N}$ for all $j \in \{0, \ldots, n\}$, $\lambda(q'_j, t'_j) \in \mathbb{N}$ for all $j \in \{0, \ldots, m\}$, $\lambda(q'_{j-1}, t'_{j-1}) \leq \lambda(q'_j, t'_j)$ for all $j \in \{1, \ldots, m\}$, and $\lambda(q'_{i-1}, t'_{i-1}) < \lambda(q'_i, t'_i)$. ↯

If, on the other hand, $\mathcal{G}$ does not contain a lasso with a rejecting state in its loop, we can easily infer a valid $(|T|\cdot|F|)$-bounded annotation by assigning to each vertex $(q, t) \in G$ of the run graph the highest number of rejecting states occurring on some path $(q_0, t_0)(q_1, t_1)\ldots(q, t)$, and by assigning _ to every pair of states $(q, t) \notin G$ not in $\mathcal{G}$. □

3.3 Estimating the bound

Theorem 2 shows that the existence of a transition system with a valid annotation is a sufficient condition for the realizability of the specification. We now show that it is also a necessary condition, i.e., there exists a bound c such that a transition system with a valid c-bounded annotation exists if the specification is realizable. If the specification is realizable, then there exists an implementation of minimal size. We compute the bound c by estimating the size of such minimal implementations.

Theorem 3 [18,24] *Given a universal co-Büchi automaton $\mathcal{U}$ with n states, we can construct an equivalent deterministic parity automaton $\mathcal{P}$ with $n!^2$ states and $2n$ colors.*

A solution to the synthesis problem is required to be input preserving, i.e., in every state, the label must accurately reflect the input. Input preservation can be checked with a deterministic safety automaton $\mathcal{D}_\mathcal{I}$, whose states are formed by the possible inputs $\mathcal{I} = 2^{O_{env}}$. In every state $i \in \mathcal{I}$, $\mathcal{D}_\mathcal{I}$ checks if the label agrees with the input i, and sends the successor state $i' \in \mathcal{I}$ into the direction i'. If $\mathcal{U}$ accepts an input-preserving transition system, then we can construct a *finite* input-preserving transition system, which is accepted by $\mathcal{U}$, by evaluating the emptiness game of the product automaton of $\mathcal{P}$ and $\mathcal{D}_\mathcal{I}$. The minimal size of such an input-preserving transition system can be estimated by the size of $\mathcal{P}$ and $\mathcal{I}$.

Corollary 1 *If a universal co-Büchi automaton $\mathcal{U}$ with n states and m rejecting states accepts an input-preserving transition system, then $\mathcal{U}$ accepts a finite input-preserving transition system $\mathcal{T}$ with $n!^2 \cdot |\mathcal{I}|$ states, where $\mathcal{I} = 2^{O_{env}}$. $\mathcal{T}$ has a valid $m \cdot n!^2 \cdot |\mathcal{I}|$-bounded annotation for $\mathcal{U}$.*

3.4 Bounded synthesis

Using the annotation function, we can reduce the synthesis problem to a simple emptiness check on safety automata. The following theorem shows that there is a deterministic safety automaton that, for a given parameter value c, accepts a transition system if, and only if, it has a valid c-bounded annotation. This leads to the following synthesis procedure:

Given a specification, represented as a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$, we construct a sequence of safety automata that check for valid bounded annotations up to the bound $c = |F| \cdot b$, where b is either the predefined bound b_A on the size of the transition system, or the sufficient bound $n!^2 \cdot |\mathcal{I}|$ from Corollary 1. If the intersection of $\mathcal{D}_\mathcal{I}$ with one of these automata is non-empty, then the specification is realizable; if the intersection with the safety automaton for the largest parameter value c is empty, then the specification is unrealizable. The emptiness of the automata can be checked by solving their emptiness games.

Theorem 4 *Given a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$, we can construct a family of deterministic safety automata $\{\mathcal{D}_c = (\Sigma, \Upsilon, S_c, s_0, \delta_c) \mid c \in \mathbb{N}\}$ such that $\mathcal{D}_c$ accepts a transition system if, and only if, it has a valid c-bounded annotation.*

Construction We choose the functions from Q to the union of $\mathbb{N}$ and a blank sign ($S = Q \to \{_\} \cup \mathbb{N}$) as the state space of a deterministic safety automaton $\mathcal{D} = (\Sigma, \Upsilon, S, s_0, \delta_\infty)$.

Each state of $\mathcal{D}$ indicates how many times a rejecting state may have been visited in some trace of the run graph that passes the current position in the transition system. The initial state of $\mathcal{D}$ maps the initial state of $\mathcal{U}$ to 0 ($s_0(q_0) = 0$) and all other states of $\mathcal{U}$ to blank ($\forall q \in Q \setminus \{q_0\}. s_0(q) = _$).

We compute the update of a mapping s after reading letter σ with the help of an auxiliary function $\delta_\infty^+(s, \sigma)$, which records, for all directions υ and successor states q', the new number of visits to rejecting states. We define $\delta_\infty^+(s, \sigma) = \{((q', s(q) + f(q)), \upsilon) \mid q, q' \in Q, s(q) \neq _, \text{and } (q', \upsilon) \in \delta(q, \sigma)\}$, where $f(q) = 1$ for all $q \in F$ and $f(q) = 0$ for all $q \notin F$.

The transition function δ_∞ is defined as follows: $\delta_\infty(s, \sigma) = \bigwedge_{\upsilon \in \Upsilon}(s_\upsilon, \upsilon)$ with $s_\upsilon(q) = max\{n \in \mathbb{N} \mid ((q, n), \upsilon) \in \delta_\infty^+(s, \sigma)\}$, where $max\{\emptyset\} = _$. The family of safety automata $\mathcal{D}_c$ is formed by restricting the states of $\mathcal{D}$ to $S_c = Q \rightarrow \{_\} \cup \{0, \dots, c\}$.

Proof Let λ be a valid c-bounded annotation of $\mathcal{T} = (T, t_0, \tau, o)$ for $\mathcal{U}$, and let λ_t denote the function with $\lambda_t(q) = \lambda(q, t)$. For two functions $s, s' : Q \rightarrow \{_\} \cup \mathbb{N}$, we write $s \leq s'$ if $s(q) \leq s'(q)$ holds for all $q \in Q$, where $_$ is the minimal element ($_ < n$ for all $n \in \mathbb{N}$). We show by induction that $\mathcal{D}_c$ has a run graph $\mathcal{G} = (G, E)$ for $\mathcal{T}$, such that $s \leq \lambda_t$ holds for all vertices $(s, t) \in G$ of the run graph. For the induction basis, $s_0 \leq \lambda_{t_0}$ holds by definition. For the induction step, let $(s, t) \in G$ be a vertex of $\mathcal{G}$. By induction hypothesis, we have $s \leq \lambda_t$. With the definition of δ_∞^+ and the validity of λ, we can conclude that $((q', n), \upsilon) \in \delta_\infty^+(s, o(t))$ implies $n \leq \lambda_{\tau(t,\upsilon)}(q')$, which immediately implies $s' \leq \lambda_{t'}$ for all successors (s', t') of (s, t) in $\mathcal{G}$.

Let now $\mathcal{G} = (G, E)$ be an accepting run graph of $\mathcal{D}_c$ for $\mathcal{T}$, and let $\lambda(q, t) = max\{s(q) \mid (s, t) \in G\}$. Then λ is obviously a c-bounded annotation. For the validity of λ, $\lambda(q_0, t_0) \in \mathbb{N}$ holds since $s_0(q_0) \in \mathbb{N}$ is a natural number and $(s_0, t_0) \in G$ is a vertex of $\mathcal{G}$. Also, if a pair (q, t) is annotated with a natural number $\lambda(q, t) = n \neq _$, then there is a vertex $(s, t) \in G$ with $s(q) = n$. If now $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $((q', n + f(q')), \upsilon) \in \delta_\infty^+(s, o(t))$ holds, and the υ-successor $(s', \tau(t, \upsilon))$ of (s, t) satisfies $s'(q') \triangleright_{q'} n$. The validity of λ now follows with $\lambda(q', \tau(t, \upsilon) \geq s'(q')$. □

Remark Since $\mathcal{U}$ may accept transition systems where the number of rejecting states occurring on a path is unbounded, the union of the languages of all $\mathcal{D}_c$ is, in general, a strict subset of the language of $\mathcal{U}$. Every finite-state transition system in the language of $\mathcal{U}$, however, is accepted by almost all $\mathcal{D}_c$.

Example Consider again the universal co-Büchi automaton shown in Fig. 3, which corresponds to the arbiter specification from Sect. 2.3. The emptiness game for $\mathcal{D}_1$ intersected with $\mathcal{D}_\mathcal{I}$ is depicted in Fig. 4.

4 Synthesis of systems with bounded size

A natural variation of the synthesis problem is to only consider implementations whose size satisfies a given limit on the number of states. As discussed in Sect. 3.3, a limit on the number of states immediately provides a bound for bounded synthesis.

We present a constraint-based approach for the synthesis of systems with bounded size. Based on an appropriate encoding of transition systems and annotations, we use a SMT solver to find both the input-preserving transition system and a valid annotation. We represent the (unknown) transition system and its annotation by uninterpreted functions. The existence of a valid annotation is, thus, reduced to the satisfiability of a constraint system in first-order logic modulo finite integer arithmetic. The advantage of this representation is that the size of the constraint system is small (bilinear in the size of $\mathcal{U}$ and the number of directions). Furthermore, the additional constraints needed for distributed and asynchronous synthesis, which will be defined in Sects. 5 and 6, have a compact representation as well (logarithmic in the number of directions of the individual processes).

Remark Integer arithmetic is useful for explaining the algorithm, but we do not need to build on integer arithmetic: the essential property is to guarantee the absence of cycles that contain a rejecting state. But to prove this absence, *any* ordered set (finite or not) suffices for the labels. Finiteness, in turn, is only used to restrict the size of the system. An efficient implementation will, therefore, build on theories with more efficient algorithms, such as real arithmetic.

The constraint system specifies the existence of a finite input-preserving 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T} = (T, t_0, \tau, o)$ that is accepted by the universal co-Büchi automaton $\mathcal{U}_\varphi = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ and has a valid annotation λ.

To encode the transition function τ, we introduce a unary function symbol τ_υ for every output $\upsilon \subseteq O_{env}$ of the environment. Intuitively, τ_υ maps a state t of the transition system $\mathcal{T}$ to its υ-successor $\tau_\upsilon(t) = \tau(t, \upsilon)$.

To encode the labeling function o, we introduce a unary predicate symbol a for every variable $a \in V$. Intuitively, a maps a state t of the transition system $\mathcal{T}$ to *true* if, and only if, it is part of the label $o(t) \ni a$ of $\mathcal{T}$ in t.

To encode the annotation, we introduce, for each state q of the universal co-Büchi automaton $\mathcal{U}$, a unary predicate symbol $\lambda_q^\mathbb{B}$ and a unary function symbol $\lambda_q^\#$. Intuitively, $\lambda_q^\mathbb{B}$ maps a state t of the transition system $\mathcal{T}$ to *true* if, and only if, $\lambda(q, t)$ is a natural number, and $\lambda_q^\#$ maps a state t of the transition system $\mathcal{T}$ to $\lambda(q, t)$ if $\lambda(q, t)$ is a natural number and is unconstrained if $\lambda(q, t) = _$.

We can now formalize that the annotation of the transition system is valid by the following first-order constraints (modulo finite integer arithmetic):

Fig. 4 Example of a safety game. The figure shows the emptiness game for the intersection of $\mathcal{D}_1$ and $\mathcal{D}_{\mathcal{I}}$ in the arbiter example (Fig. 3). *Circles* denote game positions for player *accept*, *rectangles* denote game positions for player *reject*. Game positions that are not completely expanded (i.e., that have more successors if the parameter is increased) are *dashed*. The starting position specifies $\overline{r_1 r_2}$ as the (arbitrarily chosen) root direction. Player *accept* wins the game by avoiding the move to (0, 1, 1)

$\forall t.\, \lambda^{\mathbb{B}}_q(t) \;\wedge\; \underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))} \;\rightarrow\; \lambda^{\mathbb{B}}_{q'}(\tau_\upsilon(t)) \;\wedge\; \lambda^{\#}_{q'}(\tau_\upsilon(t)) \triangleright_{q'} \lambda^{\#}_q(t)$, where $\overrightarrow{a}(t)$ represents the label $o(t)$, $\underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))}$ represents the corresponding propositional formula, and $\triangleright_{q'}$ stands for $\triangleright_{q'} \equiv >$ if $q' \in F$ and $\triangleright_{q'} \equiv \geq$ otherwise. Additionally, we require $\lambda^{\mathbb{B}}_{q_0}(0)$, i.e., we require the pair of initial states to be labeled by a natural number (without loss of generality $t_0 = 0$).

To guarantee that the resulting transition system is input-preserving, we add, for each $a \in O_{env}$ and each $\upsilon \subseteq O_{env}$, a constraint $\forall t.\, a(\tau_\upsilon(t))$ if $a \in \upsilon$, and a constraint $\forall t.\, \neg a(\tau_\upsilon(t))$ if $a \notin \upsilon$. Additionally, we require that the initial state is labeled with the root direction.

As an obvious implication of Theorem 2, this constraint system is satisfiable if, and only if, $\mathcal{U}$ accepts a finite input-preserving transition system.

Theorem 5 *The constraint system inferred from the specification is satisfiable modulo a theory with order if, and only if, the specification is finite-state realizable.*

Lemma 1 *For a specification represented as a universal co-Büchi automaton* $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$*, the inferred constraint system has size* $O(|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}|)$.

The main parameter of the constraint system is the bound b_A on the size of the transition system $\mathcal{T}_A$. If we use b_A to unravel the constraint system completely (i.e., if we resolve the universal quantification explicitly), the size of the resulting constraint system is linear in b_A.

Theorem 6 *For a specification, represented as a universal co-Büchi automaton* $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$*, and a given bound* b_A *on the size of the transition system* $\mathcal{T}_A$*, the unraveled constraint system has size* $O(b_A \cdot (|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}|))$*. It is satisfiable if, and only if, the specification is realizable in the fully informed architecture* $(\{env, p\}, V, \{I_p = O_{env}\}, \{O_{env}, O_p = V \setminus O_{env}\})$ *with respect to bound* b_A.

Example Figure 5 shows the constraint system, resulting from the specification of an arbiter by the universal co-Büchi automaton depicted in Fig. 3.

The first constraint represents the requirement that the resulting transition system must be input preserving, the second requirement represents the initialization (where $\neg r_1(0) \wedge \neg r_2(0)$ represents an arbitrarily chosen root direction), and the requirements 3 through 8 each encode one transition of the universal automaton of Fig. 3. Following the notation of Fig. 3, r_1 and r_2 represent the requests and g_1 and g_2 represent the grants.

5 Synthesizing distributed systems

We now generalize the synthesis problem to the case of distributed systems, where we synthesize several independent processes that must cooperate to guarantee that the specification is satisfied.

5.1 The distributed synthesis problem

Given a system architecture A and an LTL formula φ, the distributed synthesis problem is to decide whether or not there is an implementation for each system process in A, such that the composition of the implementations satisfies φ.

Architectures An *architecture* A is a tuple (P, env, V, I, O), where P is a set of processes consisting of a designated environment process $env \in P$ and a set of system processes $P^- = P \setminus \{env\}$. V is a set of boolean system variables (which also serve as atomic propositions), $I = \{I_p \subseteq V \mid p \in P^-\}$ assigns a set I_p of input variables to each system process $p \in P^-$, and $O = \{O_p \subseteq V \mid p \in P\}$ assigns a set O_p of output variables to each process $p \in P$ such that $\bigcup_{p \in P} O_p = V$. While the same variable $v \in V$ may occur in multiple sets in I to indicate broadcasting, the sets in O are assumed to be pairwise disjoint. If $O_{env} \subseteq I_p$ for every system process $p \in P^-$, we say the architecture is *fully informed*. Since every process in a fully informed architecture has enough information to simulate every other process, we can assume without loss of generality that a fully informed architecture contains only a single system process

1. $\forall t.\, r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 r_2}(t)) \wedge r_2(\tau_{\overline{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{\overline{r}_1 \overline{r}_2}(t))$
2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$
3. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t) \wedge \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 r_2}(t)) \geq \lambda_1^{\#}(t) \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t) \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}(t)$
4. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t)$
5. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \wedge \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)$
6. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \wedge \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)$
7. $\forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t) \wedge \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t) \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t) \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t)$
8. $\forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t) \wedge \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t) \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t) \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t)$

Fig. 5 Example of a constraint system for synthesis. The figure shows the constraint system for the arbiter example (Fig. 3)

p, and that the input variables of p are the output variables of the environment process $I_p = O_{env}$.

Distributed implementations Each system process $p \in P^-$ is implemented as a 2^{O_p}-labeled 2^{I_p}-transition system $\mathcal{T}_p = (T_p, t_p, \tau_p, o_p)$. The environment process is unconstrained except for the initial input, which we again assume to be given as a fixed set $i_0 \subseteq I$.

Let $P^- = \{p_1, p_2, \ldots, p_k\}$. The *composition* of the process implementations is the 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T}_A = (T, t_0, \tau, o)$ defined as follows: the set $T = T_{p_1} \times T_{p_2} \times \ldots \times T_{p_k} \times 2^{O_{env}}$ of states is formed by the product of the states of the process transition systems and the possible values of the output variables of the environment. The initial state $t_0 = (t_{p_1}, t_{p_2}, \ldots, t_{p_k}, e_0)$ is the tuple consisting of the initial states of the process transition systems and the initial environment output e_0. The labeling function o labels each state with the union of the labels of the process transition systems and the environment output: $o(s_{p_1}, s_{p_2}, \ldots, s_{p_k}, e) = o_{p_1}(s_{p_1}) \cup o_{p_2}(s_{p_2}) \cup \ldots \cup o_{p_k}(s_{p_k}) \cup e$. The transition function updates, for each system process p, the T_p part of the state in accordance with the transition function τ_p, using the visible part of the state label as input, and updates the $2^{O_{env}}$ part of the state with the output of the environment process: $\tau((s_{p_1}, s_{p_2}, \ldots, s_{p_k}, e), e') = (\tau_{p_1}(s_{p_1}, l \cap I_{p_1}), \tau_{p_2}(s_{p_2}, l \cap I_{p_2}), \ldots, \tau_{p_1}(s_{p_k}, l \cap I_{p_k}), e')$, where $l = o(s_{p_1}, s_{p_2}, \ldots, s_{p_k}, e)$.

Synthesis of distributed systems with bounded size We introduce bounds on the size of the process implementations and on the size of the composition. Given an architecture $A = (P, V, I, O)$, a specification φ is *realizable with respect to a family of bounds* $\{b_p \in \mathbb{N} \mid p \in P^-\}$ on the size of the system processes and a bound $b_A \in \mathbb{N}$ on the size of the composition T_A, if there exists a family of implementations $\{\mathcal{T}_p \mid p \in P^-\}$, where, for each process $p \in P^-$, $\mathcal{T}_p$ has at most b_p states, such that the composition $\mathcal{T}_A$ satisfies φ and has at most b_A states.

5.2 Synthesizing distributed systems

To solve the distributed synthesis problem for a given architecture $A = (P, V, I, O)$, we need to find a family of (finite-state) transition systems $\{\mathcal{T}_p = (T_p, t_0^p, \tau_p, o_p) \mid p \in P^-\}$ such that their composition to $\mathcal{T}_A$ satisfies the specification. The constraint system developed in the previous section can be adapted to distributed synthesis by explicitly decomposing the global state space of the combined transition system $\mathcal{T}_A$: we introduce a unary function symbol d_p for each process $p \in P^-$, which, intuitively, maps a state $t \in T_A$ of the product state space to its p-component $t_p \in T_p$.

The value of an output variable $a \in O_p$ may only depend on the state of the process transition system $\mathcal{T}_p$. We, therefore, replace every occurrence of $a(t)$ in the constraint system of the previous section by $a(d_p(t))$. Additionally, we require that every process p acts consistently on any two histories that it cannot distinguish. The update of the state of $\mathcal{T}_p$ may, thus, only depend on the state of $\mathcal{T}_p$ and the input visible to p. This is formalized by the following constraints:

1. $\forall t.\, d_p(\tau_\upsilon(t)) = d_p(\tau_{\upsilon'}(t))$ for all decisions $\upsilon, \upsilon' \subseteq O_{env}$ of the environment that are indistinguishable for p (i.e., $\upsilon \cap I_p = \upsilon' \cap I_p$).
2. $\forall t, u.\, d_p(t) = d_p(u) \wedge \bigwedge_{a \in I_p \setminus O_{env}} \big(a(d_{p_a}(t)) \leftrightarrow a(d_{p_a}(u))\big) \rightarrow d_p(\tau_\upsilon(t)) = d_p(\tau_\upsilon(u))$ for all decisions $\upsilon \subseteq O_{env} \cap I_p$ (picking one representative for each class of environment decisions that p can distinguish). $p_a \in P^-$ denotes the process controlling the output variable $a \in O_{p_a}$.

Since the combined transition system $\mathcal{T}_A$ is finite-state, the satisfiability of this constraint system modulo finite integer arithmetic (or any other theory with order) is equivalent to the distributed synthesis problem.

Theorem 7 *The constraint system inferred from the specification, represented as the universal co-Büchi automaton $\mathcal{U}$, and the architecture A are satisfiable modulo theories with order iff the specification is finite-state realizable in the architecture A.*

The constraint system for distributed synthesis is quite small:

Lemma 2 *For a specification, represented as a universal co-Büchi automaton $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, and an architecture A, the inferred constraint system for distributed synthesis has size $O(|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}| + \sum_{p \in P^-} |I_p \setminus O_{env}|)$.*

The main parameters of the constraint system for distributed synthesis are the bound b_A on the size of the transition system $\mathcal{T}_A$ and the family $\{b_p \mid p \in P^-\}$ of bounds on the process transition systems $\{\mathcal{T}_p \mid p \in P^-\}$. If we use these parameters to unravel the constraint system completely (i.e., if we resolve the universal quantification explicitly), the resulting transition system is linear in b_A, and quadratic in b_p.

Theorem 8 *For a given specification, represented as a universal co-Büchi automaton $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, an architecture $A = (P, V, I, O)$, a bound b_A on the size of the input-preserving transition system $\mathcal{T}_A$, and a family $\{b_p \mid p \in P^-\}$ of bounds on the process transition systems $\{\mathcal{T}_p \mid p \in P^-\}$, the unraveled constraint system has size $O(b_A \cdot (|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}|) + \sum_{p \in P^-} {b_p}^2 \cdot |I_p \setminus O_{env}|))$. It is satisfiable if, and only if, the specification is realizable in A with respect to the bounds b_A and $\{b_p \mid p \in P^-\}$.*

Example As an example for the reduction of the distributed synthesis problem to a constraint system, we consider the

$$
\begin{array}{rl}
4. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \to \neg g_1(d_1(t)) \vee \neg g_2(d_2(t)) \\
5. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(d_1(t)) \to \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t) \\
6. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(d_2(t)) \to \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t) \\
7. & \forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_1(t)) \to \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_2^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_2^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t) \\
8. & \forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_2(t)) \to \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_3^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_3^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t) \\
9. & \forall t.\ d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1\overline{r}_2}(t)) \wedge d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{\overline{r}_1 r_2}(t)) \wedge d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{\overline{r}_1\overline{r}_2}(t)) \\
10. & \forall t,u.\, d_1(t) = d_1(u) \wedge \big(g_2(d_2(t)) \leftrightarrow g_2(d_2(u))\big) \to d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1 r_2}(u)) \wedge d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1 r_2}(u)) \\
11. & \forall t,u.\, d_2(t) = d_2(u) \wedge \big(g_1(d_1(t)) \leftrightarrow g_1(d_1(u))\big) \to d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{r_1 r_2}(u)) \wedge d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{r_1\overline{r}_2}(u))
\end{array}
$$

Fig. 6 Example of a constraint system for distributed synthesis. The figure shows modifications and extensions to the constraint system from Fig. 5 for the arbiter example (Fig. 3) to implement the arbiter in the distributed architecture shown in Fig. 1b

problem of finding a distributed implementation to the arbiter specified by the universal automaton of Fig. 3 in the architecture of Fig. 1b. The functions d_1 and d_2 are the mappings to the processes p_1 and p_2, which receive requests r_1 and r_2 and provide grants g_1 and g_2, respectively. Figure 6 shows the resulting constraint system. Constraints 1–3, 5, and 6 are the same as in the fully informed case (Fig. 5). The consistency constraints 9–11 guarantee that processes p_1 and p_2 display the same behavior on all input histories they cannot distinguish.

6 Synthesis of asynchronous systems

The distributed synthesis problem discussed in the previous section becomes even harder if the processes are composed asynchronously: synthesis for asynchronous systems [20,26,29] is undecidable for all architectures that contain more than one black-box process [26].

In asynchronously composed systems, not all processes are scheduled at the same time. A simple way to model this is to explicitly introduce a scheduler that can activate or disable processes (including the environment) in every turn. To guarantee that the implementation works for all possible schedulings, we consider the scheduler as part of the environment.[1] In addition to being able to change the input variables (whenever the environment process is scheduled), the environment can now also disable any and all processes, including itself. When the environment is not scheduled, its output decisions stay the same. When a system process $p \in P^-$ is not scheduled, its *state*—and, as a result, its output—stays the same.

The set of environment outputs, thus, grows to

$$\mathcal{O}_{env} = \big(2^V \dot\cup \{env\}\big) \times 2^{P^-} \dot\cup 2^{P^-},$$

and the asynchronously composed system is a 2^V-labeled $\mathcal{O}_{env}$-transition system.[2]

Bounded synthesis can be extended to the asynchronous setting by introducing a restriction on the local transition systems that requires that, whenever a process p is not scheduled, the local state of p does not change: we add the constraint

$$\forall t.\forall \upsilon \not\ni p.\ d_p(t) = d_p\big(\tau_\upsilon(t)\big)$$

for every processes $p \in P^-$. Likewise, we have to add a constraint that the transition of the process only depends on its current state and the input it sees. For simplicity, we depict the local transition function of a process p, τ^p, as a function from its state and the input it sees to its successor,

$$\forall t.\forall \upsilon \ni p.\ d_p(\tau_\upsilon(t)) = d_p\big(\tau^p; \overrightarrow{\upsilon}(t)\big),$$

where $\overrightarrow{\upsilon}(t)$ is the vector of assignments to input variables to p.

[1] The distinction between scheduler and environment in [26] is motivated by the use of branching-time logics. For linear-time logics (and universal specifications in general), the distinction is unnecessary as discussed in [26].

[2] If fairness constraints are added to the system, it might become necessary (or, at least, useful) to keeping track of the set of enabled processes. In this case, we would consider $2^{P \dot\cup V}$-labeled $\mathcal{O}_{env}$-transition systems.

The parameters and arguments from the previous section remain essentially unchanged modulo the increased number of environment outputs:

Theorem 9 *For a specification, represented as a universal co-Büchi automaton $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, and an architecture $A = (P, V, I, O)$, the inferred constraint system for asynchronous synthesis has size $O(|\delta| \cdot |V| + |O_{env}| \cdot |\mathcal{O}_{env}| + \sum_{p \in P^-} |I_p \setminus O_{env}|)$.*

For a bound b_A on the size of the input-preserving transition system $\mathcal{T}_A$ and a family $\{b_p \mid p \in P^-\}$ of bounds on the process transition systems $\{\mathcal{T}_p \mid p \in P^-\}$, the unraveled constraint system has size $O(b_A \cdot (|\delta| \cdot |V| + |O_{env}| \cdot |\mathcal{O}_{env}|) + \sum_{p \in P^-} {b_p}^2 \cdot |I_p \setminus O_{env}|))$. It is satisfiable if, and only if, the specification is bounded realizable in A for the bounds b_A and $\{b_p \mid p \in P^-\}$.

In our concurrency model, we allow an arbitrary number of processes to be scheduled, which results in a large number of environment outputs. In practice, it is usually appropriate to restrict $\mathcal{O}_{env}$ to a subset of these cases. If, e.g., we assume that there is always exactly one process scheduled, then the number of outputs merely increases by $|P^-|$.

Example As an example for the reduction of the asynchronous synthesis problem to SMT, we could consider the problem of finding an implementation of the arbiter for the specification $\psi = \Box(r_1 \rightarrow \Diamond g_1) \wedge \Box(r_2 \rightarrow \Diamond g_2) \wedge \Box\neg(g_1 \wedge g_2)$ of the simple arbiter from Sect. 2.3 an the monolithic architecture of Fig. 1d. However, such an arbiter would be unrealizable in an asynchronous setting without fairness (consider the case where the arbiter is never scheduled). We, therefore, use an adjusted specification with the fairness constraint that the arbiter is scheduled infinitely many times, $\psi_f = (\Box\Diamond a) \rightarrow \psi$. Figure 7 shows a universal co-Büchi automaton for ψ_f. Figure 8 shows the resulting constraint system.

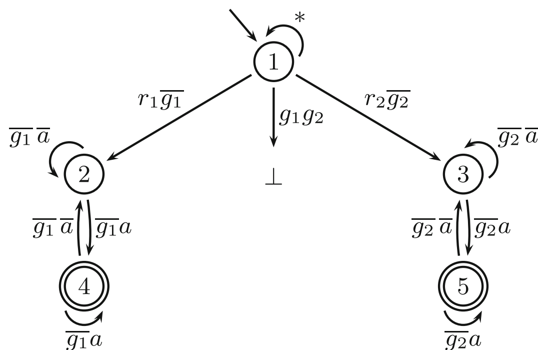


Fig. 7 Specification of an arbiter that is composed asynchronously with its environment. The specification includes a fairness requirement that excludes paths, on which the arbiter is scheduled only finitely many times, where a and $\overline{a}$ represent the situation where the arbiter was and was not scheduled, respectively

7 Additional design constraints

A key advantage of the constraint-based synthesis approach is that it is easy to add additional design constraints. In this section, we give several examples of such extensions.

A common situation in the synthesis of distributed systems is that a subset of the processes has a fixed implementation. For example, some processes may represent legacy components, or their implementation may be provided by someone else. Partially fixing an implementation can also be used as a manual step to simplify the synthesis problem, removing some of the non-determinism. This extension of the synthesis problem has been introduced as the concept of *partial designs* in [8], where the components that are known and fixed are called *white-box* components, while the processes with unknown implementations are called *black-box*.

A similar variation of the problem occurs when the processes are not be fixed, but restricted to come from some component library. As a final design constraint we consider the restriction to *symmetric* solutions. Systems that are built from identical processes are easier to build and maintain; symmetry is, therefore, often an important design goal, e.g., in VLSI designs. Additionally, the restriction to symmetric solutions may simplify the synthesis problem. When synthesizing an arbiter tree, e.g., the search will be much faster if we look for an implementation of a single component in the tree and then apply it to all components.

7.1 Partial designs

To extend the methods proposed in Sects. 4, 5, and 6 to the setting with white-box processes, we add constraints of the form

$$\forall t.\ d_p(t) = s \rightarrow d_p(\tau_v(t)) = s',$$

for all states s of the given implementation of a white-box process p, where s' is the v-successor of s. Additionally, we add constraints of the form $a(s)$ or $\neg a(s)$ that fix the value of each output variable $a \in O_p$ that belongs to the process; finally, the initial state of the white-box process is set to the global state 0 with the constraint

$$d_p(0) = 0.$$

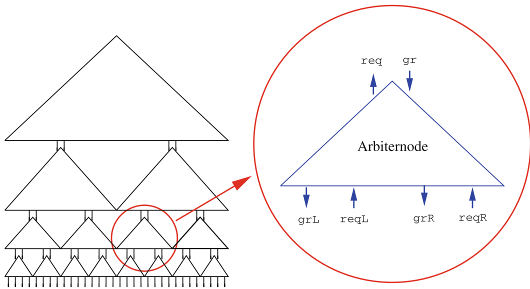
7.2 Symmetry constraints

When designing a system, we are often aware that several of its components are *similar*. A simple example for this is an

Fig. 8 Example of a constraint system for asynchronous synthesis. The figure shows the constraint system for the arbiter example (Fig. 7). Variables *env* and *a* are used to reflect whether or not the environment and the arbiter were scheduled, respectively

1. $\forall t. \forall env \in \upsilon \in \mathcal{O}_{env}.\ r_1 \in \upsilon \leftrightarrow r_1(\tau_\upsilon(t))$ and $r_2 \in \upsilon \leftrightarrow r_2(\tau_\upsilon(t))$
   $\forall t. \forall env \notin \upsilon \in \mathcal{O}_{env}.\ r_1(t) \leftrightarrow r_1(\tau_\upsilon(t))$ and $r_2(t) \leftrightarrow r_2(\tau_\upsilon(t))$
   $\forall t. \forall a \notin \upsilon \in \mathcal{O}_{env}.\ d_a(t) = d_a(\tau_\upsilon(t))$
   $\forall t. \forall a \in \upsilon \in \mathcal{O}_{env}.\ d_a(\tau_\upsilon(t)) = \tau^a(d_a(t), r_1(t), r_2(t))$
   $\forall t. \forall \upsilon \in \mathcal{O}_{env}.\ env \in \upsilon \leftrightarrow env(\tau_\upsilon(t))$ and $a \in \upsilon \leftrightarrow a(\tau_\upsilon(t))$
2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0) \wedge env(0) \wedge \neg a(0)$
3. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_1^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_1^{\#}(\tau_\upsilon(t)) \geq \lambda_1^{\#}(t)$
4. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_a(t)) \vee \neg g_2(d_a(t))$
5. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(d_a(t)) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_2^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_2^{\#}(\tau_\upsilon(t)) \geq \lambda_1^{\#}(t)$
6. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(d_a(t)) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_3^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_3^{\#}(\tau_\upsilon(t)) \geq \lambda_1^{\#}(t)$
7. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_a(t)) \wedge \neg a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_2^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_2^{\#}(\tau_\upsilon(t)) \geq \lambda_2^{\#}(t)$
8. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_a(t)) \wedge a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_4^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_4^{\#}(\tau_\upsilon(t)) > \lambda_2^{\#}(t)$
9. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_a(t)) \wedge \neg a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_3^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_3^{\#}(\tau_\upsilon(t)) \geq \lambda_3^{\#}(t)$
10. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_a(t)) \wedge a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_5^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_5^{\#}(\tau_\upsilon(t)) > \lambda_3^{\#}(t)$
11. $\forall t. \lambda_4^{\mathbb{B}}(t) \wedge \neg g_1(d_a(t)) \wedge \neg a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_2^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_2^{\#}(\tau_\upsilon(t)) \geq \lambda_4^{\#}(t)$
12. $\forall t. \lambda_4^{\mathbb{B}}(t) \wedge \neg g_1(d_a(t)) \wedge a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_4^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_4^{\#}(\tau_\upsilon(t)) > \lambda_4^{\#}(t)$
13. $\forall t. \lambda_5^{\mathbb{B}}(t) \wedge \neg g_2(d_a(t)) \wedge \neg a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_3^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_3^{\#}(\tau_\upsilon(t)) \geq \lambda_5^{\#}(t)$
14. $\forall t. \lambda_5^{\mathbb{B}}(t) \wedge \neg g_2(d_a(t)) \wedge a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_5^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_5^{\#}(\tau_\upsilon(t)) > \lambda_5^{\#}(t)$

arbiter tree: an arbiter for 2^n requests is composed of similar $2^n - 1$ arbiters components as shown in the illustration below.



Similarities between processes, as they appear in an arbiter tree, can be introduced as design constraints. We refer to such design constraints as *symmetry constraints*, as they impose symmetries between different components.

The requirements on the level of symmetry can vary: in the arbiter example, it would be reasonable to require that the processes start in the same initial state, follow the same transitions, and produce the same output in the same states. Beside this strong notion of symmetry, we can consider a weak version, where the transitions are expected to be identical, but the initial states and/or the output function may vary.

Weak symmetry is useful as a 'hint' to the synthesis algorithm that the solution is expected to exhibit some level of similarity, without restricting the search to symmetric solutions only.

To encode weak symmetry, we modify the constraints from Sect. 5 that describe a correct distribution as follows:

$$\forall t, u. \forall \upsilon, \upsilon' \subseteq O_{env} \text{ with } \overrightarrow{o}(t) \dot\cup \upsilon \sim_p \overrightarrow{o}(t) \dot\cup \upsilon'.$$
$$d_p(t) = d_p(u) \rightarrow d\big(\tau_\upsilon(t)\big) \quad = d\big(\tau_{\upsilon'}(u)\big),$$

where $\overrightarrow{o}(t)$ denotes the valuation of the processes' output variables $O \setminus O_{env}$ and $\sim_p$ is an abbreviation for 'results in indistinguishable inputs to p'.

To extend this to inter-component constraints, we can introduce a similar restriction $\sim_p^q$ that abbreviates 'the input left input looks to p like the input on the right to q', and impose the additional restrictions

$$\forall t, u. \forall \upsilon, \upsilon' \subseteq O_{env} \text{ with } \overrightarrow{o}(t) \dot\cup \upsilon \sim_p^q \overrightarrow{o}(t) \dot\cup \upsilon'.$$
$$d_p(t) = d_q(u) \rightarrow d\big(\tau_\upsilon(t)\big) = d\big(\tau_{\upsilon'}(u)\big).$$

When imposing the additional symmetry constraint that the processes share a single output function, we replace occurrences of $o_q\big(d_q(.)\big)$ by $o_p\big(d_q(.)\big)$, and to force the

9. (a) $\forall t.\ d_1(\tau_{r_1r_2}(t)) = d_1(\tau_{r_1\overline{r}_2}(t)) = \tau_1\big(d_1(t); \top, g_2(d_2(t))\big)$ $\wedge d_1(\tau_{\overline{r}_1r_2}(t)) = d_1(\tau_{\overline{r}_1\overline{r}_2}(t)) = \tau_1\big(d_1(t); \bot, g_2(d_2(t))\big)$
(b) $\forall t.\ d_2(\tau_{r_1r_2}(t)) = d_2(\tau_{\overline{r}_1r_2}(t)) = \tau_1\big(d_2(t); \top, g_1(d_1(t))\big)$ $\wedge\, d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{\overline{r}_1\overline{r}_2}(t)) = \tau_1\big(d_2(t); \bot, g_1(d_1(t))\big)$

Fig. 9 Example of a constraint system for distributed synthesis. The figure shows modifications and extensions to the constraint system from Fig. 6 for the arbiter example (Fig. 3) in order to implement the arbiter in the distributed architecture shown in Fig. 1b. The constraints 10 and 11 can be dropped

processes to start in the same initial state, we add the constraint

$$d_p(0) = d_q(0).$$

The extension to symmetry between subsets of the processes is straightforward. However, if there are multiple similar processes, it is typically more efficient to encode the local transition function explicitly instead of implicitly, i.e., to describe the transition function directly (cf. [9]), as opposed to the implicit restrictions discussed in the previous sections.

To encode the transition function τ_p, we introduce an $|I_p| + 1$-ary function symbol τ_p, where the first argument is the state and the remaining arguments are the boolean input variables presented in a predefined order. To simplify notation, we use a function input_p that takes the current values of the system output $O \smallsetminus O_{env}$ and the direction υ and orders their subset I_p in a predefined way (while dropping the rest). This leads to the constraint

$$\forall t.\ d_p\big(\tau_\upsilon(t)\big) = \tau_p\big(d_p(t); \mathsf{input}_p(\overrightarrow{o}(t), \upsilon)\big)$$

for every black-box process p.

If we use explicit instead of implicit constraints, the constraint system from Fig. 6 changes as shown in Fig. 9.

Imposing similarity on the structure of the transition system then reduces to replacing τ_q by τ_p. In the example, this leads to changing requirement 9b to

9. (b) $\forall t.\ d_2(\tau_{r_1r_2}(t)) = d_2(\tau_{\overline{r}_1r_2}(t)) = \tau_1\big(d_2(t); \top, g_1(d_1(t))\big) \wedge d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{\overline{r}_1\overline{r}_2}(t)) = \tau_1\big(d_2(t); \bot, g_1(d_1(t))\big)$.

Similarly, an equivalent output function can be imposed by replacing every occurrence of g_2 by g_1.

7.3 Component libraries

The design constraint that the implementations of certain processes must be chosen from a library can be added in a similar manner: the implementations available in the library can simply be modeled as a single white-box process with multiple initial states. Let init be the set of names for the initial states. We generalize the constraint that fixes the initial state in a partial design to the disjunction over this set:

$$\bigvee_{i \in \mathsf{init}} d_p(0) = i.$$

A particular form of synthesis from component libraries has been studied in [13], where it is assumed that the deterministic components are provided as a finite library, and we synthesize a *composer component* that decides on certain exit points in the computation which of the components should be active. Similar to the case of asynchronous synthesis discussed in Sect. 6, the composer component is unaware of the full history and decides based on the history of exit points.

The components in the library can be represented as Σ-labeled Υ-transition systems that have designated exit states in addition to the designated initial state. For simplicity, we use numbers $\mathbb{N}_n = \{0, 1, \ldots, n-1\}$ to identify the respective exit of each component. The components can then, as in the case of distributed synthesis discussed in Sect. 5, be modeled as a single white-box process with a designated set init of initial states and pairwise disjoint sets exit_i of exit states with numbers $i \in \mathbb{N}_n$, whose union is denoted by exit.

We introduce a mapping s that maps global states to the state of the joint transition system that indicates the active component and its current state and a mapping c that maps global states to the state of the composer component.

The constraints can then be formalized as follows:

1. $\forall u.\ \bigvee_{i \in \mathsf{init}} c(u) = i$, which requires that c always maps to the initial state of some component,
2. $s(0) = c(0)$, which requires that we start in an initial state, and
3. $\lambda^{\mathbb{B}}_{q_0}(0)$ to complete the initialization,
4. $\forall t.\ s(t) \notin \mathsf{exit} \rightarrow \bigwedge_{\upsilon \in \Upsilon} c(\tau_\upsilon(t)) = c(t)$,
5. $\forall t.\ s(t) \notin \mathsf{exit} \rightarrow \bigwedge_{\upsilon \in \Upsilon} s(\tau_\upsilon(t)) = \tau(s(t), \upsilon)$,
6. $\forall t \forall i \in \mathbb{N}_m.\ s(t) \in \mathsf{exit}_i \rightarrow \bigwedge_{\upsilon \in \Upsilon} c(\tau_\upsilon(t)) = \tau^c_i(c(t))$, and
7. $\forall t \forall i \in \mathbb{N}_m.\ s(t) \in \mathsf{exit}_i \rightarrow \bigwedge_{\upsilon \in \Upsilon} s(\tau_\upsilon(t)) = c(\tau^c_i(c(t)))$ for the dynamics of the system, where τ^c_i represents the ith direction of the transition of the composer, τ is the transition function from the joint transition system that represents all components in the library, and the τ_υ describe the global transition function.

Compliance with the specifying universal co-Büchi automaton can then be encoded as follows:

$$\forall t.\ \lambda^{\mathbb{B}}_q(t) \rightarrow \bigwedge_{(q',\upsilon) \in \delta(q, o(s(t)))} \lambda^{\mathbb{B}}_{q'}(\tau_\upsilon(t)) \wedge \lambda^{\#}_{q'}(\tau_\upsilon(t)) \rhd_{q'} \lambda^{\#}_q(t),$$

where $\rhd_{q'}$ again stands for $\rhd_{q'} \equiv >$ if $q' \in F$ and $\rhd_{q'} \equiv \geq$ otherwise.

8 Optimizations

We now discuss several optimizations to the bounded synthesis techniques described in the previous sections.

8.1 Edge acceptance conditions

The number of states in the universal automaton can be reduced by up to 50% by switching from the state acceptance condition to an edge acceptance condition.

A *universal co-Büchi tree automaton with edge acceptance* is a tuple $\mathcal{A} = (\Sigma, \Upsilon, Q, Q_0, E, F)$, where Σ denotes a finite set of labels, Υ denotes a finite set of directions, Q denotes a finite set of states, $Q_0 \subseteq Q$ denotes a designated set of initial states, $E \subseteq Q \times \Sigma \times (\{\bot\} \cup Q \times \Upsilon)$ denotes a set of transitions, and $F \subseteq T$ is a set of rejecting transitions.

In our setting, the automaton runs on Σ-labeled Υ-transition systems. The acceptance mechanism is again defined in terms of run graphs.

A *run graph* of an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, Q_0, E, F)$ on a Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is a minimal directed graph $\mathcal{G} = (G, D)$ with vertices G and edges D such that the following conditions are satisfied:

– The vertices $G \subseteq Q \times T$ form a subset of the product of Q and T.
– For all initial states $q \in Q_0$ of $\mathcal{A}$, the pair (q_0, t_0) of q and the initial state t_0 of $\mathcal{T}$, is a vertex of $\mathcal{G}$.
– For each vertex $(q, t) \in G$, $(q, o(t), \bot) \notin E$ is not a transition of $\mathcal{A}$.[3]
– For each vertex $g = (q, t) \in G$ of the run and each edge $\big(q, o(t), (q', \upsilon)\big) \in E$ of the automaton, $g' = \big(q', \tau(t, \upsilon)\big) \in G$ is a vertex of the run and $e = (g, g') \in D)$ is an edge of the run. We call this edge rejecting if $\big(q, o(t), (q', \upsilon)\big) \in F$ is rejecting.

A run graph is *accepting* if every infinite path $g_0 g_1 g_2 \ldots \in G^\omega$ contains only finitely many rejecting edges, and a transition system is accepted if it has an accepting run graph.

It is simple to translate universal co-Büchi automata with state acceptance into universal co-Büchi automata with edge acceptance and vice versa. To translate an automaton with state acceptance into an equivalent automaton with edge acceptance, we set $Q_0 = \{q_0\}$ and include a transition into the set of final transitions iff the target is a final state. To translate an automaton with edge acceptance into an automaton with state acceptance, we apply the following two steps:

[3] This constraint mimics the immediate reaction of an alternating automaton for $\delta(q, \sigma) = \bot$.

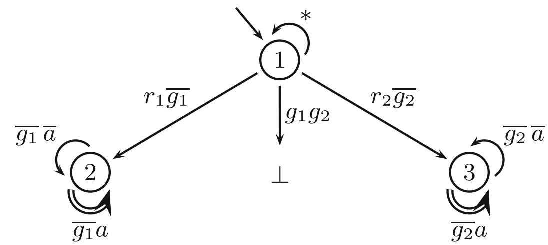


Fig. 10 Example of a universal co-Büchi automaton with edge acceptance. The automaton is equivalent to the arbiter specification as a co-Büchi automaton with state acceptance in Fig. 7. Rejecting edges are depicted with *double lines*

In the first step, we replace each state q by a final copy (q, f) and a non-final copy (q, n) of q state. The transition function is the same for both copies:

– $\delta\big((q, f), \sigma\big) = \delta\big((q, n), \sigma\big) = \mathit{false}$ if $(q, \sigma, \bot) \in T$ and
– $\delta\big((q, f), \sigma\big) = \delta\big((q, n), \sigma\big) = \bigwedge_{(q,\sigma,(q',\upsilon)) \in F} \big((q', f), \upsilon\big) \wedge \bigwedge_{(q,\sigma,(q',\upsilon)) \in T \smallsetminus F} \big((q', n), \upsilon\big)$ otherwise.

In the second step, we reduce the initial states to a singleton set, using a fresh initial state q_0 and adding, for all $q \in Q_0, \sigma \in \Sigma$, and $t \in \{\bot\} \cup (Q \times \Upsilon)$, a transition (q_0, σ, t) if $(q, \sigma, t) \in T$ is a transition. Since the new transitions will be traversed at most once, they do not need to considered for the set of rejecting transitions.

Example Figure 10 shows a universal co-Büchi automaton with edge acceptance that is equivalent to the automaton with state acceptance from Fig. 3.

8.2 Reduced annotations

We can obviously eliminate annotations corresponding to output variables if the variables do not occur in the specification. Beyond that, we can reduce the annotation of the transition system based on the internal structure of the universal co-Büchi automaton. Almost all rejecting edges in a run refer to a single strongly connected component in directed graph $(Q, \{(q, q') \mid \big(q, \sigma, (q, \upsilon)\big) \in T)$. We can, therefore, limit the annotation function $\lambda^\#$ corresponding to the edge to SCCs that actually contain the edge.

8.3 Input elimination

Analogously to the elimination of output variables, we can eliminate input variables if they are irrelevant for the specification. Eliminating input variables is particularly helpful, because it reduces the branching degree of the transition system.

Fig. 11 Example of a constraint system for the synthesis of an asynchronous arbiter. The constraint system is a simplified version of the system from Fig. 8. The constraints are obtained from the universal automaton with edge acceptance from Fig. 10. The annotation function $\lambda_1^\#$ and the variable *env*, which reflects the scheduling of the environment, have been eliminated

1. $\forall t.\forall \upsilon \in \mathcal{O}_{env}.\ r_1 \in \upsilon \leftrightarrow r_1(\tau_\upsilon(t))$ and $r_2 \in \upsilon \leftrightarrow r_2(\tau_\upsilon(t))$
   $\forall t.\forall a \notin \upsilon \in \mathcal{O}_{env}.\ d_a(t) = d_a(\tau_\upsilon(t))$
   $\forall t.\forall a \in \upsilon \in \mathcal{O}_{env}.\ d_a(\tau_\upsilon(t)) = \tau^a(d_a(t), r_1(t), r_2(t))$
   $\forall t.\forall \upsilon \in \mathcal{O}_{env}.\ a \in \upsilon \leftrightarrow a(\tau_\upsilon(t))$
2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0) \wedge \neg a(0)$
3. $\forall t.\ \lambda_1^{\mathbb{B}}(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_1^{\mathbb{B}}(\tau_\upsilon(t))$
4. $\forall t.\ \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_a(t)) \vee \neg g_2(d_a(t))$
5. $\forall t.\ \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(d_a(t)) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_2^{\mathbb{B}}(\tau_\upsilon(t))$
6. $\forall t.\ \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(d_a(t)) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_3^{\mathbb{B}}(\tau_\upsilon(t))$
7. $\forall t.\ \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_a(t)) \wedge \neg a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_2^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_2^\#(\tau_\upsilon(t)) \geq \lambda_2^\#(t)$
8. $\forall t.\ \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_a(t)) \wedge a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_2^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_2^\#(\tau_\upsilon(t)) > \lambda_2^\#(t)$
9. $\forall t.\ \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_a(t)) \wedge \neg a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_3^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_3^\#(\tau_\upsilon(t)) \geq \lambda_3^\#(t)$
10. $\forall t.\ \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_a(t)) \wedge a(t) \rightarrow \bigwedge_{\upsilon \in \mathcal{O}_{env}} \lambda_3^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_3^\#(\tau_\upsilon(t)) > \lambda_3^\#(t)$

In asynchronous systems, it is often possible to eliminate the variables that reflect whether or not a process was scheduled. If the specification does not refer to the scheduling of a system process (in particular, if there is no fairness constraint), then we can assume that the process is never scheduled, without affecting the realizability of the specification. Likewise, if the specification does not refer to the scheduling of the environment, we can assume that the environment is always scheduled.

Example The arbiter specification given as the universal automaton in Fig. 10 leads to the simplified constraint system shown in Fig. 11. The annotation function $\lambda_1^\#$ and the variable *env*, which reflects the scheduling of the environment, have been eliminated.

8.4 Semantic variations

In our semantic model, we view the interaction of an environment with the system as a sequence of environment and system outputs. There are two natural ways to couple these outputs to pairs: the system output could be coupled to the *previous* or to the *next* environment output.

While neither of these choices is more natural than the other, the choice can have a significant impact on the size of the transition system, and, hence, on the effort required for synthesis. Our definition in Sect. 2 couples the system output to the previous environment output. In input-preserving transition systems, the previous output is stored in the state. If, instead, we couple the system output to the subsequent environment output, we can map different environment decisions to the same successor states. The resulting models can, therefore, be much more concise: preserving n bits of input requires 2^n different successor states.

9 Experimental results

Our experimental results are based on the SMT solver Yices [4] version 1.0.9 on a 2.6 Ghz Opteron system. Among other theories, Yices efficiently solves constraints with uninterpreted function symbols. Yices has only incomplete support for quantifiers, and was in fact unable to determine the satisfiability of our quantified formulas. We, therefore, eliminated the quantifiers in a preprocessing step, in which universal quantifiers are replaced by explicit conjunctions.

9.1 Single-process arbiter

The first benchmark is the single-process arbiter introduced in Sect. 2.3. The arbiter can be implemented as a transition system with eight states, as illustrated in Fig. 2. Table 1 shows the time and memory consumption of Yices solving the constraints from Fig. 5 with the quantifiers unraveled for different upper bounds on the number of states. A correct implementation with eight states is found in 8 s.

Figure 12 shows a universal co-Büchi automaton with edge acceptance for an extended version of the arbiter specification. We now require that, once a grant is given, it is not retracted until there is no longer a request. The extended specification allows for implementations with a minimum of only five states.

Table 1 Experimental results from the synthesis of a single-process arbiter using the specification from Fig. 3

Bound	4	5	6	7	8	9
Result	Unsatisfiable	Unsatisfiable	Unsatisfiable	Unsatisfiable	Satisfiable	Satisfiable
# Decisions	3,957	13,329	23,881	68,628	72,655	72,655
# Conflicts	209	724	1,998	15,859	4,478	4,478
# Boolean variables	1,011	2,486	4,169	9,904	5,214	5,214
Memory (MB)	16.9102	18.1133	20.168	27.4141	26.4375	26.4414
Time (s)	0.05	0.28	1.53	35.99	7.53	7.31

The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Fig. 5, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system

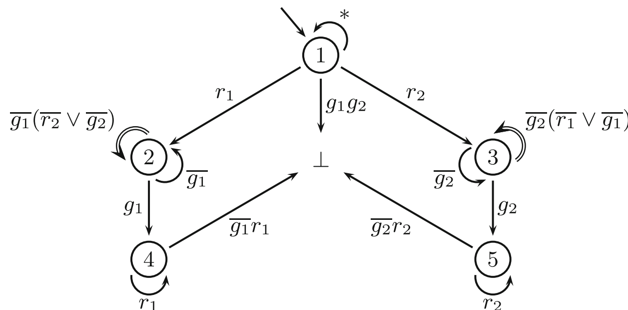


Fig. 12 Extended specification of an arbiter, represented as a universal co-Büchi automaton with edge acceptance

As shown in Table 2, Yices determines the satisfiability of the constraint system much faster (less than 1 s), even though the universal automaton has more states.

9.2 Distributed arbiter

For the synthesis of a distributed arbiter with two processes, the quantifiers in the constraint system from Fig. 9 were unraveled for different bounds on the size of the global transition system, and also with additional bounds on the size of the processes. As shown in Table 3, a correct solution with eight global states is found in 78 s if the number of process states is left unconstrained; restricting the process states explicitly to 2 (the additional bound is shown in parentheses) leads to an slows down the synthesis by a factor of two (138 s).

9.3 Symmetric distributed arbiter

Table 4 shows experimental data from the synthesis of a weakly symmetric distributed arbiter with two processes. A correct solution with eight global states is found by Yices in 71 s.

The performance for the extended arbiter specification from Fig. 12 is shown in Table 5. Yices needs only half a minute to construct a correct distributed implementation. The table also shows that borderline cases like the unsuccessful search for an implementation with eight states, but only two local states, can become very expensive; in the example, Yices needed more than 1.5 h to determine unsatisfiability, while correct implementations with eight states and three or four local states are found in approximately 30 s.

9.4 Asynchronous arbiter

Table 6 shows experimental results for the synthesis of an asynchronous arbiter based on the specification from Fig. 13, using optimization techniques from Sect. 8. A correct implementation is found in about 8.5 h.

9.5 Dining philosophers

Table 7 shows the time and memory consumption for synthesizing a strategy for the dining philosophers to satisfy the specification shown in Fig. 14. In the dining philosophers

Table 2 Experimental results from the synthesis of a single-process arbiter using the specification from Fig. 12

Bound	4	5	6	7	8
Result	Unsatisfiable	Satisfiable	Satisfiable	Satisfiable	Satisfiable
# Decisions	17,566	30,011	52,140	123,932	161,570
# Conflicts	458	800	1,375	2,614	3,987
# Boolean variables	1,850	2,854	3,734	5,406	6,319
Memory (MB)	18.3008	20.0586	22.5781	27.5000	35.7148
Time (s)	0.21	0.63	1.72	5.15	12.38

The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system

Table 3 Experimental results from the synthesis of a two-process arbiter using the specification from Fig. 3 and the architecture from Fig. 1b

Bound	4	5	6	7
Result	Unsatisfiable	Unsatisfiable	Unsatisfiable	Unsatisfiable
# Decisions	4,994	22,302	40,567	300,641
# Conflicts	253	1,206	5,376	122,863
# Boolean variables	1,393	3,274	5,793	11,427
Memory (MB)	17.6289	19.9336	22.9102	40.543
Time (s)	0.08	0.72	6.58	359.84
Bound	8	9	8 (1)	8 (2)
Result	Satisfiable	Satisfiable	Unsatisfiable	Satisfiable
# Decisions	254,465	365,158	171,117	157,829
# Conflicts	34,565	47,739	84,016	64,139
# Boolean variables	11,414	9,055	12,403	6,733
Memory (MB)	42.957	61.8477	32.1641	37.3242
Time (s)	77.95	213.6	114.88	137.89

The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Fig. 6, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in parentheses)

Table 4 Experimental results from the synthesis of a weakly symmetric two-process arbiter using the specification from Fig. 3 and the architecture from Fig. 1b

Bound	4	5	6	7
Result	Unsatisfiable	Unsatisfiable	Unsatisfiable	Unsatisfiable
# Decisions	6,041	15,008	35,977	89,766
# Conflicts	236	929	2,954	30,454
# Boolean variables	1,269	2,944	5,793	9,194
Memory (MB)	17.0469	18.4766	22.1992	33.1211
Time (s)	0.06	0.35	3.3	120.56
Bound	8	9	8 (1)	8 (2)
Result	Satisfiable	Satisfiable	Unsatisfiable	Satisfiable
# Decisions	197,150	154,315	178,350	71,074
# Conflicts	33,496	24,607	96,961	18,263
# Boolean variables	7,766	8,533	12,403	6,382
Memory (MB)	37.4297	36.2734	39.4922	29.1992
Time (s)	70.97	58.43	200.07	36.38

The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Fig. 6, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in *parentheses*)

benchmark, the size of the specification grows linearly with the number of philosophers; for 10,000 philosophers this results in systems of hundreds of thousands constraints. In spite of the large size of the resulting constraint system, the synthesis problem remains tractable; Yices solves all resulting constraint systems within a few hours, and within a minutes for small constraint systems with up to 1,000 philosophers.

10 Conclusions

Despite its obvious advantages, synthesis has been less popular than verification. While the complexity of verification is determined by the size of the implementation under analysis, standard synthesis algorithms [8,11,21,26,30] suffer from the daunting complexity determined by the theoretical upper bound on the smallest implementation, which, as

Table 5 Experimental results from the synthesis of a weakly symmetric two-process arbiter using the specification from Fig. 12 and the architecture from Fig. 1b

Bound	4	5	6	7	8	9
Result	Unsat	Unsat	Unsat	Unsat	Sat	Sat
# Decisions	16,725	47,600	91,480	216,129	204,062	344,244
# Conflicts	326	1,422	8,310	61,010	11,478	16,347
# Boolean variables	1,890	7,788	5,793	13,028	8,330	10,665
Memory (MB)	18.0273	22.2109	28.5312	43.8594	42.2344	61.9727
Time (s)	0.16	1.72	14.84	208.78	32.47	72.97
Bound	8 (1)	8 (2)	8 (3)	8 (4)		
Result	Unsat	Unsat	Sat	Sat		
# Decisions	309,700	1,122,755	167,397	208,255		
# Conflicts	92,712	775,573	13,086	13,153		
# Boolean variables	15,395	25,340	8,240	7,806		
Memory (MB)	54.1641	120.0160	42.1484	42.7188		
Time (s)	263.44	5537.68	31.12	30.36		

The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in *parentheses*)

Table 6 Experimental results from the synthesis of an asynchronous arbiter

Bound	6	7	8	9	10	11–15	16
Result	Unsatisfiable	Unsatisfiable	Unsatisfiable	Unsatisfiable	Unsatisfiable	No result	Satisfiable
# Decisions	51,865	58,976	219,396	736,917	3,387,162	–	6,543,861
# Conflicts	2,027	3,093	23,338	282,704	2,240,601	–	3,704,294
# Boolean variables	3,671	4,810	9,755	40,693	66,887	–	51,456
Memory (MB)	20.3164	21.1328	30.8516	117.016	230.406	–	269.246
Time (s)	0.91	1.64	59.63	5,217.76	68,656.6	–	28,186.6

The table shows the time and memory consumption of Yices 1.0.9 for the specification shown in Fig. 11. For 11–15 states, no result was obtained within 24 h

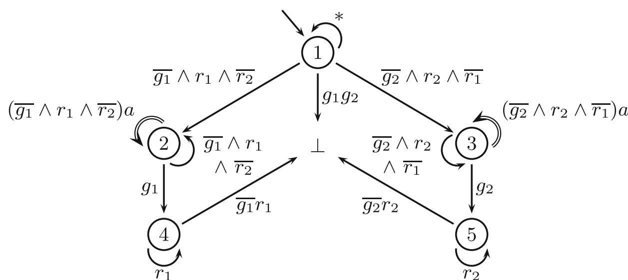


Fig. 13 Extended specification of an asynchronous arbiter, represented as a universal co-Büchi automaton with edge acceptance

shown by Rosner [22], increases by an extra exponent with each additional process for pipeline architecture in a synchronous setting. For general architectures, the situation is even worse: the realizability problem is undecidable for most distributed architectures in the synchronous setting [8,22] and all distributed architectures in the asynchronous setting [26], simply because there is no upper bound on the size of a minimal distributed implementation.

A bound on the size, however, is a practical design constraint, because, in reality, the size of an implementable program is limited. Also, the size of an implementation is a measure of its simplicity and quality. Bounded synthesis makes this design constraint explicit, by emphasizing the complexity of the synthesized implementation (the output) rather than the size of the specification (the input). The effect of moving from input to output complexity is astonishing: the problem is transformed from a technically very difficult and mostly undecidable problem, which is non-elementary even in the decidable fragment [8,22], into a problem in the tame class NP.

Once the complexity is measured in the output, the playing field between verification and synthesis, thus, appears much more leveled and, indeed, our experimental results give hope that many synthesis problems that were previously considered intractable can in fact be solved efficiently.

Table 7 Experimental results from the synthesis of a strategy for the dining philosophers using the specification from Fig. 14

# phil.	3 states			4 states			6 states		
	Time (s)	Memory (MB)	Result	Time (s)	Memory (MB)	Result	Time (s)	Memory (MB)	Result
125	1.52	23.2695	Unsat	23.84	36.2305	Unsat	236.5	87.7852	Sat
250	5.41	29.2695	Unsat	130.07	52.0859	Sat	141.36	91.1328	Sat
375	22.81	38.9727	Unsat	128.83	58.1992	Unsat	890.58	154.355	Sat
500	17.98	39.9297	Unsat	15.84	52.9336	Sat	237.04	119.309	Sat
625	35.57	49.5586	Unsat	417.05	94.7188	Unsat	486.5	130.977	Sat
750	22.25	52.3359	Unsat	20.85	69.1562	Sat	82.63	99.707	Sat
875	51.98	56.0859	Unsat	628.84	119.363	Unsat	2,546.88	255.965	Sat
1,000	168.17	70.3906	Unsat	734.74	117.703	Sat	46.18	124.691	Sat
1,125	67.14	70.1133	Unsat	1,555.18	165.922	Unsat	1,854.77	246.848	Sat
1,250	165.59	76.2227	Unsat	122.8	107.645	Sat	596.8	203.012	Sat
1,375	104.27	75.4531	Unsat	3,518.85	191.113	Unsat	8,486.18	490.566	Sat
1,500	187.25	82.8867	Unsat	85.52	129.215	Sat	232.81	214.68	Sat
1,625	85.83	88.8047	Unsat	2,651.82	246.734	Unsat	1,437.45	281.203	Sat
1,750	169.93	97.543	Unsat	107.14	126.477	Sat	257.77	185.887	Sat
1,875	174.03	105.25	Unsat	3,629.18	234.527	Unsat	4,641.03	405.781	Sat
2,000	25.86	102.125	Unsat	242.55	157.734	Sat	811.78	269.375	Sat
2,125	163.39	113.27	Unsat	5,932.24	315.711	Unsat	6,465.75	424.121	Sat
2,250	412.37	115.438	Unsat	523.87	162.391	Sat	5,034.83	456.316	Sat
2,375	201.95	120.047	Unsat	7,311.03	313.168	Unsat	4,887.76	451.332	Sat
2,500	375.29	135.535	Unsat	235.17	202.59	Sat	319.78	253.781	Sat
2,625	544.03	135.379	Unsat	6,560.53	312.355	Unsat	23,990.5	808.633	Sat
2,750	559.35	139.137	Unsat	817.41	226.082	Sat	632.28	349.992	Sat
2,875	308.36	151.727	Unsat	7,273.89	299.016	Unsat	8,638.96	551.5	Sat
3,000	666.18	155.57	Unsat	533.23	228.961	Sat	3,158.26	493.617	Sat
3,125	235.52	141.93	Unsat	12,596.6	377.328	Unsat	10,819.7	693.133	Sat
3,250	869.53	153.633	Unsat	2,089.72	308.719	Sat	21,298.8	889.285	Sat
3,375	260.88	145.918	Unsat	11,581.7	379.949	Unsat	21,560	741.09	Sat
3,500	308.23	169.348	Unsat	897.6	270.676	Sat	829.52	398.008	Sat
5,000	982.68	240.273	Unsat	3,603.7	421.832	Sat	1,357.48	582.457	Sat
7,000	2,351.87	313.277	Unsat	7,069.55	535.98	Sat	6,438.73	1,081.68	Sat
10,000	4,338.83	448.648	Unsat	4,224.28	761.008	Sat	10,504.6	1121.58	Sat

The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system

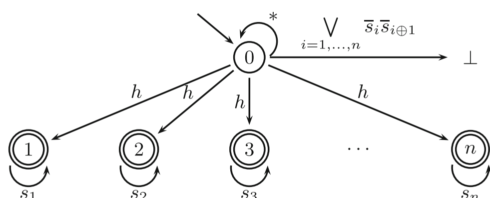


Fig. 14 Specification of a dining philosopher problem with n philosophers. The environment can cause the philosophers to become hungry (by setting h to true). The states depicted as *double circles* (1 through n) are the rejecting states in F; state i refers to the situation where philosopher i is hungry and starving (s_i). A fail state is reached when two adjacent philosophers try to reach for their common chopstick; the fail state refers to the resulting eternal philosophical quarrel that keeps the affected philosophers from eating

References

1. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 118–149 (2003)
2. Copty, F., Fix, L., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.: Benefits of bounded model checking at an industrial setting. In: Proceedings of 13th International Conference on Computer Aided Verification (CAV 2001), 18–22 July, Paris, France. Lecture Notes in Computer Science, pp. 436–453. Springer, Berlin (2001)
3. Castellani, I., Mukund, M., Thiagarajan, P.S.: Synthesizing distributed transition systems from global specification. In: Proceedings of the 19th Conferenceon Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1999), 13–15 December, Chennai, India. Lecture Notes in Computer Science, vol. 1738, pp. 219–231. Springer, Berlin (1999)

4. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for dpll(t). In: Ball, T., Jones, R.B. (eds.) CAV. Lecture Notes in Computer Science, vol. 4144, pp. 81–94. Springer, Berlin (2006)
5. Ehlers, R.: Symbolic bounded synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) In: 22nd International Conference on Computer Aided Verification. LNCS, vol. 6174, pp. 365–379. Springer, Berlin (2010)
6. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 6605, pp. 272–275. Springer, Berlin (2011)
7. Filiot, E., Jin, N., Raskin, J.-F.: An antichain algorithm for LTL realizability. In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009), June 26–July 2, Grenoble, France, Lecture Notes in Computer Science, vol. 5643, pp. 263–277. Springer-Verlag (2009)
8. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005), 26–29 June, Chicago, IL, USA, pp. 321–330. IEEE Computer Society Press, Los Alamitos (2005)
9. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Proceedings of the 2nd Workshop on Automated Formal Methods (AFM 2007), 6 November, Atlanta, Georgia, USA, pp. 69–76. ACM Press, New York (2007)
10. Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for the satisfiability (SAT) problem: a survey. In: Du, D.-Z., Gu, J., Pardalos, P. (eds.) Satisfiability Problem: Theory and applications. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 19–152. American Mathematical Society, Washington (1997)
11. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), 16–19 June, Boston, MA, USA, pp. 389–398. IEEE Computer Society Press, Los Alamitos (2001)
12. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23–25 October, Pittsburgh, PA, USA, pp. 531–540 (2005)
13. Lustig, Y., Vardi, M. Y.: Synthesis from component libraries. In: Proceedings of the Twelfth International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2009), 22–29 March, York, England, UK. Lecture Notes in Computer Science, vol. 5504, pp. 167–181. Springer, Berlin (2009)
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), 18–22 June, Las Vegas, Nevada, USA, pp. 530–535. ACM Press, New York (2001)
15. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra. Theor. Comput. Sci. **141**(1-2), 69–107 (1995)
16. Madhusudan, P., Thiagarajan, P.S.: Distributed controller synthesis for local specifications. In: Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001), 8–12 July, Crete, Greece. Lecture Notes in Computer Science, pp. 396–407. Springer, Berlin (2001)
17. Madhusudan, P., Thiagarajan, P.S.: A decidable class of asynchronous distributed controllers. In: Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002), 20–23 August, Brno, Czech Republic. Lecture Notes in Computer Science, vol. 2421, pp. 145–160. Springer, Berlin (2002)
18. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. J. Log. Methods Comput. Sci. **3**(3:5), 1–21 (2007)
19. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS, pp. 46–57. IEEE Computer Society Press, Los Alamitos (1977)
20. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP 1998), 11–15 July, Stresa, Italy. Lecture Notes in Computer Science, vol. 372, pp. 652–671. Springer, Berlin (1989)
21. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS 1990), 22–24 October, St. Louis, Missouri, USA, pp. 746–757. IEEE Computer Society Press, Los Alamitos (1990)
22. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science, Rehovot, Israel (1992)
23. Safra, S.: On the complexity of the ω-automata. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS 1988), 24–26 October, White Plains, New York, USA, pp. 319–327. IEEE Computer Society Press, Los Alamitos (1988)
24. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: Proceedings of the Twelfth International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2009), 22–29 March, York, England, UK. Lecture Notes in Computer Science, vol. 5504, pp. 167–181. Springer, Berlin (2009)
25. Schewe, S.: Software synthesis is hard and simple. In: Bodik, R., Kupferman, O., Smith, D.R. Yahav, E. (eds.) Software Synthesis. number 09501 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Germany (2010)
26. Schewe, S., Finkbeiner, B.: Synthesis of asynchronous systems. In: Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2006), 12–14 July, Venice, Italy. Lecture Notes in Computer Science, vol. 4407, pp. 127–142. Springer, Berlin (2006)
27. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 22–25 October, Tokyo, Japan, Lecture Notes in Computer Science, vol. 4762, pp. 474–488. Springer, Berlin (2007)
28. Schewe, S., Finkbeiner, B.: Distributed synthesis for alternating-time logics. In: Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 22–25 October, Tokyo, Japan. Lecture Notes in Computer Science, vol. 4762, pp. 268–283. Springer, Berlin (2007)
29. Vardi, M.Y.: An automata-theoretic approach to fair realizability and synthesis. In: Proceedings of the 7th International Conference on Computer Aided Verification (CAV 1995), 3–5 July, Liege, Belgium. Lecture Notes in Computer Science, vol. 939, pp. 267–278. Springer, Berlin (1995)
30. Walukiewicz, I., Mohalik, S.: Distributed games. In: Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2003), 15–17 December, Bombay, Mumbai, India. Lecture Notes in Computer Science, vol. 2914, pp. 338–351. Springer, Berlin (2003)