Journal of Computer and System Sciences 78 (2012) 911–938

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Synthesis of Reactive(1) designs ☆,☆☆

Roderick Bloem [a], Barbara Jobstmann [b], Nir Piterman [c,*], Amir Pnueli [d], Yaniv Sa'ar [d]

[a] Graz University of Technology, Austria
[b] CNRS/Verimag, France
[c] University of Leicester, UK
[d] Weizmann Institute of Science, Israel

ARTICLE INFO

Article history:
Received 21 April 2010
Received in revised form 25 May 2011
Accepted 5 August 2011
Available online 18 August 2011

Keywords:
Property synthesis
Realizability
Game theory

ABSTRACT

We address the problem of automatically synthesizing digital designs from linear-time specifications. We consider various classes of specifications that can be synthesized with effort quadratic in the number of states of the reactive system, where we measure effort in symbolic steps. The synthesis algorithm is based on a novel type of game called General Reactivity of rank 1 (GR(1)), with a winning condition of the form

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \rightarrow (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n),$$

where each p_i and q_i is a Boolean combination of atomic propositions. We show symbolic algorithms to solve this game, to build a winning strategy and several ways to optimize the winning strategy and to extract a system from it. We also show how to use GR(1) games to solve the synthesis of LTL specifications in many interesting cases. As empirical evidence to the generality and efficiency of our approach we include a significant case study. We describe the formal specifications and the synthesis process applied to a bus arbiter, which is a realistic industrial hardware specification of modest size.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

One of the most ambitious and challenging problems in computer science is the automatic synthesis of programs and (digital) designs from logical specifications. A solution to this problem would lift programming from the current level, which is mostly imperative, to a declarative, logical style. There is some evidence that this level is preferable, in particular when concurrency plays an important role.

The synthesis problem was first identified by Church [5]. Several methods have been proposed for its solution [6,7]. The two prevalent approaches to solving the synthesis problem were by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-person game. In these preliminary studies of the problem, the logical specification that the synthesized system should satisfy was given as an S1S formula and the complexity of synthesis is non-elementary.

The problem was considered again in [8] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). This followed two previous attempts [9,10] to synthesize programs from temporal specifications, which reduced the synthesis problem to satisfiability, ignoring the fact that the environment should be treated as an adversary. The method proposed in [8] for a given LTL specification φ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which

☆ This work was supported by the European Commission under contracts 507219 (PROSYD), 217069 (COCONUT), and 248613 (DIAMOND).
☆☆ This paper is based on the following papers: Piterman et al. (2006) [1] and Bloem et al. (2007) [2,3].
* Corresponding author.
E-mail address: nir.piterman@le.ac.uk (N. Piterman).

0022-0000/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2011.08.007

is then determinized into a deterministic Rabin automaton. This double translation necessarily causes a doubly exponential time complexity [11].

The high complexity established in [8,11] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where the synthesis problem can be solved in polynomial time, by using simpler automata or partial fragments of LTL [12–15]. Representative cases are the work in [16] which presents an efficient quadratic solution to games (and hence synthesis problems) where the acceptance condition is one of the LTL formulas $\Box p$, $\Diamond q$, $\Box\Diamond p$, or $\Diamond\Box q$. The work in [13] presents efficient synthesis approaches for various LTL fragments.

This paper can be viewed as a generalization of the results of [16] and [13] into the wider class of *Generalized Reactivity*(1) formulas (GR(1)), i.e., formulas of the form

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \rightarrow (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n). \tag{1}$$

Here, we assume that the specification is an implication between a set of *assumptions* and a set of *guarantees*.[1] Following the results of [18], we show how any synthesis problem whose specification is a GR(1) formula can be solved with effort $O(mnN^2)$, where N is the size of the state space of the design and effort is measured in symbolic steps, i.e., in the number of preimage computations [19]. Furthermore, we present a symbolic algorithm for extracting a design (program) which implements the specification.

We show that GR(1) formulas can be used to represent a relatively wide set of specifications. First, we show that we can include past LTL formulas in both the assumptions and the guarantees. Second, we show that each of the assumptions and guarantees can be a deterministic "Just Discrete System" (Büchi automaton). Thus, our method does not incur the exponential blow-ups incurred in LTL synthesis for the translation of the formula to an automaton and for the determinization of the automaton because the user provides the specification as a set of deterministic automata. (But note that the state space of the system is the product of the sizes of the automata, which may cause an exponential blowup.) Furthermore, a symbolic implementation of our algorithm is easily obtained when the automata are represented in symbolic form. One drawback is that our formalism is less expressive than LTL. In particular, Reactivity (Streett) conditions can not be expressed.

The reader may suspect that GR(1) specifications place an undue burden on the user or that the expressivity is too limited. We argue that this is not the case. Intuitively, many specifications can naturally be split into assumptions on the environment and guarantees on the system. (Cf. [20].) Often, assumptions and guarantees can naturally be written as conjunctions of simple properties that are easily expressed as deterministic automata. We substantiate this view by presenting two case studies of small but realistic industrial modules. We show that the specifications for these modules can be expressed in GR(1), that their specifications are compact and easy to read, and that they can be synthesized relatively efficiently.

The first case study concerns a generalized buffer from IBM, a tutorial design for which a good specification is available. The second concerns the arbiter for one of the AMBA buses [21], a characteristic industrial design that is not too big. This is the first time realistic industrial examples have been tackled; previous work has only considered toy examples such as a simple mutual exclusion protocol, an elevator controller, or a traffic light controller [14,1,22].

Our work stresses the compositionality of synthesis from LTL specifications and the structure of specifications as a guide to efficient synthesis. At the same time, it emphasizes the symbolic analysis of the state space through the usage of BDDs. Sohail et al. removed some of the restrictions on the expressive power imposed by our work [23,24]. They present a compositional approach in which each property is translated to a Büchi or parity automaton and the resulting generalized parity game is solved symbolically. They also show how in some cases to circumvent the construction of deterministic automata based on [25]. Morgenstern and Schneider present a similar approach. They construct an automaton that is minimal in the automata hierarchy for each of the properties in the specification [26].

In recent years significant theoretical progress has been made in approaches that emphasize the treatment of full LTL. One key result is that LTL realizability and synthesis can be reduced to games that are easier to solve than Rabin or parity games, when bounding the size of the resulting system. In [27], Kupferman and Vardi show a reduction to Büchi games that avoids the determinization procedure by going through universal co-Büchi automata. Their approach is further extended to work compositionally for specifications that are a conjunction of properties [28]. The algorithm of [27] was implemented directly in [22]. Schewe and Finkbeiner extend the reduction to co-Büchi winning conditions introduced in [27] to a reduction to Safety games [29]. They show how to use these insights to solve distributed synthesis, where the size of components is bounded. Filiot, Jin, and Raskin [30] give the same reduction to safety games and implement this approach using antichains to efficiently encode sets of states [30]. To date, these approaches are still very restricted in the scale of systems they can handle.

The paper is structured as follows. We start with presenting the notation and recalling known results (Section 2). Then, we show how to solve Generalized Reactive(1) games symbolically (Section 3), compute a winning strategy, and extract a correct program, if it exists (Section 4). In Section 5, we show how the techniques developed in Sections 3 and 4 are used to synthesize systems from temporal specifications. In Section 6, we describe the AMBA AHB arbiter case study. We give its formal specification, and show the results of synthesizing it. Finally, we discuss lessons learned and conclude in Section 7.

[1] The source of the name *reactivity* and the rank follow from the definitions of the temporal hierarchy in [17].

2. Preliminaries

2.1. Linear temporal logic

We assume a countable set of Boolean variables (propositions) $\mathcal{V}$. Without loss of generality, we assume that all variables are Boolean. The general case in which a variable ranges over arbitrary finite domains can be reduced to the Boolean case. LTL formulas are constructed as follows:

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

A model σ for a formula φ is an infinite sequence of truth assignments to propositions. Namely, if $\hat{P}$ is the set of propositions appearing in φ, then for every finite set P such that $\hat{P} \subseteq P$, a word in $(2^P)^\omega$ is a model. Given a model $\sigma = \sigma_0, \sigma_1, \ldots$, we denote by σ_i the set of propositions at position i. For a formula φ and a position $i \geqslant 0$, we say that φ *holds at position i of* σ, written $\sigma, i \models \varphi$, and define it inductively as follows:

- For $p \in P$ we have $\sigma, i \models p$ iff $p \in \sigma_i$,
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$,
- $\sigma, i \models \varphi \vee \psi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \psi$,
- $\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$,
- $\sigma, i \models \varphi\,\mathcal{U}\,\psi$ iff there exists $k \geqslant i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $i \leqslant j < k$,
- $\sigma, i \models \ominus\varphi$ iff $i > 0$ and $\sigma, i-1 \models \varphi$,
- $\sigma, i \models \varphi\,\mathcal{S}\,\psi$ iff there exists k, $0 \leqslant k \leqslant i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $k < j \leqslant i$.

If $\sigma, 0 \models \varphi$, then we say that φ *holds* on σ and denote it by $\sigma \models \varphi$. A set of models M satisfies φ, denoted $M \models \varphi$, if every model in M satisfies φ.

We use the usual abbreviations of the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ and the usual definitions for true and false. We use the temporal abbreviations $\Diamond$ (eventually), $\Box$ (globally), $\mathcal{W}$ (weakuntil), and for the past fragment $\boxminus$ (historically), $\diamondminus$ (once), and $\mathcal{B}$ (backto) which are defined as follows:

- $\Diamond\varphi = \mathsf{true}\,\mathcal{U}\,\varphi$,
- $\Box\psi = \neg\Diamond\neg\psi$,
- $\varphi\,\mathcal{W}\,\psi = (\varphi\,\mathcal{U}\,\psi) \vee \Box\varphi$,
- $\diamondminus\varphi = \mathsf{true}\,\mathcal{S}\,\varphi$,
- $\boxminus\psi = \neg\diamondminus\neg\varphi$, and
- $\varphi\,\mathcal{B}\,\psi = (\varphi\,\mathcal{S}\,\psi) \vee \boxminus\varphi$.

The following abbreviations are used in Section 6. They are inspired by the ones designed in PSL [31]. Given an atomic proposition p and two LTL formulas φ and ψ, we define

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

That is $\mathtt{raise}(p)$ indicates the raising edge of signal p, $\mathtt{fall}(p)$ indicates the falling edges of signal p, and the nested weak until $\varphi\,\mathcal{W}\,[i]\psi$ indicates that φ waits for ψ to hold i times or forever.

We distinguish between *safety* and *liveness* properties. An LTL-definable property φ is a *safety* property if for every model σ that violates φ, i.e., $\sigma \not\models \varphi$, there exists an i such that for every σ' that agrees with σ up to position i, i.e., $\forall 0 \leqslant j \leqslant i$, $\sigma'_i = \sigma_i$, σ' also violates φ. An LTL-definable property φ is a *liveness* property if for every prefix of a model $\sigma_0, \ldots, \sigma_i$ there exists an infinite model σ that starts with $\sigma_0, \ldots, \sigma_i$ and $\sigma \models \varphi$. Intuitively, safety properties specify bad things that should never happen and liveness properties specify good things that should occur. We distinguish between properties that are (i) safety, (ii) liveness, or (iii) combinations of safety and liveness.

A formula that does not include temporal operators is a *Boolean formula* (or an *assertion*). Given non-overlapping sets of Boolean variables $\mathcal{V}_1, \ldots, \mathcal{V}_k$, we use the notation $\varphi(\mathcal{V}_1, \ldots, \mathcal{V}_k)$ to indicate that φ is a Boolean formula over $\mathcal{V}_1 \cup \cdots \cup \mathcal{V}_k$. For Boolean formulas we consider models representing only a single truth assignment, i.e., given a Boolean formula $\varphi(\mathcal{V})$, we say that $s \in 2^{\mathcal{V}}$ *models* (*or satisfies*) φ, written as $s \models \varphi$, if the formula obtained from φ by replacing all variables in s by true and all other variables by false is valid. Formally, we define $s \models \varphi$ inductively by (i) for $v \in \mathcal{V}$, $s \models v$ iff $v \in s$, (ii) $s \models \neg\varphi$ iff $s \not\models \varphi$, and (iii) $s \models \varphi \vee \psi$ iff $s \models \varphi$ or $s \models \psi$. We call the set of all possible assignments to variables $\mathcal{V}$ *states* and denote them by $\Sigma_{\mathcal{V}}$ (or simply Σ, if $\mathcal{V}$ is clear from the context), i.e., $\Sigma_{\mathcal{V}} = 2^{\mathcal{V}}$. We say that s is a φ-state if $s \models \varphi$. Given a formula φ and a set of states $S \subseteq \Sigma_{\mathcal{V}}$, we say S satisfies φ denoted by $S \models \varphi$, if for all $s \in S$, $s \models \varphi$ holds. Given

a subset $\mathcal{Y} \subseteq \mathcal{V}$ of the variables and a state $s \in \Sigma_{\mathcal{V}}$, we denote by $s|_{\mathcal{Y}}$ the projection of s to $\mathcal{Y}$, i.e., $s|_{\mathcal{Y}} = \{y \in \mathcal{Y} \mid y \in s\}$. We will often use assertions over $\mathcal{V}_1 \cup \cdots \cup \mathcal{V}_k \cup \mathcal{V}'_1 \cup \cdots \cup \mathcal{V}'_k$, where $\mathcal{V}'_i$ is the set of primed versions of variables in $\mathcal{V}_i$, i.e., $\mathcal{V}'_i = \{v' \mid v \in \mathcal{V}_i\}$. Given an assertion $\varphi(\mathcal{V}_1, \ldots, \mathcal{V}_k, \mathcal{V}'_1, \ldots, \mathcal{V}'_k)$ and assignments $s_i, t_i \in \Sigma_{\mathcal{V}_i}$, we use $(s_1, \ldots, s_k, t'_1, \ldots, t'_k) \models \varphi$ to abbreviate $s_1 \cup \cdots \cup s_2 \cup t'_1 \cup \cdots \cup t'_k \models \varphi$, where $t'_i = \{v' \in \mathcal{V}'_i \mid v \in t_i\}$.

2.2. Fair discrete systems

A *fair discrete system* (FDS) [32] is a symbolic representation of a transition system with finitely many states and weak and strong fairness constraints. We use FDS to represent reactive systems such as concurrent systems that communicate by shared variables or digital circuits. Formally, an FDS $\mathcal{D} = \langle \mathcal{V}, \theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

- $\mathcal{V} = \{v_1, \ldots, v_n\}$: A finite set of Boolean variables. We define a *state* s to be an interpretation of $\mathcal{V}$, i.e., $s \in \Sigma_{\mathcal{V}}$.
- θ: The *initial condition*. This is an assertion over $\mathcal{V}$ characterizing all the initial states of the FDS. A state is called *initial* if it satisfies θ.
- ρ: A *transition relation*. This is an assertion $\rho(\mathcal{V} \cup \mathcal{V}')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successors $s' \in \Sigma$, i.e., $(s, s') \models \rho$.
- $\mathcal{J} = \{J_1, \ldots, J_m\}$: A set of *justice requirements* (weak fairness). Each requirement $J \in \mathcal{J}$ is an assertion over $\mathcal{V}$ that is intended to hold infinitely many times in every computation.
- $\mathcal{C} = \{(P_1, Q_1), \ldots, (P_n, Q_n)\}$: A set of *compassion requirements* (strong fairness). Each requirement $(P, Q) \in \mathcal{C}$ consists of a pair of assertions, such that if a computation contains infinitely many P-states, it should also hold infinitely many Q-states.

We define a *run* of the FDS $\mathcal{D}$ to be a maximal sequence of states $\sigma = s_0, s_1, \ldots$ satisfying (i) *initiality*, i.e., $s_0 \models \theta$, and (ii) *consecution*, i.e., for every $j \geqslant 0$, $(s_j, s_{j+1}) \models \rho$. A sequence σ is maximal if either σ is infinite or $\sigma = s_0, \ldots, s_k$ and s_k has no $\mathcal{D}$-successor, i.e., for all $s_{k+1} \in \Sigma$, $(s_k, s_{k+1}) \not\models \rho$.

A run σ is called a *computation* of $\mathcal{D}$ if it is infinite and satisfies the following additional requirements: (i) *justice (or weak fairness)*, i.e., for each $J \in \mathcal{J}$, σ contains infinitely many J-positions, i.e., positions $j \geqslant 0$, such that $s_j \models J$, and (ii) *compassion* (*or strong fairness*), i.e., for each $(P, Q) \in \mathcal{C}$, if σ contains infinitely many P-positions, it must also contain infinitely many Q-positions.

We say that an FDS $\mathcal{D}$ *implements* specification φ, denoted $\mathcal{D} \models \varphi$, if every run of $\mathcal{D}$ is infinite, and every computation of $\mathcal{D}$ satisfies φ. An FDS is said to be *fairness-free* if $\mathcal{J} = \mathcal{C} = \emptyset$. It is called a *just discrete system* (JDS) if $\mathcal{C} = \emptyset$. When $\mathcal{J} = \emptyset$ or $\mathcal{C} = \emptyset$ we simply omit them from the description of $\mathcal{D}$. Note that for most reactive systems, it is sufficient to use a JDS (i.e., compassion-free) model. Compassion is only needed in cases, in which the system uses built-in synchronization constructs such as semaphores or synchronous communication.

An FDS $\mathcal{D}$ is *deterministic with respect to* $\mathcal{X} \subseteq \mathcal{V}$, if (i) $\mathcal{D}$ has deterministic initial states, i.e, for all states $s, t \in \Sigma_{\mathcal{V}}$, if $s \models \theta$, $t \models \theta$, and $s|_{\mathcal{X}} = t|_{\mathcal{X}}$, then $s = t$ holds, and (ii) $\mathcal{D}$ has deterministic transitions, i.e., for all states $s, s', s'' \in \Sigma_{\mathcal{V}}$, if $(s, s') \models \rho$, $(s, s'') \models \rho$, and $s'|_{\mathcal{X}} = s''|_{\mathcal{X}}$, then $s' = s''$ holds.

An FDS $\mathcal{D}$ is *complete with respect to* $\mathcal{X} \subseteq \mathcal{V}$, if (i) for every assignment $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, there exists a state $s \in \Sigma_{\mathcal{V}}$ such that $s|_{\mathcal{X}} = s_{\mathcal{X}}$ and $s \models \theta$, and (ii) for all states $s \in \Sigma_{\mathcal{V}}$ and assignments $s'_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, there exists a state $s' \in \Sigma_{\mathcal{V}}$ such that $s'|_{\mathcal{X}} = s'_{\mathcal{X}}$ and $(s, s') \models \rho$. For every FDS and every $\mathcal{X} \subseteq \mathcal{V}$, we can construct an FDS that is complete w.r.t. $\mathcal{X}$ whose set of computations is the same as that of the original. We simply add a Boolean variable sf and set $\hat{\theta} := sf \leftrightarrow \theta$ and $\hat{\rho} := sf' \leftrightarrow (\rho \wedge sf)$ and add sf as an additional justice requirement. The set of computations of the two FDS (when projecting the value of sf) are the same.

Given an FDS $\mathcal{D}$ that is deterministic and complete w.r.t. $\mathcal{X}$, for every possible sequence $\sigma = s_0, s_1, \ldots$ of states in $\Sigma_{\mathcal{X}}$, $\mathcal{D}$ has a unique run $\tau = t_0, t_1, \ldots$ such that for all $j \geqslant 0$, $s_j|_{\mathcal{X}} = t_j|_{\mathcal{X}}$ holds. We call τ the *run of* $\mathcal{D}$ *on* σ. Note that $\mathcal{D}$ can be seen as a symbolic representation of a Mealy machine with input signal $\mathcal{X}$ and output signals $\mathcal{V} \setminus \mathcal{X}$. We say that a sequence $\sigma \in (\Sigma_{\mathcal{X}})^{\omega}$ is *accepted by* $\mathcal{D}$, if the run of $\mathcal{D}$ on σ is a computation.

For every FDS $\mathcal{D}$, there exists an LTL formula $\varphi_{\mathcal{D}}$, called the *temporal semantics* of $\mathcal{D}$, which characterizes the computations of $\mathcal{D}$. It is given by

$$\varphi_{\mathcal{D}}: \quad \theta \wedge \Box\big(\rho(\mathcal{V}, \bigcirc\mathcal{V})\big) \wedge \bigwedge_{J \in \mathcal{J}} \Box\Diamond J \wedge \bigwedge_{(P,Q) \in \mathcal{C}} (\Box\Diamond P \rightarrow \Box\Diamond Q),$$

where $\rho(\mathcal{V}, \bigcirc\mathcal{V})$ is the formula obtained from ρ by replacing each instance of primed variable v' by the LTL formula $\bigcirc v$.

Note that in the case that $\mathcal{D}$ is compassion-free (i.e., it is a JDS), then its temporal semantics has the form

$$\varphi_{\mathcal{D}}: \quad \theta \wedge \Box\big(\rho(V, \bigcirc V)\big) \wedge \bigwedge_{J \in \mathcal{J}} \Box\Diamond J.$$

Here, we are interested in open systems. That is, systems that interact with their environment: that receive some inputs and react to them. For such systems specifications are usually partitioned into assumptions and guarantees. The intended meaning is that if all assumptions hold then all guarantees should hold as well. That is, if the environment behaves as

expected then the system will behave as expected as well. In many cases, when we consider the conjunction of all assumptions (or all guarantees) the resulting formula is the temporal semantics of a JDS. That is, it is common to get specifications of the form φ_e and φ_s, where (i) φ_e and φ_s are conjunctions of smaller properties, (ii) φ_e and φ_s are the temporal semantics of JDSs, and (iii) the intended meaning is that the system should satisfy $\varphi_e \rightarrow \varphi_s$.

2.3. Game structures

We consider two-player games played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment. Formally, a *game structure* $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ consists of the following components.

- $\mathcal{V} = \{v_1, \ldots, v_n\}$: A finite set of typed *state variables* over finite domains. Without loss of generality, we assume they are all Boolean. A state and the set of states $\Sigma_{\mathcal{V}}$ are defined as before.
- $\mathcal{X} \subseteq \mathcal{V}$ is a set of *input variables*. These are variables controlled by the environment.
- $\mathcal{Y} = \mathcal{V} \setminus \mathcal{X}$ is a set of *output variables*. These are variables controlled by the system.
- θ_e is an assertion over $\mathcal{X}$ characterizing the initial states of the environment.
- θ_s is an assertion over $\mathcal{V}$ characterizing the initial states of the system.
- $\rho_e(\mathcal{V}, \mathcal{X}')$ is the transition relation of the environment. This is an assertion relating a state $s \in \Sigma$ to a possible next input value $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ by referring to unprimed copies of $\mathcal{V}$ and primed copies of $\mathcal{X}$. The transition relation ρ_e identifies a valuation $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ as a *possible input* in state s if $(s, s_{\mathcal{X}}) \models \rho_e$.
- $\rho_s(\mathcal{V}, \mathcal{X}', \mathcal{Y}')$ is the transition relation of the system. This is an assertion relating a state $s \in \Sigma$ and an input value $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ to an output value $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ by referring to primed and unprimed copies of $\mathcal{V}$. The transition relation ρ_s identifies a valuation $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ as a *possible output* in state s reading input $s_{\mathcal{X}}$ if $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$.
- φ is the winning condition, given by an LTL formula.

A state s is *initial* if it satisfies both θ_e and θ_s, i.e., $s \models \theta_e \wedge \theta_s$. For two states s and s' of G, s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. A *play* σ of G is a maximal sequence of states $\sigma = s_0, s_1, \ldots$ satisfying (i) *initiality*, i.e., s_0 is initial and (ii) *consecution*, i.e., for each $j \geqslant 0$, s_{j+1} is a successor of s_j. Let G be a game structure and σ be a play of G. Initially, the environment chooses an assignment $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that $s_{\mathcal{X}} \models \theta_e$ and the system chooses an assignment $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ such that $(s_{\mathcal{X}}, s_{\mathcal{Y}})$ is initial. From a state s, the environment chooses an input $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that $(s, s_{\mathcal{X}}) \models \rho_e$ and the system chooses an output $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ such that $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$. We say that a play starting in state s is an *s-play*.

A play $\sigma = s_0, s_1, \ldots$ is *winning for the system* if either (i) σ is finite and there is no assignment $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that $(s_n, s_{\mathcal{X}}) \models \rho_e$, where s_n is the last state in σ, or (ii) σ is infinite and it satisfies φ. Otherwise, σ is *winning for the environment*.

A *strategy* for the system is a partial function $f : M \times \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{X}} \mapsto M \times \Sigma_{\mathcal{Y}}$, where M is some *memory domain* with a designated initial value $m_0 \in M$, such that for every $s \in \Sigma_{\mathcal{V}}$, every $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, and $m \in M$ if $(s, s_{\mathcal{X}}) \models \rho_e$ then $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$, where $f(m, s, s_{\mathcal{X}}) = (m', s_{\mathcal{Y}})$. Let f be a strategy for the system. A play $s_0, s_1, \ldots$ is said to be *compliant* with strategy f if for all $i \geqslant 0$ we have $f(m_i, s_i, s_{i+1}|_{\mathcal{X}}) = (m_{i+1}, s_{i+1}|_{\mathcal{Y}})$. Notice, that the sequence $m_0, m_1, \ldots$ is implicitly defined. Strategy f is *winning* for the system from state $s \in \Sigma_{\mathcal{V}}$ if all s-plays (plays starting from s) which are compliant with f are winning for the system. We denote by W_s the set of states from which there exists a winning strategy for the system. We treat W_s as an assertion as well. For player environment, strategies, winning strategies, and the winning set W_e are defined dually. A game structure G is said to be *winning* for the system, if for all $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, if $s_{\mathcal{X}} \models \theta_e$, then there exists $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ such that $(s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \theta_s$ and $(s_{\mathcal{X}}, s_{\mathcal{Y}}) \models W_s$. We say that f *uses finite memory* or *is finite* when M is finite. When M is a singleton, we say that f is *memoryless*.

2.3.1. Realizability and synthesis

Given an LTL formula φ over sets of input and output variables $\mathcal{X}$ and $\mathcal{Y}$, we say that an FDS $\mathcal{D} = \langle \mathcal{V}, \theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ *realizes* φ if (i) $\mathcal{V}$ contains $\mathcal{X}$ and $\mathcal{Y}$, (ii) $\mathcal{D}$ is complete with respect to $\mathcal{X}$, and (iii) $\mathcal{D} \models \varphi$. Such an FDS is called a *controller* for φ, or just a controller. We say that the specification is *realizable* [8], if there exists a fairness-free FDS $\mathcal{D}$ that realizes it. Otherwise, we say that the specification is *unrealizable*. If the specification is realizable, then the construction of such a controller constitutes a solution for the *synthesis* problem.[2]

Given an LTL formula over sets of input and output variables $\mathcal{X}$ and $\mathcal{Y}$, respectively, its realizability problem can be reduced to the decision of winner in a game. Formally, $G_\varphi = \langle \mathcal{X} \cup \mathcal{Y}, \mathcal{X}, \mathcal{Y}, \mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, \varphi \rangle$ is the game where the initial conditions and the transition relations are true and the winning condition is φ. If the environment is winning in G_φ, then φ is unrealizable. If the system is winning in G_φ, then φ is realizable. Furthermore, from the winning strategy of

[2] As all the variables of FDSs are Boolean, this definition calls for realizability by a finite state system. It is well known that for LTL specifications realizability and realizability by finite state systems are the same.

the system it is possible to extract a controller that realizes φ. Realizability for general LTL specifications is 2EXPTIME-complete [33]. It is well known that for LTL specifications it is sufficient to consider finite memory strategies. In this paper we are interested in a subset of LTL for which we solve realizability and synthesis in time exponential in the size of the LTL formula and polynomial in the resulting controller.

More generally, consider a game $G : \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$. The system wins in G iff the following formula is realizable[3]:

$$\varphi_G = (\theta_e \to \theta_s) \wedge \big(\theta_e \to \square\big((\boxminus \rho_e) \to \rho_s\big)\big) \wedge (\theta_e \wedge \square \rho_e \to \varphi).$$

Formally, we have the following.

Theorem 1. *The system wins in a game G iff φ_G is realizable.*

The proof of this theorem resembles the proof of Theorem 4 and is omitted.

3. Generalized Reactive(1) games

In [18], we consider the case of *Generalized Reactive(1)* games (called there *generalized Streett(1)* games). In these games the winning condition is an implication between conjunctions of recurrence formulas ($\square \diamondsuit J$ where J is a Boolean formula). We repeat the main ideas from [18] and show how to solve GR(1) games, by computing the winning states of each of the players. We start with a definition of μ-calculus over game structures. We then give the μ-calculus formula that characterizes the set of winning states of the system; and explain how to implement this solution symbolically. We defer the extraction of a controller from this computation to Section 4. We finish this section by explaining the straightforward usage of GR(1) games in synthesis from LTL. In Section 5 we include a thorough discussion of usage of GR(1) games for LTL synthesis.

3.1. μ-Calculus over game structures

We define μ-calculus [34] over game structures. Consider a game structure $G : \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$. For every variable $v \in \mathcal{V}$ the formulas v and $\neg v$ are *atomic formulas*. Let $Var = \{X, Y, \ldots\}$ be a set of *relational variables*. The μ-calculus formulas are constructed as follows:

$$\varphi ::= v \mid \neg v \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \diamondsuit \varphi \mid \boxminus \varphi \mid \mu X \varphi \mid \nu X \varphi.$$

A formula ψ is interpreted as the set of G-states in Σ in which ψ is true. We write such set of states as $[[\psi]]_G^{\mathcal{E}}$ where G is the game structure and $\mathcal{E} : Var \to 2^{\Sigma}$ is an *environment*. The environment assigns to each relational variable a subset of Σ. We denote by $\mathcal{E}[X \leftarrow S]$ the environment such that $\mathcal{E}[X \leftarrow S](X) = S$ and $\mathcal{E}[X \leftarrow S](Y) = \mathcal{E}(Y)$ for $Y \neq X$. The set $[[\psi]]_G^{\mathcal{E}}$ is defined inductively as follows.[4]

- $[[v]]_G^{\mathcal{E}} = \{s \in \Sigma \mid s[v] = 1\}$.
- $[[\neg v]]_G^{\mathcal{E}} = \{s \in \Sigma \mid s[v] = 0\}$.
- $[[X]]_G^{\mathcal{E}} = \mathcal{E}(X)$.
- $[[\varphi \vee \psi]]_G^{\mathcal{E}} = [[\varphi]]_G^{\mathcal{E}} \cup [[\psi]]_G^{\mathcal{E}}$.
- $[[\varphi \wedge \psi]]_G^{\mathcal{E}} = [[\varphi]]_G^{\mathcal{E}} \cap [[\psi]]_G^{\mathcal{E}}$.
- $$[[\diamondsuit \varphi]]_G^{\mathcal{E}} = \left\{ s \in \Sigma \;\middle|\; \begin{array}{c} \forall s_{\mathcal{X}} \in \Sigma_{\mathcal{X}},\ (s, s_{\mathcal{X}}) \models \rho_e \to \exists s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}} \text{ such that} \\ (s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s \text{ and } (s_{\mathcal{X}}, s_{\mathcal{Y}}) \in [[\varphi]]_G^{\mathcal{E}} \end{array} \right\}.$$

  A state s is included in $[[\diamondsuit \varphi]]_G^{\mathcal{E}}$ if the system can force the play to reach a state in $[[\varphi]]_G^{\mathcal{E}}$. That is, regardless of how the environment moves from s, the system can choose an appropriate move into $[[\varphi]]_G^{\mathcal{E}}$.
- $$[[\boxminus \varphi]]_G^{\mathcal{E}} = \left\{ s \in \Sigma \;\middle|\; \begin{array}{c} \exists s_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \text{ such that } (s, s_{\mathcal{X}}) \models \rho_e \text{ and } \forall s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}, \\ (s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s \to (s_{\mathcal{X}}, s_{\mathcal{Y}}) \in [[\varphi]]_G^{\mathcal{E}} \end{array} \right\}.$$

  A state s is included in $[[\boxminus \varphi]]_G^{\mathcal{E}}$ if the environment can force the play to reach a state in $[[\varphi]]_G^{\mathcal{E}}$. As the environment moves first, it chooses an input $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that for all choices of the system the successor is in $[[\varphi]]_G^{\mathcal{E}}$.

[3] Technically, ρ_e and ρ_s contain primed variables and are not LTL formulas. This can be easily handled by using the next operator ($\bigcirc$). We ignore this issue in the rest of the paper.

[4] Only for finite game structures.

- $[[\mu X\varphi]]^{\mathcal{E}}_G = \bigcup_i S_i$ where $S_0 = \emptyset$ and $S_{i+1} = [[\varphi]]^{\mathcal{E}[X \leftarrow S_i]}_G$.
- $[[\nu X\varphi]]^{\mathcal{E}}_G = \bigcap_i S_i$ where $S_0 = \Sigma$ and $S_{i+1} = [[\varphi]]^{\mathcal{E}[X \leftarrow S_i]}_G$.

When all the variables in φ are bound by either μ or ν the initial environment is not important and we simply write $[[\varphi]]_G$. In case that G is clear from the context we write $[[\varphi]]$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. A μ-calculus formula defines a symbolic algorithm for computing $[[\varphi]]$ [35] (i.e., an algorithm that manipulates sets of states rather than individual states). For a μ-calculus formula of alternation depth k, this symbolic algorithm requires the computation of at most $O(|\Sigma|^{k+1})$ symbolic next step operations. By saving intermediate results of the symbolic computation it is possible to reduce the number of symbolic next step operations of the symbolic algorithm to $O(|\Sigma|^{\lceil \frac{k+1}{2} \rceil})$ [36]. In general, if the number of transitions of G is m, then it is known that a μ-calculus formula over G can be evaluated in time proportional to $O(m|\Sigma|^{\lfloor \frac{k}{2} \rfloor})$ [37]. For a full exposition of μ-calculus we refer the reader to [38]. We often abuse notations and write a μ-calculus formula φ instead of the set $[[\varphi]]$.

In some cases, instead of using a very complex formula, it may be more readable to use *vector notation* as in Eq. (2):

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \begin{bmatrix} \mu Y(\bigcirc Y \vee p \wedge \bigcirc Z_2) \\ \mu Y(\bigcirc Y \vee q \wedge \bigcirc Z_1) \end{bmatrix}. \tag{2}$$

Such a formula, may be viewed as the mutual fixpoint of the variables Z_1 and Z_2 or equivalently as an equal formula where a single variable Z replaces both Z_1 and Z_2 and ranges over pairs of states [39]. The formula above characterizes the set of states from which system can force the game to visit p-states infinitely often and q-states infinitely often. We can characterize the same set of states by the following 'normal' formula[5]:

$$\varphi = \nu Z\big(\big[\mu Y(\bigcirc Y \vee p \wedge \bigcirc Z)\big] \wedge \big[\mu Y(\bigcirc Y \vee q \wedge \bigcirc Z)\big]\big).$$

3.2. Solving GR(1) games

Let G be a game where the winning condition is of the following form:

$$\varphi = \bigwedge_{i=1}^{m} \Box\Diamond J_i^e \rightarrow \bigwedge_{j=1}^{n} \Box\Diamond J_j^s.$$

Here J_i^e and J_j^s are Boolean formulas. We refer to such games as Generalized Reactivity(1) games, or GR(1) in short. In [18] we term these games as generalized Streett(1) games and provide the following μ-calculus formula to solve them. Let $j \oplus 1 = (j \bmod n) + 1$,

$$\varphi_{gr} = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y(\bigvee_{i=1}^{m} \nu X(J_1^s \wedge \bigcirc Z_2 \vee \bigcirc Y \vee \neg J_i^e \wedge \bigcirc X)) \\ \mu Y(\bigvee_{i=1}^{m} \nu X(J_2^s \wedge \bigcirc Z_3 \vee \bigcirc Y \vee \neg J_i^e \wedge \bigcirc X)) \\ \vdots \\ \mu Y(\bigvee_{i=1}^{m} \nu X(J_n^s \wedge \bigcirc Z_1 \vee \bigcirc Y \vee \neg J_i^e \wedge \bigcirc X)) \end{bmatrix}. \tag{3}$$

Intuitively, for $j \in [1..n]$ and $i \in [1..m]$ the greatest fixpoint $\nu X(J_j^s \wedge \bigcirc Z_{j\oplus 1} \vee \bigcirc Y \vee \neg J_i^e \wedge \bigcirc X)$ characterizes the set of states from which the system can force the play either to stay indefinitely in $\neg J_i^e$ states (thus violating the left-hand side of the implication) or in a finite number of steps reach a state in the set $J_j^s \wedge \bigcirc Z_{j\oplus 1} \vee \bigcirc Y$. The two outer fixpoints make sure that the system wins from the set $J_j^s \wedge \bigcirc Z_{j\oplus 1} \vee \bigcirc Y$. The least fixpoint μY makes sure that the unconstrained phase of a play represented by the disjunct $\bigcirc Y$ is finite and ends in a $J_j^s \wedge \bigcirc Z_{j\oplus 1}$ state. Finally, the greatest fixpoint νZ_j is responsible for ensuring that, after visiting J_j^s, we can loop and visit $J_{j\oplus 1}^s$ and so on. By the cyclic dependence of the outermost greatest fixpoint, either all the sets in J_j^s are visited or, getting stuck in some inner greatest fixpoint, some J_i^e is visited only finitely many times.

Lemma 2. *(See [18].)* $W_s = [[\varphi]]$.

[5] This does not suggest a canonical translation from vector formulas to plain formulas. The same translation works for the formula in Eq. (3) below. Note that the formula in Eq. (2) and the formula in Eq. (3) have a very similar structure.

```
public BDD calculate_win() {
   BDD Z = TRUE;
   for (Fix fZ = new Fix(); fZ.advance(Z);) {
      mem.clear();
      for (int j = 1; j <= sys.numJ(); j++) {
         BDD Y = FALSE;
         for (Fix fY = new Fix(); fY.advance(Y);) {
            BDD start = sys.Ji(j).and(cox(Z)).or(cox(Y));
            Y = FALSE;
            for (int i = 1; i <= env.numJ(); i++) {
               BDD X = Z;
               for (Fix fX = new Fix(); fX.advance(X);)
                  X = start.or(env.Ji(i).not().and(cox(X)));
               mem.addX(j, i, X); // store values of X
               Y = Y.or(X);
            }
            mem.addY(j, Y); // store values of Y
         }
         Z = Y;
      }
   }
   return Z;
}
```

Fig. 1. JTLV implementation of Eq. (3).

We include in Fig. 1 a (slightly simplified) code of the implementation of this μ-calculus formula in JTLV ([40]). We denote the system and environment players by `sys` and `env`, respectively. We denote J_i^e and J_j^s by `env.Ji(i)` and `sys.Ji(j)`, respectively. $\diamondsuit$ is denoted by `cox`. We use `Fix` to iterate over the fixpoint values. The loop terminates if two successive values are the same. We use `mem` to collect the intermediate values of `Y` and `X`. We denote by `mY` the two dimensional vector ranging over $1..n$, and $1..k$, where k is the depth of the least fixpoint iteration of `Y`. We denote by `mX` a three dimensional vector ranging over $1..n$, $1..k$, and $1..m$. We use the sets `mY`$[j][r]$ and their subsets `mX`$[j][r][i]$ to define n memoryless strategies for the system. The strategy f_j is defined on the states in Z_j. We show that the strategy f_j either forces the play to visit J_j^s and then proceed to $Z_{j\oplus 1}$, or eventually avoid some J_i^e. We show that by combining these strategies, either the system switches strategies infinitely many times and ensures that the play be satisfies the right-hand side of the implication or eventually uses a fixed strategy ensuring that the play does not satisfy the left-hand side of the implication. Essentially, the strategies are "go to `mY`$[j][r]$ for minimal r" until getting to a J_j^s state and then switch to strategy $j \oplus 1$ or "stay in `mX`$[j][r][i]$".

It follows that we can solve realizability of LTL formulas in the form that interests us in polynomial (quadratic) time.

Theorem 3. *(See [18].) A game structure G with a* GR(1) *winning condition of the form $\varphi = \bigwedge_{i=1}^{m} \Box\diamondsuit J_i^e \to \bigwedge_{j=1}^{n} \Box\diamondsuit J_j^s$ can be solved by a symbolic algorithm that performs $O(nm|\Sigma|^2)$ next step computations, where Σ is the set of all possible assignments to the variables in φ.*

A straightforward implementation of the fixpoint computation gives a cubic upper bound. Implementing the approach of [36] reduces the complexity in $|\Sigma|$ to quadratic, as stated. Their approach starts computations of fixpoints from earlier approximations of their values. Thus, the fixpoint is not restarted from scratch leading to significant savings. An enumerative algorithm (i.e., an algorithm that handles states individually and not sets of states) can solve a GR(1) game structure in time $O(nm|\Sigma||T|)$, where $|T|$ is the number of transitions in the game [41].[6]

Following Theorem 1, we prove the following about the connection between solving GR(1) games and realizability. Consider a GR(1) game G: $\langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi\rangle$, where $\varphi = \bigwedge_{i=1}^{m} \Box\diamondsuit J_i^e \to \bigwedge_{j=1}^{n} \Box\diamondsuit J_j^s$. Let

$$\varphi_G = (\theta_e \to \theta_s) \wedge \big(\theta_e \to \Box\big((\boxminus \rho_e) \to \rho_s\big)\big) \wedge \big((\theta_e \wedge \Box\rho_e) \to \varphi\big).$$

[6] We note that in the previous versions [18,1] the analysis of complexity was not accurate and in particular higher than stated above. This led to some confusion in the exact complexity of solving GR(1) games. In particular, in [42] an enumerative algorithm for GR(1) games is suggested whose complexity is *higher* than the complexity stated above. It is, however, better than the stated complexity in previous publications.

Intuitively, this formula is split into three levels: initial, safety, and liveness levels. In order to realize this formula the system needs to satisfy the same levels the environment does. For instance, if the environment chooses an initial assignment satisfying θ_e, the system cannot choose an initial assignment violating θ_s even if the environment later violates ρ_e.

Theorem 4. *The system wins in G iff φ_G is realizable.*

Proof. Recall that if the system wins G finite memory suffices. Let M be some memory domain and m_0 its designated initial value. Suppose that $f : M \times \Sigma \times \Sigma_{\mathcal{X}} \to M \times \Sigma_{\mathcal{Y}}$ is a winning strategy for the system in G. Furthermore, for every $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that $s_{\mathcal{X}} \models \theta_e$ there exists a $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ such that $(s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \theta_e \wedge \theta_s$ and $(s_{\mathcal{X}}, s_{\mathcal{Y}}) \in W_s$. We use f to construct a fairness-free FDS that realizes φ_G.

Let $|M| = k$ and let $\mathcal{M} = \{m_1, \ldots, m_{\lceil \log(k) \rceil}\}$ be Boolean variables. It follows that an assignment to $\mathcal{M}$ characterizes a value in M. By abuse of notations, we denote by m the value in M, the assignment to $\mathcal{M}$ that represents that value, and the assertion over $\mathcal{M}$ whose unique satisfying assignment is m. Similarly, for a state $s \in \Sigma$, we denote by s the assertion whose unique satisfying assignment is s. Consider the fairness-free FDS $\mathcal{D} = \langle \hat{\mathcal{V}}, \hat{\theta}, \hat{\rho} \rangle$ with the following components.

- $\hat{\mathcal{V}} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{M}$.
- $\hat{\theta} = \theta_e \to (\theta_s \wedge m_0 \wedge W_s)$.
  That is, if the assignment to $\mathcal{X}$ satisfies θ_e then the assignment to $\mathcal{Y}$ ensures θ_s and the joint assignment to $\mathcal{X}$ and $\mathcal{Y}$ (i.e., the state) is in W_s. Furthermore, the initial memory value is m_0. If the assignment to the input variables does not satisfy θ_e then the choice of m and $s_{\mathcal{Y}}$ is arbitrary.
- For the definition of $\hat{\rho}$ we write the strategy f as an assertion as follows:

$$\hat{f} = \bigwedge_{m \in M} \bigwedge_{s \in W_s} \bigwedge_{s'_{\mathcal{X}} \in \Sigma_{\mathcal{X}}} \left((m \wedge s \wedge s'_{\mathcal{X}}) \to f(m, s, s'_{\mathcal{X}})' \right).$$

  That is, depending on the current value of m, s, and $s'_{\mathcal{X}}$, the assignment to m' and $s'_{\mathcal{Y}}$ respects the strategy f.
  Finally, $\hat{\rho}$ is the following assertion:

$$\hat{\rho} = (W_s \wedge \rho_e) \to \hat{f}.$$

  That is, if the current state s is winning for system (W_s) and the environment chooses an input $s'_{\mathcal{X}}$ such that $(s, s'_{\mathcal{X}}) \models \rho_e$, then the system is going to update the memory and choose outputs $s'_{\mathcal{Y}}$ according to f. If the current state is winning for the environment or the environment does not satisfy its transition, the choice of memory value and output is arbitrary.

We have to show that $\mathcal{D}$ is complete with respect to $\mathcal{X}$ and that $\mathcal{D} \models \varphi_G$. Completeness of $\mathcal{D}$ follows from the definition of winning in G and from the definition of the strategy f. Indeed, as system wins G, for every $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that $s_{\mathcal{X}} \models \theta_e$ there exists a state $s \in \Sigma$ such that $s \in W_s$ and $s \models \theta_e \wedge \theta_s$. Furthermore, if $s_{\mathcal{X}} \not\models \theta_e$ then, by definition of $\hat{\theta}$, for every state s such that $s|_{\mathcal{X}} = s_{\mathcal{X}}$ we have $s \models \hat{\theta}$. Similarly, for every $m \in M$, $s \in \Sigma$, and $s'_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, if $s \in W_s$ and $(s, s'_{\mathcal{X}}) \models \rho_e$ then $\hat{f}$ defines values $s'_{\mathcal{Y}}$ and m' such that $(s, m, s'_{\mathcal{X}}, m', s'_{\mathcal{Y}}) \models \hat{\rho}$. If $s \notin W_s$ or $(s, s'_{\mathcal{X}}) \not\models \rho_e$ then for every $s'_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ we have $(s, m, s'_{\mathcal{X}}, m', s'_{\mathcal{Y}}) \models \hat{\rho}$.

We have to show that $\mathcal{D} \models \varphi_G$. Consider an infinite computation $\sigma\colon s_0, s_1, \ldots$ of $\mathcal{D}$. Clearly, if $s_0 \not\models \theta_e$ then $\sigma \models \varphi_G$. Assume that $s_0 \models \theta_e$, then by definition of $\hat{\theta}$ we have $s_0 \models \theta_s$ and $s_0 \in W_s$ as well. Suppose now that for some i' we have $(s_{i'}, s_{i'+1}) \not\models \rho_e$. Let i_0 be the minimal such that $(s_{i_0}, s_{i_0+1}) \not\models \rho_e$. We can show by induction that for every $i < i_0$ we have $s_i \in W_s$ and $(s_i, s_{i+1}) \models \rho_s$. It follows that $\sigma \models \square(\boxminus \rho_e \to \rho_s)$ as required. Finally, as $\sigma \not\models \square \rho_e$ the third clause holds as well. The remaining case is when $\sigma \models \square \rho_e$. We can show by induction that for every $i \geqslant 0$ we have $(s_i, s_{i+1}) \models \rho_s$. We have to show that $\sigma \models \varphi$ as well. However, $\sigma|_{\mathcal{X} \cup \mathcal{Y}}$ is a play in G that is compliant with f. It follows that $\sigma \models \varphi$ as required. Overall, $\mathcal{D} \models \varphi_G$.

Suppose that there exists a fairness-free FDS $\mathcal{D} = \langle \hat{\mathcal{V}}, \hat{\theta}, \hat{\rho} \rangle$ that is complete with respect to $\mathcal{X}$ and such that $\mathcal{D} \models \varphi$. We use the states of $\mathcal{D}$ as the memory domain for construction of a strategy f. Let t^{in} be a new state to be used as the initial value of the memory. Formally, for a memory value t we define $f(t, s, s'_{\mathcal{X}}) = (t', s'_{\mathcal{Y}})$ as follows.

- If $t = t^{in}$ then we define t' and $s'_{\mathcal{Y}}$ as follows.
  - If $s \models \theta_e \wedge \theta_s$ and there exists a state $t_0 \models \hat{\theta}$ such that $t_0|_{\mathcal{X} \cup \mathcal{Y}} = s$ then, by completeness of $\mathcal{D}$, there exists a successor t' of t_0 such that $t'|_{\mathcal{X}} = s'_{\mathcal{X}}$ and we set $s'_{\mathcal{Y}} = t'|_{\mathcal{Y}}$.
  - If $s \not\models \theta_e \wedge \theta_s$ or there is no state $t_0 \models \hat{\theta}$ such that $t_0|_{\mathcal{X} \cup \mathcal{Y}} = s$ then we choose arbitrary t' and $s'_{\mathcal{Y}}$ (if at all).
- If $t \neq t^{in}$ then we define t' and $s'_{\mathcal{Y}}$ as follows.
  - If $t|_{\mathcal{X} \cup \mathcal{Y}} = s$, then, by completeness of $\mathcal{D}$, the state t has a successor t'' such that $t''|_{\mathcal{X}} = s'_{\mathcal{X}}$. We set $t' = t''$ and $s'_{\mathcal{Y}} = t''|_{\mathcal{Y}}$.
  - If $t|_{\mathcal{X} \cup \mathcal{Y}} \neq s$, then t' and $s'_{\mathcal{Y}}$ are arbitrary.

We claim that this strategy is winning from every state s for which there exists a state t_0 such that $t_0 \models \hat{\theta}$ and $t_0|_{\mathcal{X}\cup\mathcal{Y}} = s$. Consider such a state s_0. Then for every $s'_{\mathcal{X}}$ such that $(s_0, s'_{\mathcal{X}}) \models \rho_e$ there exists a t_1 and $s'_{\mathcal{Y}}$ such that $(t_0, t_1) \models \rho_s$, $t_1|_{\mathcal{X}} = s'_{\mathcal{X}}$, and $t_1|_{\mathcal{Y}} = s'_{\mathcal{Y}}$. Consider a play $\sigma\colon s_0, \ldots, s_n$ compliant with f, where the sequence of memory values is $\tau\colon t_0, \ldots, t_n$. It is simple to show by induction that for every $j \geqslant 1$ we have $t_j|_{\mathcal{X}\cup\mathcal{Y}} = s_j$. Consider a value $s'_{\mathcal{X}}$ such that $(s_n, s'_{\mathcal{X}}) \models \rho_e$. By completeness of $\mathcal{D}$ there exists a memory value t_{n+1} such that $(t_n, t_{n+1}) \models \hat{\rho}$, $t_{n+1}|_{\mathcal{X}} = s'_{\mathcal{X}}$ so the strategy f is defined. Furthermore, from $\mathcal{D} \models \varphi_G$ it follows that $(t_n, t_{n+1}) \models \rho_s$. Thus, $t_{n+1}|_{\mathcal{Y}}$ is a valid choice of the strategy f.

Consider an infinite play $\sigma\colon s_0, \ldots$ compliant with f, where $\tau\colon t_0, \ldots$ is the associated sequence of memory values. Then, as τ is a computation of $\mathcal{D}$ (modulo the initial state), it follows that $\tau \models \varphi_G$. We conclude that $\sigma \models \varphi$.

Finally, we have to show that for every value $s'_{\mathcal{X}}$ such that $s'_{\mathcal{X}} \models \theta_e$ there exists a value $s'_{\mathcal{Y}}$ such that $(s'_{\mathcal{X}}, s'_{\mathcal{Y}}) \models \theta_e \wedge \theta_s$. However, this follows from the completeness of $\mathcal{D}$ and from the inclusion of θ_e on the left-hand side of every implication in φ_G. □

3.3. *Symbolic* JDS *specifications*

We would like to use GR(1) games to solve realizability directly from LTL formulas. In many practical cases, the specification is partitioned to assumptions and guarantees. Each assumption or guarantee is relatively simple; and together they have the semantics that the conjunction of all assumptions implies the conjunction of all guarantees. To support this claim, we will demonstrate in Section 6 the application of the synthesis method to a realistic industrial specification. Here we suggest to embed such specifications directly into a GR(1) game, giving rise to the *strict semantics* of the implication. In Section 5 we discuss the differences between the strict semantics and the simple implication.

Recall that a temporal semantics of a JDS $\mathcal{D}$ has the following form:

$$\varphi_{\mathcal{D}} : \theta \wedge \Box\big(\rho(\mathcal{V}, \bigcirc\mathcal{V})\big) \wedge \bigwedge_{J\in\mathcal{J}} \Box\Diamond J.$$

Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets of typed input and output variables, respectively and let $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$. We say that a JDS is *output independent* if θ does not relate to $\mathcal{Y}$ and ρ does not depend on the value of $\mathcal{Y}$ in the next state. That is, ρ can be expressed as an assertion over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}'$. A JDS can be represented by a triplet $\langle \varphi_i, \Phi_t, \Phi_g \rangle$ with the following parts.

- φ_i is an assertion which characterizes the initial states (i.e., θ above).
- $\Phi_t = \{\psi_i\}_{i\in I_t}$ is a set of Boolean formulas ψ_i, where each ψ_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc v$ where $v \in \mathcal{X}$ if the JDS is output independent, and $v \in \mathcal{X} \cup \mathcal{Y}$ otherwise.
  That is $\rho(\mathcal{V}, \bigcirc\mathcal{V})$ is the conjunction of all the assertions in Φ_t.
- $\Phi_g = \{J_i\}_{i\in I_g}$ is a set of Boolean formulas (i.e, Φ_g is a different name for $\mathcal{J}$).

The intended semantics of the triplet $\langle \varphi_i, \Phi_t, \Phi_g \rangle$ is

$$\varphi_i \wedge \bigwedge_{i\in I_t} \Box\psi_i \wedge \bigwedge_{i\in I_g} \Box\Diamond J_i.$$

Consider the case where assumptions and guarantees have the following forms: (i) ψ for an assertion over $\mathcal{V}$, (ii) $\Box\psi$ for an assertion over $\mathcal{V} \cup \mathcal{V}'$, or (iii) $\Box\Diamond\psi$ for an assertion over $\mathcal{V}$. Then, we can partition the Boolean components of assumptions or guarantees to triplets as explained above.

Let $S_\alpha = \langle \varphi_i^\alpha, \Phi_t^\alpha, \Phi_g^\alpha \rangle$ for $\alpha \in \{e, s\}$ be two specifications as described above, where S_e is output independent. Here S_e is a description of the environment (i.e., results from the assumptions) and S_s is the description of the system (i.e., results from the guarantees). The specifications S_e and S_s naturally give rise to the following game. The *strict realizability game for* S_e *and* S_s is $G^{sr}_{e,s}$: $\langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ with the following components.[7]

- $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$.
- $\theta_e = \varphi_i^e$.
- $\theta_s = \varphi_i^s$.
- $\rho_e = \bigwedge_{i\in I_t^e} \psi_i^e$.
- $\rho_s = \bigwedge_{i\in I_t^s} \psi_i^s$.
- $\varphi = \bigwedge_{i\in I_g^e} \Box\Diamond J_i^e \rightarrow \bigwedge_{i\in I_g^s} \Box\Diamond J_i^s$.

By Theorem 4 the game $G^{sr}_{e,s}$ is winning for system iff the following formula is realizable

[7] The name strict realizability when referring to such a composition was coined in [43].

$$\varphi^{sr}_{e,s} = \big(\varphi^e_i \to \varphi^s_i\big) \wedge \big(\varphi^e_i \wedge \Box\big((\boxminus\rho_e) \to \rho_s\big)\big) \wedge \Big(\varphi^e_i \wedge \Box\rho_e \wedge \bigwedge_{i\in I^e_g} \Box\Diamond J^e_i \to \bigwedge_{i\in I^s_g} \Box\Diamond J^s_i\Big).$$

The proof of Theorem 4 also tells us how to extract an implementation for $\varphi^{sr}_{e,s}$ from the winning strategy in $G^{sr}_{e,s}$.

3.4. Example: Lift controller

As an example, we consider a simple lift controller. We specify a lift controller serving n floors. We assume the lift has n button sensors $(b_1, \ldots, b_n)$ controlled by the environment. The lift may be requested on every floor, once the lift has been called on some floor the request cannot be withdrawn. Initially, on all floors there are no requests. The location of the lift is modeled by n Boolean variables $(f_1, \ldots, f_n)$ controlled by the system. Once a request has been fulfilled it is removed. Formally, the specification of the environment is $S_e = \langle \varphi^e_i, \{\psi^e_{1,1}, \ldots, \psi^e_{1,n}, \psi^e_{2,1}, \ldots, \psi^e_{2,n}\}, \emptyset\rangle$, where the components of S_e are as follows:

$$\varphi^e_i = \bigwedge_j \neg b_j,$$
$$\psi^e_{1,j} = b_j \wedge f_j \to \bigcirc\neg b_j,$$
$$\psi^e_{2,j} = b_j \wedge \neg f_j \to \bigcirc b_j.$$

We expect the lift to initially start on the first floor. We model the location of the lift by an n bit array. Thus we have to demand mutual exclusion on this array. The lift can move at most one floor at a time, and eventually satisfy every request. Formally, the specification of the system is $S_s = \langle \varphi^s_i, \{\psi^s_1, \psi^s_{2,1}, \ldots, \psi^s_{2,n}, \psi^s_{3,1}, \ldots, \psi^s_{3,n}\}, \{J^s_1, \ldots, J^s_{n+1}\}\rangle$, where the components of S_s are as follows:

$$\varphi^s_i = \bigwedge_j (j = 1 \wedge f_j) \vee \big((j \neq 1) \wedge \neg f_j\big),$$
$$\psi^s_1 = up \to sb,$$
$$\psi^s_{2,j} = \bigwedge_{k\neq j} \neg(f_j \wedge f_k),$$
$$\psi^s_{3,j} = f_j \to \bigcirc(f_j \vee f_{j-1} \vee f_{j+1}),$$
$$J^s_j = b_j \to f_j,$$
$$J^s_{n+1} = f_1 \vee sb,$$

where $up = \bigvee_i (f_i \wedge \bigcirc f_{i+1})$ denotes that the lift moves one floor up, and $sb = \bigvee_i b_i$ denotes that at least one button is pressed. The requirement ψ^s_1 states that the lift should not move up unless some button is pressed. The liveness requirement J^s_{n+1} states that either some button is pressed infinitely many times, or the lift parks at floor f_1 infinitely many times. Together they imply that when there is no active request, the lift should move down and park at floor f_1.

The strict realizability game for S_e and S_s is won by system, implying that there is a controller realizing $\varphi^{sr}_{e,s}$.

4. GR(1) strategies

In this section we discuss how to extract a program from the solution of the GR(1) game. First, we show how to analyze the intermediate values and how to extract from them a winning strategy for the system. Then, we show how this strategy can be reduced in size in some cases. Finally, we show how to extract from the symbolic BDD representation of the strategy a deterministic strategy that can be used for creating an HDL description of a resulting circuit.

4.1. Extracting the strategy

We show how to use the intermediate values in the computation of the fixpoint to produce an FDS that implements φ. The FDS basically follows the strategies explained in Section 3.2. Recall that the combined strategy does one of two things. It either iterates over strategies $f_1, .., f_n$ infinitely often, where strategy f_j ensures that the play reaches a J^s_j state. Thus, the play satisfies all liveness guarantees. Or, it eventually uses a fixed strategy ensuring that the play does not satisfy one of the liveness assumptions.

$$\rho_1 = \bigvee_{j\in[1..n]} (\mathcal{Z}_n = j) \wedge J_j^e \wedge \rho_s \wedge \mathtt{Z}' \wedge (\mathcal{Z}'_n = j \oplus 1),$$

$$\rho_2(j) = \bigvee_{r>1} \mathtt{mY}[j][r] \wedge \neg \mathtt{mY}[j][{\prec}r] \wedge \rho_s \wedge \mathtt{mY}'[j][{\prec}r],$$

$$\rho_2 = \bigvee_{j\in[1..n]} (\mathcal{Z}_n = \mathcal{Z}'_n = j) \wedge \rho_2(j),$$

$$\rho_3(j) = \bigvee_{r} \bigvee_{i\in[1..m]} \mathtt{mX}[j][r][i] \wedge \neg \mathtt{mX}[j][{\prec}(r,i)] \wedge \neg J_i^s \wedge \rho_s \wedge \mathtt{mX}'[j][r][i],$$

$$\rho_3 = \bigvee_{j\in[1..n]} (\mathcal{Z}_n = \mathcal{Z}'_n = j) \wedge \rho_3(j).$$

Fig. 2. The transitions definition.

Let $\mathcal{Z}_n = \{z_0, \ldots, z_k\}$ be a set of Boolean variables that encode a counter ranging over $[1..n]$. We denote by $\mathcal{Z}_n = j$ the variable assignment that encodes the value j. Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets of input and output variables, respectively, and φ be a GR(1) winning condition. Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ be a game structure (where $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$). We show how to construct a fairness-free FDS $\mathcal{D} = \langle \mathcal{V}_\mathcal{D}, \theta_\mathcal{D}, \rho \rangle$, where $\mathcal{V}_\mathcal{D} = \mathcal{V} \cup \mathcal{Z}_n$, such that $\mathcal{D}$ is complete with respect to $\mathcal{X}$. the BDD representing the set of winning states. Following Theorem 4, we set $\theta_\mathcal{D} = \theta_e \rightarrow (\theta_s \wedge \mathcal{Z}_n = 1 \wedge \mathtt{Z})$. Recall, that $\mathtt{Z}$ is the variable representing the winning states for the system (as in Fig. 1). The variable $\mathcal{Z}_n$ is used to store internally which strategy should be applied. The transition relation ρ is $(\rho_e \wedge \mathtt{Z}) \rightarrow (\rho_1 \vee \rho_2 \vee \rho_3)$, where ρ_1, ρ_2, and ρ_3 are formally defined in Fig. 2, and described below.

We use the sets $\mathtt{mY}[j][r]$ and their subsets $\mathtt{mX}[j][r][i]$ to construct the strategies $f_1, \ldots, f_n$ collected for the system, where j ranges over the number of strategies, r ranges over the number of iterations of the least fixpoint at the j^{th} strategy, and i ranges over the number of assumptions. Let $\mathtt{mY}[j][{\prec}r]$ denote the set $\bigcup_{l\in[1..r-1]} \mathtt{mY}[j][l]$. We write $(r', i') \prec (r, i)$ to denote that the pair (r', i') is lexicographically smaller than the pair (r, i). That is, either $r' < r$ or $r' = r$ and $i' < i$. Let $\mathtt{mX}[j][{\prec}(r,i)]$ denote the set $\bigcup_{(r',i')\prec(r,i)} \mathtt{mX}[j][r'][i']$.

Transitions in ρ_1 are taken when a J_j^s state is reached and we change strategy from f_j to $f_{j\oplus 1}$. The counter $\mathcal{Z}_n$ is updated accordingly. Transitions in ρ_2 are taken when we can get closer to a J_j^s state. These transitions go from states in $\mathtt{mY}[j][r]$ to states in $\mathtt{mY}[j][r']$ where $r' < r$. We require that r' is strictly smaller than r to ensure that the phase of the play, where neither the guarantees are satisfied nor the assumptions are violated, is bounded. Note that there is no need to add transitions that start from states in $\mathtt{mY}[j][1]$ to $\rho_2(j)$, because these transitions are already included in ρ_1. The conjunct $\neg\mathtt{mY}[j][{\prec}r]$ appearing in $\rho_2(j)$ ensures that each state is considered once in its minimal entry.

Transitions in ρ_3 start from states $s \in \mathtt{mX}[j][r][i]$ such that $s \models \neg J_i^e$ and take us back to states in $\mathtt{mX}[j][r][i]$. Repeating such a transition forever will also lead to a legitimate computation because it violates the environment requirement of infinitely many visits to J_i^e-states. Again, to avoid redundancies we apply this transition only to states s for which (r, i) are the lexicographically minimal indices such that $s \in \mathtt{mX}[j][r][i]$. The conjuncts $\neg\mathtt{mX}[j][{\prec}(r,i)]$ appearing in transitions $\rho_3(j)$ ensure that each state is considered once in its minimal entry.

Note that the above transition relation can be computed symbolically. We show the JTLV code that symbolically constructs the transition relation of the synthesized FDS in Fig. 3. We denote the resulting controller by `ctrl`. The functionality of all used methods is self-explanatory.

4.2. Minimizing the strategy

In the previous section, we have shown how to create an FDS that implements an LTL goal φ. The set of variables of this FDS includes the given set of input and output variables as well as the 'memory' variables $\mathcal{Z}_n$. This FDS follows a very liberal policy when choosing the next successor in the case of a visit to J_j^s, i.e., it chooses an arbitrary successor in the winning set. In the following, we use this freedom to minimize (symbolically) the resulting FDS. Notice, that our FDS is deterministic with respect to $\mathcal{X} \cup \mathcal{Y}$. That is, for every state and every possible assignment to the variables in $\mathcal{X} \cup \mathcal{Y}$ there exists at most one successor state with this assignment.[8] As $\mathcal{X}$ and $\mathcal{Y}$ and the restrictions on their possible changes are part of the specification, removing transitions seems to be of lesser importance. We concentrate on removing redundant states.

Since we are using the given sets of variables $\mathcal{X}$ and $\mathcal{Y}$ the only possible candidate states for merging are states that agree on the values of variables in $\mathcal{X} \cup \mathcal{Y}$ and disagree on the value of $\mathcal{Z}_n$. If we find two states s and s' such that $\rho(s, s')$, $s|_{\mathcal{X}\cup\mathcal{Y}} = s'|_{\mathcal{X}\cup\mathcal{Y}}$, and $s'|_{\mathcal{Z}_n} = s|_{\mathcal{Z}_n} \oplus 1$, we remove state s. We direct all its incoming arrows to s' and remove its outgoing arrows. Intuitively, we can do that because the specification does not relate to the variable $\mathcal{Z}_n$. Consider a computation where the sequence (s_0, s', s_1) appears and results from separate transitions (s_0, s) and (s', s_1). Consider the case that there is no successor s'_1 of s such that $s'_1|_{\mathcal{X}\cup\mathcal{Y}} = s_1|_{\mathcal{X}\cup\mathcal{Y}}$ and similarly for a predecessor s'_0 of s'. By $s|_{\mathcal{X}\cup\mathcal{Y}} = s'_{\mathcal{X}\cup\mathcal{Y}}$ we conclude

[8] On the other hand, the FDS may be non-deterministic with respect to $\mathcal{X}$. That is, for a given state s and a given assignment $s'_\mathcal{X}$ to $\mathcal{X}$, there may be multiple $s'_\mathcal{Y}$ such that $(s, s'_\mathcal{X}, s'_\mathcal{Y})$ satisfies the transition of $\mathcal{D}$.

```
public void build_symbolic_controller() {
   ctrl = new FDS("symbolic_controller");
   Zn = ctrl.newBDDDomain("Zn", 1, sys.numJ());
   BDD tr12 = sys.trans().and(env.trans());
   for (int j = 1; j <= sys.numJ(); j++) {
      BDD rho1 = (Zn.eq(j)).and(Z).and(sys.Ji(j)).and(tr12)
         .and(next(Z)).and(next(Zn).eq((j % sys.numJ()) + 1));
      ctrl.disjunctTrans(rho1);
   }
   for (int j = 1; j <= sys.numJ(); j++) {
      BDD low = mem.Y(j, 1);
      for (int r = 2; r <= mem.maxr(j); r++) {
         BDD rho2 = (Zn.eq(j)).and(mem.Y(j, r)).and(low.not())
            .and(tr12).and(next(low)).and(next(Zn).eq(j));
         low = low.or(mem.Y(j, r));
         ctrl.disjunctTrans(rho2);
      }
   }
   for (int j = 1; j <= sys.numJ(); j++) {
      BDD low = FALSE;
      for (int r = 2; r <= mem.maxr(j); r++) {
         for (int i = 1; i <= env.numJ(); i++) {
            BDD rho3 = (Zn.eq(j)).and(mem.X(j, r, i))
               .and(low.not()).and(env.Ji(i).not()).and(tr12)
               .and(next(mem.X(j, r, i))).and(next(Zn).eq(j));
            low = low.or(mem.X(j, r, i));
            ctrl.disjunctTrans(rho3);
         }
      }
   }
}
```

Fig. 3. The symbolic construction of the FDS.

that $(s_0, s') \models \rho_e \wedge \rho_s$. Furthermore, if some J is visited in s then the same J is visited in s' and the progress of $\mathcal{Z}_n$ ensures that an infinite computation satisfies all required liveness constraints.

The symbolic implementation of the minimization is given in Fig. 4. The transition `obseq` includes all possible assignments to $\mathcal{V}$ and $\mathcal{V}'$ such that all variables except $\mathcal{Z}_n$ maintain their values. It is enough to consider the transitions from $\mathcal{Z}_n = j$ to $\mathcal{Z}_n = j \oplus 1$ for all j and then from $\mathcal{Z}_n = n$ to $\mathcal{Z}_n = j$ for all j to remove all redundant states. This is because the original transition just allows to increase $\mathcal{Z}_n$ by one.

This minimization can significantly reduce the numbers of states and so lead to smaller explicit-state representations of a program. However, it turns out that the minimization increases the size of the symbolic representation, i.e., the BDDs. Depending on the application, we may want to keep the size of BDDs minimal rather than minimize the FDS. In the next section, we minimize the symbolic representation to reduce the size of the resulting circuit.

4.3. Generating circuits from BDDs

In this section, we describe how to construct a Boolean circuit from the strategy in Section 4.1. A strategy is a BDD over the variables $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}_n$, $\mathcal{X}'$, $\mathcal{Y}'$, and $\mathcal{Z}'_n$ where $\mathcal{X}$ are input variables, $\mathcal{Y}$ are output variables, $\mathcal{Z}_n$ are the variables encoding the memory of the strategy, and the primed versions represent next state variables. The corresponding circuit contains $|\mathcal{X}| + |\mathcal{Y}| + |\mathcal{Z}_n|$ *flipflops* to store the values of the inputs and outputs in the last clock tick as well as the extra memory needed for applying the strategy (see Fig. 5). In every step, the circuit reads the next input values $\mathcal{X}'$ and determines the next output values $\mathcal{Y}'$ (and $\mathcal{Z}'_n$) using combinational logic with inputs $\mathcal{I} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}_n \cup \mathcal{X}'$. Note that the strategy does not prescribe a unique combinational output for every combinational input. In most cases, multiple outputs are possible, in states that do not occur when the system adheres to the strategy, no outputs may be allowed. Both issues need to be solved before translation to a combinational circuit. That is, we have to fix exactly one possible output for every possible value of the flipflops and the inputs.

The extant solution [44] yields a circuit that can generate, for a given input, any output allowed by the strategy. To this end, it uses a set of extra inputs to the combinational logic. Note that this is more general than what we need: a circuit

```
public void reduce() {
   For (j = 1; j <= sys.numJ(); j++)
      reduce_helper(j, (j % sys.numJ()) + 1);
   For (j = 1; j < sys.numJ(); j++)
      reduce_helper(sys.numJ(), j);
}

public void reduce_helper(j, k) {
   BDD init = ctrl.initial();
   BDD trans = ctrl.trans();
   BDD states = (trans.and(obseq).and(Zn.eq(j))
      .and(next(Zn).eq(k))).exist(next(V));

   BDD Znj_states = states.and(Zn.eq(j));
   BDD rm_trans = next(Znj_states).or(Znj_states);
   ctrl.conjunctTrans(rm_trans.not());
   BDD add_trans = (trans.and(next(states))
      .and(next(Zn).eq(j))).exist(Zn);
   ctrl.disjunctTrans(add_trans.and(next(Zn).eq(k)));

   BDD rm_init = states.and(Zn.eq(j));
   ctrl.conjunctIni(rm_init.not());
   BDD add_init = init.and(states).and(Zn.eq(j)).exist(Zn);
   ctrl.disjunctIni(add_init.and(Zn.eq(k)));
}
```

Fig. 4. The symbolic algorithm of the minimization.

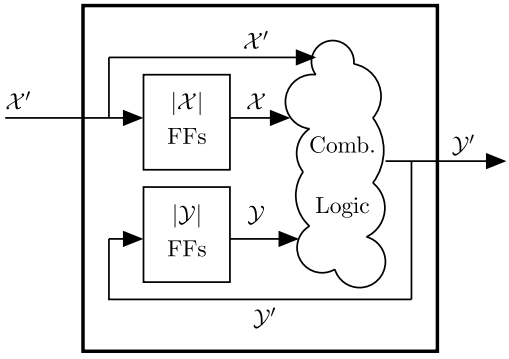


Fig. 5. Diagram of generated circuit.

that always yields one valid output given an input. Our experience shows that this may come at a heavy price in terms of the size of the logic [3].

Due to these scalability problems of [44], we devised the following method to extract a combinational circuit from a BDD that matches our setting. Our method uses the pseudo code shown in Fig. 6.

We write `outputs` and `inputs` to denote the set of all combinational outputs and inputs, respectively. We denote by `set_minus(outputs,y)` the functionality which excludes `y` from the set `outputs`. For every combinational output `y` we construct a function `f_y` in terms of $\mathcal{X}$ that is compatible with the given strategy BDD. The algorithm proceeds through the combinational outputs `y` one by one: First, we build `trans_y` to get a BDD that restricts only `y` in terms of $\mathcal{X}$. Then we build the *positive* and *negative cofactors* (`p,n`) of `trans_y` with respect to `y`, that is, we find the sets of inputs for which `y` can be 1 and the sets of inputs for which `y` can be 0. For the inputs that occur both in the positive cofactor and in the negative cofactor, both 0 and 1 are possible values. The combinational inputs that are neither in the positive cofactor nor in the negative cofactor are outside the winning states and thus represent situations that cannot occur (as long as the environment satisfies the assumptions). Thus, `f_y` has to be 1 in `p` $\cap$ `!n` and 0 in (`!p` $\cap$ `n`), which give us the set of care states. We minimize the positive cofactors with the care set to obtain the function `f_y`. Finally, we substitute variable `y` in `comb` by `f_y`, and proceed with the next variable. The substitution is necessary since a combinational output may also depend on other combinational outputs.

```
public void build_circuit() {
   comb = ctrl.trans();
   for (BDDDomain y : outputs) {
      BDD trans_y = comb.exist(set_minus(outputs, y));
      BDD p = trans_y.and(y.eq(TRUE)).exist(y);
      BDD n = trans_y.and(y.eq(FALSE)).exist(y);
      // (*)
      // NOTE: p and n in general incomparable
      BDD careset = p.and(n.not()).or(p.not().and(n));
      BDD f_y = p.simplify(careset);
      // keep relation between outputs
      comb = comb.relprod(f_y, y);
   }
}
```

Fig. 6. Algorithm to construct a circuit from a BDD.

```
p = p.and(n.not());
n = n.and(p.not());
// where p and n overlap, output can be anything
for (BDDDomain x : inputs) {
   BDD p_x = p.exist(x);
   BDD n_x = n.exist(x);
   if (p_x.and(n_x).isZero()) {
      p = p_x; n = n_x;
   }
}
```

Fig. 7. Extension to algorithm in Fig. 6.

The resulting circuit is constructed by writing the BDDs for the functions using CUDD's DumpBlif command [45].

In the following we describe two extensions that are simple and effective.

4.3.1. Optimizing the cofactors

The algorithm presented in Fig. 6 generates a function in terms of the combinational inputs for every combinational output. Some outputs may not depend on all inputs and we would like to remove unnecessary inputs from the functions. Consider the positive cofactor and the negative cofactor of a variable `y`. If the cofactors do not overlap when we existentially quantify variable `x`, then variable `x` is not needed to distinguish between the states where `y` has to be 1 and where `y` has to be 0. Thus, variable `x` can be simply left out. We adapt the algorithm in Fig. 6 by inserting the code shown in Fig. 7 at the spot marked with `(*)`.

4.3.2. Removing dependent variables

After computing the combinational logic, we perform *dependent variables analysis* [46] on the set of reachable states to simplify the generated circuit.

Definition 1. (See [46].) Given a Boolean function f over $v_0, v_1, \ldots v_n$, a variable v_i is *functionally dependent* in f iff $\forall v_i . f = 0$.

Note that if v_i is functionally dependent, it is uniquely determined by the remaining variables of f. Thus, the value of v_i can be replaced by a function $g(v_0, \ldots v_{i-1}, v_{i+1} \cdots v_n)$.

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

5. LTL Synthesis

In this section we show that the techniques developed in Sections 3 and 4 are strong enough to handle synthesis of LTL specifications in many interesting cases. In particular, the specifications of many hardware designs that we encountered as part of the PROSYD project fall into this category [47]. Given a specification of a realizability problem, we show how to

embed this problem into the framework of GR(1) games. We start from a simple subset of LTL (that is interesting in its own right) and show how to extend the types of specifications that can be handled.

5.1. Implication of Symbolic JDS over input and output variables

We have already argued that in many practical cases the specification calls for a JDS that realizes the environment and a JDS that realizes the system. We suggested to embed the different parts of such specifications into a GR(1) game and defined the strict realizability of the implication of such specifications. Here we highlight the differences between strict realizability of the implication and realizability of the implication; show how to embed the implication of the specifications as a GR(1) game; and (in the following subsections) explain how to extend the fragment of LTL handled by these techniques.

Recall, that the temporal semantics of a JDS $\mathcal{D}$ has the following form:

$$\theta \wedge \Box\rho \wedge \bigwedge_{J\in\mathcal{J}} \Box\Diamond J.$$

Accordingly, when assumptions or guarantees taken together give rise to a JDS they can be arranged as a specification as follows. Let $S_\alpha = \langle \varphi_i^\alpha, \Phi_t^\alpha, \Phi_g^\alpha\rangle$ be a specification, where φ_i^α is an assertion over $\mathcal{V}$, $\Phi_t^\alpha = \{\psi_i\}_{i\in I_t^\alpha}$ is a set of assertions over $\mathcal{V}\cup\mathcal{V}'$, and $\Phi_g^\alpha = \{J_i\}_{i\in I_g^\alpha}$ is a set of assertions over $\mathcal{V}$. Given S_e and S_s of this form, we defined strict realizability in Section 3. Consider now the implication between these two specifications. Formally, let $\varphi_{e,s}^{\rightarrow}$ be the following formula

$$\left(\varphi_i^e \wedge \Box\rho_e \wedge \bigwedge_{i\in I_g^e} \Box\Diamond J_i^e\right) \rightarrow \left(\varphi_i^s \wedge \Box\rho_s \wedge \bigwedge_{i\in I_g^s} \Box\Diamond J_i^s\right), \tag{4}$$

where $\rho_e = \bigwedge_{i\in I_g^e} \psi_i$ and $\rho_s = \bigwedge_{i\in I_g^s} \psi_i$. Namely, $\varphi_{e,s}^{\rightarrow}$ says that if the environment satisfies its specification then the system guarantees to satisfy its specification. The formula $\varphi_{e,s}^{\rightarrow}$ seems simpler and more intuitive than $\varphi_{e,s}^{sr}$. This simplified view, however, leads to dependency between the fulfillment of the systems safety and the liveness of the environment. Thus, specifications that should intuitively be unrealizable turn out to be realizable.

Notice that $\varphi_{e,s}^{sr}$ implies $\varphi_{e,s}^{\rightarrow}$. Thus, if $\varphi_{e,s}^{sr}$ is realizable, a controller for $\varphi_{e,s}^{sr}$ is also a controller for $\varphi_{e,s}^{\rightarrow}$. The following example shows that the other direction is false.

Example 1. Let $\mathcal{X} = \{x\}$ and $\mathcal{Y} = \{y\}$, where both x and y are Boolean variables. Let $S_e = \langle \text{true}, \{\bigcirc x\}, \{x \leftrightarrow y\}\rangle$ and $S_s = \langle \text{true}, \{\bigcirc x \leftrightarrow \bigcirc y\}, \{\neg y\}\rangle$. Intuitively, the environment specification says that the environment should keep x asserted and make sure that x and y are equal infinitely often.[9] The system specification says that the system should keep y equal to x and make sure that y is off infinitely often. Consider the two specifications $\varphi_{e,s}^{\rightarrow}$ and $\varphi_{e,s}^{sr}$,

$$\varphi_{e,s}^{\rightarrow} = \big(\Box \bigcirc x \wedge \Box\Diamond(x \leftrightarrow y)\big) \rightarrow \big(\Box(\bigcirc x \leftrightarrow \bigcirc y) \wedge \Box\Diamond\neg y\big),$$
$$\varphi_{e,s}^{sr} = \Box\big(\boxminus \bigcirc x \rightarrow (\bigcirc x \leftrightarrow \bigcirc y)\big) \wedge \big(\Box \bigcirc x \wedge \Box\Diamond(x \leftrightarrow y) \rightarrow \Box\Diamond\neg y\big).$$

While $\varphi_{e,s}^{\rightarrow}$ is realizable, $\varphi_{e,s}^{sr}$ is unrealizable. Indeed, in the first case, the strategy that always sets y to the inverse of x is a winning strategy. The system may violate its safety but it ensures that the environment cannot fulfill its liveness. On the other hand, $\varphi_{e,s}^{sr}$ is unrealizable. Indeed, as long as the environment does not violate its safety the system has to satisfy safety. An infinite play that satisfies safety will satisfy the liveness of the environment but not of the system. We find that $\varphi_{e,s}^{sr}$ better matches what we want from such a system. Indeed, if the only way for the system to satisfy its specification is by violating its safety requirement, then we would like to know that this is the case. Using $\varphi_{e,s}^{sr}$ and its unrealizability surfaces this problem to the user.

We now contrast two examples.[10]

Example 2. Consider the case where $\mathcal{X} = \{x\}$ and $\mathcal{Y} = \{y\}$ but this time x ranges over $\{1,\ldots,10\}$ and y ranges over $\{1,\ldots,5\}$. Let $S_e = \langle x = 0, \{\bigcirc x > x\}, \{\text{true}\}\rangle$ and $S_s = \langle y = 0, \{\bigcirc y > y\}, \{\text{true}\}\rangle$. Intuitively, both the system and the environment are doomed. Both cannot keep increasing the values of x and y as both variables range over a finite domain. In this case $\varphi_{e,s}^{\rightarrow}$ is realizable and $\varphi_{e,s}^{sr}$ is unrealizable. Dually, if x ranges over $\{1,\ldots,5\}$ and y ranges over $\{1,\ldots,10\}$ both $\varphi_{e,s}^{\rightarrow}$ and $\varphi_{e,s}^{sr}$ are realizable. Again, we find that the behavior of $\varphi_{e,s}^{sr}$ matches better our intuition of what it means to be realizable. Indeed, only when the environment is the first to violate its safety the specification is declared realizable.

[9] This example and the observation that the implication between strict realizability and realizability is only one way is due to M. Roveri, R. Bloem, B. Jobstmann, A. Tchaltsev, and A. Cimatti.

[10] Due to O. Maler.

In general, the kind of dependency that is created in the realizability of $\varphi^{\rightarrow}_{e,s}$ is related to machine closure of specifications [48] (cf. also discussion in [49,50]). In general, we find that specifications that allow this kind of dependency between safety and liveness are not well structured and using strict realizability informs us of such problems.

We now turn to the question of realizability of $\varphi^{\rightarrow}_{e,s}$ and show how to reduce it to the solution of a GR(1) game. Intuitively, we add to the game a memory of whether the system or the environment violate their initial requirements or their safety requirements. Formally, we have the following.

Let $S_\alpha = \langle \varphi^\alpha_i, \Phi^\alpha_t, \Phi^\alpha_g \rangle$ for $\alpha \in \{e, s\}$ be two specifications. The *realizability game for* $\varphi^{\rightarrow}_{e,s}$ is $G^{\rightarrow}_{e,s}$: $\langle \mathcal{V}, \mathcal{X}', \mathcal{Y}', \theta_e, \theta_s, \rho_e, \rho_s, \varphi' \rangle$ with the following components.

- $\mathcal{X}' = \mathcal{X}$.
- $\mathcal{Y}' = \mathcal{Y} \cup \{sf_e, sf_s\}$.
- $\mathcal{V} = \mathcal{X}' \cup \mathcal{Y}'$.
- $\theta_e = \mathsf{true}$.
- $\theta_s = (\varphi^e_i \leftrightarrow sf_e) \wedge (\varphi^s_i \leftrightarrow sf_s)$.
- $\rho_e = \mathsf{true}$.
- $\rho_s = ((\bigwedge_{i \in I^e_t} \psi^e_i \wedge sf_e) \leftrightarrow sf'_e) \wedge ((\bigwedge_{i \in I^s_t} \psi^s_i \wedge sf_s) \leftrightarrow sf'_s)$.
- $\varphi' = (\square \diamondsuit sf_e \wedge \bigwedge_{i \in I^e_g} \square \diamondsuit J^e_i) \rightarrow (\square \diamondsuit sf_s \wedge \bigwedge_{i \in I^s_g} \square \diamondsuit J^s_i)$.

We show that the game $G^{\rightarrow}_{e,s}$ realizes the goal $\varphi^{\rightarrow}_{e,s}$.

Theorem 5. *The game $G^{\rightarrow}_{e,s}$ is won by system iff $\varphi^{\rightarrow}_{s,e}$ is realizable.*

Proof. By Theorem 4 we have that $G^{\rightarrow}_{e,s}$ is won by system iff the following specification ψ^{sr} is realizable

$$\psi^{sr} = \big((\varphi^e_i \leftrightarrow sf_e) \wedge (\varphi^s_i \leftrightarrow sf_s)\big) \wedge \square\Bigg(\Bigg(\Bigg(\bigwedge_{i \in I^e_t} \psi^e_i \wedge sf_e\Bigg) \leftrightarrow sf'_e\Bigg) \wedge \Bigg(\Bigg(\bigwedge_{i \in I^s_t} \psi^s_i \wedge sf_s\Bigg) \leftrightarrow sf'_s\Bigg)\Bigg)$$
$$\wedge \Bigg(\square \diamondsuit sf_e \wedge \bigwedge_{i \in I^e_g} \square \diamondsuit J^e_i\Bigg) \rightarrow \Bigg(\square \diamondsuit sf_s \wedge \bigwedge_{i \in I^s_g} \square \diamondsuit J^s_i\Bigg).$$

Consider an FDS that realizes ψ^{sr}. Let $\sigma\colon s_0, s_1, \ldots$ be a computation of this FDS. We show that $\sigma \models \varphi^{\rightarrow}_{e,s}$. If σ does not satisfy one of the conjuncts on the left-hand side of $\varphi^{\rightarrow}_{e,s}$ then clearly $\sigma \models \varphi^{\rightarrow}_{e,s}$. Assume that σ satisfies all the conjuncts on the left-hand side of $\varphi^{\rightarrow}_{e,s}$. As $\sigma \models \varphi^e_i$ it follows that $\sigma \models sf_e$. As $\sigma \models \square(\bigwedge_{i \in I^e_t} \psi^e_i)$ it follows that $\sigma \models \square sf_e$. As $\sigma \models \bigwedge_{i \in I^e_g} \square \diamondsuit J^e_i$ and clearly $\sigma \models \square \diamondsuit sf_e$ if follows that $\sigma \models \square \diamondsuit sf_s$ and $\sigma \models \bigwedge_{i \in I^s_g} \square \diamondsuit J^s_i$. As there are infinitely many positions where sf_s holds, by using $\square((\bigwedge_{i \in I^s_t} \psi^s_i \wedge sf_s) \leftrightarrow sf'_s)$ we conclude that $\sigma \models \square sf_s$ and $\sigma \models \square(\bigwedge_{i \in I^s_t} \psi^s_i)$. Finally, as $\sigma \models sf_s$ we conclude that $\sigma \models \varphi^s_i$. Thus, σ satisfies all the conjuncts on the right-hand side of $\varphi^{\rightarrow}_{e,s}$ as well.

In the other direction, consider an FDS $\mathcal{D}$ that satisfies $\varphi^{\rightarrow}_{e,s}$. We create the system $\hat{\mathcal{D}}$ by adding to $\mathcal{D}$ the variables sf_e and sf_s and use the augmented initial condition $\hat{\theta} = \theta \wedge \theta_s$ and the augmented transition relation $\hat{\rho} = \rho \wedge \rho_s$. The addition of sf_e and sf_s does not restrict the behavior of $\mathcal{D}$. Furthermore, the values of sf_s and sf_e are determined according to the values of other variables of $\hat{\mathcal{D}}$. Consider a computation $\sigma\colon s_0, s_1, \ldots$ of the augmented system $\hat{\mathcal{D}}$. By definition of $\hat{\mathcal{D}}$ we have $\sigma \models \theta_s$ and $\sigma \models \square((\boxminus \rho_e) \rightarrow \rho_S)$. We have to show $\sigma \models \varphi'$. As σ is also a computation of $\mathcal{D}$ we have $\sigma \models \varphi^{\rightarrow}_{e,s}$.

- Suppose that $\sigma \not\models \varphi^e_i$. Then, $\sigma \models \square \neg sf_e$ and $\sigma \models \varphi'$.
- Suppose that $\sigma \models \varphi^e_i$ and $\sigma \not\models \square \bigwedge_{i \in I^e_t} \psi^e_i$. Then, $\sigma \models \diamondsuit \square \neg sf_e$ and $\sigma \models \varphi'$.
- Suppose that $\sigma \models \varphi^e_i \wedge \square \bigwedge_{i \in I^e_t} \psi^e_i$, and that $\sigma \not\models \bigwedge_{i \in I^e_g} \square \diamondsuit J^e_i$. Then, $\sigma \models \varphi'$ as the left-hand side of the implication does not hold.
- Suppose that σ satisfies all the conjuncts on the left-hand side of $\varphi^{\rightarrow}_{e,s}$. Then, σ also satisfies all the conjuncts on the right-hand side of $\varphi^{\rightarrow}_{e,s}$. It follows that $\sigma \models \varphi^s_i$ implying $\sigma \models sf_s$. It follows that $\sigma \models \square(\bigwedge_{i \in I^s_t} \psi^s_i)$ implying $\sigma \models \square sf_s$. Finally, $\sigma \models \square \diamondsuit sf_s$ and φ' holds as well. □

5.2. Incorporating the past

We have discussed the case where the specification is a combination of assumptions and guarantees of the forms $\square\psi$ and $\square \diamondsuit J$, where ψ is a Boolean formula restricting transitions of the environment or of the system, and J is a Boolean formula. In this subsection we show how to reduce to GR(1) games the case where specifications include parts J and ψ containing past temporal formulas (or temporal patterns that can be translated to past temporal formulas). As before, one

could distinguish between realizability and strict realizability. To simplify presentation we concentrate on strict realizability.

An LTL formula φ is a past formula if it does not use the operators $\bigcirc$ and $\mathcal{U}$. That is, it belongs to the following grammar.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \ominus\varphi \mid \varphi \,\mathcal{S}\, \varphi.$$

For example, the formula $\psi = (\neg g)\,\mathcal{S}\, r$ is a past formula. If r is a request and g is a grant, then ψ holds at time i if there is a pending request in the past that was not granted.

For every LTL formula φ over variables $\mathcal{V}$, one can construct a *temporal tester*, a JDS T_φ, which has a distinguished Boolean variable x_φ such that the following hold.

- T_φ is complete with respect to $\mathcal{V}$.
- For every computation $\sigma\colon s_0, s_1, s_2, \ldots$ of T_φ, $s_i[x_\varphi] = 1$ iff $\sigma, i \models \varphi$.
- For every sequence of states $\sigma\colon s_0, s_1, s_2, \ldots$ there is a corresponding computation $\sigma'\colon s'_0, s'_1, s'_2, \ldots$ of T_φ such that for every i we have s_i and s'_i agree on the interpretation of the variables of φ.

For further details regarding the construction and merits of temporal testers, we refer the reader to [51,52]. It is well known that temporal testers for *past* LTL formulas are fairness-free and *deterministic*. It follows that for every past LTL formula φ, there exists a fairness-free FDS $T_\varphi\colon \langle \mathcal{V}, \theta, \rho\rangle$ such that $\mathcal{V}$ contains the set of variables of φ and θ and ρ are deterministic. Using these fairness-free FDS we can now handle assumptions and guarantees that contain past LTL formulas.

Let $\mathcal{X}$ and $\mathcal{Y}$ be the set of input and output variables, respectively, and let $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$. Consider an assumption or guarantee $\Box\psi$, where ψ is a Boolean combination of past formulas over $\mathcal{V}$ and expressions of the form $\bigcirc v$, where $v \in \mathcal{X}$ if $\Box\psi$ is an assumption, and $v \in \mathcal{X} \cup \mathcal{Y}$ if $\Box\psi$ is a guarantee. For every maximal past temporal formula γ appearing in ψ,[11] there is a temporal tester T_γ with the distinguished variable x_γ. Consider the following specification $\Box\hat{\psi}$, where $\hat{\psi}$ is obtained from ψ by replacing γ by x_γ. It follows that the specification $\Box\hat{\psi}$ is of the required form as in Section 3. Given an assumption or guarantee $\Box\diamondsuit J$, where J is a past formula over $\mathcal{V}$, we treat it in a similar way.

Here we extend the specifications described previously by incorporating referral to past in them. The specifications we consider are

$$S_\alpha = \left\langle \varphi_i^\alpha, \Phi_t^\alpha, \Phi_g^\alpha \right\rangle \quad \text{for } \alpha \in \{e, s\}, \text{ where } \Phi_t^\alpha = \left\{\psi_i^\alpha\right\}_{i\in I_t^\alpha} \text{ and } \Phi_g^\alpha = \left\{J_i^\alpha\right\}_{i\in I_j^\alpha}$$

and ψ_i^α and J_i^α may relate to past formulas.

Given such a specifications S_e and S_s, let $\hat{S}_\alpha$ denote the specification obtained by treating S_α as explained above. Namely, replacing referral to past formulas by referral to the outputs of temporal testers. In particular, let $\hat{\psi}_i^\alpha$ denote the specification obtained from ψ_i^α by replacing maximal past formulas γ by x_γ and similarly for $\hat{J}_i^\alpha$. Let $T_{\gamma_1}, \ldots, T_{\gamma_n}$ be the temporal testers whose variables are used in $\hat{S}_\alpha$ for $\alpha \in \{e, s\}$. The strict realizability game for S_e and S_s is $G_{e,s}^{sr}$: $\langle \mathcal{V} \cup \mathcal{V}_t, \mathcal{X}, \mathcal{Y} \cup \mathcal{V}_t, \theta_e, \theta_s, \rho_e, \rho_s, \varphi\rangle$ with the following components.

- $\mathcal{V}_t$ is the set of variables of $T_{\gamma_1}, \ldots, T_{\gamma_n}$ that are not in $\mathcal{V}$.
- $\theta_e = \varphi_i^e$.
- $\theta_s = \varphi_i^s \wedge \bigwedge_{i=1}^n \theta_i$, where θ_i is the initial condition of T_{γ_i}.
- $\rho_e = \bigwedge_{i\in I_t^e} \hat{\psi}_i^e$.
- $\rho_s = (\bigwedge_{i\in I_t^s} \hat{\psi}_i^s) \wedge (\bigwedge_{i=1}^n \rho_i)$, where ρ_i is the transition of T_{γ_i}.
- $\varphi = \bigwedge_{i\in I_g^e} \Box\diamondsuit \hat{J}_i^e \rightarrow \bigwedge_{i\in I_g^s} \Box\diamondsuit \hat{J}_i^s$.

That is, all the variables of $T_{\gamma_1}, \ldots, T_{\gamma_n}$ are added as variables of the system. The initial condition of the system is extended by all the initial conditions of the temporal testers and the transition of the system is extended by the transitions of the temporal testers. Notice, that variables and transitions of temporal testers that come from assumptions as well as guarantees are added to the system side of the game. This is important as the next state of temporal testers may depend on the next state of both the inputs and the outputs.

Theorem 6. *The game $G_{e,s}^{sr}$ is won by system iff $\varphi_{e,s}^{sr}$ is realizable.*

Notice that $\varphi_{e,s}^{sr}$ uses ψ_i^α and not $\hat{\psi}_i^\alpha$, and similarly for J_i^α.

[11] Subformula γ is a maximal past formula in ψ if every subformula γ' of ψ that contains γ includes the operator $\bigcirc$.

Proof. This follows from Theorem 4 and the correctness of the temporal testers $T_{\psi_1}, \ldots, T_{\psi_n}$. The argument relies on temporal testers for past being fairness-free, deterministic, and complete with respect to $\mathcal{X} \cup \mathcal{Y}$. □

We note that the inclusion of the past allows us to treat many interesting formulas. For example, consider formulas of the form $\square(r \rightarrow \diamondsuit g)$, where r is a request and g a guarantee. As this formula is equivalent to $\square\diamondsuit\neg(\neg g\, \mathcal{S}\, (\neg g \wedge r))$, it is simple to handle using the techniques just introduced. Similarly, $\square(a \wedge \bigcirc b \rightarrow \bigcirc^2 c)$ can be rewritten to $\square(\ominus a \wedge b \rightarrow \bigcirc c)$ and $\square(a \rightarrow a\, \mathcal{U}\, b)$ is equivalent to $\square \boxminus (\ominus(a \wedge \neg b) \rightarrow (a \vee b)) \wedge \square\diamondsuit(\neg a \vee b)$. In practice, many interesting future LTL formulas (that describe deterministic properties) can be rewritten into the required format.

In the next subsection we use deterministic FDS to describe very expressive specifications whose realizability can be reduced to GR(1) games.

5.3. Implication of symbolic JDS

We now proceed to an even more general case of specifications, where each of the assumptions or guarantees is given as (or can be translated to) a deterministic JDS. The main difference between this section and previous sections is in the inclusions of additional variables as part of the given specifications. These variables are then added to the game structure and enable a clean treatment of this kind of specifications. Notice that it is hard to impose strict realizability as there is no clean partition of specifications to safety and liveness.

Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets of typed input and output variables, respectively. In this subsection we consider the case where specifications are given as a set of complete deterministic JDS. Formally, let $S_\alpha = \{D_i^\alpha\}_{i \in I_\alpha}$ for $\alpha \in \{e, s\}$ be a pair of specifications, where $D_i^\alpha = \langle \mathcal{V}_i^\alpha, \theta_i^\alpha, \rho_i^\alpha, \mathcal{J}_i^\alpha \rangle$ is a complete and deterministic JDS with respect to $\mathcal{X} \cup \mathcal{Y}$ for every i and α.

The *realizability game for S_e and S_s* is $G_{e,s}^d$: $\langle \mathcal{V}, \mathcal{X}', \mathcal{Y}', \mathsf{true}, \theta_s, \mathsf{true}, \rho_s, \varphi \rangle$ with the following components.

- $\mathcal{V} = \mathcal{X} \cup \mathcal{Y} \cup (\bigcup_{i \in I_e} \mathcal{V}_i^e) \cup (\bigcup_{i \in I_s} \mathcal{V}_i^s)$.
- $\mathcal{X}' = \mathcal{X}$.
- $\mathcal{Y}' = \mathcal{Y} \cup (\bigcup_{i \in I_e} \mathcal{V}_i^e) \cup (\bigcup_{i \in I_s} \mathcal{V}_i^s)$.
- $\theta_s = (\bigwedge_{i \in I_e} \theta_i^e) \wedge (\bigwedge_{i \in I_s} \theta_i^s)$.
- $\rho_s = (\bigwedge_{i \in I_e} \rho_i^e) \wedge (\bigwedge_{i \in I_s} \rho_i^s)$.
- $\varphi = (\bigwedge_{i \in I^e} (\bigwedge_{J \in \mathcal{J}_i^e} \square\diamondsuit J)) \rightarrow (\bigwedge_{i \in I^s} (\bigwedge_{J \in \mathcal{J}_i^s} \square\diamondsuit J))$.

We show that the game $G_{e,s}^d$ realizes the goal of implication between these sets of deterministic JDS. Let

$$\varphi_{e,s}^d = \bigwedge_{i \in I_e} D_i \rightarrow \bigwedge_{i \in I_s} D_i.$$

We say that $\sigma \models \varphi_{e,s}^d$ if either (i) there exists an $i \in I_e$ such that there is no computation of $\mathcal{D}_i$ that agrees with σ on the variables in $\mathcal{X} \cup \mathcal{Y}$ or (ii) for every $i \in I_s$ there is a computation of $\mathcal{D}_i$ that agrees with σ on the variables in $\mathcal{X} \cup \mathcal{Y}$. The specification $\varphi_{e,s}^d$ is *realizable* if there exists an FDS that is complete with respect to $\mathcal{X}$ that implements this specification.

Theorem 7. *The game $G_{e,s}^d$ is won by system iff $\varphi_{e,s}^d$ is realizable.*

Proof. By Theorem 4 we have that $G_{e,s}^d$ is won by system iff the following specification ψ^{sr} is realizable.

$$\psi^{sr} = \left(\bigwedge_{i \in I_e} \theta_i^e\right) \wedge \left(\bigwedge_{i \in I_s} \theta_i^s\right) \wedge \square\left(\left(\bigwedge_{i \in I_e} \rho_i^e\right) \wedge \left(\bigwedge_{i \in I_s} \rho_i^s\right)\right) \wedge \left(\bigwedge_{i \in I^e}\left(\bigwedge_{J \in \mathcal{J}_i^e} \square\diamondsuit J\right)\right) \rightarrow \left(\bigwedge_{i \in I^s}\left(\bigwedge_{J \in \mathcal{J}_i^s} \square\diamondsuit J\right)\right).$$

Consider an FDS that realizes ψ^{sr}. Let σ: $s_0, s_1, \ldots$ be a computation of this FDS. Let τ: $t_0, t_1, \ldots$ be the computation over $\mathcal{X} \cup \mathcal{Y}$ such that for every $j \geqslant 0$ we have $s_j|_{\mathcal{X} \cup \mathcal{Y}} = t_j$. We show that $\sigma \models \varphi_{e,s}^d$. Consider an FDS $\mathcal{D}_i$. As $\mathcal{D}_i$ is deterministic the assignment to the variables in $\mathcal{V}_i$ in σ is the unique assignment that is possible to accompany τ. It follows that $\tau \models \mathcal{D}_i$ iff the $\sigma \models \bigwedge_{J \in \mathcal{J}_i} \square\diamondsuit J$. It follows that $\sigma \models \varphi_{e,s}^d$.

In the other direction, consider an FDS $\mathcal{D}$ that satisfies $\varphi_{e,s}^d$. From determinism and completeness of $\mathcal{D}_i$ for every i it follows that $\mathcal{D}$ also satisfies ψ^{sr}. □

To summarize, we have presented three possible fragments of specifications and their translation to GR(1) games. In general, when one is presented with specifications in LTL or PSL, a combination of the approaches in the previous sections

should be taken. Simple specifications of the form $\square\psi$ or $\square\diamondsuit J$, where ψ or J are either Boolean formulas or past formulas, should be treated by adding them to the game as explained previously. More complicated specifications should be translated to deterministic JDS and treated by inclusion of the additional variables in this JDS as part of the game. In a sense, the treatment of past formulas and of deterministic JDS is very similar in that it requires the inclusion of additional variables (except the input and the output) in the structure of the game.

For some specifications, it may be impossible to translate them to deterministic JDS. We find that these specifications are not very common. Generalizations of our techniques as presented, e.g., in [23] might be applicable. Otherwise, techniques that handle general LTL formulas may be required [53,27,28,25,24].

6. AMBA AHB case study

We demonstrate the application of the synthesis method by shortly summarizing a case study that we performed on one of the AMBA (*Advanced Microcontroller Bus Architecture*) [21] buses of ARM. More details about this case study can be found in [2]. In order to obtain further insights on the applicability and performance of the method, we refer the interested reader to a second case study [3] based on a tutorial design from IBM.

6.1. Protocol

ARM's *Advanced Microcontroller Bus Architecture* (AMBA) [21] defines the *Advanced High-Performance Bus* (AHB), an on-chip communication standard connecting such devices as processor cores, cache memory, and DMA controllers. Up to 16 *masters* and up to 16 *slaves* can be connected to the bus. The masters initiate communication (read or write) with a slave of their choice. Slaves are passive and can only respond to a request. Master 0 is the *default master* and is selected whenever there are no requests for the bus.

The AHB is a pipelined bus. This means that different masters can be in different stages of communication. At one instant, multiple masters can request the bus, while another master transfers address information, and a yet another master transfers data. A bus *access* can be a single *transfer* or a *burst*, which consists of a specified or unspecified number of transfers. Access to the bus is controlled by the *arbiter*, which is the subject of this section. All devices that are connected to the bus are Moore machines, that is, the reaction of a device to an action at time t can only be seen by the other devices at time $t+1$.

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

We will now introduce the signals used in the AHB. The notation S[n:0] denotes an $(n+1)$-bit signal.

- HBUSREQ[i] – A request from master i to access the bus. Driven by the masters.
- HLOCK[i] – A request from master i to receive a locked (uninterruptible) access to the bus (raised in combination with HBUSREQ[i]). Driven by the masters.
- HMASTER[3:0] – The master that currently owns the address bus (binary encoding). Driven by the arbiter.
- HREADY – High if the slave has finished processing the current data. Change of bus ownership and commencement of transfers only takes place when HREADY is high. Driven by the slave.
- HGRANT[i] – Signals that if HREADY is high, HMASTER $= i$ will hold in the next tick. Driven by the arbiter.
- HMASTLOCK – Indicates that the current master is performing a locked access. If this signal is low, a burst access may be interrupted when the bus is assigned to a different master. Driven by the arbiter

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

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

The list of signals does not contain the data transfer signals as these do not concern the arbiter (ownership of the data bus follows ownership of the address bus in a straightforward manner). Bursts of length 8 or 16 are not taken into account, nor are the different addressing types for bursts. Adding longer bursts only lengthens the specification and the addressing types do not concern the arbiter. Furthermore, as an optional feature of the AHB, a slave is allowed to "split" a burst access and request that it be continued later. We have left this feature out for simplicity, but it can be handled by our approach.

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

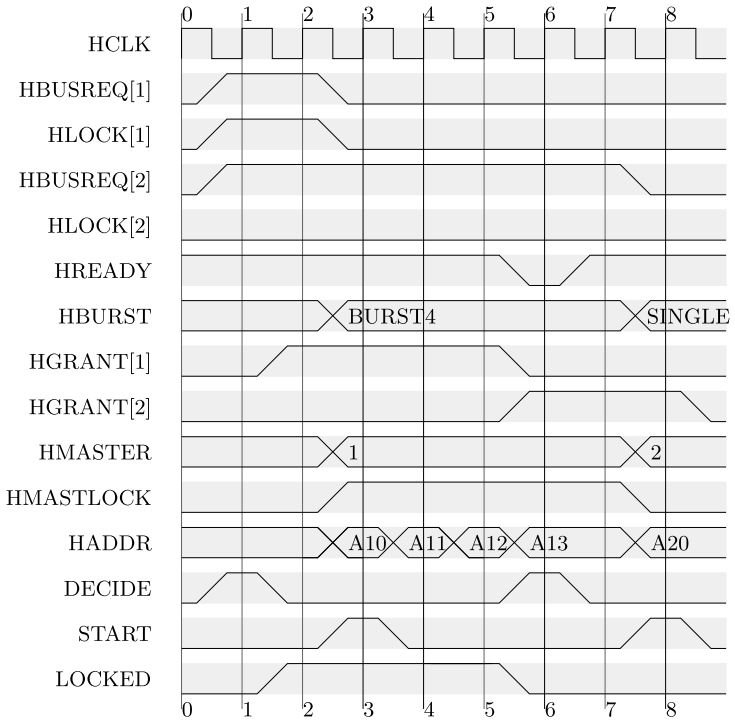


Fig. 8. An example of AMBA bus behavior.

6.2. Formal specification

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

We group the properties into three sets. The first set of properties defines when a new access is allowed to start, the second describes how the bus has to be handed over, and the third describes which decisions the arbiter makes.

All properties are stated using LTL formulas. Some properties are assumptions on the environment, the others are guarantees the system has to satisfy. As explained in Section 5, not all LTL specifications can be synthesized directly using GR(1) games. In order to apply the presented synthesis approach, we aim for a specification that can be expressed using Eq. (4) in Section 5. The separation of the properties into assumptions and guarantees facilitates this translation: the conjunction of all formulas used to describe assumptions form the premiss (left part) of the implication in Eq. (4). Formulas describing guarantees form to the consequent (right part). Now, we only need to ensure that every formula that we use can be mapped into one of the parts of Eq. (4), i.e., into (1) φ_i^x, (2) $\Box\rho_x$, or (3) $\bigwedge_{i\in I_g^x}\Box\diamondsuit J_i^x$ with $x=\{e,s\}$. (Recall that φ_i^x and J_i^x are Boolean formulas over the variables, and ρ_x is a Boolean formula over the variables potentially prefixed with the next operator $\bigcirc$.) Furthermore, note that $\Box\varphi_1\wedge\Box\varphi_2=\Box(\varphi_1\wedge\varphi_2)$ for arbitrary φ_1 and φ_2. Therefore, we can write Part (2) also as conjunction of formulas starting with the always operator $\Box$.

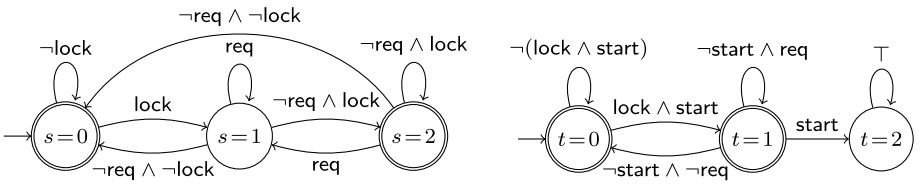
Most formulas we use to describe the desired properties of the arbiter are already in the required format. For the properties (Assumption 1, Guarantees 2 and 3) that are initially not in the right format, we give a corresponding translation.

6.2.1. Starting an access

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

$$\Box\big((\text{HMASTLOCK}\wedge\text{HBURST}=\text{INCR})\rightarrow\bigcirc\diamondsuit\neg\text{HBUSREQ[HMASTER]}\big).$$

The expression HBUSREQ[HMASTER] *is not part of the* LTL *syntax. The formula can be replaced by adding for every master i, the formula* $\Box((\text{HMASTLOCK}\wedge\text{HBURST}=\text{INCR}\wedge\text{HMASTER=i})\rightarrow\bigcirc\diamondsuit\neg\text{HBUSREQ[i]})$. *Alternative, we can introduce a new variable* (*e.g.,* BUSREQ) *and add the following two formulas*:



(a) Assumption (A1.2) (b) Guarantee (2)

Fig. 9. Automata representing Assumption (A1.2) and Guarantee (2). The formulas HMASTLOCK ∧ HBURST = INCR, BUSREQ, and START are abbreviated by lock, req, and start, respectively.

$$\square\Big(\bigwedge_i \text{HMASTER=i} \rightarrow (\text{BUSREQ} \leftrightarrow \text{HBUSREQ[i]})\Big), \tag{A1.1}$$

$$\square\big((\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR}) \rightarrow \bigcirc \Diamond \neg \text{BUSREQ}\big). \tag{A1.2}$$

We chose the latter option, since it made the synthesis computation more efficient.

Assumption (A1.1) is in the right format. We translated Assumption (A1.2) into a deterministic FDS encoding the automaton shown in Fig. 9(a), i.e., we replace Assumption (A1.2) by the three formulas (A1.3), (A1.4), and (A1.5) referring to a new variable s ranging over $\{0, 1, 2\}$. (See Section 5.3 for references on how to obtain this FDS.)

$$s = 0, \tag{A1.3}$$

$$\begin{aligned}
&\square\big(s = 0 \wedge \neg(\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR}) \rightarrow \bigcirc(s = 0)\big) \wedge \\
&\square\big(s = 0 \wedge \text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \rightarrow \bigcirc(s = 1)\big) \wedge \\
&\square\big((s = 1 \vee s = 2) \wedge \text{BUSREQ} \rightarrow \bigcirc(s = 1)\big) \wedge \\
&\square\big((s = 1 \vee s = 2) \wedge \neg\text{BUSREQ} \wedge \\
&\qquad \text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \rightarrow \bigcirc(s = 2)\big) \wedge \\
&\square\big((s = 1 \vee s = 2) \wedge \neg\text{BUSREQ} \wedge \\
&\qquad \neg(\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR}) \rightarrow \bigcirc(s = 0)\big),
\end{aligned} \tag{A1.4}$$

$$\square \Diamond (s = 0 \vee s = 2). \tag{A1.5}$$

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

$$\square \Diamond \text{HREADY}. \tag{A2}$$

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

$$\bigwedge_i \square(\text{HLOCK[i]} \rightarrow \text{HBUSREQ[i]}). \tag{A3}$$

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

$$\square\big(\neg\text{HREADY} \rightarrow \bigcirc(\neg\text{START})\big). \tag{G1}$$

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

$$\begin{aligned}
&\square\big((\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{START}) \rightarrow \\
&\quad \bigcirc\big(\neg\text{START}\, \mathcal{W}\, (\neg\text{START} \wedge \neg\text{HBUSREQ[HMASTER]})\big)\big).
\end{aligned}$$

We treat the expression HBUSREQ[HMASTER] *in the same way as in Assumption* 1*, i.e., we use the variable BUSREQ introduced previously and obtain the following formula.*

$$\begin{aligned}
&\square\big((\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{START}) \rightarrow \\
&\quad \bigcirc\big(\neg\text{START}\, \mathcal{W}\, (\neg\text{START} \wedge \neg\text{BUSREQ})\big)\big).
\end{aligned} \tag{G2.1}$$

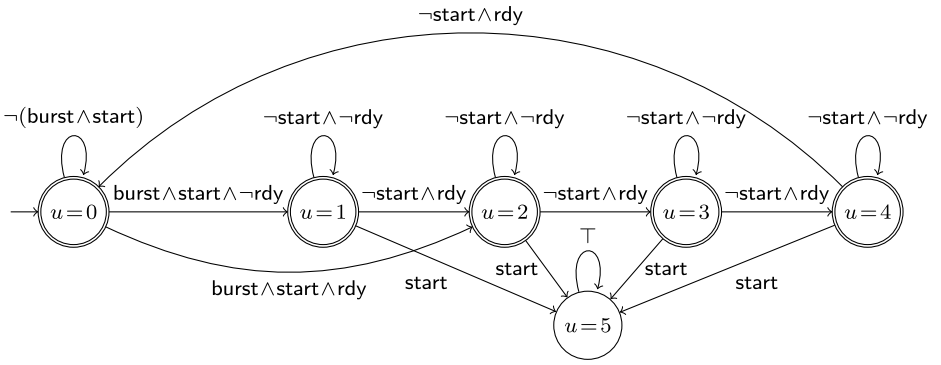


Fig. 10. Automaton encoding Guarantee (G3.1) and (G3.2). We use burst, start, and rdy to abbreviate HMASTLOCK∧HBURST = BURST4, START, and HREADY, respectively.

Guarantee (2) has the form $\square(a \rightarrow \bigcirc(b\,\mathcal{W}\,(b \wedge c)))$, which is equivalent to the past formula $\square(\neg(\neg b \wedge \ominus(\neg c\,\mathcal{S}\,\ominus a)))$. As explained in Section 5.2, for every past LTL formula, there exists a corresponding fairness-free[12] FDS. Fig. 9(b) shows an automaton that encodes Guarantee (G2.1) and corresponds to the FDS that is given by the formulas (G2.2), (G2.3), and (G2.4) referring to the new Boolean variable t.

$$t = 0, \tag{G2.2}$$

$$\begin{aligned}
&\square\big(t = 0 \wedge \neg(\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{START}) \rightarrow \bigcirc(t = 0)\big)\wedge\\
&\square\big(t = 0 \wedge \text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{START} \rightarrow \bigcirc(t = 1)\big)\wedge\\
&\square\big(t = 1 \wedge \neg\text{START} \wedge \neg\text{BUSREQ} \rightarrow \bigcirc(t = 0)\big)\wedge\\
&\square\big(t = 1 \wedge \neg\text{START} \wedge \text{BUSREQ} \rightarrow \bigcirc(t = 1)\big)\wedge\\
&\square\big(t = 1 \wedge \text{START} \rightarrow \bigcirc(t = 2)\big)\\
&\square\big(t = 2 \rightarrow \bigcirc(t = 2)\big),
\end{aligned} \tag{G2.3}$$

$$\square\diamondsuit(t = 0 \vee t = 1). \tag{G2.4}$$

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

$$\begin{aligned}
&\square\big((\text{HMASTLOCK} \wedge \text{HBURST} = \text{BURST4} \wedge \text{START} \wedge \text{HREADY}) \rightarrow\\
&\quad\bigcirc\big(\neg\text{START}\,\mathcal{W}\,[3](\neg\text{START} \wedge \text{HREADY})\big)\big),
\end{aligned} \tag{G3.1}$$

$$\begin{aligned}
&\square\big((\text{HMASTLOCK} \wedge \text{HBURST} = \text{BURST4} \wedge \text{START} \wedge \neg\text{HREADY}) \rightarrow\\
&\quad\bigcirc\big(\neg\text{START}\,\mathcal{W}\,[4](\neg\text{START} \wedge \text{HREADY})\big)\big).
\end{aligned} \tag{G3.2}$$

In order to express Guarantee (G3.1) and (G3.2) in the right format, we translate them into a deterministic FDS in the same way as for Guarantee (G2.1). Fig. 10 shows the automaton this FDS encoding. We use a new variable u, ranging over $\{0, 1, 2, 3, 4, 5\}$, and three formulas (G3.3), (G3.4), and (G3.5) to encode the initial, transition, and final condition of the corresponding FDS, respectively. Since the encoding is done in the same way as the encoding for Assumption (A1.2) and Guarantee (G2.1), we omit the detailed descriptions of (G3.3), (G3.4), and (G3.5).

6.2.2. Granting the bus

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

$$\bigwedge_i \square\big(\text{HREADY} \rightarrow \big(\text{HGRANT[i]} \leftrightarrow \bigcirc(\text{HMASTER} = i)\big)\big). \tag{G4}$$

[12] Note that if we remove the state $t = 2$, which has an empty language, from the automaton shown in Fig. 9(b), then the automaton is fairness-free. However, in order to ensure that the semantics of realizability and strict realizability are the same for our specification (cf. Section 5.1) we give the translation for the complete automaton with fairness.

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

$$\square\big(\text{HREADY} \rightarrow \big(\text{LOCKED} \leftrightarrow \bigcirc(\text{HMASTLOCK})\big)\big). \tag{G5}$$

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

For each master i,

$$\square\big(\bigcirc(\neg\text{START}) \rightarrow \big(\big(\text{HMASTER} = i \leftrightarrow \bigcirc(\text{HMASTER} = i)\big) \wedge \big(\text{HMASTLOCK} \leftrightarrow \bigcirc(\text{HMASTLOCK})\big)\big)\big). \tag{G6}$$

6.2.3. Deciding the next access

Signal DECIDE indicates the time slot in which the arbiter decides who the next master will be, and whether its access will be locked. The decision is based on HBUSREQ[i] and HLOCK[i]. For instance, DECIDE is high in Step 1 and 6 in Fig. 8. Note that a decision is executed at the next START signal, which can occur at the earliest two time steps after the HBUSREQ[i] and HLOCK[i] signals are read. See Fig. 8, the signals are read in Step 1 and the corresponding access starts at Step 3.

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

$$\bigwedge_i \square\big(\big(\text{DECIDE} \wedge \bigcirc(\text{HGRANT[i]})\big) \rightarrow \big(\text{HLOCK[i]} \leftrightarrow \bigcirc(\text{LOCKED})\big)\big). \tag{G7}$$

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

$$\square\big(\neg\text{DECIDE} \rightarrow \bigwedge_i \big(\text{HGRANT[i]} \leftrightarrow \bigcirc(\text{HGRANT[i]})\big)\big) \wedge \square\big(\neg\text{DECIDE} \rightarrow \big(\text{LOCKED} \leftrightarrow \bigcirc(\text{LOCKED})\big)\big). \tag{G8}$$

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

$$\bigwedge_i \square\diamondsuit\big(\neg\text{HBUSREQ[i]} \vee \text{HMASTER} = i\big). \tag{G9}$$

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

$$\bigwedge_{i\neq 0} \big(\neg\text{HGRANT[i]}\, \mathcal{W}\, \text{HBUSREQ[i]}\big), \tag{G10.1}$$

$$\square\big(\big(\text{DECIDE} \wedge \bigwedge_i \neg\text{HBUSREQ[i]}\big) \rightarrow \bigcirc\big(\text{HGRANT[0]}\big)\big). \tag{G10.2}$$

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

$$\text{DECIDE} \wedge \text{START} \wedge \text{HGRANT[0]} \wedge \text{HMASTER} = 0 \wedge \neg\text{HMASTLOCK} \wedge \bigwedge_{i\neq 0} \neg\text{HGRANT[i]}. \tag{G11}$$

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

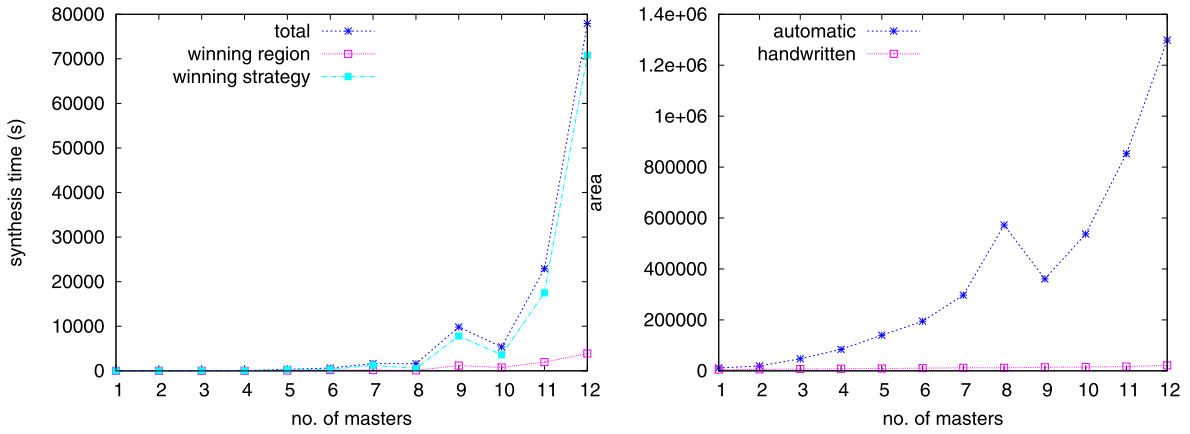
$$\bigwedge_i \big(\neg\text{HBUSREQ[i]} \wedge \neg\text{HLOCK[i]}\big) \wedge \neg\text{HREADY}. \tag{A4}$$

6.3. Synthesis

The final specification in the right form is an implication $\varphi_e \rightarrow \varphi_s$, where φ_e is the conjunction of all the formulas referring to assumptions and φ_s is the conjunction of all the formulas referring to guarantees. In the following we use the equation numbers to abbreviate for the corresponding formulas.

$$\varphi_e = (\text{A1.1}) \wedge (\text{A1.3}) \wedge (\text{A1.4}) \wedge (\text{A1.5}) \wedge (\text{A2}) \wedge (\text{A3}) \wedge (\text{A4}),$$

$$\varphi_s = (\text{G1}) \wedge (\text{G2.2}) \wedge (\text{G2.3}) \wedge (\text{G2.4}) \wedge (\text{G3.3}) \wedge (\text{G3.4}) \wedge (\text{G3.5}) \wedge (\text{G4}) \wedge (\text{G5}) \wedge (\text{G6}) \wedge (\text{G7}) \wedge (\text{G8}) \wedge (\text{G9}) \wedge (\text{G10.1}) \wedge (\text{G10.2}) \wedge (\text{G11}).$$



(a) Time to synthesize AMBA arbiter (b) Size of the AMBA arbiter circuits

Fig. 11. Synthesis of AMBA arbiter results.

Given a specification in the right form, we synthesize a strategy and construct a circuit as described in Section 4.3. Subsequently, the circuit is optimized and mapped to standard cells using ABC [54].

We note that using an extra variable (BUSREQ) for Assumption 1 afforded a considerable increase in capacity of the technique. The time for synthesis is shown in Fig. 11(a) and ranges from a few seconds for 2 masters to about 1.5 h for 10 masters and 21 h for 12 masters. Computing the set of winning states, which allows us to decide if the specification is realizable, takes only a small fraction of the total time. Most of the time is spent in constructing the winning strategy. A more precise analysis showed that our tool spends most of this time reordering BDDs to keep them small. We do not know why synthesis for ten masters is faster than for nine.

In Fig. 11(b), we show the areas of the arbiter as a function of the number of masters using our algorithm compared with a manual implementation. For one master the manual and the automatically generated implementation have approximately the same size. The automatically generated implementations grow rapidly with the number of masters, while the manual implementations are nearly independent of the number of masters. The automatically generated implementation for ten master is about a hundred times larger than the manual implementation. We do not know why size of arbiter for nine masters is smaller than for eight.

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

7. Discussion and conclusions

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

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

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

The effort for and the size of a manual implementation of the AMBA arbiter does not depend much on the number of senders. The same is not true for automatic synthesis: the time to synthesize the arbiter grows quickly with the number of masters as does the size of the generated circuit. Moreover, the size of the system depends strongly on the formulation of the specification. Godhal, Chatterjee, and Henzinger present a formulation of the AHB specification that can be synthesized more efficiently than ours, and present recommendation for writing specifications for synthesis [55].

The gate-level output that our tool produces is complicated and cannot be easily modified manually. The resulting circuit can likely be improved further by using more intelligent methods to generate the circuits, which is an important area for future research. The problem is related to synthesis of partially specified functions [56] with the important characteristic that the space of allowed functions is very large.

On the upside, the resulting specification is short, readable, and easy to modify, much more so than a manual implementation in VERILOG. There is anecdotal evidence that the specification in the form given in this paper can easily be understood

by people with no experience in formal methods: The ARM helpdesk very quickly found some errors in the specification in a preliminary version of this paper.[13] For the arbiter, we expect that it is easier to learn the way the design functions from the formal specification than from a manual VERILOG implementation. The synthesis algorithm was also a great tool to get the specifications to be consistent and complete. We doubt whether we would have gotten to a complete and consistent specification without it.

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

Although this approach removes the need for verification of the resulting circuit, the specification itself still needs to be validated. This is not quite trivial, as the specification is not executable. In our experience, mistakes in the specification are immediately apparent: either the specification becomes unrealizable, or the resulting system behaves nonsensically. Finding the cause, however, is not at all easy. Debugging of specifications has been addressed in [57]. In [58] and [43], methods were developed to extract a core from an unrealizable (or incorrect) specification and to extract a compact explanation of unrealizability. Chatterjee, Henzinger, and Jobstmann consider the modification of unrealizable specifications by making the environment assumptions (minimally) stricter [59].

A need for quantitative specifications to state that an event should happen "as soon as possible," "as infrequently" as possible, etc. was identified in [60], but requires a more expensive synthesis algorithm.

The algorithm presented in this paper has the disadvantage that the resulting system can behave arbitrarily as soon as the environment assumptions are violated. In [61,42], we developed algorithms that synthesize systems that behave "reasonably" in the presence of environment failures.

The algorithm presented here generates synchronous systems. Pnueli and Klein [62] show an incomplete algorithm to reduce asynchronous synthesis [63] of GR(1) properties to the problem of synchronous GR(1) synthesis, making the algorithm presented here applicable to that domain as well.

The work described in this paper has given rise to several implementations. The algorithm is implemented as part of TLV [64] and JTLV [40], as a stand-alone tool called Anzu [4], as a realizability checker in the requirements analysis tool RAT [65] and in the synthesis tool RATSY [66]. RATSY in particular allows for graphical input of the specification automata and contains the debugging algorithm described above.

Finally, our algorithm and its implementation have been used also for applications in robotics and user programming. Kress-Gazit, Conner, et al. use our algorithm to produce robot controllers [67,68]. They combine the discrete controller with continuous control and achieve, for example, controllers for cars that autonomously search for parking. Further, they start exploring domain-specific languages for synthesis of robot controllers [69]. Similar applications are considered in [70–72], where additional effort is exerted to analyze huge state spaces. In the context of user programming our algorithm is used to produce programs from live sequence charts [73,74], and to develop AspectLTL – an aspect-oriented programming language for LTL specifications [75].

References

[1] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, in: Proc. of the 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, in: Lecture Notes in Comput. Sci., vol. 3855, Springer-Verlag, 2006, pp. 364–380.
[2] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, M. Weiglhofer, Automatic hardware synthesis from specifications: A case study, in: Design Automation and Test in Europe, ACM, 2007, pp. 1188–1193.
[3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, M. Weiglhofer, Specify, compile, run: Hardware from PSL, in: 6th Int. Workshop on Compiler Optimization Meets Compiler Verification, in: Electron. Notes Theor. Comput. Sci., vol. 190, 2007, pp. 3–16.
[4] B. Jobstmann, S. Galler, M. Weiglhofer, R. Bloem, Anzu: A tool for property synthesis, in: Proc. of the 19th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 4590, Springer-Verlag, 2007, pp. 258–262.
[5] A. Church, Logic, arithmetic and automata, in: Proc. 1962 Int. Congr. Math. Upsala, 1963, pp. 23–25.
[6] J. Büchi, L. Landweber, Solving sequential conditions by finite-state strategies, Trans. Amer. Math. Soc. 138 (1969) 295–311.
[7] M. Rabin, Automata on Infinite Objects and Churc's Problem, CBMS Reg. Conf. Ser. Math., vol. 13, Amer. Math. Soc., 1972.
[8] A. Pnueli, R. Rosner, On the synthesis of an asynchronous reactive module, in: Proc. of the 16th Int. Colloq. Aut. Lang. Prog., in: Lecture Notes in Comput. Sci., vol. 372, Springer-Verlag, 1989, pp. 652–671.
[9] E. Clarke, E. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Proc. IBM Workshop on Logics of Programs, in: Lecture Notes in Comput. Sci., vol. 131, Springer-Verlag, 1981, pp. 52–71.
[10] Z. Manna, P. Wolper, Synthesis of communicating processes from temporal logic specifications, ACM Trans. Prog. Lang. Syst. 6 (1984) 68–93.
[11] R. Rosner, Modular synthesis of reactive systems, PhD thesis, Weizmann Institute of, Science, 1992.
[12] N. Wallmeier, P. Hütten, W. Thomas, Symbolic synthesis of finite-state controllers for request-response specifications, in: Proceedings of the International Conference on the Implementation and Application of Automata, in: Lecture Notes in Comput. Sci., vol. 2759, Springer-Verlag, 2003, pp. 11–22.
[13] R. Alur, S.L. Torre, Deterministic generators and games for LTL fragments, ACM Trans. Comput. Log. 5 (1) (2004) 1–25.
[14] A. Harding, M. Ryan, P. Schobbens, A new algorithm for strategy synthesis in LTL games, in: Tools and Algorithms for the Construction and the Analysis of Systems, in: Lecture Notes in Comput. Sci., vol. 3440, Springer-Verlag, 2005, pp. 477–492.
[15] B. Jobstmann, A. Griesmayer, R. Bloem, Program repair as a game, in: Proc. of the 17th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 3576, Springer-Verlag, 2005, pp. 226–238.
[16] E. Asarin, O. Maler, A. Pnueli, J. Sifakis, Controller synthesis for timed automata, in: IFAC Symposium on System Structure and Control, Elsevier, 1998, pp. 469–474.
[17] Z. Manna, A. Pnueli, A hierarchy of temporal properties, in: Proc. 9th ACM Symp. Princ. of Dist. Comp., 1990, pp. 377–408.

[13] We take this opportunity to acknowledge the help of Margaret Rugira, Chris Styles, and Colin Campbell at the ARM helpdesk.

[18] Y. Kesten, N. Piterman, A. Pnueli, Bridging the gap between fair simulation and trace inclusion, Inform. and Comput. 200 (1) (2005) 36–61.
[19] R. Bloem, H.N. Gabow, F. Somenzi, An algorithm for strongly connected component analysis in $n \log n$ symbolic steps, Formal Methods Syst. Des. 28 (1) (2006) 37–56.
[20] A. Pnueli, In transition from global to modular temporal reasoning about programs, Logics Models Concurrent Syst. 13 (1985) 123–144.
[21] A. Ltd., AMBA specification (rev. 2), available from www.arm.com, 1999.
[22] B. Jobstmann, R. Bloem, Optimizations for LTL synthesis, in: Proc. of the 6th Int. Conf. on Formal Methods in Computer-Aided Design, IEEE, 2006, pp. 117–124.
[23] S. Sohail, F. Somenzi, K. Ravi, A hybrid algorithm for LTL games, in: Proc. of the 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, in: Lecture Notes in Comput. Sci., vol. 4905, Springer-Verlag, 2008, pp. 309–323.
[24] S. Sohail, F. Somenzi, Safety first: A two-stage algorithm for LTL games, in: Proc. of the 9th Int. Conf. on Formal Methods in Computer-Aided Design, IEEE, 2009, pp. 77–84.
[25] T. Henzinger, N. Piterman, Solving games without determinization, in: Proc. of the 15th Annual Conf. of the European Association for Computer Science Logic, in: Lecture Notes in Comput. Sci., vol. 4207, Springer-Verlag, 2006, pp. 394–410.
[26] A. Morgenstern, Symbolic controller synthesis for LTL specifications, PhD thesis, Universität Kaiserslautern, 2010.
[27] O. Kupferman, M. Vardi, Safraless decision procedures, in: Proc. of the 46th IEEE Symp. on Foundations of Computer Science, 2005, pp. 531–542.
[28] O. Kupferman, N. Piterman, M. Vardi, Safraless compositional synthesis, in: Proc. of the 18th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 4144, Springer-Verlag, 2006, pp. 31–44.
[29] S. Schewe, Bounded synthesis, in: Automated Technology for Verification and Analysis, 2007, pp. 474–488.
[30] E. Filiot, N. Jin, J.-F. Raskin, An antichain algorithm for ltl realizability, in: Proc. of the 21st Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 5643, Springer-Verlag, 2009, pp. 263–277.
[31] C. Eisner, D. Fisman, A Practical Introduction to PSL, Springer-Verlag, 2006.
[32] Y. Kesten, A. Pnueli, Verification by augmented finitary abstraction, Inform. and Comput. 163 (2000) 203–243.
[33] A. Pnueli, R. Rosner, Distributed reactive systems are hard to synthesize, in: Proc. of the 31st IEEE Symp. Found. of Comp. Sci., 1990, pp. 746–757.
[34] D. Kozen, Results on the propositional μ-calculus, Theoret. Comput. Sci. 27 (1983) 333–354.
[35] E.A. Emerson, C.L. Lei, Efficient model-checking in fragments of the propositional modal μ-calculus, in: Proc. of the 1st IEEE Symp. Logic in Comp. Sci., 1986, pp. 267–278.
[36] D. Long, A. Brown, E. Clarke, S. Jha, W. Marrero, An improved algorithm for the evaluation of fixpoint expressions, in: Proc. of the 6th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 818, Springer-Verlag, 1994, pp. 338–350.
[37] M. Jurdziński, Small progress measures for solving parity games, in: Proc. of the 17th Symp. on Theoretical Aspects of Computer Science, in: Lecture Notes in Comput. Sci., vol. 1770, Springer-Verlag, 2000, pp. 290–301.
[38] E. Emerson, Model checking and the μ-calculus, in: N. Immerman, P. Kolaitis (Eds.), Descriptive Complexity and Finite Models, American Mathematical Society, 1997, pp. 185–214.
[39] O. Lichtenstein, Decidability, completeness, and extensions of linear time temporal logic, PhD thesis, Weizmann Institute of Science, 1991.
[40] A. Pnueli, Y. Sa'ar, L.D. Zuck, JTLV: A framework for developing verification algorithms, in: Proc. of the 22nd Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 6174, Springer-Verlag, 2010, pp. 171–174, http://jtlv.ysaar.net/.
[41] S. Juvekar, N. Piterman, Minimizing generalized Büchi automata, in: Proc. of the 18th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 4144, Springer-Verlag, 2006, pp. 45–58.
[42] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, B. Jobstmann, Robustness in the presence of liveness, in: Proc. of the 22nd Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 6174, Springer-Verlag, 2010, pp. 410–424.
[43] R. Koenighofer, G. Hofferek, R. Bloem, Debugging formal specifications using simple counterstrategies, in: Proc. of the 9th Int. Conf. on Formal Methods in Computer-Aided Design, IEEE, 2009, pp. 152–159.
[44] J.H. Kukula, T.R. Shiple, Building circuits from relations, in: Proc. of the 12th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 1855, Springer-Verlag, 2000, pp. 113–123.
[45] F. Somenzi, CUDD: CU Decision Diagram package, University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.
[46] A.J. Hu, D. Dill, Reducing BDD size by exploiting functional dependencies, in: Proc. of the Design Automation Conference, Dallas, TX, 1993, pp. 266–271.
[47] Prosyd – Property-Based System Design, http://www.prosyd.org/, EU grant 507219, 2004–2007.
[48] M. Abadi, L. Lamport, The existence of refinement mappings, Theoret. Comput. Sci. 82 (2) (1991) 253–284.
[49] F. Dederichs, R. Weber, Safety and liveness from a methodological point of view, Inform. Process. Lett. 36 (1) (1990) 25–30.
[50] M. Abadi, B. Alpern, K.R. Apt, N. Francez, S. Katz, L. Lamport, F.B. Schneider, Preserving liveness: Comments on "safety and liveness from a methodological point of view", Inform. Process. Lett. 40 (3) (1991) 141–142.
[51] Y. Kesten, A. Pnueli, L. Raviv, Algorithmic verification of linear temporal logic specifications, in: Proc. of the 25th Int. Colloq. Aut. Lang. Prog., in: Lecture Notes in Comput. Sci., vol. 1443, Springer-Verlag, 1998, pp. 1–16.
[52] A. Pnueli, A. Zaks, On the merits of temporal testers, in: 25 Years of Model Checking, in: Lecture Notes in Comput. Sci., vol. 5000, Springer-Verlag, 2008, pp. 172–195.
[53] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proc. of the 16th ACM Symp. Princ. of Prog. Lang., 1989, pp. 179–190.
[54] B.L. Synthesis, V. Group, Abc: A system for sequential synthesis and verification, release 61208, http://www.eecs.berkeley.edu/~alanmi/abc/.
[55] Y. Godhal, K. Chatterjee, T.A. Henzinger, Synthesis of AMBA AHB from formal specification, Tech. Rep. abs/1001.2811, CORR, 2010.
[56] G.D. Hachtel, F. Somenzi, Logic Synthesis and Verification Algorithms, Kluwer Academic Publishers, Boston, MA, 1996.
[57] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, A. Cimatti, Formal analysis of hardware requirements, in: Proc. of the Design Automation Conference, 2006, pp. 821–826.
[58] A. Cimatti, M. Roveri, V. Schuppan, A. Tchaltsev, Diagnostic information for realizability, in: Proc. of the 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, in: Lecture Notes in Comput. Sci., vol. 4905, Springer-Verlag, 2008, pp. 52–67.
[59] K. Chatterjee, T. Henzinger, B. Jobstmann, Environment assumptions for synthesis, in: Int. Conf. on Concurrency Theory (CONCUR), in: Lecture Notes in Comput. Sci., vol. 5201, Springer-Verlag, 2008, pp. 147–161.
[60] R. Bloem, K. Chatterjee, T. Henzinger, B. Jobstmann, Better quality in synthesis through quantitative objectives, in: Proc. of the 21st Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 5643, Springer-Verlag, 2009, pp. 140–156.
[61] R. Bloem, K. Greimel, T. Henzinger, B. Jobstmann, Synthesizing robust systems, in: Proc. of the 9th Int. Conf. on Formal Methods in Computer-Aided Design, IEEE, 2009, pp. 85–92.
[62] A. Pnueli, U. Klein, Synthesis of programs from temporal property specifications, in: Proc. Formal Methods and Models for Co-Design (MEMOCODE), IEEE, 2009, pp. 1–7.
[63] M. Abadi, L. Lamport, P. Wolper, Realizable and unrealizable specifications of reactive systems, in: Proc. of the 16th Int. Colloq. Aut. Lang. Prog., in: Lecture Notes in Comput. Sci., vol. 372, Springer-Verlag, 1989, pp. 1–17.
[64] A. Pnueli, E. Shahar, A platform for combining deductive with algorithmic verification, in: Proc. of the 8th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 1102, Springer-Verlag, 1996, pp. 184–195.

[65] R. Bloem, R. Cavada, I. Pill, M. Roveri, A. Tchaltsev, Rat: A tool for the formal analysis of requirements, in: Proc. of the 19th Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 4590, Springer-Verlag, 2007, pp. 263–267.
[66] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Koenighofer, M. Roveri, V. Schuppan, R. Seeber, RATSY — a new requirements analysis tool with synthesis, in: Proc. of the 22nd Int. Conf. on Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 6174, Springer-Verlag, 2010, pp. 425–429.
[67] H. Kress-Gazit, G.E. Fainekos, G.J. Pappas, Where's waldo? sensor-based temporal logic motion planning, in: Conf. on Robotics and Automation, IEEE, 2007, pp. 3116–3121.
[68] D.C. Conner, H. Kress-Gazit, H. Choset, A.A. Rizzi, G.J. Pappas, Valet parking without a valet, in: Conf. on Intelligent Robots and Systems, IEEE, 2007, pp. 572–577.
[69] H. Kress-Gazit, G. Fainekos, G. Pappas, From structured English to robot motion, in: Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IEEE, 2007, pp. 2717–2722.
[70] T. Wongpiromsarn, U. Topcu, R.M. Murray, Receding horizon temporal logic planning for dynamical systems, in: Proc. of the 48th IEEE Conf. on Decision and Control, IEEE, 2009, pp. 5997–6004.
[71] T. Wongpiromsarn, U. Topcu, R.M. Murray, Receding horizon control for temporal logic specifications, in: Proc. of the 13th ACM Int. Conf. on Hybrid Systems: Computation and Control, ACM, 2010, pp. 101–110.
[72] T. Wongpiromsarn, U. Topcu, R.M. Murray, Automatic synthesis of robust embedded control software, in: In AAAI Spring Symposium on Embedded Reasoning: Intelligence in Embedded Systems, 2010, pp. 104–110.
[73] H. Kugler, C. Plock, A. Pnueli, Controller synthesis from LSC requirements, in: Proc. Fundamental Approaches to Software Engineering, in: Lecture Notes in Comput. Sci., vol. 5503, Springer-Verlag, 2009, pp. 79–93.
[74] H. Kugler, I. Segall, Compositional synthesis of reactive systems from live sequence chart specifications, in: Proc. of the 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Comput. Sci., vol. 5505, Springer-Verlag, 2009, pp. 77–91.
[75] S. Maoz, Y. Sa'ar, Aspectltl: an aspect language for ltl specifications, in: Proc. of the 10th Int. Conf. on Aspect-Oriented Software Development, ACM, 2011, pp. 19–30.