SYNT 2017

6th Workshop on Synthesis
July 22, 2017 (collocated with CAV 2017)
PRE-Proceedings

The papers in the present volume are to be presented at SYNT 2017 on July 22, 2017. They are not final versions of papers. When referencing these papers, please consult the EPTCS POST-proceedings of SYNT 2017 available from `http://www.eptcs.org/`

Contents

Synthesis Challenges in Building a Multi-Robot Task Server (Keynote Talk)

Rupak Majumdar
MPI-SWS
rupak@mpi-sws.org

In this talk, I will talk about synthesis challenges that arose in our attempts to build Antlab, an end-to-end system that takes streams of user task requests and executes them using collections of robots. In Antlab, each request is specified declaratively in linear temporal logic extended with quantifiers over robots. The user does not program robots individually, nor know how many robots are available at any time or the precise state of the robots. The Antlab runtime system manages the set of robots, schedules robots to perform tasks, automatically synthesizes robot motion plans from the task specification, and manages the co-ordinated execution of the plan.

We are using Antlab as a vehicle to test out different ideas in synthesis. I will describe a repeated re-planning and dynamic conflict resolution algorithm and its hierarchical version. On the theoretical side, I will describe a game problem with dynamic and stochastic rewards which get discounted over time. I will conclude with our unfinished attempts to understand the "right" notion of compositionality in synthesis.

This talk represents joint work with Rayna Dimitrova, Ivan Gavran, Adrian Leva, Kaushik Mallik, Thomas Moor, Vinayak Prabhu, Indranil Saha, Anne-Kathrin Schmuck, Sadegh Soudjani, and Damien Zufferey.

To appear in EPTCS.

Quantitative Assume Guarantee Synthesis (Invited Talk)

Jan Oliver Ringert
Tel Aviv University
ringert@post.tau.ac.il

In assume-guarantee synthesis, we are given a specification (A,G), describing an assumption on the environment and a guarantee for the system, and we construct a system that interacts with an environment and is guaranteed to satisfy G whenever the environment satisfies A. While assume-guarantee synthesis is 2EXPTIME-complete for specifications in LTL, researchers have identified the GR(1) fragment of LTL, which supports assume-guarantee reasoning and for which synthesis has an efficient symbolic solution. In recent years we see a transition to quantitative synthesis, in which the specification formalism is multi-valued and the goal is to generate high-quality systems, namely ones that maximize the satisfaction value of the specification.

We study quantitative assume-guarantee synthesis. We start with specifications in LTL[F], an extension of LTL by quality operators. The satisfaction value of an LTL[F] formula is a real value in [0,1], where the higher the value is, the higher is the quality in which the computation satisfies the specification.

We define the quantitative extension GR(1)[F] of GR(1). We show that the implication relation, which is at the heart of assume-guarantee reasoning, has two natural semantics in the quantitative setting. Indeed, in addition to max1-A,G, which is the multi-valued counterpart of Boolean implication, there are settings in which maximizing the ratio G/A is more appropriate. We show that GR(1)[F] formulas in both semantics are hard to synthesize. Still, in the implication semantics, we can reduce GR(1)[F] synthesis to GR(1) synthesis and apply its efficient symbolic algorithm. For the ratio semantics, we present a sound approximation, which can also be solved efficiently.

This talk presents joint work with Shaull Almagor, Orna Kupferman, and Yaron Velner, published at CAV 2017.

To appear in EPTCS.

Synthesizing Universally-Quantified Inductive Invariants (Invited Talk)

Sharon Shoham
Tel Aviv University
sharon.shoham@gmail.com

A fundamental approach for safety verification is the use of inductive invariants — properties that hold initially, imply the safety property, and are preserved by every step of the system. A common practice is to model a system using logical formulas, and then come up with an inductive invariant in the form of a logical formula. For infinite-state systems (such as programs that manipulate dynamic memory, or distributed algorithms that are designed to run on any number of nodes), these formulas are in many cases quantified. In this talk I will discuss recent approaches for synthesizing inductive invariants in the form of universally-quantified formulas in uninterpreted first-order logic. The key idea is to generalize from concrete counterexamples to induction into universally-quantified clauses based on the logical notion of a diagram.

To appear in EPTCS.

SyGuS Techniques in the Core of an SMT Solver (Invited Talk)

Andrew Reynolds
University of Iowa
andrew.j.reynolds@gmail.com

Recent work has shown that SMT solvers, instead of just acting as subroutines for synthesis tasks, can be instrumented to perform synthesis themselves. This talk will overview two techniques for synthesis conjectures in SMT solvers. The first is based on first-order quantifier instantiation, and can be used to tackle a restricted but fairly common class of properties, known as single invocation properties. The second relies on a deep embedding of the synthesis problem into the theory of inductive datatypes, which can then be solved using enumerative syntax-guided techniques. This talk will discuss current challenges for these two techniques. For the first, we describe challenges for devising quantifier instantiation techniques for synthesis for new background theories and ways of minimizing solutions. For the second, we describe how a DPLL(T)-based solver can perform enumerative syntax-guided search while incorporating techniques that prune search space, including those are specialized for programming-by-examples.

To appear in EPTCS.

CTL* Synthesis via LTL Synthesis

Roderick Bloem[1], Sven Schewe[2], Ayrat Khalimov[1] *

[1] Graz University of Technology, Austria
[2] University of Liverpool, UK

Abstract. We reduce synthesis for CTL* properties to synthesis for LTL. In the context of model checking this is impossible, but in synthesis we can choose how a system should look: in particularly, we can add new outputs. This way, we can construct an LTL formula, over old and new outputs, and original inputs, which is realizable if and only if the original CTL* property is realizable. Thus we can use existing LTL synthesizers to solve CTL* synthesis problem.

1 Introduction

In reactive synthesis we automatically derive a system from a given specification in some temporal logic. Originally, Alonzo Church considered S1S logic [4], then later Amir Pnueli introduced linear temporal logic (LTL) [11].

Recent years showed a great progress in the LTL synthesis. The first solutions used Safra construction [13,10] to translate a nondeterministic Büchi automaton expressing a given LTL formula into a (deterministic) game. The strategy to win the game, if exists, can be used to construct a system that satisfies the original LTL formula. This approach, namely the Safra construction, is difficult to implement efficiently. Safraless decision methods [9,14] try to overcome the difficult construction and go via universal co-Büchi automata. Rougly, they reduce the LTL synthesis problem to safety LTL synthesis, which is amenable to efficient implementations. In a similar spirit, a restricted fragment of LTL, GR(1) [3] is amenable to efficient implementations. Furthemore, there is the synthesis competetion SYNTCOMP [6], where synthesisers compete on (i) safety LTL expressed as AIGER circuits [2], and (ii) full LTL.

All this is about *linear* temporal logics, while less progress was made for synthesis from branching logics. The branching logics can express important properties unexpressible in linear logics, like resettability: "from every state there is a way to get to the 'reset' state". Also, CTL* is the first step towards logics like ATL useful for verifying and synthesizing concurrent systems, which are hard to implement correctly.

* The authors-order was decided by the fair coin.

A few works explored synthesisers for branching logics [7,8,5,12]. In [8], the authors encode CTL synthesis problem into so-called monothonic SAT, while in [7], the authors encoded CTL* synthesis into SMT. In both papers, the authors developed specialized encoders for CTL* synthesis.

Since LTL synthesizers gain efficiency, one might ask

Can we re-use optimized state-of-the-art LTL synthesizers to solve CTL* synthesis problem?

In this paper we positively answer this question by providing sound and complete encoding of CTL* synthesis problem into LTL synthesis.

As it often happens, the above question was not our original motivation. Originally, we studied Bounded Synthesis for CTL* [7], in which we search for systems of increasing size until the upper bound is reached. Due to 2EXPTIME completeness of the CTL* synthesis problem, the upper bound is doubly exponential in the size of the formula, and is impractical to reach. This diminishes the usefulness of the bounded synthesis for unrealisable CTL* specifications, raising the question:

Is there an efficient way to check unrealisability of CTL* properties?

Note that the approach that is usually taken in the LTL synthesis, namely synthesizing the negated specification (thus a model of environment), does not work for CTL*. Take for example, the property $\mathsf{A}g$ where g is the output. Both $\mathsf{A}g$ and $\mathsf{E}\neg g$ are realizable. Our translation from CTL* synthesis to LTL synthesis is "if and only if". Thus we can translate CTL* synthesis into LTL synthesis problem and then run known unrealisability checks on the LTL synthesis problem. Although this does not seem to be very efficient due to the size and structure of LTL formulas, it can still be faster than iterating all system sizes up to $2^{2^{|\Phi|}}$.

2 Definitions

Notation: $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ is the set of Boolean values, $\mathbb{N}$ is the set of natural numbers (excluding 0), $[i, j]$ for integers $i \leq j$ is the set $\{i, ..., j\}$, $[k]$ is $[1, k]$ for $k \in \mathbb{N}$. By default, we use natural numbers. Also, for an arbitrary set I, the calligraphic writing $\mathcal{I}$ denotes 2^I.

2.1 Systems and Automata

In this paper we consider *finite* systems and automata.

A *(Moore) system* M is a tuple $(I, O, T, t_0, \tau, out)$ where I and O are disjoint sets of input and output variables, T is the set of states, $t_0 \in T$ is the initial state, $\tau : T \times 2^I \to T$ is a transition function, $out : T \to 2^O$ is the output function that labels each state with a set of output variables. Note that systems have no dead ends and have a transition for every input. We write $t \xrightarrow{io} t'$ when $t' = \tau(t, i)$ and $out(t) = o$; let $\mathcal{I} \triangleq 2^I$ and $\mathcal{O} \triangleq 2^O$.

For the rest of the section, fix a system $M = (I, O, T, t_0, \tau, out)$.

A *system path* is a sequence $t_1 t_2 ... \in T^\omega$ such that for every i there is $e \in \mathcal{I}$ with $\tau(t_i, e) = t_{i+1}$. An *input-labeled system path* is a sequence $(t_1, e_1)(t_2, e_2)... \in (T \times \mathcal{I})^\omega$ where $\tau(t_i, e_i) = t_{i+1}$ for every i. A *system trace starting from* $t_1 \in T$ is a sequence $(o_1 \cup e_1)(o_2 \cup e_2)... \in (\mathcal{I} \cup \mathcal{O})^\omega$ for which there exists an input-labeled system path $(t_1, e_1)(t_2, e_2)...$ and $o_i = out(t_i)$ for every i. Note that since systems are Moore, the output o_i cannot "react" to input e_i, the outputs are "delayed" with respect to inputs.

A *(word) automaton* A is a tuple $(\Sigma, Q, Q_0, \delta, acc)$ where Σ is an alphabet, Q is a set of states, $Q_0 \subseteq Q$ are initial states, $\delta : Q \times \Sigma \to 2^Q$ is a transition relation, $acc : Q^\omega \to \mathbb{B}$ is a path acceptance condition. Note that automata have no dead ends and have a transition for every letter of the alphabet.

For the rest of the section, fix automaton $A = (\Sigma, Q, Q_0, \delta, acc)$ with $\Sigma = 2^{I \cup O}$.

A path in automaton A is a sequence $q_1 ... \in Q^\omega$ starting in an initial state such that there exists $a_i \in \Sigma$ for every i such that $(q_i, a_i, q_{i+1}) \in \delta(q_i)$. *A sequence* $a_1 ... \in \Sigma^\omega$ *generates a path* $\pi = q_1 \dots$ iff for every i: $(q_i, a_i, q_{i+1}) \in \delta$. A *path* π *is accepted* iff $acc(\pi)$ holds.

We define two acceptance conditions. For a given sequence $\pi \in Q^\omega$, let $Inf(\pi)$ be the elements of Q appearing in π infinitely often and let $Fin(\pi) = Q \setminus Inf(\pi)$. Then:

- *Büchi acceptance* is defined by a set $F \subseteq Q$: $acc(\pi)$ holds iff $Inf(\pi) \cap F \neq \emptyset$.
- *Co-Büchi acceptance* is defined by a set $F \subseteq Q$: $acc(\pi)$ holds iff $F \subseteq Fin(\pi)$.

We distinguish two types of automata: universal and non-deterministic. The type defines when the automaton accepts a given infinite sequence. A *nondeterministic automaton A accepts a sequence* from Σ^ω iff there exists an accepted path generated by the sequence. Universal automata require *all* paths generated by the sequence to be accepted. We write $L(A)$ for the set of all infinite sequences accepted by A.

We distinguish between two *path quantifiers*, E and A: $M \models \mathsf{E}(A)$ iff there is a system trace $(o_0 \cup e_0)(o_1 \cup e_1)...$ accepted by the automaton; $M \models \mathsf{A}(A)$ iff every system trace is accepted by the automaton.

The *product* $M \times A$ is the automaton $(Q \times T, Q_0 \times \{t_0\}, \Delta, acc')$ such that for all $(q, t) \in Q \times T$: $\Delta(q, t) = \{(\delta(q, i \cup out(t)), \tau(q, i)) \mid i \in \mathcal{I}\}$. Define acc' to return true for a given $\pi \in (Q \times T)^\omega$ iff acc returns true for the corresponding projection of π into Q. Note that $M \times A$ has the 1-letter alphabet (not shown in the tuple).

Abbreviations. NBA means nondeterministic Büchi automaton, UCA—universal co-Büchi automaton.

2.2 CTL* with Inputs (release PNF)

For this section, fix a system $M = (I, O, T, t_0, \tau, out)$. Below we define CTL* with inputs (in release positive normal form). The definition differentiates inputs and outputs (see Remark 1) and is specific to Moore machines.

Syntax of CTL* with inputs. *State formulas* have the grammar:

$$\Phi = \mathsf{true} \mid \mathsf{false} \mid o \mid \neg o \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathsf{A}\varphi \mid \mathsf{E}\varphi$$

where $o \in O$ and φ is a path formula. *Path formulas* are defined by the grammar:

$$\varphi = \Phi \mid i \mid \neg i \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi,$$

where $i \in I$. The temporal operators G and F are defined as usual.

The above grammar describes the CTL* formulas in positive normal form. The general CTL* formula (in which negations can appear anywhere) can be converted into the formula of this form with no size blowup, using equivalence $\neg(a\,\mathsf{U}\,b) \equiv \neg a\,\mathsf{R}\,\neg b$.

Semantics of CTL* with inputs. We define the semantics of CTL* with respect to a system M. The definition is very similar to the standard one [1], except for a few cases involving inputs (marked with "+").

Let $t \in T$ and $o \in O$. Then:

- $t \not\models \Phi$ iff $t \models \Phi$ does not hold
- $t \models \mathsf{true}$ and $t \not\models \mathsf{false}$
- $t \models o$ iff $o \in out(t)$, $t \models \neg o$ iff $o \notin out(t)$
- $t \models \Phi_1 \wedge \Phi_2$ iff $t \models \Phi_1$ and $t \models \Phi_2$. Similarly for $\Phi_1 \vee \Phi_2$.
+ $t \models \mathsf{A}\varphi$ iff for all *input-labeled* system paths π starting from t: $\pi \models \varphi$. For $\mathsf{E}\varphi$, replace "for all" with "there exists".

Let $\pi = (t_1, e_1)(t_2, e_2)... \in (T \times 2^I)^\omega$ be an input-labeled system path and $i \in I$. For $k \in \mathbb{N}$, define $\pi_{[k:]} = (t_k, e_k)...$, i.e., the suffix of π starting from (t_k, e_k). Then:

- $\pi \models \Phi$ iff $t_1 \models \Phi$
+ $\pi \models i$ iff $i \in e_1$, $\pi \models \neg i$ iff $i \notin e_1$
- $\pi \models \varphi_1 \wedge \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$. Similarly for $\varphi_1 \vee \varphi_2$.
- $\pi \models \mathsf{X}\,\varphi$ iff $\pi_{[2:]} \models \varphi$
- $\pi \models \varphi_1\,\mathsf{U}\,\varphi_2$ iff $\exists l \in \mathbb{N} : (\pi_{[l:]} \models \varphi_2 \wedge \forall m \in [1, l-1] : \pi_{[m:]} \models \varphi_1)$
- $\pi \models \varphi_1\,\mathsf{R}\,\varphi_2$ iff $(\forall l \in \mathbb{N} : \pi_{[l:]} \models \varphi_2) \vee (\exists l \in \mathbb{N} : \pi_{[l:]} \models \varphi_1 \wedge \forall m \in [1, l] : \pi_{[m:]} \models \varphi_2)$

A system M satisfies a CTL* state formula Φ, written $M \models \Phi$, iff the initial state satisfies it.

Remark 1 (Subtleties). Note that $M \models i \wedge o$ is not defined, since $i \wedge o$ is not a state formula. Let $r \in I$ and $g \in O$. By the semantics, $\mathsf{E}r \equiv \mathsf{true}$ and $\mathsf{E}\neg r \equiv \mathsf{true}$, while $\mathsf{E}g \equiv g$ and $\mathsf{E}\neg g \equiv \neg g$. Similar claims hold for a Boolean combination of inputs or that of outputs.

3 Converting CTL* to LTL for Synthesis

3.1 Reducing CTL* synthesis to LTL synthesis

Let us first look at a standard algorithm for CTL* synthesis. When synthesising a system that realises a CTL* specification, we normally

- Turn the CTL* formula to an alternating tree automaton A.
- We move from computation trees to annotated computation trees that move the (memoryless) strategy of the verifier[1] into the label of the computation tree. This allows for using the derived universal tree automaton U.
- We determinise U to a deterministic automaton D.
- We play an emptiness game for D.
- If the verifier wins, his winning strategy (after projection of the additional labels) defines a system, if the spoiler wins, the specification is unrealisable.

We draw from this construction and use particular properties of the alternating automaton A. The properties of A that we use are that it is not a general alternating automaton, but one that is composed from universal and existential word automata. These universal and existential word automata start at any system state [tree node] where a universally and existentially, respectively, quantified subformula is marked as true in the annotated model [annotated computation tree]. We use the term "existential word automata" to emphasise that the automaton is not only a non-deterministic word automaton, but it is also used in the alternating automaton in a way, where the verifier can pick the path along which it has to accept.

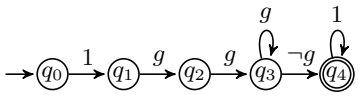
Example 1 (Word and tree automata). Consider formula $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. Figure 1 shows non-deterministic word automata for the subformulas, and the alternating (actually, nondeterministic) tree automaton for the whole formula.

Given a computation tree, the verifier moves from a node according to the alternation of the automaton (either in all directions or in one direction) and maps each node to an automaton state. The decision, in which direction to move and which automaton state to pick, constitutes the strategy of the verifier. Each time the verifier has to move in several tree directions (this can happen when the node is annotated with a universal automaton state), we spawn a new version of the verifier, for each transition of the universal automaton.

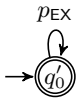
The strategy of the verifier is simply a mapping of states of the existential word automata to the selected decisions, which consist of the tree direction (the continuation of the tree path along which the automaton shall accept) and the automaton transition. This is a mapping $dec : Q \to \mathcal{I} \times Q$ such that $dec(q) = (I, q')$ implies that $q' \in \delta(q; O, I)$ (where δ corresponds to the existential word automaton to which q belongs)[2]. This strategy is memoryless wrt. the history of automata states.

[1] Such a strategy maps, in each tree node, an automaton state to a next automaton state and direction.

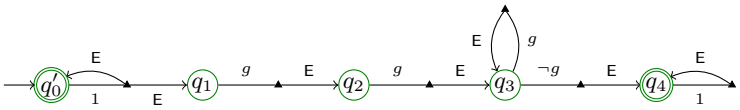
[2] The verifier, when in the tree node or system state, moves according to this strategy.



(a) NBW for $\mathsf{X}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. Transitions to the non-final state *sink* are not shown.



(b) NBW for $\mathsf{G}(p_{\mathsf{EX}})$. The transition to the non-final state *sink* is omitted.



(c) Alternating tree automaton for $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. The green color of the states indicate that they have the Büchi acceptance condition. To get an alternating automaton for $\mathsf{AG}\,\mathsf{EX}(...)$, replace in the self-loop edge of q_0' label E with A, and make the state non-final co-Büchi.

Fig. 1: Word and tree automata.

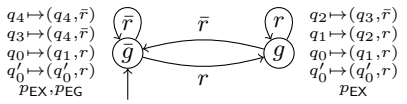
We call a model in which every state is additionally annotated with a verifier strategy an *annotated model*. Similarly, an *annotated computation tree* is a computation tree in which every node is additionally annotated with a verifier strategy. Thus, in both cases, every system state [node] is labeled with: (i) original propositional labeling $out : \mathcal{O} \to \mathbb{B}$, (ii) propositional labeling for (universal and existential) subformulas, $f\_prop : F \to \mathbb{B}$, and (iii) decision labeling $dec : Q \to \mathcal{I} \times Q$ where Q are the states of all existential automata.

Example 2. Figure 2 shows an annotated model and computation tree.

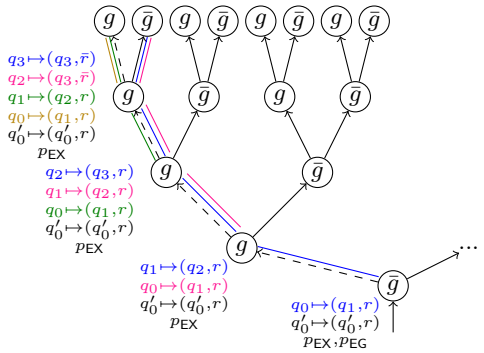
The verifier strategy (encoded in the annotated computation tree) encodes both, the words on which the nondeterministic automata are interpreted and witnesses of acceptance (accepting automata paths on those words). For the encoding in LTL, we will later use that it is enough to map out the automaton word, and replace the witnesses by what it actually means: that the automaton word satisfies the respective path formula.

Example 3. In Figure 2b, the verifier strategy in the root node maps out the word $(\bar{g}, p_{\mathsf{EX}}, r)(\bar{g}, p_{\mathsf{EX}}, \bar{r})^\omega$ on which the NBW in Figure 1b is run, and the witness of acceptance $(q_0')^\omega$. The blue path encodes the word $(\bar{g}, r)(g, r)(g, r)(g, \bar{r})(\bar{g}, \bar{r})^\omega$ and the witness $q_0 q_1 q_2 q_3 q_3 q_4^\omega$ for the NBW in Figure 1a.

To map out the word, we look at the set of tree paths that is mapped out in an annotated computation tree and define equivalence classes on them. Two tree paths are *equivalent* if they share a tail (or, equivalently, if one is the tail of the other).



(a) An annotated model satisfying $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. Near the nodes is the winning strategy of the verifier.



(b) An annotated computation tree that satisfies $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. Let proposition p_{EG} correspond to $\mathsf{EG}(p_{\mathsf{EX}})$, and p_{EX}—to $\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$ (not shown in the figure). A winning strategy for the verifier is depicted using dashed and colored paths. The black dashed path witnesses p_{EG}, the blue path witnesses p_{EX} starting in the root node, the pink path—starting in the left child, and so on. The pink and blue paths share the tail. The verifier strategy annotation is on the left side of nodes, and decisions for non mapped states are irrelevant. Note that this particular annotated computation tree is *not* and unfolding of the annotated model above—here we postpone the right-turn of the blue path in order to illustrate that paths can share the tail.

Fig. 2: Annotated model and computation tree.

There is a simple sufficient condition for two tree paths to be equivalent: if they pass through the same node of the annotated computation tree in the same automaton state, then they have the same future, and are therefore equivalent.

Example 4. In Figure 2b the blue and pink paths are equivalent, since they share the tail. The sufficient condition fires in the top node, where the paths meet in automaton state q_3

The sufficient condition implies that we cannot have more non-equivalent tree paths passing through a node than there are states in all existential automata, call this number k. For each node, we assign unique numbers from $\{1, ..., k\}$ to equivalence classes, and thus any two non-equivalent paths that go through the same node have different numbers. As this is an intermediate step in our translation, we are wasteful with the labeling:

(1) we map states to numbers (IDs) using a label $id : Q \to \{1, \ldots, k\}$, we choose the direction to take $dir : \{1, \ldots, k\} \to \mathcal{I}$, and choose the successor state, $succ : Q \to Q$, such that $succ(q) \in \delta\Big(q; O, dir\big(id(q)\big)\Big)$, *and*

(2) we maintain the same state ID along the chosen direction: $id(q) = id(succ(q))$.

Note that (1) alone can be viewed as a re-phrasing of the labeling *dec* that we had before on page 5. The requirement (2) is satisfiable, because a tree path maintains its equivalence class. Therefore any annotated computation tree can be re-labeled as stated.

Example 5. A re-labeled computation tree is in Figure 3.

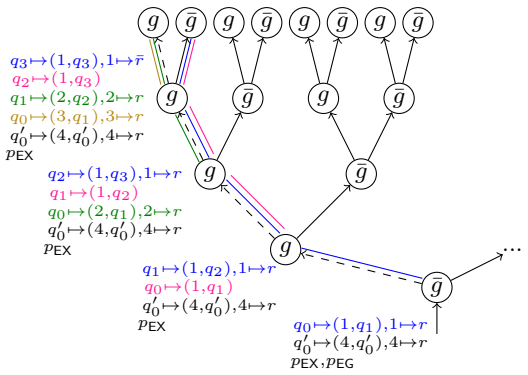


Fig. 3: A re-labeled computation tree. Notation "$q_0 \mapsto (1, q_1)$" means $id(q_0) = 1$ and $succ(q_0) = q_1$, and "$1 \mapsto r$" means $dir(1) = r$. Since the blue and pink paths are equivalent, the label *id* maps the corresponding automata states in the nodes to the same number, 1. The IDs of the green and yellow paths differ implying that they are not equivalent and hence do not share the tail (their tails cannot be seen).

In the new annotation with labels $(out, f\_prop, id, dir, succ)$, labeling *dir* alone maps out the tree path for each ID, the remainder of the information is mainly there to establish that the corresponding word is accepted by the respective word automaton (equiv.: satisfies the respective path formula). If we use only *dir*, then the only missing information is where the path starts and which path formula it belongs to—the information originally encoded by f_prop.

We address these two points by using *numbered* computation trees. Recall that the annotated computation trees have a *propositional* labeling $f\_prop : F \to \mathbb{B}$ that labels nodes with subformulas. In the numbered computation trees, we replace f_prop for *existential* subformulas by labeling $v : F \to \{0, ..., k\}$, where value 0 encodes that no claim that the path formula holds is made in a node (similar to the proposition being "false" in the annotated tree), whereas a value

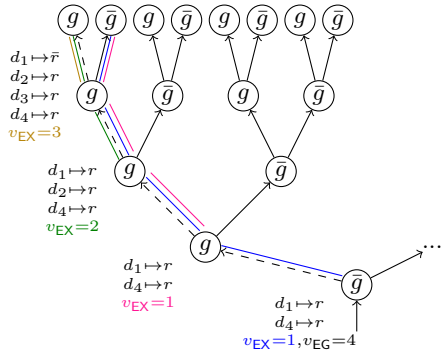


Fig. 4: Numbered computation tree after stripping annotations.

$ID \in \{1, ..., k\}$ is interpreted similarly to the proposition being "true", but also requires that a respective witness is encoded on the tree path mapped out by ID.

Example 6. The tree in Figure 3 becomes a numbered computation tree if we replace the propositional labels p_{EX} and p_{EG} with ID numbers as follows. The root node and its left child have $v_{\mathsf{EX}} = 1$ and $v_{\mathsf{EG}} = 4$, the left-left child has $v_{\mathsf{EX}} = 2$ and $v_{\mathsf{EG}} = 4$, the left-left-left child has $v_{\mathsf{EX}} = 3$ and $v_{\mathsf{EG}} = 4$. Note that $id(q_0) = v_{\mathsf{EX}}$ and $id(q'_0) = v_{\mathsf{EG}}$ whenever those vs are non-zero. The nodes outside of the dashed path have $v_{\mathsf{EX}} = v_{\mathsf{EG}} = 0$, meaning that no claims about satisfaction of the path formulas has to be witnessed there.

Initially, we use ID labeling v in addition with $(out, id, dir, succ, f\_prop_{univ})$, and then there is no relevant change in the automata with which the decision-less verifier works. I.e., a numbered computation tree can be turned into annotated computation tree, and vice versa, such that the numbered tree is accepted by the corresponding automaton iff the annotated tree is accepted by the corresponding automaton.

Once we have done this, we can observe that the only use of the existential automata that is left is to help to check that the tree paths mapped out by dir are indeed models of the respective existentially quantified LTL formula (see Figure 4). Thus, we can replace this acceptance check by, instead, requiring that the labeling of the path is a model of this LTL formula. What is more, this is easy to encode in LTL:

– For each existentially quantified path formula $\mathsf{E}\varphi$, we introduce an integer value between 0 and k, and we label system states with such values. The value $v_{\mathsf{E}\varphi} = 0$ encodes that we do *not* claim that $\mathsf{E}\varphi$ holds at this state. The value $v_{\mathsf{E}\varphi} = j \neq 0$ means that $\mathsf{E}\varphi$ holds in this state, and the system path along the j-labeled directions encodes a witness. Thus, for each $\mathsf{E}\varphi$, we get

an LTL formula

$$\bigwedge_{j\in\{1,\ldots,k\}} \mathsf{G}\Big[v_{\mathsf{E}\varphi} = j \;\rightarrow\; \big(\mathsf{G}\, d_j \rightarrow \varphi'\big)\Big], \tag{1}$$

where φ' is obtained from φ by replacing the subformulas of the form $\mathsf{E}\psi$ by $v_{\mathsf{E}\psi} \neq 0$, and the subformulas of the form $\mathsf{A}\psi$ by $p_{\mathsf{A}\psi}$.

– For each subformula of the form $\mathsf{A}\varphi$, we simply take

$$\mathsf{G}\Big[p_{\mathsf{A}\varphi} \;\rightarrow\; \varphi'\Big], \tag{2}$$

where φ' is obtained from φ as above.

Thus, the overall LTL formula is the conjunction $\bigwedge_{\mathsf{E}\varphi}$ Eq.1 $\wedge \bigwedge_{\mathsf{A}\varphi}$ Eq.2.

The whole above discussion leads us to the theorem.

Theorem 1. *Let Φ_{LTL} be derived from a given Φ_{CTL^*} as described above, $\mathcal{I}$ is the set of inputs and $\mathcal{O}$ is the set outputs. Then:*

$$\Phi_{CTL^*} \text{ is realizable} \quad\Leftrightarrow\quad \Phi_{LTL} \text{ is realizable.}$$

The synthesis costs blow-up is not much worse than before: the individual specifications are not very complex (tiny blow-up compared to the initial specification). We need to satisfy their conjunction—but the size of the resulting UCW is additive in the individual UCWs. Thus we get:

Theorem 2. *Let Φ_{LTL} be derived from a given Φ_{CTL^*} as described above, $\mathcal{I}$ is the set of inputs and $\mathcal{O}$ is the set outputs. Then realizing Φ_{LTL} is of the same complexity class as that of Φ_{CTL^*}, namely, 2EXPTIME-complete.*

While we have emptiness equivalence for sufficiently large k, k is a parameter, where much smaller k might suffice. In the spirit of bounded synthesis, it is possible to use smaller parameters in the hope of finding a model. These models might be of interest in that they guarantee a limited entanglement of different runs, as they cap the number of tails of runs that go through the same node of a computation tree. They are therefore simple in some formal sense, and this sense is independent of the representation by an automaton. (As opposed to a lower bound of a sufficiently high number k, for which we have explicitly used the representation by an automaton.)

Example 7 (LTL translation). Let $I = \{r\}$, $O = \{g\}$. Consider the CTL formula

$$\underbrace{\mathsf{EG}\,\neg g}_{v_2} \wedge \mathsf{AG}\,\underbrace{\mathsf{EF}\,\neg g}_{v_1} \wedge \underbrace{\mathsf{EF}\, g}_{v_3}.$$

The sum of states of individual NBWs is 5 (assuming the natural translations), so we introduce integer propositions v_1, v_2, v_3, all varying over $\{0, ..., 5\}$, and

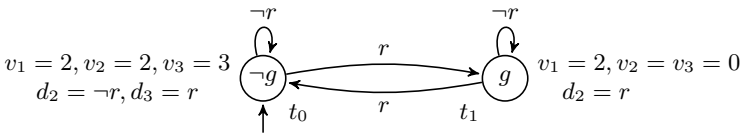


Fig. 5: A Moore machine for Example 7. The witness for $\mathsf{EG}\,\neg g$ (expressed by v_2) is: $v_2(t_0) = 2$ and hence we move along $d_2 = \neg r$ and thus loop in t_0. The witness for $\mathsf{EF}\,g$ (expressed by v_3): since $v_3(t_0) = 3$, we move along $d_3 = r$ to t_1, where d_3 is not restricted, so let us set $d_3 = \neg r$, then the witness for $\mathsf{EF}\,g$ is $t_0(t_1)^\omega$. The witness for $\mathsf{AGEF}\,\neg g$ is that every state should have $v_1 \neq 0$, which is true. In t_0 we have $\neg g$, so $\mathsf{EF}\,\neg g$ is satisfied; for t_1 we have $v_1(t_1) = 2$ hence we move $t_1 \xrightarrow{r} t_0$ and $\mathsf{EF}\,\neg g$ is also satisfied.

five Boolean propositions d_1, ..., d_5. The LTL formula is:

$$v_2 \neq 0 \wedge \mathsf{G}(v_1 \neq 0) \wedge v_3 \neq 0 \wedge \bigwedge_{j\in[1,5]} \begin{pmatrix} v_1 = j \;\Rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{F}\,\neg g) \\ v_2 = j \;\Rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{G}\,\neg g) \\ v_3 = j \;\Rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{F}\,g) \end{pmatrix}$$

Figure 5 shows a model satisfying the LTL specification.

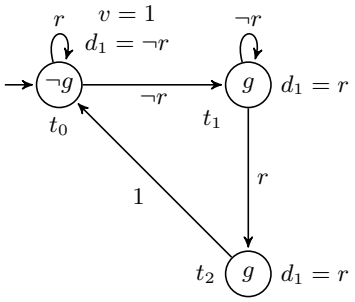
Example 8 (Non-minimality). Let $I = \{r\}$, $O = \{g\}$, and consider the CTL* safety property

$$\mathsf{E}(\mathsf{X}(g \wedge \mathsf{X}(g \wedge \mathsf{X}\,\neg g)))$$

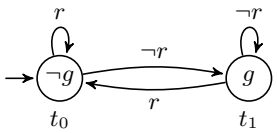
The NBA automaton has 5 states, so we introduce integer proposition v varying over $\{0, ..., 5\}$ and Boolean propositions d_1, d_2, d_3, d_4, d_5. The LTL formula is

$$v \neq 0 \;\wedge \bigwedge_{j\in[1,5]} \Big(v = j \;\Rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{X}(g \wedge \mathsf{X}(g \wedge \mathsf{X}\,\neg g)))\Big)$$

A smallest system for this LTL formula is in Figure 6a. It is of size is 3, while a smallest system for the CTL* property is 2 (Figure 6b). Thus, although the encoding can handle the full CTL*, it might produce non-minimal systems.



(a) A smallest Moore machine satisfying the LTL formula. The witness for $\mathsf{E}(\mathsf{X}(g \wedge \mathsf{X}(g \wedge \mathsf{X}\,\neg g)))$ is $t_0 \xrightarrow{\neg r} t_1 \xrightarrow{r} t_2 \xrightarrow{r} t_0 \ldots$.



(b) A smallest Moore machine satisfying the CTL* formula.

Fig. 6: Systems synthesized from CTL* and LTL (Example 8).

4 Checking Unrealisability of CTL*

Our encoding also allows for checking unrealisability: we do have an LTL formula, and that can be complemented and the game dualised, just like in the standard LTL synthesis approach. But this is problematic. Not only can k be exponential in the size of the CTL* formula, all is multiplicative here. What one could try is to let the new system player in the dualised game choose a number of disjunctive formulas to follow, and allow it to revoke the choice whenever it likes to. This is conservative: if following d different disjuncts in the dualised formula is enough to win, then the new system wins. There does not seem to be good complexity guarantees that go with this, but with a bit of luck that might work. Also, parts of the disjunction might work well; this could then be handled precisely.

Acknowledgements. This work was supported by the Austrian Science Fund (FWF) under the RiSE National Research Network (S11406).

References

1. Baier, C., Katoen, J.P.: Principles of model checking, vol. 26202649. MIT press Cambridge (2008)
2. Biere, A.: Aiger format and toolbox, http://fmv.jku.at/aiger/
3. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences 78, 911–938 (2012)
4. Church, A.: Logic, arithmetic, and automata. In: International Congress of Mathematicians (Stockholm, 1962), pp. 23–35. Institute Mittag-Leffler, Djursholm (1963)
5. De Angelis, E., Pettorossi, A., Proietti, M.: Synthesizing concurrent programs using answer set programming. Fundamenta Informaticae 120(3-4), 205–229 (2012)
6. Jacobs, S., Bloem, R., Brenguier, R., Ehlers, R., Hell, T., Könighofer, R., Pérez, G.A., Raskin, J.F., Ryzhyk, L., Sankur, O., et al.: The first reactive synthesis competition (syntcomp 2014). arXiv preprint arXiv:1506.08726 (2015)

7. Khalimov, A., Bloem, R.: Bounded synthesis for Streett, Rabin, and CTL*. In: Computer Aided Verification - 29th International Conference, CAV 2017 (2017), to appear
8. Klenze, T., Bayless, S., Hu, A.J.: Fast, flexible, and minimal ctl synthesis via smt. In: International Conference on Computer Aided Verification. pp. 136–156. Springer (2016)
9. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS. pp. 531–542 (2005)
10. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: LICS. pp. 255–264. IEEE Computer Society (2006), http://dx.doi.org/10.1109/LICS.2006.28
11. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57. IEEE (1977)
12. Prezza, N.: Ctl (computation tree logic) sat solver, https://github.com/nicolaprezza/CTLSAT
13. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 319–327. IEEE Computer Society (1988), http://dx.doi.org/10.1109/SFCS.1988.21948
14. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Automated Technology for Verification and Analysis (ATVA'07). pp. 474–488 (2007)

Symbolic vs. Bounded Synthesis for Petri Games*

Bernd Finkbeiner
Universität des Saarlandes
Saarbrücken, Germany
finkbeiner@react.uni-saarland.de

Manuel Gieseking
Carl von Ossietzky Universität Oldenburg
Oldenburg, Germany
gieseking@informatik.uni-oldenburg.de

Jesko Hecking-Harbusch
Universität des Saarlandes
Saarbrücken, Germany
hecking-harbusch@react.uni-saarland.de

Ernst-Rüdiger Olderog
Carl von Ossietzky Universität Oldenburg
Oldenburg, Germany
olderog@informatik.uni-oldenburg.de

Petri games are a multiplayer game model for the automatic synthesis of distributed systems. We compare two fundamentally different approaches for solving Petri games. The symbolic approach decides the existence of a winning strategy via a reduction to a two-player game over a finite graph, which in turn is solved by a fixed point iteration based on binary decision diagrams (BDDs). The bounded synthesis approach encodes the existence of a winning strategy, up to a given bound on the size of the strategy, as a quantified Boolean formula (QBF). In this paper, we report on initial experience with a prototype implementation of the bounded synthesis approach. We compare bounded synthesis to the existing implementation of the symbolic approach in the synthesis tool ADAM. We present experimental results on a collection of benchmarks, including one new benchmark family, modeling manufacturing and workflow scenarios with multiple concurrent processes.

1 Introduction

The *synthesis problem* asks, given a formal specification, for the existence of an implementation and derives it automatically if existent [2]. This problem can be described as a game between the system and the environment. A strategy is winning for the system and therefore corresponds to an implementation if the strategy fulfills the winning condition of the game against all behaviors of the environment. The synthesized implementation for a given specification is correct by construction, which is beneficial for error-prone implementation tasks. Synthesis has been applied successfully to implement several practical applications like the protocol of the AMBA bus circuit [1].

Synthesis is especially interesting for *distributed systems* where several concurrent components can communicate with each other to fulfill the specification. Pnueli and Rosner defined the first setting for distributed synthesis [14]. *Information forks* have been identified as a necessary and sufficient criterion for the undecidability of the synthesis problem in this setting [9], which prevents most practical applications. Even for the decidable cases without information forks like pipelines and rings, the synthesis problem has non-elementary complexity [15, 11]. More recently, Zielonka's asynchronous automata were introduced as another framework for distributed synthesis. Whereas the synthesis problem for some cases has non-elementary complexity, the general complexity of synthesis for asynchronous automata remains open [18, 12].

We take Petri games as a starting point to synthesize distributed systems [8]. Petri games define a multiplayer game model where several distributed system players cooperatively play against several

*This work was partially supported by the European Research Council (ERC) Grant OSARES (No. 683300) and by the German Research Foundation through the Research Training Group (DFG GRK 1765) SCARE.

Submitted to:
SYNT 2017

distributed environment players. Petri games are based on an underlying Petri net where each token represents a player. Each player in a Petri game has only local information about other players it synchronized with on joint transitions.

We compare two fundamentally different approaches to synthesize winning strategies of Petri games: The *symbolic* approach is based on the result that the synthesis problem for Petri games with a bounded number of system players, a single environment player, and the avoidance of bad places as winning condition can be solved in single exponential time [8]. This approach performs a reduction to a two-player game over a finite graph with full information. The implementation is realized in the tool ADAM using a BDD-based fixed point iteration [6]. The restriction to only a single environment player is an impediment for convenient modeling.

The second approach to find winning strategies in a Petri game is *bounded* synthesis [10]. Here, the size of possible strategies and of the proof that the strategy is winning is limited. It is increased incrementally when no strategy of the previous size can be found. This steers the search towards small winning strategies at the cost of being unable to prove the non-existence of a winning strategy. Bounded synthesis can find winning strategies for Petri games with more than one environment player [5]. The bounded approach for Petri games limits the number of fired transitions in the proof of the strategy being winning. It further takes a second bound on the size of the memory each player can utilize as the memory of players is infinite in general. For the pair of bounds, bounded synthesis searches for termination or loops in the strategy and can thereby prove the existence of an infinite winning strategy. This search is encoded in a quantified Boolean formula (QBF) and solved by a QBF solver. We report on our experience with a first prototype implementation of bounded synthesis generating the QBF and solving it with the QBF solver QUABS [17] in comparison to the symbolic approach implemented in ADAM.

ADAM has been used to synthesize several case studies from manufacturing and workflow scenarios. These case studies have been extended into scalable benchmark families to evaluate the behavior of the implementation of the symbolic approach and to show the applicability of Petri games to synthesize distributed systems [6].

The key contributions of this paper are the following:

- We add the new benchmark family of a distributed *alarm system* to the benchmark families of ADAM. The new benchmark family serves to secure several locations with an alarm system such that a burglary at any location is indicated at all locations including the information where exactly the burglary takes place.
- We state our experience with the prototype implementation for the generation of the QBFs representing the bounded synthesis problem for the given pair of bounds and with the solving of the generated QBFs. We found out that the bounded unfolding of Petri games benefits from pruning techniques and that solvers for non-CNF QBFs refining an abstraction based on counterexamples show the best performance for solving.
- We empirically compare the symbolic synthesis approach implemented in ADAM with the bounded synthesis approach realized by our prototype implementation. The symbolic approach solves more instances overall from the extended set of benchmark families whereas bounded synthesis derives strategies of smaller size. We present reasons for the observed behavior based on the number of variables in the two approaches.

The remainder of the paper is structured as follows. Section 2 recaps the theory of Petri games on an abstract level and introduces the distributed alarm system as a running example. In Sec. 3, the symbolic and the bounded approach for solving Petri games are presented, including their application to the distributed alarm system. The experimental results comparing the two approaches are given in Sec. 4.

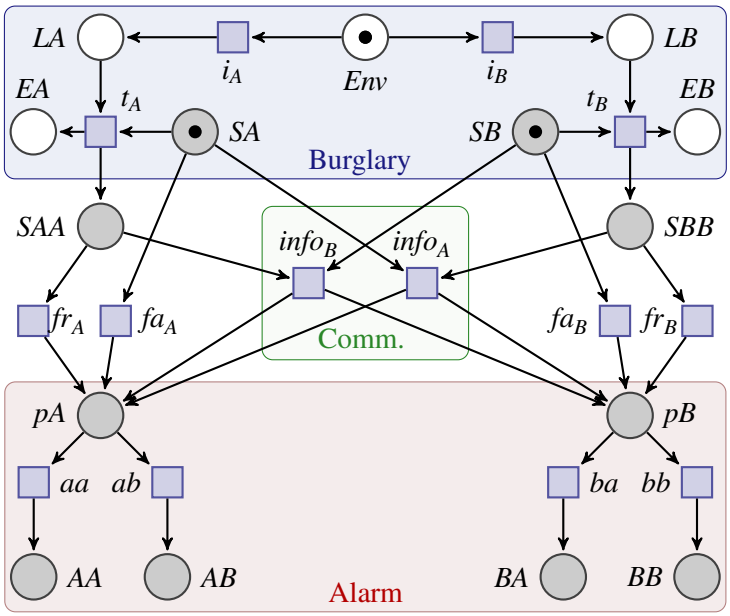


Figure 1: *The Petri game depicting the synthesis problem of an alarm system being distributed over two independent locations with the possibility to communicate. The alarm system has to detect a burglary and set off alarms at both locations.*

2 Petri Games

Petri games are a multiplayer game model for the synthesis of distributed systems [8]. They define a game on an underlying Petri net by characterizing certain places as bad such that the system has to cooperate to avoid reaching these places to win. The distinction between system and environment is achieved by distributing each place of the Petri net to either belong to the system or to the environment. System places are depicted gray whereas environment places remain white. The players in a Petri game are the tokens flowing through the underlying Petri net. If a token resides in a system place then it is controlled by the token's strategy whereas the behavior of tokens in environment places is uncontrollable. Each token in a Petri game has local memory, which is only exchanged with the other participating tokens of joint transitions. The causal history of tokens is utilized by their local strategy to make decisions on which transitions to fire. Therefore, the strategy of the system in a Petri game is a local controller for each process and not a global controller with information about the state of tokens in all system places.

Example. *We model a distributed alarm system. A burglar (modeled by the environment player residing in place Env) decides to intrude one of several secured locations. In Fig. 1, the situation is displayed for the distributed locations A (shown to the left) and B (shown to the right). This situation has been used as the introductory example in [8] and is extended to a benchmark family for an arbitrary number of secured locations later in this paper. The goal of the game is to find a strategy for each of the two alarm processes initially residing in places SA and SB. A token in place XY (for* $X, Y \in \{A, B\}$*) represents that in location X an alarm is set off indicating that the strategy of the alarm system presumes an intrusion at location Y. If the burglar intrudes location A by entering place LA, the strategies should steer the token in SA to place AA and the token in SB to place BA in order to correctly indicate the burglar's intrusion, and similarly for the burglar intruding location B. If a token resides in one of the system places SA, SB, SAA, SBB, pA, or pB then the strategy of the player has to resolve nondeterminism between the outgoing*

transitions. For this, the strategies for the players in SA and SB need to collect sufficient information about the moves of the other system player and the burglar. For example, the player in place SB does not know whether the alarm system in SA has fired transition t_A unless it gets informed by the communication transition $info_B$. We omit the bad places and transitions to them representing false alarms and false reports, which have to be avoided by a correct alarm system. A false alarm occurs when an intrusion is indicated before the burglar actually intruded the object whereas a false report occurs when the alarm system at a certain location indicates an intrusion at a location where no intrusion occurred.

Formally, a Petri game $\mathscr{P} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, In, \mathscr{B})$ is based on a Petri net $\mathscr{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In)$ with the set of places $\mathscr{P} = \mathscr{P}_S \cup \mathscr{P}_E$, the set of transitions $\mathscr{T}$, the flow relation $\mathscr{F} \subseteq (\mathscr{P} \times \mathscr{T}) \cup (\mathscr{T} \times \mathscr{P})$, and the initial marking In. A Petri game divides the places to either belong to the system ($\mathscr{P}_S$) or to the environment ($\mathscr{P}_E$) and it further defines some places as bad ($\mathscr{B} \subseteq \mathscr{P}$). We restrict ourselves to *safe* Petri games, i.e., at most one token can reside in each place. We identify a token residing in a place by the name of the place. A *marking* is a set of places where one token resides at each place. From a marking, a transition is *enabled* if all places in the transition's preset have one token residing in them. The *firing* of an enabled transition removes one token from each place in the transition's preset ($^\bullet t$) and creates one token in each place in the transition's postset ($t^\bullet$). We also refer to the sets of transitions preceding and following a place as the place's preset ($^\bullet p$) and postset ($p^\bullet$). The behavior of Petri games is defined by sequences of *reachable markings*. These sequences start from the initial marking and between each pair of subsequent markings one transition is fired. We say that a place p is *reachable* if there exists a sequence of reachable markings such that one of the markings contains p.

The notion of *unfoldings* from Petri nets [4] translates to Petri games. In an unfolding, all cycles, i.e., all reachable sequences of markings starting and ending with the same marking, are unrolled infinitely and all backward branches, i.e., places with at least two transitions merging into them, are unfolded. A place is unfolded by copying the place and all its outgoing transitions including the following subgames and changing one of the incoming flows of a transition from the place's preset to the copied place. Therefore, the unfolding explicitly represents the unique causal history of each process. As for Petri nets, the unfolding can be infinite for Petri games. In *bounded unfoldings* of Petri games [5], loops and backward branches are only unfolded up to a given bound b for the number of copies per place. Upon reaching the bound for a place, both the original place and its existing copies can be part of a loop or can have backward branch. Therefore, the bounded unfolding is finite even for Petri games with infinite unfoldings. One bounded unfolding for the Petri game in Fig. 1 is depicted in Fig. 5 in Sec. 3.2. The places *AA*, *AB*, *BA*, and *BB* are not unfolded despite them being reachable with different causal pasts, i.e., having backward branches.

A *strategy* of a Petri game is a restriction to the unfolding of the Petri game where certain branches of transitions and places are removed. We assume that this removal is based on the decisions of system players not to fire certain transitions. The behavior of a strategy is defined by the remaining sequences of reachable markings. The following four requirements have to hold for a strategy to be *winning*:

1. *Safety.* No marking containing a bad place is reachable in the strategy.
2. *Determinism.* For all system places in all reachable markings of the strategy, at most one outgoing transition is enabled.
3. *Deadlock avoidance.* For all reachable markings in the strategy, if a transition is enabled for that marking in the unfolding then one transition is enabled in the strategy as well. This requirement prevents trivial solutions where the system does not fire any transition to avoid reaching a marking containing a bad place.

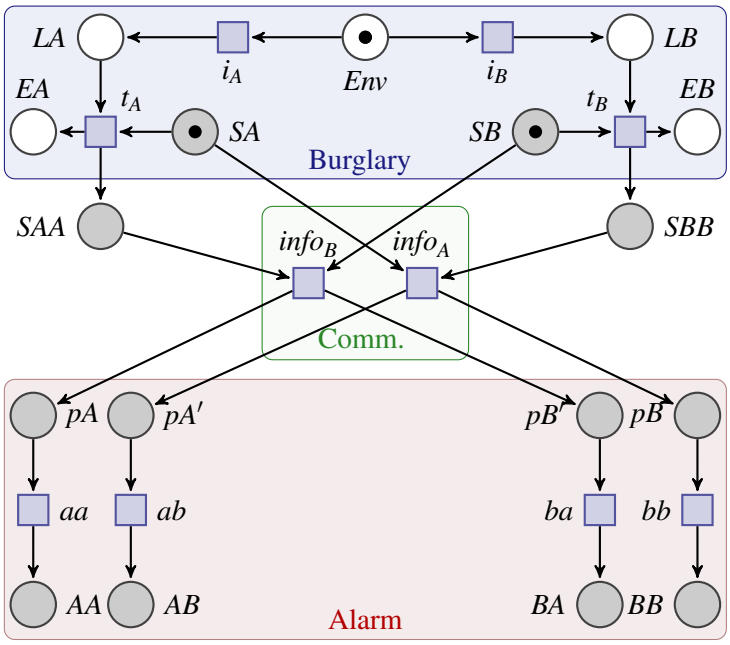


Figure 2: *The winning strategy for the system players for the Petri game depicted in Fig. 1 modeling an alarm system. To win, both system players have to communicate the detection of the burglary before setting off their alarms.*

4. *Justified refusal.* When a transition is removed from the unfolding to create the strategy then there is at least one system place in the removed transition's preset for which all copies of the removed transition are removed as well. This prevents the system from differentiating copies of transitions resulting from the unfolding.

Petri games with a bounded number of system tokens, one environment token, and bad places as winning condition can be solved in single exponential time [8].

Example. *The winning strategy for the alarm system from Fig. 1 is depicted in Fig. 2. The displayed strategy avoids bad places because no false alarm can occur, as the alarm is always triggered after the burglar intruded, and because no false report can occur, as both system tokens exchange information and utilize it to indicate the correct location of intrusion. The strategy is deterministic because only one of the two respective outgoing transitions of SA and SB is enabled, depending on the location of intrusion, and all other system places have only one outgoing transition. The strategy is deadlock-avoiding because after indicating the alarm, the system terminates as no transitions are enabled. The unfolding of the game only allows justified refusal by the system. Therefore, the displayed strategy is winning.*

The strategy contains the local controllers for the alarm systems at location A and at location B. The local controller for the alarm system at A behaves in the following way. The alarm system at SA waits until it either recognizes an intrusion at A via transition t_A or is informed about an intrusion at B via transition $info_A$. The place pA' is reached after getting informed about an intrusion at B from which the transition ab is fired setting off an alarm at A indicating an intrusion at B. The place SAA is reached after recognizing an intrusion at A. The local alarm system informs the local alarm system at B with the transition $info_B$ and afterwards fires the transition aa to reach the place AA. This place represents an alarm at location A that there was an intrusion at location A. The two local system controllers can be found in Fig. 3. Note that they can only behave correctly as they rely on the other local alarm system to

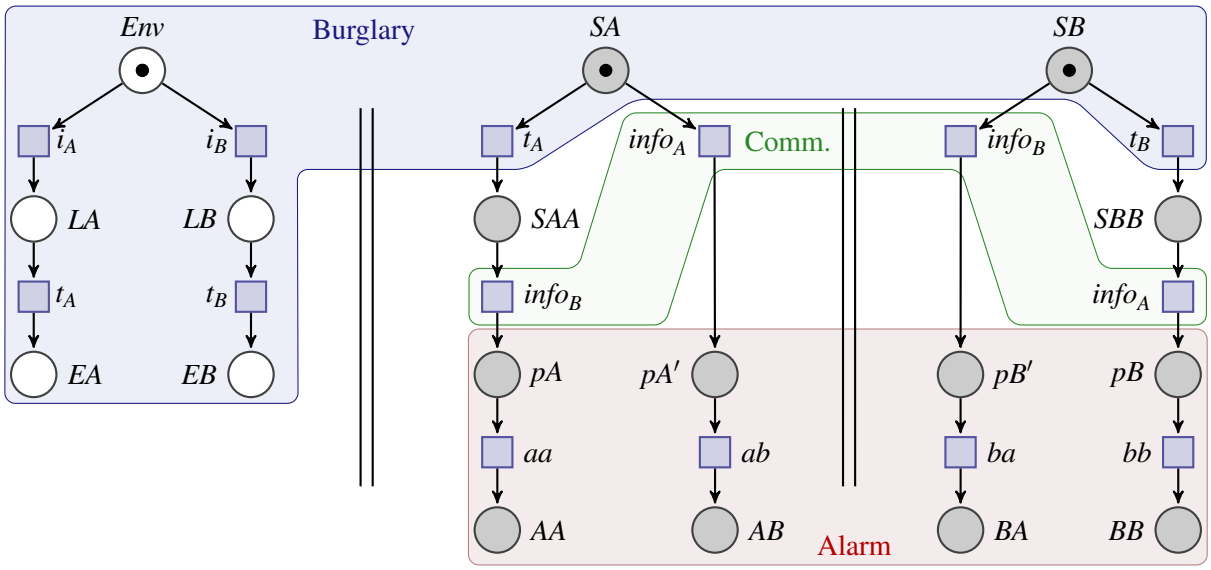


Figure 3: *The distributed local controllers for each player of the winning strategy depicted in Fig. 2. The parallel bars indicate the parallel composition of Petri nets [13] for the environment and the two system controllers, with synchronization on equally labeled transitions. First, the burglary is detected, then the detection is communicated, and afterwards the alarm is set off.*

faithfully inform them about an ongoing burglary, i.e., the system players cooperate. This distribution of a winning strategy into local controllers is possible for all winning strategies of safe, concurrency-preserving Petri games with only one environment player [8].

A *bounded strategy* for a bounded unfolding is generated in the same way as a strategy is generated for the unfolding. Based on the decisions of system players, transitions and the following sub-games are removed. The bounded strategy has to fulfill the same requirements as a strategy for an unfolding but may have fewer system places where transitions can be removed. Places are not unfolded infinitely often in the bounded unfolding, which implies that certain histories are aggregated in one place for which the same decision is repeated. A bounded strategy for a bounded unfolding can easily be extended into a strategy for the general unfolding by repeating the same decisions at places, which were not unfolded in the bounded unfolding. The converse direction is not true, i.e., a strategy for the unfolding cannot in general be translated into a strategy for a bounded unfolding [5].

3 Symbolic and bounded solving

We recall the symbolic solving approach implemented in ADAM and the bounded solving approach implemented as a prototype. Both approaches are compared in Sec. 4. Symbolic solving is based on the reduction of Petri games to a two-player game on finite graphs. This is solved using BDDs in ADAM. The bounded solving approach utilizes two bounds n on the size of the proof and b on the available memory. The question of the existence of a strategy within these bounds is encoded in a 2-QBF. Our prototype automates the generation of the 2-QBF, invokes the QBF solver QUABS to solve it, and generates a winning strategy if the QBF is satisfiable.

3.1 Symbolic Solving

In the symbolic synthesis approach for Petri games introduced in [7], the existence of a winning strategy for the system players is decided via a reduction to a two-player game over a finite graph with complete information. We recap the ideas of the reduction in this section for the comparison to the bounded approach presented in Sec. 3.2. In this paper, we only consider the case of one environment and a bounded number of system players for the symbolic synthesis approach. This case can be solved with a safety objective in single-exponential time [8]. Furthermore, we stick to safe Petri nets, since the implementation of ADAM and the bounded synthesis approach are limited to safe nets.

The general approach for the symbolic solving of Petri games is done in three steps: Firstly, from a given safe Petri game with one environment player, a bounded number of system players, and a safety objective, a two-player game over a finite graph is constructed. The environment player is represented by Player 1 (depicted as white rectangles with sharp corners) and all system players together are represented by Player 0 (depicted as gray rectangles with rounded corners). Secondly, a winning strategy of the two-player game is constructed such that the system players can cooperatively play against all behaviors of the hostile environment without encountering any bad situations. Thirdly, the winning strategy for the system players (Player 0) of the two-player game over a finite graph is transformed into a winning strategy of the system players in the Petri game and distributed into local controllers for each system player.

The two-player game over a finite graph simulates a subset of the behavior of the Petri game in such a way that the game over the graph can be considered as *completely informed*, i.e., both players have full information about their opponent at all times. Even though only a subset of the behavior is considered, [8] shows the existence of a strategy of the Petri game if and only if a strategy for the two-player game exists. Intuitively, the omitted behavior corresponds to situations where the system players exploit knowledge about the environment player's behavior of which they had not been informed. The key idea to achieve complete information is to delay every *environment transition*, i.e., transitions $t \in \mathscr{T}$ with ${}^\bullet t \cap \mathscr{P}_E \neq \emptyset$, until every next possible action of the system has to be done directly or indirectly in interaction with the environment (or there is no future interaction with the environment needed at all). Those states of the two-player game where the system players have progressed maximally are called *mcut*s. In an mcut, all system players will be informed of the environment's decision when executing their next step (or they will never be informed of the environment's decision). This idea restricts the proposed solving technique to only one environment player. The states corresponding to an mcut are assigned to the environment (Player 1) and all other states to the system (Player 0).

To simulate the Petri game, the states of the two-player game correspond to *cuts*, i.e., maximal sets of concurrent places. For the sophisticated handling of the causal memory model of Petri games, each place of a cut is enriched by a *commitment set*, i.e., a set of transitions currently selected by the corresponding system player to be allowed to fire. The transition relation of the two-player game mimics the firing of *chosen* (and enabled) transitions of the Petri net between the corresponding cuts.

Additionally, there is one extra kind of transitions in the two-player game allowing the system players to chose new commitment sets. Therefore, each place in a state is equipped with a Boolean flag $\top$. It is set to true for a place p in a state s if and only if s is a successor of an mcut reached by firing transition t and $p \in t^\bullet$ holds. Note that it is only harmful for successors of mcuts to directly choose their commitment sets without such an intermediate state with a true $\top$-flag, since for a winning strategy of the system players, *all* successors of the environment states have to avoid bad situations. Thus, the commitment sets of system state successors are directly chosen. The resolving of the $\top$, i.e., choosing new commitment sets, has to be made before any transition is allowed to fire to ensure the correct modeling of the players'

informedness. It is therefore guaranteed that every decision of the system, which should be independent of the environment's decision, is actually taken independently.

Since environment decisions are delayed until the system players have maximally progressed, possibly infinite calculations can be encountered when the system players can infinitely proceed without any interaction with the environment. To prevent these infinite behaviors, a further Boolean flag $\texttt{type}_2$ for each place in a state is introduced. This flag set to true prohibits the corresponding player to maximally progress. Thus, in an mcut all non-$\texttt{type}_2$-typed places are blocked until the environment makes its next move due to the non-existence of an enabled and chosen transition and the $\texttt{type}_2$-typed places are blocked by definition. This ensures that the system players cannot pass over the environment's decision by just playing infinitely on their own.

There are three different types of bad situations in the two-player game. Firstly, a state s is bad if two different transitions t_1 and t_2 with ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ are enabled and chosen in s. Those situations are called *nondeterministic*. Secondly, a state represents a bad situation if it contains bad places. Thirdly, *deadlock*s are bad situations. Deadlocks are states s where a transition exists which is enabled in the corresponding cut of the underlying Petri net, but there is no enabled and chosen transition in s. For more details, we refer the reader to [7, 6, 8].

Example. *We describe the two-player game over a finite graph obtained by the reduction of the Petri game of Fig. 1. In Fig. 4, a part of this game is visualized. Each state s is depicted as tuple* $s \in \mathscr{P}_E \times \mathbb{P}(\mathscr{P}_S \times \{0,1\} \times \{\top, !\top\} \times \mathbb{P}(\mathscr{T}))$*, where* 0 *indicates that the corresponding place is* $\texttt{type}_2$ *typed. In this example, there is no possibility for the system players to play infinitely long without any further interaction with the environment. Thus, all places are typed as not* $\texttt{type}_2$ *(indicated by* 1*).*

The initial state (labeled with 1) corresponds to the cut of the initial marking of the Petri net, where all system tokens initially have to decide on their commitment sets. A commitment set for a place p has to be chosen from the powerset of $p^\bullet$*. Hence, for the initial state the player on SA chooses from* $\mathbb{P}(\{t_A, fa_A, info_A\})$ *and SB from* $\mathbb{P}(\{t_B, fa_B, info_B\})$*. All of these possible combinations yield a successor of the initial state. Here, only four successors are displayed. In general, dashed borders designate that not all successors of a state are depicted in this figure.*

The checkerboard patterned states designate the bad situations of the game. Consider for example the upper left state. There, the token in pA has two possible chosen and enabled transitions (aa and ab). This situation corresponds to a nondeterministic choice in the Petri game. The other two depicted bad states correspond to deadlock situations, since the players in SA and SB decided not to allow to fire any transition, but the underlying Petri net can still fire in the corresponding cut $\{LA, SA, SB\}$ *(e.g., transition* t_A *is enabled). The situations where the game enters a state containing a bad place are not directly visualized, but reaching a bad place cannot be prevented in both situations representing a false alarm (depicted as the left branch in Fig. 4) and a false report (depicted as the right branch). In the depicted case of a false alarm, the alarm system of location A decides to use transition* fa_A *and since the environment is delayed until all system behavior is maximally processed, the alarm system will show a burglary before it has happened (cf. state 2 with a transition* $t_\perp$ *leading to a bad place and* ${}^\bullet t_\perp = \{Env, AA\}$*). Even if the system decides to not use any of the bad transitions, it cannot avoid a bad situation because it will end up in a deadlock (cf. state 3, where reaching a deadlock is mandatory). This is similarly in the depicted case of a false report. Since alarm system B decided to use transition* fr_B *(cf. state 4) and thus does not report the burglary at its location to the alarm system in A, the token in SA triggers a deadlock. If SA would have chosen some of the other possible transitions (*t_A *or* fa_A*), it would still have been a deadlock or a false alarm. The only possible solution for the alarm systems is to wait for the burglary and then use their information channel (*$info_A$ *and* $info_B$*) to inform the other player of the burglary. This situation corresponds to the orange underlaid states resulting in a winning strategy.*

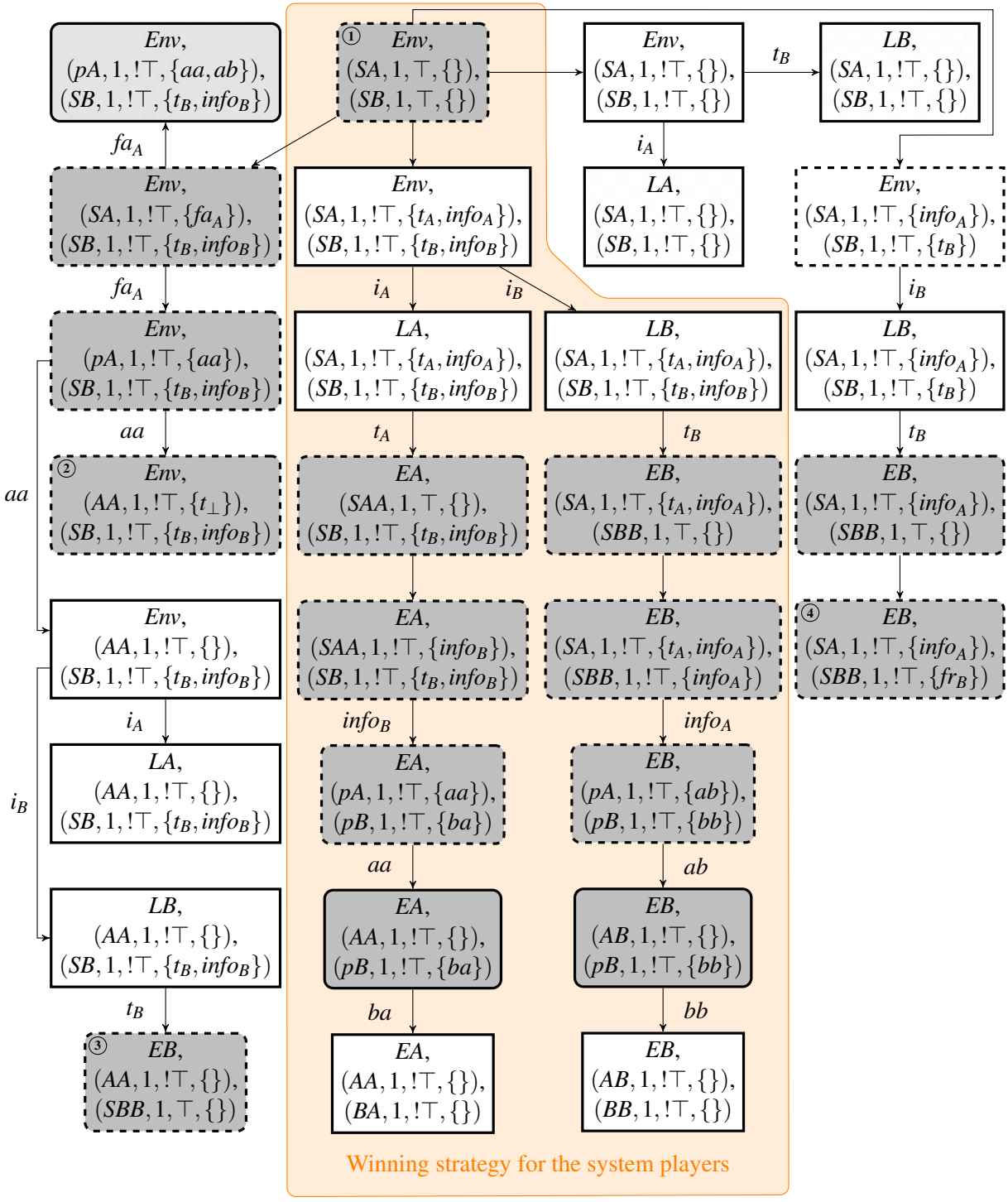


Figure 4: *The part of the two-player game over a finite graph constructed from the Petri game depicted in Fig. 1. The states belonging to the environment player are white with sharp corners whereas the states belonging to all system players together are gray with rounded corners. The winning strategy for the system players is underlaid in orange. Dashed states indicate that not all possible successors are depicted. Checkerboard patterned states designate bad states of the two-player game.*

3.2 Bounded Solving

We recall the bounded synthesis approach for Petri games [5]. In bounded synthesis, a bound is introduced to limit the search space of possible winning strategies to small strategies. Therefore, bounded synthesis can find small implementations fast. The bound is increased incrementally if no winning strategy can be found. If a winning strategy for a certain bound is found, bounded synthesis ensures that this solution is winning in general. We denote this bound by n. Bounded synthesis constitutes a semi-decision procedure, i.e., it can prove the existence of a winning strategy but not the non-existence of winning strategies in general. Bounded synthesis finds strategies, which are, because of their small size, interesting for practical applications as they avoid unnecessary (and possibly expensive) computation steps.

In Petri games, each local player can have different information about the other players. Recall that only participating players of a fired transition exchange their complete causal history. A place can have infinitely many different histories, which are represented explicitly in the possibly infinite unfolding. For bounded synthesis of Petri games, we have to introduce a second bound b on the size of the memory for each place in order to retain a finite representation of the bounded synthesis problem. A player residing in the place can only differentiate causal history up to this bound and further history is treated on par with some previous history. The original bound of bounded synthesis n limits the size of the proof of correctness for the strategy. It defines how many transitions are fired until the game has to terminate or has to repeat its behavior in a loop while fulfilling the requirements for a winning strategy. The conditions for a winning strategy are safety, determinism, deadlock-avoidance, and justified refusal as discussed in Sec. 2.

We utilize 2-QBFs to realize the bounded synthesis approach for Petri games. 2-QBFs restrict QBFs to only have one quantifier alternation. A QBF starts with an alternation of either existentially ($\exists$) or universally ($\forall$) quantified sets of Boolean variables. This prefix is followed by the matrix which is a Boolean formula using the standard operators ($\wedge$, $\vee$, $\neg$) and abbreviated operators ($\Longrightarrow$, $\Longleftrightarrow$) on Boolean variables. We focus on 2-QBFs of the form $\exists V_1.\forall V_2.\phi$ where $V_1 \cup V_2$ are all Boolean variables in ϕ. The meaning of a 2-QBF is that there exists an assignment for the Boolean variables in V_1 such that for all assignments to the Boolean variables in V_2 the formula ϕ over the assigned Boolean variables is satisfied.

For bounded synthesis of Petri games, the bound b is used to build a bounded unfolding $\mathscr{P}^b$ of the Petri game $\mathscr{P}$. $\mathscr{P}^b$ is again a Petri game explicitly modeling all available decision points for the bounded strategy. For a Petri game, the existence of a winning strategy for a play of length n can be encoded as a 2-QBF $\exists S.\forall M.\phi_n$. The set S describes the strategy and contains pairs (p,t) to indicate whether the system place $p \in \mathscr{P}^b_S$ decides to fire the transition $t \in p^\bullet$ or not. We further ensure that a decision for or against t does not violate justified refusal such that all bounded strategies fulfill this condition. The set M describing the sequence of markings contains pairs (p,i) to indicate that on place p resides a token at time point $1 \leq i \leq n$. The formula ϕ_n ensures that if M represents a play following the decisions by the strategy S and the rules of a Petri game for n steps, then the play is winning. This approach can handle finite and infinite plays by accepting termination before the last simulation step is reached and checking for loops if the last simulation step is reached.

The encoding for bounded synthesis has the following form:

$$\phi_n \overset{Def.}{=} \left(\bigwedge_{i \in \{1,\dots,n-1\}} sequence_i \implies win_i \right) \wedge (sequence_n \implies win_n \wedge loop)$$

$$sequence_i \overset{Def.}{=} initial \wedge \bigwedge_{j \in \{1,\dots,i-1\}} flow_j$$

$$win_i \overset{Def.}{=} nobadplaces_i \wedge deterministic_i \wedge deadlocksterm_i$$

$$deadlocksterm_i \overset{Def.}{=} deadlock_i \implies terminating_i$$

$$loop \overset{Def.}{=} \bigvee_{j,k \in \{1,\dots,n\}, j<k} \left(\bigwedge_{p \in \mathscr{P}} (p,j) \iff (p,k) \right)$$

For each time point $1 \leq i \leq n$, it is tested whether the variables in M represent a correct *sequence* of markings corresponding to a play in the Petri game. If this is the case then win_i ensures that the strategy fulfills the requirements at i to be winning. If $i = n$, i.e., the limit on the simulation length is reached, then it is additionally tested that a *loop* occurred. A correct sequence is defined by the play starting from the *initial* marking and firing one enabled and (by the strategy) chosen transition at each time step ($flow_j$). The play is winning if *no bad places* are reached, the system makes only *deterministic* decisions, and each *deadlock* is caused by *termination*. A deadlock occurs when all transitions are either not enabled or not chosen by the strategy. Meanwhile, termination occurs when no transition is enabled, i.e., only the lack of tokens in the preset of transitions is responsible for this and not the decisions of the system. Therefore, $deadlocksterm_i$ ensures that the system does not prevent the reaching of bad places by just stopping to fire transitions but deadlocks are only allowed when the whole game terminated. A loop in a Petri game occurs when the exact marking is repeated at two different time points j and k. Since it is tested that between j and k the strategy is deterministic, this behavior is repeated infinitely often such that the strategy is also winning in an infinite play. For further details on the definition of *initial*, $flow_j$, $deterministic_i$ etc., we refer the interested reader to [5].

Example. *In Fig. 5, a bounded unfolding of the distributed alarm system is given. Only the places pA and pB are unfolded three times resulting in the four respective places without labels. pA's unfolded places are on the lefthand side whereas pB's are on the righthand side. For the four places of pA from left to right, we thereby can differentiate the situation that the alarm system in the corresponding wing successfully tested for the environment and then decided not to inform the other system player, did not test for the environment at all, successfully tested for the environment and then informed the other system player, or was informed by the other system player about an intrusion at the other location. The same holds in converse direction for pB. We did not unfold the places AA, AB, BA, and BB because they are used only to determine bad behavior. This is only possible in the* bounded *unfolding.*

We argue why bounded synthesis rejects and accepts certain strategies for the bounded unfolding from Fig. 5. The Petri game is the running example of an alarm system from Fig. 1 in Sec. 2. For example, the strategy (SA,t_A), (SA,fa_A), $(SA,info_A)$ *activated and all other transitions deactivated is not winning because for the allowed sequence of markings* $(Env,1)$, $(SA,1)$, $(SB,1)$, $(LA,2)$, $(SA,2)$, $(SB,2)$ *in M there is nondeterminism between the enabled transitions* t_A *and* fa_A*. Meanwhile, the strategy allowing* (SA,t_A), $(SA, info_A)$, (SB,t_B), $(SB,info_B)$, $(SAA,info_B)$, $(SBB,info_A)$, (pA_A,a), (pA_B,b), (pB_A,a), (pB_B,b) *is winning because for all markings that represent a valid play of the game no bad place is reached, all decisions are deterministic, and all deadlocks are caused by termination.* pA_A, pA_B *and* pB_A, pB_B *describe the unfolded places of pA and pB reached after firing* $info_B$ *and* $info_A$*, respectively.*

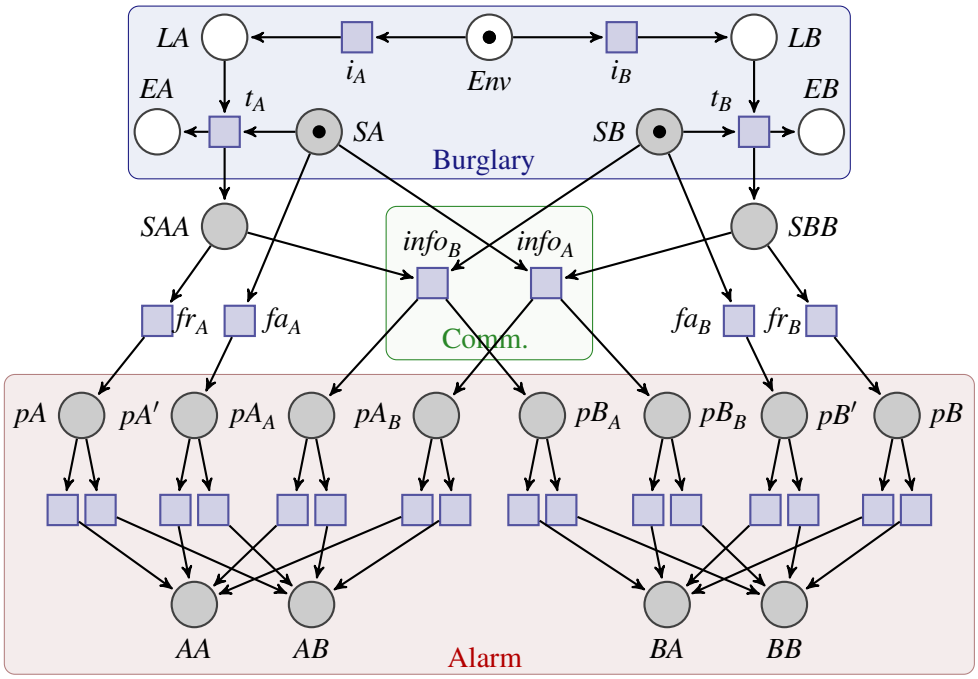


Figure 5: *A bounded unfolding for the Petri game depicted in Fig. 1 modeling an alarm system. The places pA and pB are unfolded three times, respectively, into the eight unlabeled places whereas the places AA, AB, BA, and BB are not unfolded.*

For the pairs of unlabeled transitions, the left transitions are based on the original transition a and the right ones on b. pA, pA', pB and pB' are not reached by the strategy and arbitrary decisions can be made there. We choose not to fire any transitions in this case. For example, the sequence of markings $(Env,1),(SA,1),(SB,1),(LA,2),(LB,2),\ldots$ *in M does not represent a valid play because both outgoing transitions of Env have been fired, which is illegal in a Petri game (*$flow_1$ *is violated).*

4 Experimental Results

We compare the implementation of the symbolic approach in the tool ADAM against our prototype implementation of the bounded synthesis approach on an extended set of benchmarks. We take the benchmark set of ADAM and add the benchmark family of an alarm system. At first, we describe all benchmark families. Then, we outline the technical details of our comparison framework and implementation specific particularities of the two approaches. Afterwards, we state our observations and explanations concerning the times for finding winning strategies and the sizes of these strategies.

4.1 Benchmark families

The results in the table from Fig. 6 refer to the following scalable benchmark families where the alarm system is the new benchmark family:

- AS: *Alarm System*. There are *m* secured locations belonging to one person. A burglar can intrude one of the locations. Each location has a local alarm system, which can communicate with all

other local alarm systems. The goal is that the alarm system in each location has to indicate the position where the burglar intruded. Furthermore, the alarm system should not issue unsubstantiated warnings of an intrusion at any location.
Parameters: m locations

- CM: *Concurrent Machines*. There are m machines which should process k orders. Each machine is allowed to process at most one order. The hostile environment disables one arbitrary machine such that it cannot process any order. The system's goal is to still process all k orders.
Parameters: m machines and k orders.
- SR: *Self-reconfiguring Robots*. There are m robots having m different tools at their disposal each. The robots can only equip one tool at a time. From a global perspective, all robots together have to maintain a functioning state such that material can be processed by each of the m different tools. The environment can destroy k tools in total. This can occur at the same robot or on different robots. The robots have to reconfigure theirselves to maintain a functioning global state for the processing of material.
Parameters: m robots with m tools each and k destroyed tools in total.
- JP: *Job Processing*. A job requires processing by a, from the environment chosen, subset of m processors in ascending order.
Parameter: m processors.
- DW: *Document Workflow*. There are m clerks having to unanimously endorse or reject a document. The environment decides which clerk gets the document first. In DWs, it is required that all clerks endorse the document.
Parameter: m clerks.

4.2 Comparison framework

We compare the symbolic synthesis approach implemented in ADAM with our prototype implementation of the bounded synthesis approach. ADAM and the bounded synthesis approach are the only tools existing to find winning strategies of Petri games but they are inherently different. On the one hand, the symbolic approach models, in theory, infinite memory and unbounded firing sequences of transitions. On the other hand, bounded synthesis has two parameters n and b, which can be increased to find a winning strategy. Therefore, the bounded synthesis approach can be parallelized easily because the QBF-solver can be called twice for two different pairs (n,b) and the resulting encodings. We report in the following on the runtime results for the minimal b and the corresponding n such that a winning strategy exists because b turned out to be more expensive than n in terms of runtime. Notice that $b = 1$ enforces that the bounded unfolding is the original game, i.e., no bounded unfolding is utilized when searching for a winning bounded strategy of the corresponding Petri game.

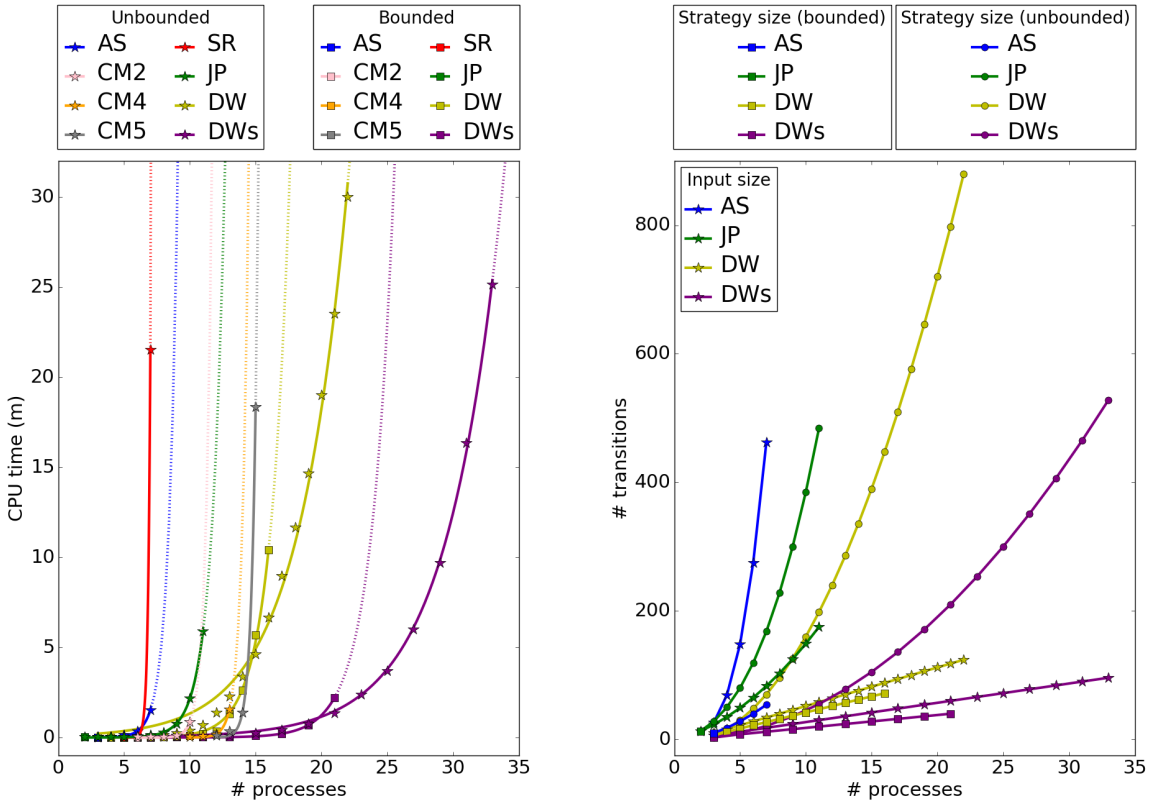
The table in Fig. 6 shows the results of ADAM and our prototype implementation of bounded synthesis for the previously described benchmark families. The results were obtained on an Intel i7-2700K CPU with 3.50 GHz, 32 GB RAM, and a timeout of 1800 seconds. For each benchmark (column *Ben.*), we report on the attempted parameters of the benchmark (*Par.*), on the size of the Petri game (number of tokens (#*Tok*), places ($\#\mathscr{P}$), and transitions ($\#\mathscr{T}$)), and on the respective *time* and *memory* for solving and on the respective number of places ($\#\mathscr{P}_{str}$) and transitions ($\#\mathscr{T}_{str}$) of the winning strategies synthesized by ADAM and by our prototype implementation of bounded synthesis. The elapsed CPU time is measured in seconds and the used memory in gigabyte. For bounded synthesis, we additionally report the smallest b and corresponding n to find winning bounded strategies with the least memory requirement.

					Symbolic Synthesis				*Bounded Synthesis*					
Ben.	*Par.*	*#Tok*	$\#\mathscr{P}$	$\#\mathscr{T}$	*time*	*memory*	$\#\mathscr{P}_{str}$	$\#\mathscr{T}_{str}$	*n*	*b*	*time*	*memory*	$\#\mathscr{P}_{str}$	$\#\mathscr{T}_{str}$
AS	2	3	17	26	**1.9**	.30	**17**	**10**	7	2	18.0	**.18**	**17**	**10**
	3	4	28	69	2.5	.41	31	18	7	3	timeout			
			…			…				…				
	6	7	73	462	91.0	4.65	97	54	7	6	timeout			
	7				timeout				7	7	timeout			
CM	2/1	6	13	10	1.4	.30	14	**8**	6	3	**.6**	**.06**	**13**	**8**
	2/2	7	18	16	2.0	.30	-	-	-	-				
			…			…				…				
	2/5	10	33	34	50.9	2.74	-	-	-	-				
	2/6				timeout				-	-				
	3/1	8	18	15	2.0	.30	26	12	6	3	**1.7**	**.12**	**18**	**9**
	3/2	9	25	24	2.4	.30	36	18	6	4	timeout			
	3/3	10	32	33	3.8	.39	-	-	-	-				
	3/4	11	39	42	17.2	1.28	-	-	-	-				
	3/5				timeout									
	4/1	10	23	20	**2.3**	.30	42	16	6	3	6.0	**.19**	**21**	**12**
	4/2	11	32	32	5.0	.40	55	24	6	4	timeout			
	4/3	12	41	44	10.9	.84	68	32	6	5	timeout			
	4/4	13	50	56	92.2	4.17	-	-	-	-				
	4/5				out of memory				-	-				
	5/1	12	28	25	**7.2**	.39	62	20	6	3	11.1	**.19**	**22**	**11**
	5/2	13	39	40	20.8	.79	78	30	6	4	timeout			
	5/3	14	50	55	82.1	2.67	94	40	6	5	timeout			
	5/4	15	61	70	1101.3	16.70	110	50	6	6	timeout			
	5/5				out of memory				-	-				
	6/1	14	33	30	41.5	.80	86	24	6	3	**23.6**	**.31**	**25**	**12**
	6/2	15	46	48	183.4	2.67	105	36	6	4	timeout			
	7/1	16	38	35	289.5	5.35	114	28	6	3	**26.0**	**.36**	**27**	**13**
	8/1	18	43	40	1657.4	15.73	146	32	6	3	**94.7**	**.65**	**27**	**14**
	9/1	20	48	45	out of memory				6	3	152.4	1.22	36	25
			…			…				…				
	15/1	32	78	75	out of memory				6	3	1259.5	23.24	66	49
SR	2/1	5	18	17	**1.9**	.30	32	16	6	2	2.7	**.16**	**18**	**10**
	2/2	6	24	26	4.3	.39	-	-	-	-				
	2/3	7	30	35	1290.3	5.36	-	-	-	-				
	2/4				out of memory				-	-				
	3/1				out of memory				7	2	219.4	.61	27	19
JP	2	3	12	13	**1.3**	.30	**16**	**13**	7	3	1.5	**.08**	**16**	**13**
	3	4	18	23	1.8	.30	34	28	8	4	timeout			
			…			…				…				
	11	12	102	175	353.3	16.78	706	484	16	12	timeout			
	12				out of memory				17	13	timeout			
DW	1	3	12	10	1.1	.30	**10**	**6**	8	1	**.3**	**.04**	**10**	**6**
	2	4	19	16	2.0	.30	24	16	10	1	**.4**	**.05**	**16**	**12**
			…			…				…				
	11	13	82	70	137.6	2.82	420	286	28	1	**78.3**	**2.26**	**70**	**57**
	12	14	89	76	201.8	**2.82**	494	336	30	1	**157.3**	3.38	**76**	**62**
	13	15	96	82	**277.9**	**4.24**	574	390	32	1	341.1	8.18	**82**	**67**
	14	16	103	88	**400.1**	**4.22**	660	448	34	1	624.5	11.80	**88**	**72**
	15	17	110	94	537.2	4.87	752	510	36	1	timeout			
			…			…				…				
	20	22	145	124	1799.8	11.69	1302	880	46	1	timeout			
	21				timeout				48	1	timeout			
DWs	1	3	11	6	.7	.29	**8**	**3**	5	1	**.2**	**.04**	**8**	**3**
	2	5	21	12	1.6	.30	23	10	7	1	**.3**	**.05**	**17**	**8**
			…			…				…				
	7	15	71	42	14.4	.91	218	105	17	1	**5.3**	**.87**	**57**	**28**
	8	17	81	48	24.9	**1.52**	281	136	19	1	**11.7**	3.18	**65**	**32**
	9	19	91	54	45.4	**2.83**	352	171	21	1	**41.3**	8.84	**73**	**36**
	10	21	101	60	**80.0**	**2.85**	431	210	23	1	132.8	12.96	**81**	**40**
	11	23	111	66	142.6	4.42	518	253	25	1	timeout			
			…			…				…				
	16	33	161	96	1508.3	16.30	1073	528	35	1	timeout			
	17				timeout				37	1	timeout			

'-' means no winning strategy exists.

Figure 6: Comparison between the symbolic synthesis approach and the bounded synthesis approach.

When ADAM and our prototype implementation both synthesized a winning strategy, then we mark the minimal running time, the minimal memory usage, and the minimal number of places and of transitions in the winning strategy in bold, respectively.



(a) The CPU running time (in minutes) for a selection of benchmarks and the respective number of processes (tokens). The running time for the bounded approach is given as squares and the running time for the symbolic case is given by stars. The dotted lines designate the expected running time after the timeout of 30 minutes.

(b) The sizes (in the number of transitions) for a selection of benchmarks and the respective number of processes (tokens). The number of transitions of the Petri game is designated by stars, the number of transitions for the bounded strategy by filled circles, and the number of transitions of the strategy in the bounded case by squares.

Figure 7: Comparison of the symbolic synthesis approach and the bounded synthesis approach by the running times and sizes of the strategies.

A selection of these values and benchmark families are plotted in Fig. 7. In Fig. 7(a), the CPU times in minutes for the symbolic and the bounded approach on selected benchmark families are plotted for an increasing number of processes, i.e., the number of tokens of the underlying net. *CMi*, for $i \in \{2,4,5\}$, represents the subset of the concurrent machines benchmark, where the first parameter m for the number of machines is fixed to i. Therefore, the number of orders increases the number of processes in *CMi*. The dotted lines designate the expected running time obtained by fitting exponential curves through the actual values. In Fig. 7(b), we plotted the number of transitions of the input Petri games and of the corresponding winning strategies of the two approaches for an increasing number of processes from selected benchmark families.

4.3 Implementation details

During the implementation of our prototype for bounded synthesis of Petri games, we observed that a translation of the matrix ϕ_n into conjunctive normalform (CNF) and the usage of a QBF solver requiring input in CNF has poor performance. We found out that the `QCIR` file format for QBFs [16] allows competitive performance as ϕ_n does not need to be translated into CNF. As we have a 2-QBF not in CNF, solvers using counterexamples to refine an abstraction (CEGAR-based) [3] showed the best performance. We therefore decided for the solver QuAbS [17] as it combines fast parsing with fast solving. Given a bound on the length of the proof of correctness for a strategy, we further pruned the bounded unfolding of unreachable places and unreachable transitions to remove unnecessary variables from the 2-QBF, which increased the overall performance.

The running times of both approaches are highly dependent on the number of variables which are respectively used in the BBD and in the QBF. For the symbolic approach, the number of variables in the BDD grows significantly with the number of players in the Petri game (which are represented by tokens in the underlying net). The two-player game over a finite graph with complete information is represented by a BDD for each flow encoding the flow's source and target state. For optimizing the size of the BDD, we partition $\mathscr{P}_S$ into k disjunct sets $\mathscr{P}_{S_i} \subseteq \mathscr{P}_S$, for $i \in \{1,\ldots,k\}$, such that for every reachable marking each place of the marking belongs uniquely to exactly one of the sets $\mathscr{P}_{S_i}$. In general, the number k corresponds to the number of system processes in the game. A state of the finite graph is encoded by a binary encoding of the ID of the environment place and the maximally k IDs of the system places of the marking, which results in encodings of logarithmic size. Whether a process's commitment set contains a transition is encoded explicitly by a Boolean flag. An explicit encoding is used because the size of the commitment set varies for each process depending on the place it resides in. In general, this approach yields smaller BDDs. The two additional variables per system process encode whether the process's commitment set has to be renewed, i.e., the $\top$-flag is set or not, and whether the place is a $\texttt{type}_2$-typed place. Thus, the number of variables for a BDD can be calculated by

$$2 \cdot \left(\texttt{log}_2(|\mathscr{P}_E|) + \sum_{i=1}^{k} (\texttt{log}_2(|\mathscr{P}_{S_i}|) + |\mathscr{T}_i| + 2) \right)$$

where $\mathscr{T}_i = \bigcup_{p \in \mathscr{P}_{S_i}} p^\bullet$.

For the bounded approach, we distinguish existentially and universally quantified variables and gate variables. The existentially quantified variables describe the system's strategy and the universally quantified variables encode all sequences of markings. The gate variables are used to describe the bounded synthesis problem in the matrix ϕ_n given the existentially and universally quantified variables. Thus, the size of the QBF is growing with the structure of the underlying net of the Petri game, i.e., with the number of places and transitions.

In Fig. 8, we compare the number of variables in the two approaches for benchmark families where several instances are solved by both approaches. DW, DWs, and CM with parameter $k = 1$ qualify for this comparison. For the symbolic approach, the number of variables in the BDD is given ($\#Var_{symbolic}$). For the bounded approach, the total number of variables in the QBF ($\#Var_{bounded}$) and the number of existentially ($\#Var_\exists$) and universally ($\#Var_\forall$) quantified variables thereof are given. The number of gate variables ($\#Var_{\phi_n}$) used to build the matrix ϕ_n is stated as the remaining variables of $\#Var_{bounded}$, i.e., $\#Var_\exists + \#Var_\forall + \#Var_{\phi_n} = \#Var_{bounded}$. From the size difference of the respective numbers of variables in the two approaches, one can derive that they are used for a different purpose in the respective approach. For the symbolic approach, the number of variables grows linearly for all benchmarks. For the

Ben.	*Par.*	$\#Var_{symbolic}$	n	b	$\#Var_{bounded}$	$\#Var_{\exists}$	$\#Var_{\forall}$	$\#Var_{\phi_n}$
CM	2/1	66	6	3	2743	53	162	2528
	3/1	92	6	3	7678	109	300	7269
	4/1	120	6	3	17849	197	462	17190
	5/1	146	6	3	25848	266	570	25012
	6/1	172	6	3	35287	343	678	34266
	7/1	198	6	3	46166	428	786	44952
	8/1	226	6	3	58485	521	894	57070
DW	1	46	8	1	1144	12	96	1036
	2	72	10	1	2591	20	190	2381
	3	98	12	1	4838	28	312	4498
	4	124	14	1	8052	36	462	7554
	5	148	16	1	12404	44	640	11720
	6	172	18	1	18059	52	846	17161
	7	198	20	1	25186	60	1080	24046
	8	224	22	1	33953	68	1342	32543
	9	248	24	1	44528	76	1632	42820
	10	272	26	1	57079	84	1950	55045
	11	296	28	1	71744	92	2296	69356
	12	320	30	1	88781	100	2670	86011
	13	344	32	1	108268	108	3072	105088
	14	368	34	1	130403	116	3502	126785
DWs	1	36	5	1	440	9	55	376
	2	70	7	1	1414	17	147	1250
	3	102	9	1	3204	25	279	2900
	4	136	11	1	6050	33	451	5566
	5	168	13	1	10192	41	663	9488
	6	200	15	1	15870	49	915	14906
	7	232	17	1	23324	57	1207	22060
	8	266	19	1	32794	65	1539	31190
	9	298	21	1	44520	73	1911	42536
	10	330	23	1	58742	81	2323	56338

Figure 8: Comparison between the numbers of different variables in the two approaches.

bounded approach, the number of existentially quantified variables grows linearly, the number of universally quantified variables grows quadratically, and the total number of variables grows cubicly in DW and DWs. For CM, all variables in the bounded encoding grow exponentially which is surprising as n and b remain constant. We suppose that this increase is caused by the construction of the bounded unfolding which produces an exponentially growing number of transitions for this benchmark family despite $b = 3$ remaining constant.

We further detected that the QBF problem files can become large, which requires a QBF solver with fast parsing of the problem file. The largest solved `QCIR` file is of size 40 MB and contains 208.877 variables (benchmark *concurrent machines* with parameters $m = 15$ and $k = 1$ and bounds $n = 6$ and $b = 3$). A very large `QCIR` file for a benchmark which ADAM solved but for which the QBF solver timed out (benchmark *job processing* with parameter $m = 4$ and bounds $n = 9$ and $b = 5$) has a size of 275 MB and contains 2.722.512 variables.

In summary, the size of the BDD for the symbolic approach is dominated by the number of tokens whereas the size of the 2-QBF for the bounded approach is dominated by the number of places and transitions of the underlying net of the Petri game.

4.4 Comparison

Both approaches work especially well on certain aspects of distributed synthesis. The symbolic approach implemented in ADAM solves more instances than the bounded synthesis approach for all benchmark families but for *concurrent machines* (CM) with one defective machine ($k = 1$). ADAM can further show the non-existence of a winning strategy for instantiations of benchmark families for which bounded synthesis is not applicable. For example, ADAM shows that for CM no strategy exists when equally many or more orders as machines are placed because one machine can process at most one order and the environment marks one machine as unable to process an order.

The bounded approach is well suited for finding small winning strategies. It holds for all winning strategies produced by the two approaches that the respective winning strategies from bounded synthesis have equally many or fewer places and transitions. For small instances, these differences are negligible as for the first instances of *alarm system* (AS) and both versions of *document workflow* (DW and DWs) the respective strategies are of equal size. The larger the benchmark instances become, the larger the differences in strategy size get. For DW with parameter $m = 14$, the strategy from ADAM has 660 places and 448 transitions whereas the strategy from the prototype implementation of bounded synthesis has only 88 places and 72 transition. This comes at a higher solving time of 625 seconds in contrast to 400 seconds and at using 12 GB of memory in comparison to 4 GB.

The difference in size of the strategies can also be observed from Fig. 7(b) where the size of the input and of the produced strategies by the two approaches are compared for a given number of processes. This difference in size is based on the different structure of the strategies in the two approaches. In bounded unfoldings and bounded strategies, more than one transition can merge into one place, if the different history of the token is not needed. In contrast, the symbolic approach has to unfold a place in every case notwithstanding the need for differentiation of its causal past. This becomes apparent in the benchmarks DW and DWs. In the symbolic case, for every choice of the environment which clerk has to decide on endorsing the document first, the places and transitions of each clerk are copied and put into the right order. The bounded algorithm detects that it is not necessary to unfold all these places, since the memory is not needed for finding a winning strategy and thus yields a much smaller strategy.

The bounded approach can be more subtle in choosing when to unfold places and therewith generally finds smaller strategies than the symbolic approach. This illustrates the difference between a bounded

strategy (produced by the bounded synthesis approach) and a strategy (produced by the symbolic synthesis approach). Bounded strategies are based on the bounded unfolding which can consolidate different causal pasts into one system place for which the corresponding strategy has to make the same decision. In contrast, a strategy is based on the unfolding, which explicitly represents every causal past of a system place. At each such system place, an individual decision can be made. Therefore, when the same decision suffices for each causal past, these decisions are represented explicitly with each unfolded place. From the visualization of Fig. 7(b), we can conjecture for the displayed subset of benchmark families that the symbolic approach can only find strategies which grow in size exponentially because the unfolding is exponential in the number of places and transitions. In contrast, the bounded approach can find strategies which grow in size linearly when a linearly growing bounded unfolding suffices to represent the necessary causal history.

For the sizes of the solution in the symbolic case, we can see that in general the strategy sizes increase faster and also are larger than the sizes of the input in the benchmark families from Fig. 7(b). This is caused by our benchmarks mostly increasing linearly in the input sizes (e.g., by adding a new robot or a new machine). Meanwhile, the solution is getting more difficult due to the additional abilities and behaviors of the whole system (e.g., the factory) and the symbolic approach has to consider more different flows of tokens with different causal histories. In general, the unfolding and the strategy increase in size stronger than the input. One exception to the linear increase of the input is the new *alarm system* benchmark. There we also add only linearly bounded many places and transitions for every new alarm system, but we have to add an additional alarm signal at each already existing alarm system and, furthermore, transitions leading to bad places for all additional combinations of bad situations. Since those transitions are not present in the strategy, the size of the solution is smaller than the size of the input for the alarm system benchmark.

For the running time, we can see in Fig. 6 and in Fig. 7(a) that the bounded approach outperforms the symbolic one for smaller instances but increases more sharply. This stems from the different parameters in both approaches discussed in Sec. 4.3, which are responsible for the solving complexity. For the symbolic case, the number of processes are principally responsible for the increasing number of variables of the BDD and thus for the complexity. For the bounded approach, this is different because the number of QBF variables is more dependent on the structure of the net than on the number of tokens.

For the benchmarks of DW and DWs, the bounded synthesis approach outperforms ADAM for the first eleven respectively nine parameters in terms of runtime and memory usage. On the next three parameters (DW) respectively on the next parameter (DW), ADAM outperforms bounded synthesis. After that, bounded synthesis already reaches the time limit while ADAM can solve further five parameters for DW and DWs each.

The bounded synthesis approach further showed that no unfolding is necessary to solve instances of DW and DWs. It also revealed that for CM it is possible to find winning strategies for benchmark instances of growing size while maintaining the same values for n and b.

5 Conclusion

We added the new benchmark family of a distributed alarm system to the set of benchmark families for distributed synthesis with Petri games collected during the implementation of ADAM. The new benchmark family models an alarm system for a person with a scalable number of independent locations she needs to secure. Each of these locations has a local alarm system, which can detect the intrusion by a burglar. Furthermore, all alarm systems can communicate with each other and each alarm system can

indicate the position of a detected burglary. We synthesized strategies to detect the position of a burglary and indicate it at the alarm systems of *all* independent locations.

We found out that the translation of bounded synthesis into 2-QBF resulted in large non-CNF formulas, which were solved best by a CEGAR-based QBF solver like QuAbS. The automatic construction of the bounded unfolding benefits from a removal of unreachable places and transitions, which implies that there is still room for improvement when constructing the bounded unfolding.

We compared the symbolic synthesis approach implemented in the tool Adam with the bounded synthesis approach on the extended set of benchmarks. We found out that symbolic synthesis can overall synthesize strategies for larger problems for all benchmark families except the benchmark family of concurrent machines (with parameter $k = 1$). For the smaller instances, bounded synthesis is faster but the running time grows at a higher rate such that Adam can solve more instances overall. At the same time, bounded synthesis finds smaller strategies in the number of places and transitions. This difference is negligible for small instances but grows for larger instances. The difference in size of the strategies is caused by the distinction between the bounded unfolding and the unfolding. In the bounded unfolding, different causal pasts can be consolidated into one place whereas in the unfolding, unique causal pasts have to be differentiated. Therefore, bounded strategies can profit in terms of size from situations where only some causal past is needed. This adds to the general benefit of bounded synthesis in comparison with symbolic synthesis to steer the search to small strategies.

We identified that the number of variables in the BDD of the symbolic approach grows in the number of tokens in the underlying net of the Petri game whereas the number of variables in the QBF of the bounded approach grows in the number of places and transitions of the underlying net of the Petri game. We further showed that for the benchmark families DW and DWs local strategies for each system player suffice as the bounded unfolding is the same as the original game. This proved that both symbolic and bounded synthesis are well-suited for certain aspects of distributed synthesis with Petri games.

References

[1] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Interactive presentation: Automatic hardware synthesis from specifications: a case study*. In: *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, pp. 1188–1193, doi:10.1145/1266366.1266622.

[2] Alonzo Church (1963): *Application of recursive arithmetic to the problem of circuit synthesis*.

[3] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2000): *Counterexample-Guided Abstraction Refinement*. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pp. 154–169, doi:10.1007/10722167_15. Available at `https://doi.org/10.1007/10722167_15`.

[4] Javier Esparza & Keijo Heljanko (2008): *Unfoldings - A Partial-Order Approach to Model Checking*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-540-77426-6.

[5] Bernd Finkbeiner (2015): *Bounded Synthesis for Petri Games*. In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pp. 223–237, doi:10.1007/978-3-319-23506-6_15.

[6] Bernd Finkbeiner, Manuel Gieseking & Ernst-Rüdiger Olderog (2015): *Adam: Causality-Based Synthesis of Distributed Systems*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pp. 433–439, doi:10.1007/978-3-319-21690-4_25.

[7] Bernd Finkbeiner & Ernst-Rüdiger Olderog (2014): *Petri Games: Synthesis of Distributed Systems with Causal Memory*. In: *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014.*, pp. 217–230, doi:10.4204/EPTCS.161.19.

[8] Bernd Finkbeiner & Ernst-Rüdiger Olderog (2017): *Petri games: Synthesis of distributed systems with causal memory*. *Inf. Comput.* 253, pp. 181–203, doi:10.1016/j.ic.2016.07.006.

[9] Bernd Finkbeiner & Sven Schewe (2005): *Uniform Distributed Synthesis*. In: *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pp. 321–330, doi:10.1109/LICS.2005.53.

[10] Bernd Finkbeiner & Sven Schewe (2013): *Bounded synthesis*. *STTT* 15(5-6), pp. 519–539, doi:10.1007/s10009-012-0228-z.

[11] Orna Kupferman & Moshe Y. Vardi (2001): *Synthesizing Distributed Systems*. In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pp. 389–398, doi:10.1109/LICS.2001.932514.

[12] P. Madhusudan, P. S. Thiagarajan & Shaofa Yang (2005): *The MSO Theory of Connectedly Communicating Processes*. In: *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, pp. 201–212, doi:10.1007/11590156_16.

[13] Ernst-Rüdiger Olderog (1991): *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship*. Cambridge University Press.

[14] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pp. 179–190, doi:10.1145/75277.75293.

[15] Amir Pnueli & Roni Rosner (1990): *Distributed Reactive Systems Are Hard to Synthesize*. In: *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pp. 746–757, doi:10.1109/FSCS.1990.89597.

[16] QBF Gallery 2014: *QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas*. Available at `http://qbf.satisfiability.org/gallery/qcir-gallery14.pdf`.

[17] Leander Tentrup (2016): *Solving QBF by Abstraction*. *CoRR* abs/1604.06752. Available at `http://arxiv.org/abs/1604.06752`.

[18] Wieslaw Zielonka (1987): *Notes on Finite Asynchronous Automata*. *ITA* 21(2), pp. 99–135.

A Class of Control Certificates to Ensure Reach-While-Stay for Switched Systems

Hadi Ravanbakhsh and Sriram Sankaranarayanan
Department of Computer Science
University of Colorado, Boulder
Boulder, CO, USA
firsname.lastname@colorado.edu

In this article, we consider the problem of synthesizing switching controllers for temporal properties through the composition of simple primitive reach-while-stay (RWS) properties. Reach-while-stay properties specify that the system states starting from an initial set I, must reach a goal (target) set G in finite time, while remaining inside a safe set S. Our approach synthesizes switched controllers that select between finitely many modes to satisfy the given RWS specification. To do so, we consider control certificates, which are Lyapunov-like functions that represent control strategies to achieve the desired specification. However, for RWS problems, a control Lyapunov-like function is often hard to synthesize in a simple polynomial form. Therefore, we combine control barrier and Lyapunov functions with an additional compatibility condition between them. Using this approach, the controller synthesis problem reduces to one of solving quantified nonlinear constrained problems that are handled using a combination of SMT solvers. The synthesis of controllers is demonstrated through a set of interesting numerical examples drawn from the related work, and compared with the state-of-the-art tool SCOTS. Our evaluation suggests that our approach is computationally feasible, and adds to the growing body of formal approaches to controller synthesis.

1 Introduction

The problem of synthesizing switching controllers for reach-while-stay (RWS) specifications is examined in this article. RWS properties are an important class, since we may decompose more complex temporal specifications into a sequence of RWS specifications [10]. The plant model is a switched system that consists of finitely many (controllable) modes, and the dynamics for each mode are specified using ODEs. Furthermore, we consider nonlinear ODEs for each mode, including rational, trigonometric, and exponential functions. The goal of the controller is to switch between the appropriate modes, so that the resulting closed loop traces satisfy the specification.

RWS properties specify that a goal set G must be reached by all behaviors of the closed-loop system while staying inside a safe set S. Specifically, the state of the system is assumed to be initialized to any state in the set S. RWS properties include safety properties (stay inside a safe set S), reachability properties (reach a goal set G), and "control-to-facet" problems [7, 8].

The controller synthesis is addressed in two phases: (a) formulating a *control certificate* whose existence guarantees the existence of a non-Zeno switching control law for the given RWS specification, and (b) solving for a certificate of a particular form as a feasibility problem. The control certificates are control Lyapunov-like functions which represent a strategy for the controller to satisfy the specifications. Additionally, this strategy can be effectively implemented as a feedback law using a controller that respects min dwell time constraints. In the second phase, a counterexample guided inductive synthesis (CEGIS) framework [27], — an approach that uses SMT solvers at its core — is used to discover such control certificates. However, this procedure is used off the shelf, building upon the previous work of

Submitted to:
SYNT 2017

Ravanbakhsh et al. [22]. This procedure uses a specialized solver for finding a certificate of a given parametric form that handles quantified formulas by alternating between a series of quantifier free formulas using existing SMT solvers [18, 4].

The contributions of the paper are as follows: first, we show that a straightforward formulation of the control certificate for the RWS problem yields an exponential number of conditions, and hence can be computationally infeasible. Next, we introduce a class of control certificates which (i) has a concise logical structure that makes the problem of discovering the certificates computationally feasible; and (ii) we show that such certificates yield corresponding switching strategies with a min-dwell time property unlike the conventional control certificates. Next, we extend our approach to the initialized RWS (IRWS) property that additionally restricts the set of initial conditions of the system using a class of "control zero-ing" barrier functions [34, 37] . Also, a suitable formulation for these functions is provided within our framework. Finally, we provide numerical examples to demonstrate the effectiveness of the method, including comparisons with recently developed state-of-art automatic control synthesis tool SCOTS [26].

1.1 Related Work

The broader area of temporal logic synthesis seeks to synthesize formally guaranteed controllers from the given plant model and specifications. The dominant approach is to build a discrete abstraction of the given plant that is related to the original system [36, 14, 15, 26, 17]. Once a suitable abstraction is found, these approaches use a systematic temporal logic-based controller design approach over the abstraction [33]. The properties of interest in these systems include the full linear temporal logic (LTL) and an efficiently synthesizable subset such as GR(1) [36, 14]. These approaches differ in how the abstraction can be constructed in a guaranteed manner. One class of approaches works by fixing a time step, gridding the state-space, and simulating one point per cell [17, 15, 26, 38, 32]. The resulting abstraction, however, is not always approximately bisimilar to the original system. Nevertheless, conditions such as open loop incremental stability of the plant can be used to obtain bisimilarity [5]. Alternatively, the abstraction can be built without time discretization [20, 14] by considering infeasible transitions. And furthermore, the abstraction can be iteratively refined through a counter-example refinement scheme [19]. Our work here does not *directly* focus on building abstractions. Rather, our focus is on deductive approaches for a narrow class of temporal logic properties namely RWS properties. Using our approach, control systems for richer properties can be built from solving a series of RWS problems.

Our approach is closely related to work of Habets et al. [6] and Kloetzer et al. [10]. In these methods, an abstraction is obtained by solving local control-to-facet problems instead of reachability analysis. However, continuous feedback is synthesized for each control-to-facet problem. The key difference in this paper is that the control-to-facet problems themselves are solved using switching. Furthermore, we consider initialized problems, where the initial states are also restricted to belong to a set. We find that IRWS problems can often be realized through a controller even when the corresponding RWS problem (for which the initial condition is not restricted) cannot be synthesized.

Another related class of solutions is based on synthesizing "a deductive proof of correctness" simultaneously with "a control strategy". The goal of these approaches also consists of finding a control certificate, which yields a (control) strategy to guarantee the property. This typically takes the form of a control Lyapunov-like function. The idea of control Lyapunov functions goes back to Artstein [1] and Sontag [28]. The problem of *discovering* a control Lyapunov function is usually formulated using bilinear matrix inequalities (BMI) [31]. Also, instead of solving such NP-hard problems, usually alternating optimization (V-K iteration or policy iteration) is used to conservatively find a solution [31, 3].

Wongpiromsarn et al. [35] discuss *verification* of temporal logic properties using barrier certificates.

For synthesis, Xu et al. [37] discuss conditions for the so-called "control zeroing" barrier functions for safety and their properties. They also, consider their combination with control Lyapunov functions. In this article, we provide an alternative condition that is based on "exponential condition" barrier functions [11] and enforcing a compatibility condition between the control actions suggested by the control barrier and control Lyapunov functions. Also, Dimitrova et. al. [2] have shown that control certificates can be extended to address more complicated specifications i.e. parity games. While these results show that constraint solving based methods can be applied on more complicated specification, no method of finding such certificates is provided.

The use of SMT solvers in control synthesis has also been well-studied. Taly et. al [29, 30] use a constraint solving approach to find control certificates for reachability and safety. They adapt a technique known as Counter-Example Guided Inductive Synthesis (CEGIS), originally proposed for program synthesis [27], to solve the control problems using a combination of an SMT solver with numerical simulations. Ravanbakhsh et al. [22] propose a combination of SMT and SDP solvers for finding control certificates. However, their method is only applicable to stability or simple reachability properties, involving the use of a single Lyapunov function. In a subsequent paper, their approach is extended to handle disturbance inputs [23]. The use of SMT solvers to solve for Lyapunov-like functions is used in our paper as well. However, this paper focuses on defining a more tractable class of control certificates for RWS problems. Furthermore, we show how these problems can be composed for more complex temporal objectives. In particular, our use of the CEGIS procedure is not a contribution of this paper. Furthermore, in order to handle nonlinear systems and also to guarantee numerical soundness of these solvers, we use the nonlinear SMT solver dReal [4].

Huang et. al. [9] also propose control certificates to solve the RWS problem for piecewise affine systems, using SMT solvers. Their approach uses piecewise constant functions as control certificates and partitions the state space into small enough cells in order to define such functions. By using this technique, any function can be approximated, which makes the method relatively complete.

As mentioned earlier, past work by Habets et al. and Klutzier et al. [6, 10] build a finite abstraction by repeatedly solving control-to-facet problems. These problems seek to find a feedback law inside a polytope P that guarantees all the resulting trajectories exit P through a specific facet F of P. Habets et. al. [7] show necessary and sufficient conditions for the existence of a control strategy for the control-to-facet problem on simplices. This condition is sufficient but not necessary for polytopes. They extract a unique certificate from each problem instance and check whether the condition holds for the certificate. Subsequently, Roszak et al. [24] and Helwa et al. [8] extend this approach and solve reachability to a set of facets by introducing flow condition, which combined with invariant condition serves as a control certificate similar to those used in this paper. From the published results, these methods are more efficient, but are only applicable to affine systems over polytopes. In contrast, the dynamics in this article can be non-linear involving rational, trigonometric, and exponential functions. In this article, we demonstrate that our method can be used to solve such problems and it can be integrated into other methods which build an abstraction for the system.

2 Background

2.1 Notation

Given a function $f(t)$, let $f^+(t)$ ($f^-(t)$) be the right (left) limit of f at t, and $\dot{f}(t)$ represent the *right derivative* of f at time t. For a set $S \subseteq \mathbb{R}^n$, ∂S and $int(S)$ are its boundary and interior, respectively.

Definition 1 *(Nondegenerate Basic Semialgebraic Set):*

A nondegenerate basic semialgebraic set K *is a nonempty set defined by a conjunction polynomial inequalities:*

$$K: \{\mathbf{x} \mid p_{K,1}(\mathbf{x}) \leq 0 \wedge \cdots \wedge p_{K,i}(\mathbf{x}) \leq 0\},$$

where $\mathbf{x} \in \mathbb{R}^n$*. For each* $j \in [1,i]$*, we define*

$$H_{K,j} = \{\mathbf{x} \mid \mathbf{x} \in K \wedge p_{K,j}(\mathbf{x}) = 0\}.$$

It is required that (a) each $H_{K,j}$ is nonempty, (b) the boundary ∂K and the interior $int(K)$ are given by $\bigvee_{j=1}^{i} H_{K,j}$ and $\bigwedge_{j=1}^{i} p_{K,j}(\mathbf{x}) < 0$, respectively, and (c) the interior is nonempty. We use "basic semialgebraic" and "nondegenerate basic semialgebraic" interchangeably.

2.2 Switched Systems

We consider continuous-time switched system plants, controlled by a memoryless controller that provides continuous-time switching feedback. The state of the plant $\mathscr{P}$ is defined by n continuous variables $\mathbf{x}$ in a state space $X \subseteq \mathbb{R}^n$, along with a finite set of modes $Q = \{q_1, \ldots, q_m\}$. The trace of the system $(q(t), \mathbf{x}(t))$ maps time to mode $q(.) : \mathbb{R}^+ \to Q$, and state $\mathbf{x}(.) : \mathbb{R}^+ \to X$. The mode $q \in Q$ is controlled by an external switching input $q(t)$. The state of the plant inside each mode evolves according to (time invariant) dynamics:

$$\dot{\mathbf{x}}(t) = f_{q(t)}(\mathbf{x}(t)), \tag{1}$$

wherein $f_q : X \to \mathbb{R}^n$ is a Lipschitz continuous function over X, describing the vector field of the plant for mode q.

The controller $\mathscr{C}$ is defined as a function $\mathscr{K} : Q \times X \to Q$, which given the current mode and state of the plant, decides the mode of the plant at the next time instant. Formally:

$$q^+(t) = \mathscr{K}(q(t), \mathbf{x}(t)). \tag{2}$$

The closed loop $\langle \mathscr{P}, \mathscr{C} \rangle$ produces traces $(q(t), \mathbf{x}(t))$ defined jointly by equations (1) and (2). However, care must be taken to avoid *Zenoness*, wherein the controller can switch infinitely often in a finite time interval. Such controllers are physically unrealizable. Therefore, we will additionally ensure that the $\mathscr{K}$ function satisfies a *minimum dwell time* requirement that guarantees a minimum time $\delta > 0$ between mode switches.

Definition 2 *(Minimum Dwell Time): A controller* $\mathscr{C}$ *has a minimum dwell time* $\delta > 0$ *with respect to a plant* $\mathscr{P}$ *iff for all traces and for all switch times* T *(*$q(T) \neq q^+(T)$*, the controller does not switch during the times* $t \in [T, T+\delta)$*: i.e,* $\mathscr{K}(q(t), \mathbf{x}(t)) = q^+(T)$ *for all* $t \in [T, T+\delta)$*.*

Once the function $\mathscr{K}$ is defined with a minimum dwell time guarantee, given initial mode $(q(0))$, and initial state $(\mathbf{x}(0))$, a unique trace is defined for the system.

Specifications: Generally, specifications describe desired sequences of plant states $\mathbf{x}(t)$ over time $t \geq 0$ that we wish to control for. In this paper, we focus on *reach-while-stay* (RWS) specifications involving three sets: *initial set* $I \subseteq X$, *safe set* $S \subseteq X$ and *goal set* $G \subseteq X$.

Definition 3 *(Initialized Reach-While-Stay (RWS) Specification): A trace* $\mathbf{x}(t)$ *for* $t \in [0, \infty)$ *satisfies a reach-while-stay (RWS) specification w.r.t sets* $\langle I, S, G \rangle$ *iff whenever* $\mathbf{x}(0) \in I$*, there exists a time* $T \geq 0$ *s.t. for all* $t \in [0, T)$*,* $\mathbf{x}(t) \in S$*, and* $\mathbf{x}(T) \in G$*.*

In other words, whenever the system is initialized inside the set I, it stays inside the safe set S until it reaches the goal set G. Alternatively, we may express the specification in temporal logic as $I \implies (S \mathscr{U} G)$, where $\mathscr{U}$ is the temporal operator "until".

We will assume that set S is a compact basic semialgebraic set. Typical examples include polytopes defined by linear inequalities or ellipsoids, that can be easily checked for the properties such as compactness and nondegeneracy. Also, sets I and G are compact semialgebraic sets.

The special case when $I = S$ will be called *uninitialized* RWS. Such a property simply states that the system initialized inside the set S continues to remain in S until it reaches a goal state $\mathbf{x} \in G$ at some finite time instant T. This case is suitable for building a finite abstraction as mentioned in Sec. 1.

2.3 Control Certificates

Encoding verification and synthesis problems into (control) certificates, which are defined by a set of conditions, is a standard approach. For example Lyapunov functions have been used for ensuring stability and barrier functions are employed to reason about safety properties. However, these functions are not usually known in advance. To discover such a function in the first place, we solve a constrained problem in which certificates are parameterized. Usually, certificates are defined over polynomials with unknown coefficients and the problem reduces to finding proper coefficients for polynomials [21, 2]. For example, to find a Lyapunov function, first, a template for Lyapunov function V is chosen: $V = \sum_i \mathbf{c}_\alpha \mathbf{x}^\alpha$, where $\mathbf{x}^\alpha$ is a monomial with degree greater than zero. Then, solving the following constrained problem yields a Lyapunov function for proving stability to origin: $(\exists \mathbf{c})\ (\forall \mathbf{x} \neq 0)\left(V(\mathbf{x}) > 0 \wedge \dot{V}(\mathbf{x}) < 0\right)$, where $\dot{V}$ is $\nabla V.f(\mathbf{x})$. In these techniques, it is essential to define *control* certificate with a simple structure that can be discovered automatically. In the subsequent we combine the certificates for safety and liveness to obtain a certificate for RWS properties.

3 RWS for Basic Semialgebraic Safe Sets

In this section, we first focus on the uninitialized RWS problem ($I = S$) and provide solutions for the case when S is a basic nondegenerate semialgebraic set (see Def. 1).

Let S be a nondegenerate basic semialgebraic sets, as in Def. 1. Let ∂S be partitioned into nonempty facets $F_1, \ldots, F_{l_k}$. Each facet F_k is, in turn, defined by two sets of polynomial inequalities $F_k^<$ of inactive constraints and $F_k^=$ of active constraints: $F_k = \{\bigwedge_{p_{S,j} \in F_k^<} p_{S,j}(\mathbf{x}) < 0 \ \wedge \ \bigwedge_{p_{S,j} \in F_k^=} p_{S,j}(\mathbf{x}) = 0\}$.

For each state on a facet and not in G, we require the existence of a mode q, whose vector field points inside S. Additionally, we will require a certificate V to decrease everywhere in $S \setminus G$. For any polynomial p, let $\dot{p}_q : (\nabla p) \cdot f_q(\mathbf{x})$. By combining conditions for safety and liveness, one can obtain the following conditions:

$$\begin{cases} \mathbf{x} \in int(S) \setminus G \implies (\exists\, q)\ \dot{V}_q(\mathbf{x}) < -\varepsilon \\ \mathbf{x} \in F_1 \setminus G \implies (\exists\, q) \left(\dot{V}_q(\mathbf{x}) < -\varepsilon \ \wedge \bigwedge_{p \in F_1^=} \dot{p}_q(\mathbf{x}) < -\varepsilon \right) \\ \vdots \\ \mathbf{x} \in F_{l_k} \setminus G \implies (\exists\, q) \left(\dot{V}_q(\mathbf{x}) < -\varepsilon \ \wedge \bigwedge_{p \in F_{l_k}^=} \dot{p}_q(\mathbf{x}) < -\varepsilon \right). \end{cases} \tag{3}$$

The first condition in Eq. (3) states that V must strictly decrease everywhere in the set $int(S) \setminus G$. The subsequent conditions treat each facet F_j of the set S and posit the existence of a mode q for each state that causes the active constraints and the function V to decrease.

However, we note that as the number of state variables increases, the number of facets can be exponential in the number of inequalities that define S [8] . This poses a serious limitation to the applicability of Eq. (3).

Our solution to this problem, is based partly on the idea of exponential barriers discussed by Kong et al. [11]. Rather than force the vector field to point inwards at each facet, we simply ensure that each polynomial inequality $p_{S,j} \leq 0$ that defines S, satisfies a decrease condition outside set G. Thus, Eq. (3) is replaced by a simpler (relaxed) condition:

$$\mathbf{x} \in S \setminus G \implies (\exists\, q)\ \dot{V}_q(\mathbf{x}) < -\varepsilon\ \wedge \bigwedge_j \left(\ (\dot{p_{S,j,q}}(\mathbf{x}) + \lambda\ p_{S,j}(\mathbf{x}))\ < -\varepsilon\ \right). \tag{4}$$

Here $\lambda > 0$ is a user specified parameter. This rule is a relaxation of (3). The rule is made stronger for larger values of λ. However, larger values of λ can cause numerical difficulties in practice while searching for a control certificate.

For safety constraints, we require $\dot{p}_q$ to be numerically $\leq -\varepsilon$ mainly, to avoid numerical issues. This can be restrictive for cases where $\dot{p_{S,j,q}}$ is simply zero. To go around this, we define a set of facets $J_q = \{j | (\exists x)\ \dot{p_{S,j,q}}(x) > 0\}$ for each mode q. Informally speaking, J_q is set of all facets for which change of $p_{S,j}$ must be considered when mode q is selected. Because for each facet $j \notin J_q$, $p_{S,j}$ will never increase as long as mode q is selected. Then, the conditions become:

$$\mathbf{x} \in S \setminus G \implies (\exists\, q)\ \dot{V}_q(\mathbf{x}) < -\varepsilon\ \wedge \bigwedge_{j \in J_q} \left((\dot{p_{S,j,q}}(\mathbf{x}) + \lambda\ p_{S,j}(\mathbf{x}))\ < -\varepsilon\right). \tag{5}$$

As mentioned earlier, the problem of control synthesis consists of two phases. The first phase deals with the problem of *finding a control certificate* $V(\mathbf{x})$ that satisfies (5). We use a *counter-example guided inductive synthesis* (CEGIS) framework to find such certificates. In the second phase, a switching strategy is extracted from the control certificate to design the final controller. We now examine each phase, in turn.

3.1 Discovering Control Certificates

We now explain the CEGIS framework that searches for a suitable control certificate V. To synthesize a control certificate, we start with a parametric form $V_{\mathbf{c}}(\mathbf{x}) = V(\mathbf{c}, \mathbf{x}) : \sum_{i=1}^{N} c_i g_i(\mathbf{x})$ with some (nonlinear) basis functions $g_1(\mathbf{x}), \ldots, g_N(\mathbf{x})$ chosen by the user, and unknown coefficients $\mathbf{c} : (c_1, \ldots, c_N)$, s.t. $\mathbf{c} \in C$ for a compact set $C \subseteq \mathbb{R}^N$. The certificate V is a linear function over $\mathbf{c}$.

The constraints from Eq. (5) become as follows:

$$(\exists \mathbf{c} \in C)\ (\forall \mathbf{x} \in X)\ \mathbf{x} \in S \setminus G \implies \bigvee_q \left(\dot{V}_q < -\varepsilon \wedge \bigwedge_{j \in J_q} (\dot{p_{S,j,q}}(\mathbf{x}) + \lambda\, p_{S,j}(\mathbf{x}) < -\varepsilon)\right). \tag{6}$$

The constraints in Eq. (6) has a complex quantifier alternation structure involving the $\exists \mathbf{c}$ quantifier nested outside the $\forall \mathbf{x}$ quantifier. First, we note that J_q is computed separately and here we assume it is given. Next, we modify an algorithm commonly used for program synthesis problems to the problem of synthesizing the coefficients $\mathbf{c} \in C$ [27].

The counterexample guided inductive synthesis (CEGIS) approach has its roots in program synthesis, wherein it was proposed as a general approach to solve $\exists\forall$ constraints that arise in such problems [27].

The key idea behind the CEGIS approach is to find solutions to such constraints while using a satisfiability (feasibility) solver for quantifier-free formulas that check whether a given set of constraints without quantifiers have a feasible solution.

Solvers like Z3 allow us to solve many different classes of constraints with extensive support for linear arithmetic constraints [18]. On the other hand, general purpose nonlinear delta-satisfiability solvers like dReal, support the solving of quantifier-free nonlinear constraints involving polynomials, trigonometric, and rational functions [4]. However, the presence of quantifiers drastically increases the complexity of solving these constraints. Here, we briefly explain the idea of CEGIS procedure for $\exists\forall$ constraints of the form

$$(\exists\, \mathbf{c} \in C)\ (\forall\, \mathbf{x} \in X)\ \Psi(\mathbf{c},\mathbf{x}).$$

Here, $\mathbf{c}$ represents the unknown coefficients of a control certificate and $\mathbf{x}$ represents the state variables of the system. Our goal is to find one witness for $\mathbf{c}$ that makes the overall quantified formula true. The overall approach constructs, maintains, and updates two sets iteratively:

1. $X_i \subseteq X$ is a finite set of *witnesses*. This is explicitly represented as $X_i = \{\mathbf{x}_1, \ldots, \mathbf{x}_i\}$.
2. $C_i \subseteq C$ is a (possibly infinite) subset of available *candidates*. This is implicitly represented by a constraint $\psi_i(\mathbf{c})$, s.t. $C_i :\ \{\mathbf{c}\ \in C \mid \psi_i(\mathbf{c})\}$.

In the beginning, $X_0 = \{\}$ and $\psi_0 :\ \mathit{true}$ representing the set $C_0 :\ C$.

At each iteration, we perform the following steps:

(a) *Choose* a candidate solution $\mathbf{c}_{i+1} \in C_i$. This is achieved by checking the feasibility of the formula ψ_i. Throughout this paper, we will maintain ψ_i as a *linear arithmetic* formula that involves boolean combinations of linear inequality constraints. Solving these problems is akin to solving linear optimization problems involving disjunctive constraints. Although the complexity is NP-hard, solvers like Z3 integrate fast LP solvers with Boolean satisfiability solvers to present efficient solutions [18].

(b) *Test* the current candidate. This is achieved by testing the satisfiability of $\neg\Psi(\mathbf{c},\mathbf{x})$ for fixed $\mathbf{c} = \mathbf{c}_{i+1}$. In doing so, we obtain a set of nonlinear constraints over $\mathbf{x}$. We wish to now check if it is feasible.

If $\neg\Psi(\mathbf{c}_{i+1},\mathbf{x})$ has no feasible solutions, then $\Psi(\mathbf{c}_{i+1},\mathbf{x})$ is true (valid) for all $\mathbf{x}$. Therefore, we can stop with $\mathbf{c} = \mathbf{c}_{i+1}$ as the required solution for $\mathbf{c}$.

Otherwise, if $\neg\Psi(\mathbf{c},\mathbf{x})$ is feasible for some $\mathbf{x} = \mathbf{x}_{i+1}$, we add it back as a witness: $X_{i+1} :\ X_i \cup \{\mathbf{x}_{i+1}\}$. The formula ψ_{i+1} is given by

$$\psi_{i+1} :\ \psi_i\ \wedge\ \Psi(\mathbf{c},\mathbf{x}_{i+1})\,.$$

Note that $\psi_{i+1} \implies \psi_i$, and $\mathbf{c}_{i+1}$ is no longer a feasible point for ψ_{i+1}. The set C_{i+1} described by ψ_{i+1} is:

$$C_{i+1} :\ \{\mathbf{c} \in C \mid \Psi(\mathbf{c},\mathbf{x}_i) \text{ holds for each } \mathbf{x}_i \in X_{i+1}\}\,.$$

The CEGIS procedure either (i) runs forever, or (ii) terminates after i iterations with a solution $\mathbf{c} : \mathbf{c}_i$, or (iii) terminates with a set of witness points X_i proving that no solution exists.

We now provide further details of the CEGIS procedure adapted to find a certificate that satisfies Eq. (6). In the CEGIS procedure, the formula $\Psi(\mathbf{c},\mathbf{x})$ will have the following form:

$$\begin{cases} \mathbf{x} \in R_1 \implies \varphi_1(\mathbf{c},\mathbf{x}) \\ \mathbf{x} \in R_2 \implies \varphi_2(\mathbf{c},\mathbf{x}) \\ \ldots \\ \mathbf{x} \in R_{N_j} \implies \varphi_{N_j}(\mathbf{c},\mathbf{x})\,, \end{cases} \tag{7}$$

and each φ_j for $j = 1, \ldots, N_j$ has the form

$$\bigvee_k \bigwedge_l p_{j,k,l}(\mathbf{c}, \mathbf{x}) > 0\,, \tag{8}$$

where $p_{j,k,l}(\mathbf{c}, \mathbf{x})$ is a function linear in $\mathbf{c}$ and possibly nonlinear in $\mathbf{x}$, depending on the dynamics and template used for the control certificate.

The CEGIS procedure involves two calls to solvers: (a) Testing satisfiability of $\psi_i(\mathbf{c})$ and (b) Testing the satisfiability of $\neg\Psi(\mathbf{c}_{i+1}, \mathbf{x})$. We shall discuss each of these problems in the following paragraphs.

Finding Candidate Solutions: Given a finite set of witnesses X_i, a solution exists for ψ_{i+1} iff there exists $\mathbf{c} \in C$ s.t.

$$\bigwedge_{\mathbf{x} \in X_i} \bigwedge_{j=1}^{N_j} \left(\mathbf{x} \in R_j \implies \bigvee_k \bigwedge_l p_{j,k,l}(\mathbf{c}, \mathbf{x}) > 0 \right),$$

and since $p_{j,k,l}$ is a linear function in $\mathbf{c}$, such $\mathbf{c}$ can be found by solving a formula in Linear Arithmetic Theory ($\mathscr{LA}$).

Finding Witnesses: Finding a witness for a given candidate solution $\mathbf{c}_i$ involves checking the satisfiability $\neg\Psi$. Whereas Ψ is a conjunction of N_j clauses, $\neg\Psi$ is a disjunction of clauses. The j^{th} clause in $\neg\Psi$ ($1 \leq j \leq N_j$) has the form

$$\mathbf{x} \in R_j \wedge \bigwedge_k \bigvee_l p_{j,k,l}(\mathbf{c}_i, \mathbf{x}) \leq 0\,. \tag{9}$$

We will test each clause separately for satisfiability. Assuming that $p_{j,k,l}$ is a general nonlinear function over $\mathbf{x}$, SMT solvers like dReal [4] can be used to solve this over a compact set R_j. Numerical SMT solvers like dReal can either conclude that the given formula is unsatisfiable or provide a solution to a "nearby" formula that is δ close. The parameter δ is adjusted by the user. As a result, dReal can correctly conclude that the current candidate yields a valid certificate. On the other hand, its witness may not be a witness for the original problem. In this case, using the spurious witness may cause the CEGIS procedure to potentially continue (needlessly) even when a solution $\mathbf{c}_i$ has been found. Nevertheless, the overall procedure produces a correct result whenever it terminates with an answer.

Example 1 *This example is adopted from [19]. There are two variables and three control modes with the dynamics given below:*

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} -x_2 - 1.5x_1 - 0.5x_1^3 \\ x_1 \end{bmatrix} + B_q,\ B_{q1} = \begin{bmatrix} 0 \\ -x_2^2 + 2 \end{bmatrix},\ B_{q2} = \begin{bmatrix} 0 \\ -x_2 \end{bmatrix},\ B_{q3} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}.$$

The goal is to reach the target set $G : (x_1 + 0.75)^2 + (x_2 - 1.75)^2 \leq 0.25^2$*, a circle centered at* $(-0.75, 1.75)$*, as shown in Figure 1a, while staying in the safe region given by the rectangle* $S_0 : [-2, 2] \times [-2, 3]$*:*

$$S_0 : \{\mathbf{x} | (x_1 + 2)(x_1 - 2) \leq 0 \wedge (x_2 + 2)(x_2 - 3) \leq 0\}\,.$$

First, we shift co-ordniates to transform $(-0.75, 1.75)$ *as the new origin. Then, we use a quadratic template for* V $(c_1 x_1^2 + c_2 x_1 x_2 + c_3 x_2^2)$ *,* $\varepsilon = 1$*,* $\lambda = 5$*. The solution* V *is found in* 5 *iterations. Then, we translate the function back to the original co-ordinates:*

$$V(x_1, x_2) : 37.782349x_1^2 - 2.009762x_1x_2 + 60.190607x_1 + 4.415093x_2^2 - 16.960145x_2 + 37.411604\,.$$

Example 2 *A unicycle [25] has three variables. x and y are position of the car and θ is its angle. The dynamics of the system is $\dot{x} = u_1 cos(\theta)$, $\dot{y} = u_1 sin(\theta)$, $\dot{\theta} = u_2$, where u_1 and u_2 are inputs. Assuming a switched system, we consider $u_1 \in \{-1,0,1\}$ and $u_2 \in \{-1,0,1\}$. The safe set is $[-1,1]\times[-1,1]\times[-\pi,\pi]$ and the target facet is $x = 1$. We use a template that is linear in (x,y) and quadratic in θ. Using $\varepsilon = 0.1$ and $\lambda = 0.5$, the following CLF is found after 22 iterations:*

$$V(\mathbf{x}) : -x - y - 0.5881\theta + \theta^2 - 0.1956\theta x + \theta y .$$

Example 3 *This example is adopted from [7]. There are four variables and two control inputs. The dynamic is as follows:*

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \end{bmatrix} = \begin{bmatrix} x_1 + x_2 + 8 \\ -x_2 + x_3 + 1 \\ -2x_3 + 2x_4 + 1 \\ -3x_4 + 1 \end{bmatrix} + \begin{bmatrix} u_1 \\ -u_2 \\ -2u_1 \\ u_2 \end{bmatrix} .$$

The region of interest S is hyber-box $[-1,1]^4$ and the input belongs to set $[0,1]\times[0,2]$. The goal is to reach facet $x_1 = 1$, while staying in S as the safe region.

First, we discretize the control input to model the system as a switched system. For this purpose, we assume $u_1 \in \{0,1\}$ and $u_2 \in \{0, 0.5, 1, 1.5, 2\}$. Then, we use a linear template for the CLF ($c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4$), $\varepsilon = 0.1$, $\lambda = 5$. CEGIS framework finds certificate $V(\mathbf{x}) : -0.13333344(x_1 + x_2 + x_3 + x_4)$.

3.2 Control Design

Thus far, we discussed the CEGIS framework for finding a control certificate. Extracting the $\mathscr{K}$ function from the certificate is now considered. Given a control certificate V satisfying Eq. (5), the choice of a switching mode is dictated by a function $\eta_q(\mathbf{x})$ defined for each state $\mathbf{x} \in X$ and mode $q \in Q$ as follows:

$$\eta_q(\mathbf{x}) : \max\left(\dot{V}_q(\mathbf{x}),\ \eta_{S,1,q}(\mathbf{x}),\ \cdots, \eta_{S,k,q}(\mathbf{x}) \right) ,$$

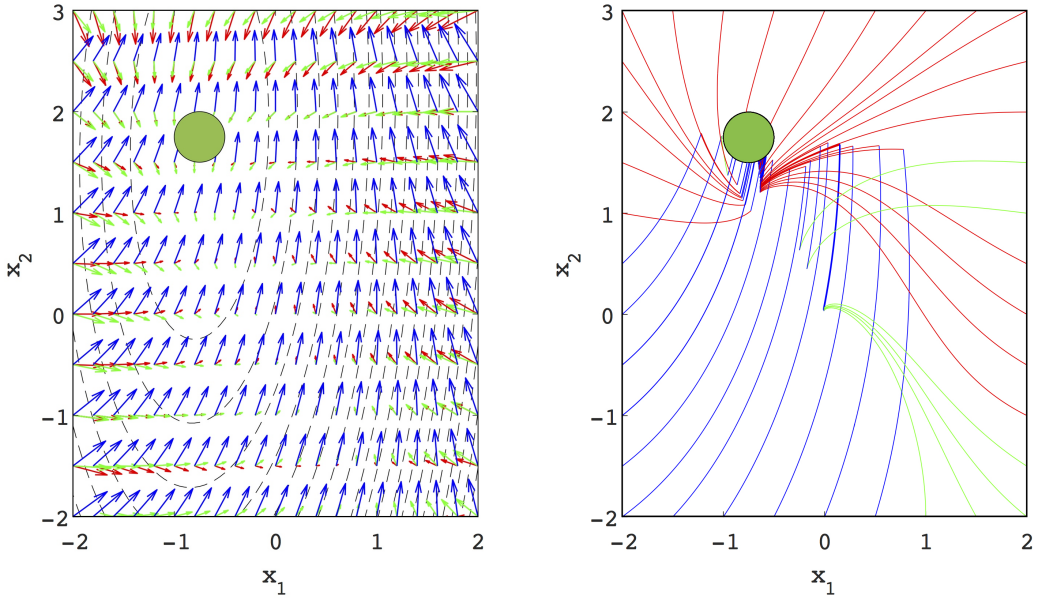
where for all $j \in J_q$, $\eta_{S,j,q}$ is $\dot{p_{S,j,q}} + \lambda p_{S,j}$ and for $j \notin J_q$, $\eta_{S,j,q} = -\infty$ or equivalently $\eta_{S,j,q} = -L$ for some large constant L.

The idea is that whenever (at time t) the controller chooses a mode q s.t. $\eta_q(\mathbf{x}(t)) < -\varepsilon$, one can guarantee that $\eta_q(\mathbf{x}(t)) < 0$ holds for all $t \in [T, T+\delta)$, for some minimum time δ. Therefore, for some fixed ε_s ($0 < \varepsilon_s < \varepsilon$), the function $\mathscr{K}$ for any $\mathbf{x} \in S \setminus G$ can be defined as

$$\mathscr{K}(q,\mathbf{x}) := \begin{cases} \hat{q} & \text{if } \left(\ \eta_q(\mathbf{x}) \geq -\varepsilon_s \wedge\ \eta_{\hat{q}}(\mathbf{x}) < -\varepsilon \ \right) \\ \\ q & \text{otherwise}. \end{cases} \tag{10}$$

In other words, the controller state persists in a given mode q until $\eta_q(\mathbf{x}) \geq -\varepsilon_s$. Then, given that $\mathbf{x} \in S \setminus G$, Eq. (5) will provide us a new control mode $\hat{q}$ that satisfies $\eta_{\hat{q}}(\mathbf{x}) < -\varepsilon$. This mode is chosen as the next mode to switch to.

Example 4 *Consider once again, the problem from Ex. 1. Using the defined function $V(x_1,x_2)$, Eq. (10) yields a controller. Figure 1b shows some of the simulation traces of this closed loop system, demonstrating the RWS property.*



(a) (b)

Figure 1: (a) Region G for Example 1 is shown shaded in the center, and the vector fields for modes q_1, q_2 and q_3 are shown in red, green and blue, respectively. Level-sets of V are shown with black dashed lines. (b) Closed loop trajectories for Example 1 using the controller defined by Eq. (10). The segments shown in colors red, green and blue correspond to the modes q_1, q_2 and q_3, respectively.

We now establish the key result that provides a minimum dwell time guarantee.

Lemma 1 *There exists a $\delta > 0$ s.t. for all initial conditions $\mathbf{x}(T) \in S \setminus G$, if $\eta_q(\mathbf{x}(T)) < -\varepsilon$, and if the mode of the system is set to q at time T, then*

$$(\forall t \in [T, T+\delta])\ (\mathbf{x}(t) \in S \setminus G) \implies \eta_q(\mathbf{x}(t)) \leq -\varepsilon_s\,.$$

Proof 1 *Let $T+\delta$ be the earliest time instant, where $\eta_q(\mathbf{x}(T+\delta)) \geq -\varepsilon_s$ while at the same time*

$$(\forall t \in [T, T+\delta])\ q(t) = q,\ \mathbf{x}(t) \in S \setminus G\,.$$

At time T, $\eta_q(\mathbf{x}(T)) < -\varepsilon$ and at time $T+\delta$, $\eta_q(\mathbf{x}(T+\delta)) = -\varepsilon_s$. Note that $\eta_q(\mathbf{x})$ is defined as $\max(\alpha_1(\mathbf{x}), \ldots, \alpha_m(\mathbf{x}))$ for some smooth functions $\alpha_1, \ldots, \alpha_m$. As a result, Since S is a bounded set, and p, f_q, and V are bounded over S, there exists a constant $\Lambda > 0$ s.t.

$$(\forall\, \mathbf{x} \in S)\ \dot{\alpha}_{i,q} \leq \Lambda\,. \tag{11}$$

Therefore, for each α_i, we have

$$\alpha_i(\mathbf{x}(T+\delta)) = \alpha_i(\mathbf{x}(T)) + \int_{t=T}^{T+\delta} \dot{\alpha}_{i,q}(\mathbf{x}(t))dt \leq \alpha_i(\mathbf{x}(T)) + \Lambda\delta\,.$$

As a result, we conclude that

$$\eta_q(\mathbf{x}(T+\delta)) = \max_i \alpha_i(\mathbf{x}(T+\delta)) = \alpha_{j^*}(\mathbf{x}(T+\delta)) \leq \alpha_{j^*}(\mathbf{x}(T)) + \Lambda\delta \leq \eta_q(T) + \Lambda\delta\,.$$

Therefore, we can conclude $-\varepsilon_s < -\varepsilon + \Lambda\delta \implies \frac{\varepsilon-\varepsilon_s}{\Lambda} < \delta$ *and there exists a fixed* $\delta > \frac{\varepsilon-\varepsilon_s}{\Lambda} > 0$ *s.t.*

$$(\forall t \in [T, T+\delta))\ \eta_q(\mathbf{x}(t)) < -\varepsilon_s\,.$$

Eq. (10) gives a switching strategy which respects the min-dwell time and as long as $\mathbf{x}(t) \in S$, the controller guarantees $\eta_{q(t)}(\mathbf{x}(t)) \leq -\varepsilon_s$. I.e. for all $j \in J_q$, $\dot{V}_q(\mathbf{x}(t)) \leq -\varepsilon_s$ and $\dot{p_{S,j,q}}(\mathbf{x}(t)) + \lambda\, p_{S,j}(\mathbf{x}(t)) \leq -\varepsilon_s$.

Theorem 1 *Given nondegenerate basic semialgebraic set S, a semialgebraic set G, and a function V (satisfying Equation* (5)*), the control strategy defined by Eq.* (10) *respects the min-dwell time property and guarantees the RWS property defined by* S, G*:* $S \implies S\mathscr{U}G$.

Proof 2 *As discussed, there exists a controller which respects the min-dwell time property. Also, the controller guarantees* $\dot{V}_q(\mathbf{x}) \leq -\varepsilon_s$ *and* $(\dot{p_{S,j,q}}(\mathbf{x}) + \lambda\, p_{S,j}(\mathbf{x}) \leq -\varepsilon_s$ *(for all* $j \in J_q$*), as long as* $\mathbf{x} \in S \setminus G$.

Assume $\mathbf{x}(t)$ *is on the boundary of S (and not in G) at some time t. Because S is assumed to be a nondegenerate basic semialgebraic set, there exists at least one j s.t.* $p_{S,j}(\mathbf{x}(t)) = 0$. *If* $j \notin J_q$*, by definition,* $\dot{p_{S,j,q}}$ *is negative for all states and* $p_{S,j,q}$ *remains* ≤ 0 *as long as mode q is selected. Otherwise (*$j \in J_q$*), we obtain* $\dot{p_{S,j,q}}(\mathbf{x}(t)) \leq -\varepsilon_s < 0$. *Therefore, there exits* $\tau_j > 0$*, s.t.* $s \in (t, t+\tau_j)$*, we conclude that* $p_{S,j,q}(\mathbf{x}(s)) < 0$. *As a result, the trajectory cannot leave the set S.*

Thus, the trace cannot leave S, unless it reaches G. Now, we show that the trajectory cannot stay inside $S \setminus G$ *forever. By the construction of the controller, we can conclude time diverges (because the controller respects the min-dwell time property) and that V decreases (*$\dot{V}_q(\mathbf{x}(t)) \leq -\varepsilon_s$*). However, the value of V is bounded on bounded set* $S \setminus G$*. Therefore,* $\mathbf{x}$ *cannot remain in* $S \setminus G$ *and the only possible outcome for the trace is to reach G.*

4 RWS for Semialgebraic Safe Set

As Habets et al [6] discussed, control-to-facet problems can be used to build an abstraction. Here, we demonstrate that the method described so far can be integrated in this framework to tackle more complicated problems. First, we briefly explain how the method works. For a more detailed discussion, the reader can refer to [6] or [10].

First, state space is decomposed into polytopes according to the specifications. Here, we can use basic semialgebraic sets instead of polytopes. Then, for each such a set u, we consider an abstract state $\mathscr{A}(u)$. Furthermore, for each of its $n-1$ dimensional facet F, a control-to-facet problem is solved. The corresponding problem is to find a control strategy to reach F starting from u. If the control-to-facet problem is solved successfully, then for each basic semialgebraic set v with a $n-1$ dimensional facet $F' \subseteq F$, an edge from $\mathscr{A}(u)$ to $\mathscr{A}(v)$ (with label/action F) is added to the abstraction. Also for each basic semialgebraic set u, one can check if u is a control invariant to build self loops. However, for RWS properties, self loops are redundant and we skip them here. After building the abstract system, we use standard techniques to solve the problem for finite systems. If the problem could be solved for the abstract system, then, one can design a controller.

First, for each abstract state $\mathscr{A}(u)$, there is at least one action F that agrees with the winning strategy for the abstract system. Let that action be $\mathscr{F}(\mathscr{A}(u))$. The idea is to implement transition $\mathscr{F}(\mathscr{A}(u))$, using controller $\mathscr{K}_{u,\mathscr{F}(\mathscr{A}(u))}$ for the corresponding control-to-facet problem [6]. Formally, the controller

can be defined as follows:

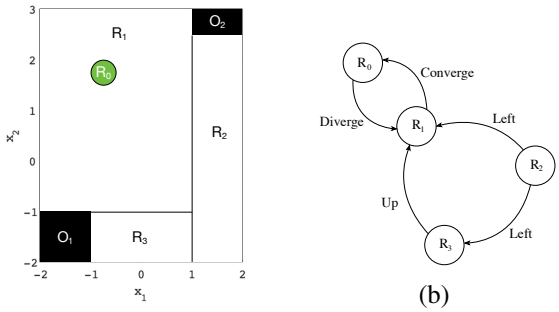
$$\mathscr{K}(q,\mathbf{x}) = \begin{cases} \mathscr{K}_{u_1,\mathscr{F}(\mathscr{A}(u_1))}(q,\mathbf{x}) & \mathbf{x} \in u_1 \\ \vdots & \\ \mathscr{K}_{u_s,\mathscr{F}(\mathscr{A}(u_s))}(q,\mathbf{x}) & \mathbf{x} \in u_s\,. \end{cases} \tag{12}$$

When $\mathbf{x}$ belongs to multiple sets, one can break the tie by some ordering, where states in the winning set have priorities. It is worth mentioning that combining these controllers together, does not produce any Zeno behavior as it is guaranteed that each abstract state is visited only once for RWS properties. However, superdense switching is possible as two facets of a polytope can get arbitrarily close.

If one is interested in LTL properties (not just reach-while-stay) or min-dwell time property, one possible solution is to use *fat* facets, where the target sets are n dimensional goal sets. This extends the domain of the control-to-facet problem to adjacent basic semialgebraic sets as well. Also, it allows the controller to continue using current sub-controller for some minimum time (if min-dwell time requirement is not met), before changing the sub-controller (at the switch time).

Example 5 *Consider again the system from Example 1, with the addition of some obstacles [19]. More precisely, as shown in Fig 2a, safe set is $S = S_0 \setminus (O_1 \cup O_2)$. First, the safe set is decomposed into four basic semialgebraic sets, which are shown with R_0 to R_3 in Fig. 2a.*

R_0 is the target set. Next, we build a transition relation between four abstract states, representing four basic semialgebraic sets. This is done by solving seven RWS problems for basic semialgebraic sets. For R_1 to R_0, we use a quadratic template for V, and for other problems, we use linear template. The abstract system is shown in Fig. 2b. Next, the problem is solved for the abstract system. The solution to the abstract system is simple: if the state is in R_2, the controller uses the left facet to reach R_1 or R_3. Otherwise, if the state is in R_3, the controller uses the upper facet to reach R_1 and finally, if the state is in R_1, the controller makes sure the state reaches R_0.



(a) (b)

Figure 2: (a) Schematic view of state decomposition. (b) Finite abstraction for the original problem.

Example 6 *This example is a path planning problem for the unicycle [26]. Projection of safe set on x and y yields a maze. The target set is placed at the right bottom corner of the maze (Fig. 3). Using specification-guided technique, we modeled the system with 53 polyhedra. Each polyhedron is treated as a single state and a transition relation is built by solving 113 control-to-facet problems. Then, the problem is solved over the finite graph. The total computation took 1484 seconds. The figure also shows a single trajectory of the closed loop system.*

Example 7 *This example is similar to Example 6, except for the fact that there is no direct control over the angular velocity. More precisely, only the angular acceleration is controllable and the system would have the following dynamics $\dot{x} = u_1\cos(\theta)$, $\dot{y} = u_1\sin(\theta)$, $\dot{\theta} = \omega$, $\dot{\omega} = u_2$. Also, we assume $\omega \in [-1,1]$. By changing the coordinates one can use $r = \sqrt{x^2+y^2}$, $z_1 = x\cos(\theta) + y\sin(\theta)$ and $z_2 = y\cos(\theta) - x\sin(\theta)$ to define position and angle of the car(cf. [12] for details). Then, we use the following template $V(x,y,\theta,\omega) = c_1 r^2 + c_2 z_1 + c_3 z_2\omega + c_4\omega^2$, where the origin is located just outside of the target facet. Using this template, we find control certificates for all 113 control-to-facet problems in 5296 seconds.*

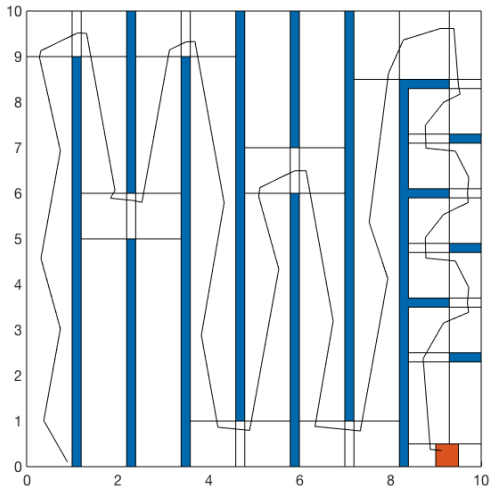


Figure 3: Region G is shown shaded in Orange, and unsafe regions are shown in blue. An execution trace of the car is shown for x and y variables.

5 Initialized Reach-While-Stay

So far, we discussed uninitialized RWS specifications ($S \implies S\mathscr{U}G$). In these systems, we use boundary of safe set as barrier. However, as pointed out by Lin et al. [13], this may not be the case. Now, we consider the initialized problem for a given initial set I ($I \implies S\mathscr{U}G$). To avoid technical difficulties, we assume that $I \subseteq int(S)$. The solution is to create a composite barrier that is formed by the boundary of S as well as other a priori unknown barrier functions.

Barrier Functions: We recall that for a control barrier function [30, 37], the following conditions are considered

$$\begin{aligned} &\mathbf{x} \in \partial S \implies B(\mathbf{x}) > 0 \\ &\mathbf{x} \in I \implies B(\mathbf{x}) < 0 \\ &\mathbf{x} \in S \implies \Big(B(\mathbf{x}) = 0 \implies (\exists q)\dot{B}_q(\mathbf{x}) < -\varepsilon\Big)\,. \end{aligned} \tag{13}$$

This ensures that $B(\mathbf{x}) = 0$ is a barrier and ∂S is unreachable. Eq. (13), combined with the smoothness of B and f_q ensures that as soon as the state is sufficiently "close" to the barrier, it is possible to choose a control mode that ensures the local decrease of the B.

The condition in Equation (13) can be encoded into the CEGIS framework. However, the presence of the equality $B(\mathbf{x}) = 0$ poses practical problems. In particular, it requires for each candidate $B_{\mathbf{c}}$, to find a counterexample $\mathbf{x}$ s.t. $B_{\mathbf{c}}(\mathbf{x}) \neq 0$. Unfortunately, such an assertion is easy to satisfy, resulting in the procedure always exceeding the maximum number of iterations permitted.

Again, we find that the following relaxation of the third condition is particularly effective in our experiments

$$\begin{aligned} &\mathbf{x} \in \partial S \implies B(\mathbf{x}) > 0 \\ &\mathbf{x} \in I \implies B(\mathbf{x}) < 0 \\ &\mathbf{x} \in S \implies \bigvee\nolimits_q \big(\ \dot{B}_q(\mathbf{x}) - \lambda B(\mathbf{x}) < -\varepsilon \vee \dot{B}_q(\mathbf{x}) + \lambda B(\mathbf{x}) < -\varepsilon\ \big)\,, \end{aligned} \tag{14}$$

for some constant λ.

Intuitively, by choosing $\lambda = 0$, the condition is similar to that of Lyapunov functions, whereas as $|\lambda| \to \infty$, the condition gets less conservative and in the limit, it is equivalent to the original condition.

In fact, for smaller $|\lambda|$ CEGIS terminates faster, but at the cost of missing potential solutions. On the other hand, using larger $|\lambda|$, is less conservative at the cost of CEGIS timing out. We also note that this formulation is less conservative than the one introduced by Kong et al. [11] as our formulation uses two exponential conditions which only forces decrease of value of B around $B^* = \{\mathbf{x} \mid B(\mathbf{x}) = 0\}$.

To solve the RWS in general form, we define a finite set of barriers $\mathscr{B}$ with the following conditions:

$$\begin{array}{l} \mathbf{x} \in \partial S \implies \bigvee_{B\in\mathscr{B}} B(\mathbf{x}) > 0 \\ \mathbf{x} \in I \implies \bigwedge_{B\in\mathscr{B}} B(\mathbf{x}) < 0\,. \end{array} \tag{15}$$

Also for each mode q, $\mathscr{B}_q$ is defined as $\mathscr{B}_q = \{B \in \mathscr{B} | (\exists \mathbf{x}) \dot{B}_q(\mathbf{x}) > 0\}$. Then, existence of a proper mode can be encoded as the following:

$$\mathbf{x} \in S \setminus G \implies \left(\left(\dot{V}_q(\mathbf{x}) < -\varepsilon\right) \wedge \bigwedge_{B\in\mathscr{B}_q} \left(\begin{array}{l} \dot{B}_q(\mathbf{x}) + \lambda B(\mathbf{x}) < -\varepsilon \vee \\ \dot{B}_q(\mathbf{x}) - \lambda B(\mathbf{x}) < -\varepsilon \end{array} \right) \right). \tag{16}$$

Theorem 2 *Given nondegenerate basic semialgebraic set S, semialgebraic sets I and G, function V, and a non-empty set of functions $\mathscr{B}$ (satisfying Equation* (15) *and* (16)*), there is a control strategy that respects the min-dwell time property and guarantees the RWS property: $I \implies S\mathscr{U}G$.*

To simplify these constraints and reduce the number of unknowns, one can use some of $p_{S,i}$'s to fix some of these barriers, which yields conditions similar to the ones used for the uninitialized problem. This trick is demonstrated in the following example.

Example 8 *This example is taken from [16], in which a DC-DC converter is modeled with two variables i and v.*

The system has two modes q_1 and q_2, with the following dynamics:

$$q_1 : \begin{cases} \dot{i} = 0.0167i + 0.3333 \\ \dot{v} = -0.0142v \end{cases}$$

$$q_2 : \begin{cases} \dot{i} = -0.0183i - 0.0663v + 0.3333 \\ \dot{v} = -0.0711i - 0.0142v\,. \end{cases}$$

The safe set is $S : [0.65, 1.65] \times [4.95, 5.95]$ and the goal set is $G : [1.25, 1.45] \times [5.55, 5.75]$. We assume initial set to be $I : [0.85, 0.95] \times [5.15, 5.25]$ (Fig. 4). Then, we use 5 barriers $B_0, \ldots, B_4$. Using boundaries of S, we choose $B_1, \ldots, B_4$ as follows:

$$\begin{array}{ll} B_1 = 0.65 - i + \varepsilon_b & B_2 = 1.65 - i + \varepsilon_b \\ B_3 = 4.95 - v + \varepsilon_b & B_4 = 5.95 - v + \varepsilon_b\,, \end{array}$$

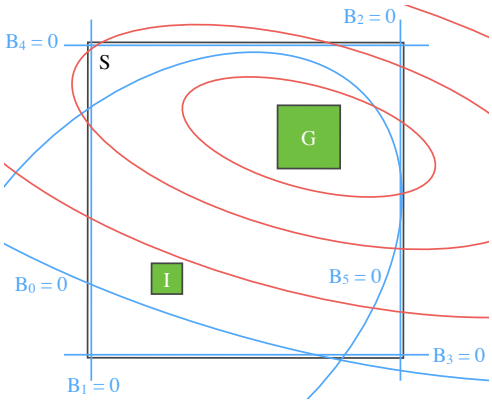


Figure 4: The blue lines are the barriers and the red lines are level-sets of the Lyapunov function.

where $\varepsilon_b > 0$ is small enough that $I \subset int(\bigcap_{i=1}^{4} B_i)$. In this case, we choose $\varepsilon_b = 0.01$. Notice that such ε_b always exists by the definition. Next, we assume $B_0 = V$ and both have the following template:

$$B_0 = V : c_1(i - 1.35)^2 + c_2(i - 1.35)(v - 5.65) + c_3(v - 5.65)^2 - 1\,.$$

This template is chosen in a way that V is a quadratic function with minimum value of -1 for the point of interest $i = 1.35$, $v = 5.65$. So far, we used these tricks to reduce the number of unknowns for barriers.

Table 1: Results of Comparison with SCOTS on examples

Legend: n: # state variables, itr : # iterations, Time: total computation time, η: state discretization step, τ: time step. All timings are in seconds and rounded, TO: timeout ($>$ 10 hours).

Problem		SCOTS				CEGIS	
ID	n	η	τ	itr	Time	δ	Time
Ex. 5	2	0.16^2	0.12	18	0	10^{-4}	3
Ex. 8	2	0.01^2	1.0	106	1	10^{-4}	39
Ex. 6	3	$0.2^2\times0.1$	0.3	404	989	10^{-4}	1484
Ex. 3	4	0.03×0.1^3	0.005	48	304	10^{-5}	3
Ex. 7	4	$0.1^2\times0.05^2$	0.3	TO		10^{-4}	5296

However, our method fails to find a certificate. Next, we add one more barrier (B_5) to the formulation and we use the following template for $B_5 : c_4(i-0.9)^2 + c_5(i-0.9)(v-5.2) + c_6(v-5.05)^2 - 1$, which is a quadratic function with minimum value of -1 for initial point $i = 0.9$, $v = 5.2$. This time, we can successfully find a control certificate. The final barriers and level-sets of the Lyapunov function is shown in Fig. 4.

Comparison: While abstraction based methods can provide a near optimal solution (are relatively complete), these methods can be computationally expensive. On the other hand, our method is a Lyapunov-based method and the solution is not necessarily (relatively) complete. For example, our approach assumes that control certificates with a given form (that is given as input by user) exist . As such, the existence of such certificates is not guaranteed and thus, our approach lacks the general applicability of a fixed-point based synthesis. Also, for initialized problems our method needs an initial set as input, while for the abstraction based methods, maximum controllable region can be obtained without the need for specifying the initial set. However, our method is relatively more scalable thanks to recent development in SMT solvers. Here, for the sake of completeness, we provide a brief comparison with SCOTS toolbox [26] for the examples provided in this article. To compare Example 3, we use *fat* facet and assume target set has a volume (otherwise, because of time discretization, SCOTS cannot find a solution). More precisely, we use target set $[1, 1.2] \times [-1, 1]^3$ instead of $[1, 1] \times [-1, 1]^3$.

All the experiments are ran on a laptop with Core i7 2.9 GHz CPU and 16GB of RAM. The results are reported in Table 1. We also note that if we use larger values for SCOTS parameters, SCOTS fails to solve these problems (initial set is not a subset of controllable region). Table 1 shows that SCOTS performs much better for Example 5 and 8 for which there are only 2 state variables. For Example 6, both methods have similar performances. And for Example. 3 and Example.7, which have 4 state variables, our method is faster.

6 Conclusions

In this paper, given a switched system, we addressed controller synthesis problems for RWS with composite barriers. Specifically, we addressed uninitialized problems which are useful for building an abstraction, as well as initialized problems. For each problem, we provided sufficient conditions in terms of "existence of a control certificate". Also, we demonstrated that searching for a control certificate can be encoded into constrained problems and solving these problems is computationally feasible. In the future,

we wish to investigate how the initialized RWS problems can be extended to be used along fixed-point computation based techniques as it allows more flexible switching strategies.

Acknowledgments

This work was funded in part by NSF under award numbers SHF 1527075 and CPS 1646556. All opinions expressed are those of the authors and not necessarily of the NSF.

References

[1] Zvi Artstein (1983): *Stabilization with relaxed controls*. Nonlinear Analysis: Theory, Methods & Applications 7(11), pp. 1163–1173.

[2] Rayna Dimitrova & Rwitajit Majumdar (2014): *Deductive control synthesis for alternating-time logics*. In: Embedded Software (EMSOFT), 2014 International Conference on, IEEE, pp. 1–10.

[3] L ElGhaoui & V Balakrishnan (1994): *Synthesis of fixed-structure controllers via numerical optimization*. In: Decision and Control, 1994., Proceedings of the 33rd IEEE Conference on, 3, IEEE, pp. 2678–2683.

[4] Sicun Gao, Soonho Kong & Edmund M. Clarke (2013): *dReal: An SMT Solver for Nonlinear Theories over the Reals*. In: Intl. Conference on Automated Deduction (CADE), pp. 208–214.

[5] Antoine Girard, Giordano Pola & Paulo Tabuada (2010): *Approximately bisimilar symbolic models for incrementally stable switched systems*. IEEE Transactions on Automatic Control 55(1), pp. 116–126.

[6] LCGJM Habets, Pieter J Collins & Jan H van Schuppen (2006): *Reachability and control synthesis for piecewise-affine hybrid systems on simplices*. IEEE Transactions on Automatic Control 51(6), pp. 938–948.

[7] LCGJM Habets & Jan H Van Schuppen (2004): *A control problem for affine dynamical systems on a full-dimensional polytope*. Automatica 40(1), pp. 21–35.

[8] Mohamed K Helwa & Mireille E Broucke (2013): *Monotonic reach control on polytopes*. Automatic Control, IEEE Transactions on 58(10), pp. 2704–2709.

[9] Zhenqi Huang, Yu Wang, Sayan Mitra, Geir E Dullerud & Swarat Chaudhuri (2015): *Controller synthesis with inductive proofs for piecewise linear systems: An SMT-based algorithm*. In: 2015 54th IEEE Conference on Decision and Control (CDC), IEEE, pp. 7434–7439.

[10] Marius Kloetzer & Calin Belta (2008): *A fully automated framework for control of linear systems from temporal logic specifications*. Automatic Control, IEEE Transactions on 53(1), pp. 287–297.

[11] Hui Kong, Fei He, Xiaoyu Song, William NN Hung & Ming Gu (2013): *Exponential-condition-based barrier certificate generation for safety verification of hybrid systems*. In: Computer Aided Verification, Springer, pp. 242–257.

[12] Daniel Liberzon (2012): *Switching in systems and control*. Springer Science & Business Media.

[13] Zhiyun Lin & Mireille E Broucke (2007): *Reachability and control of affine hypersurface systems on polytopes*. In: Decision and Control, 2007 46th IEEE Conference on, IEEE, pp. 733–738.

[14] Jun Liu, Necmiye Ozay, Ufuk Topcu & Richard M Murray (2013): *Synthesis of reactive switching protocols from temporal logic specifications*. Automatic Control, IEEE Transactions on 58(7), pp. 1771–1785.

[15] Manuel Mazo Jr, Anna Davitian & Paulo Tabuada (2010): *Pessoa: A tool for embedded controller synthesis*. In: Computer Aided Verification, Springer, pp. 566–569.

[16] Sebti Mouelhi, Antoine Girard & Gregor Gössler (2012): *CoSyMA: A Tool for Controller Synthesis Using Multi-scale Abstractions*. Research Report RR-8108, INRIA. Available at `https://hal.inria.fr/hal-00743982`.

[17] Sebti Mouelhi, Antoine Girard & Gregor Gössler (2013): *CoSyMA: a tool for controller synthesis using multi-scale abstractions*. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*, ACM, pp. 83–88.

[18] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, *LNCS* 4963, Springer, pp. 337–340.

[19] Petter Nilsson & Necmiye Ozay (2014): *Incremental Synthesis of Switching Protocols via Abstraction Refinement*. In: *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, IEEE.

[20] Necmiye Ozay, Jun Liu, Priyanka Prabhakar & Richard M Murray (2013): *Computing augmented finite transition systems to synthesize switching protocols for polynomial switched systems*. In: *American Control Conference (ACC), 2013*, IEEE, pp. 6237–6244.

[21] Stephen Prajna, Antonis Papachristodoulou & Pablo A Parrilo (2002): *Introducing SOSTOOLS: A general purpose sum of squares programming solver*. In: *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, 1, IEEE, pp. 741–746.

[22] Hadi Ravanbakhsh & Sriram Sankaranarayanan (2015): *Counter-Example Guided Synthesis of Control Lyapunov Functions for Switched Systems*. In: *Decision and Control (CDC), 2015 IEEE 54rd Annual Conference on*, IEEE.

[23] Hadi Ravanbakhsh & Sriram Sankaranarayanan (2016): *Robust Controller Synthesis of Switched Systems Using Counterexample Guided Framework*. In: *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, ACM, pp. 8:1–8:10, doi:10.1145/2968478.2968485. Available at `http://doi.acm.org/10.1145/2968478.2968485`.

[24] Bartek Roszak & Mireille E Broucke (2006): *Necessary and sufficient conditions for reachability on a simplex*. *Automatica* 42(11), pp. 1913–1918.

[25] Matthias Rungger, Manuel Mazo & Paulo Tabuada (2012): *Scaling up controller synthesis for linear systems and safety specifications*. In: *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, IEEE, pp. 7638–7643.

[26] Matthias Rungger & Majid Zamani (2016): *SCOTS: A tool for the synthesis of symbolic controllers*. In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, ACM, pp. 99–104.

[27] Armando Solar-Lezama (2008): *Program synthesis by sketching*. ProQuest.

[28] Eduardo D Sontag (1989): *A 'universal' construction of Artstein's theorem on nonlinear stabilization*. *Systems & control letters* 13(2), pp. 117–123.

[29] Ankur Taly, Sumit Gulwani & Ashish Tiwari (2011): *Synthesizing switching logic using constraint solving*. *International journal on software tools for technology transfer* 13(6), pp. 519–535.

[30] Ankur Taly & Ashish Tiwari (2010): *Switching logic synthesis for reachability*. In: *Proceedings of the tenth ACM international conference on Embedded software*, ACM, pp. 19–28.

[31] Weehong Tan & Andrew Packard (2004): *Searching for control Lyapunov functions using sums of squares programming*. In: *Allerton conference on communication, control and computing*, pp. 210–219.

[32] Yuichi Tazaki & Jun-ichi Imura (2012): *Discrete abstractions of nonlinear systems based on error propagation analysis*. *IEEE Transactions on Automatic Control* 57(3), pp. 550–564.

[33] Wolfgang Thomas, Thomas Wilke et al. (2002): *Automata, logics, and infinite games: a guide to current research*. 2500, Springer Science & Business Media.

[34] Peter Wieland & Frank Allgöwer (2007): *Constructive safety using control barrier functions*. *IFAC Proceedings Volumes* 40(12), pp. 462–467.

[35] Tichakorn Wongpiromsarn, Ufuk Topcu & Andrew Lamperski (2016): *Automata theory meets barrier certificates: Temporal logic verification of nonlinear systems*. *IEEE Transactions on Automatic Control* 61(11), pp. 3344–3355.

[36] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu & Richard M Murray (2011): *TuLiP: a software toolbox for receding horizon temporal logic planning*. In: *Proceedings of the 14th international conference on Hybrid systems: computation and control*, ACM, pp. 313–314.

[37] Xiangru Xu, Paulo Tabuada, Jessy W Grizzle & Aaron D Ames (2015): *Robustness of control barrier functions for safety critical control*. *IFAC-PapersOnLine* 48(27), pp. 54–61.

[38] Majid Zamani, Giordano Pola, Manuel Mazo & Paulo Tabuada (2012): *Symbolic models for nonlinear control systems without stability assumptions*. *IEEE Transactions on Automatic Control* 57(7), pp. 1804–1809.

Performance Heuristics for GR(1) Synthesis and Related Algorithms

Elizabeth Firman Shahar Maoz Jan Oliver Ringert
School of Computer Science
Tel Aviv University, Israel

Reactive synthesis for the GR(1) fragment of LTL has been implemented and studied in many works. In this workshop paper we present and evaluate a list of heuristics to potentially reduce running times for GR(1) synthesis and related algorithms. The list includes early detection of fixed-points and unrealizability, fixed-point recycling, and heuristics for unrealizable core computations. We evaluate the presented heuristics on SYNTECH15, a total of 78 specifications of 6 autonomous Lego robots, written by 3rd year undergraduate computer science students in a project class we have taught, as well as on several benchmarks from the literature. The evaluation investigates not only the potential of the suggested heuristics to improve computation times, but also the difference between existing benchmarks and the robot's specifications in terms of the effectiveness of the heuristics.

1 Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [28]. Rather than manually constructing a system and using model checking to verify its compliance with its specification, synthesis offers an approach where a correct implementation of the system is automatically obtained, if such an implementation exists.

GR(1) is a fragment of LTL, which has an efficient symbolic synthesis algorithm [1, 27] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [6, 19]. GR(1) synthesis has been used and extended in different contexts and for different application domains, including robotics [16], scenario-based specifications [23], aspect languages [22], event-based behavior models [5], and device drivers [30], to name a few.

In this workshop paper we present and investigate performance heuristics for algorithms for GR(1) synthesis in case a specification is realizable and Rabin(1) synthesis [14, 24] in case the specification is unrealizable. For the case of unrealizability we also investigate heuristics for speeding up the calculation of unrealizable cores [4, 14], i.e., minimal unrealizable subsets that explain a cause of unrealizability. For each heuristics we present (1) its rationale including the source of the heuristics, if one exists, (2) how we implement it on top of the basic algorithms, and (3) one example where the heuristics is very effective and one example where it does not yield an improvement of performance.

All heuristics we have developed and studied, satisfy three main criteria. First, they are generic, i.e., they are not optimized for a specific specification or family of specifications. Second, they are all low risk heuristics, i.e., in the worst case they may only have small negative effects on performance. Finally, they are conservative, i.e., none of the heuristics changes the results obtained from the algorithms.

We evaluate the presented heuristics on two sets of specifications. The first set, SYNTECH15, consists of 78 specifications of 6 autonomous Lego robots, written by 3rd year undergraduate computer science students in a project class we have taught. The second set consists of specifications for the ARM AMBA AHB Arbiter (AMBA) and a Generalized Buffer from an IBM tutorial (GenBuf), which are the most popular GR(1) examples in literature, used, e.g., in [1, 4, 14, 31]. Our evaluation addresses the

To appear in EPTCS.

effectiveness of each of the heuristics individually and together, and whether there exists a difference in effectiveness with regard to different sets of specifications.

To the best of our knowledge, a comprehensive list of heuristics for GR(1) and its systematic evaluation have not yet been published.

2 Preliminaries

LTL and synthesis We repeat some of the standard definitions of linear temporal logic (LTL), e.g., as found in [1], a modal temporal logic with modalities referring to time. LTL allows engineers to express properties of computations of reactive systems. The syntax of LTL formulas is typically defined over a set of atomic propositions AP with the future temporal operators `X` (next) and `U` (until).

The syntax of LTL formulas over AP is $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \texttt{X}\varphi \mid \varphi\texttt{U}\varphi$ for $p \in AP$. The semantics of LTL formulas is defined over computations. For $\Sigma = 2^{AP}$, a computation $u = u_0u_1.. \in \Sigma^\omega$ is a sequence where u_i is the set of atomic propositions that hold at the i-th position. For position i we use $u,i \models \varphi$ to denote that φ holds at position i, inductively defined as:

- $u,i \models p$ iff $p \in u_i$;
- $u,i \models \neg\phi$ iff $u,i \not\models \phi$;
- $u,i \models \varphi_1 \vee \varphi_2$ iff $u,i \models \varphi_1$ or $u,i \models \varphi_2$;
- $u,i \models \texttt{X}\varphi$ iff $u,i{+}1 \models \varphi$;
- $u,i \models \varphi_1\texttt{U}\varphi_2$ iff $\exists k \geq i\colon\ u,k \models \varphi_2$ and $\forall j, i \leq j < k\colon\ u,j \models \varphi_1$.

We denote $u,0 \models \varphi$ by $u \models \varphi$. We use additional LTL operators `F` (finally), `G` (globally), `ONCE` (at least once in the past) and `H` (historically, i.e., always in the past) defined as:

- $\texttt{F}\varphi := \texttt{true U}\ \varphi$;
- $\texttt{G}\varphi := \neg\texttt{F}\neg\varphi$;
- $u,i \models \texttt{ONCE}\varphi$ iff $\exists 0 \leq k \leq i\colon\ u,k \models \varphi$;
- $u,i \models \texttt{H}\varphi$ iff $\forall 0 \leq k \leq i\colon\ u,k \models \varphi$.

LTL formulas can be used as specifications of reactive systems where atomic propositions are interpreted as environment (input) and system (output) variables. An assignment to all variables is called a state. Winning states are states from which the system can satisfy its specification. A winning strategy for an LTL specification φ prescribes the outputs of a system that from its winning states for all environment choices lead to computations that satisfy φ. A specification φ is called realizable if a strategy exists such that for all initial environment choices the initial states are winning states. The goal of LTL synthesis is, given an LTL specification, to find a strategy that realizes it, if one exists.

μ-Calculus and Fixed-Points The modal μ-calculus is a fixed-point logic [15]. It extends modal logic with least (μ) and greatest (ν) fixed points. We use the μ-calculus over the power set lattice of a finite set of states S, i.e., the values of fixed-points are subsets of S. For monotonic functions ψ over this lattice and by the Knaster-Tarski theorem the fixed points $\mu X.\psi(X)$ and $\nu Y.\psi(Y)$ are uniquely defined and guaranteed to exist. The fixed-points can be computed iteratively [10] in at most $|S|$ iterations due to monotonicity of ψ:

- $\mu X.\psi(X)$: From $X_0 := \bot$ and $X_{i+1} := \psi(X_i)$ obtain $\mu X.\psi(X) := X_f$ for $X_f = \psi(X_f)$ (note $f \leq |S|$)
- $\nu Y.\psi(Y)$: From $Y_0 := \top$ and $Y_{i+1} := \psi(Y_i)$ obtain $\nu Y.\psi(Y) := Y_f$ for $Y_f = \psi(Y_f)$ (note $f \leq |S|$)

The fixed-point computation is linear in $|S|$. When states are represented by a set of atomic propositions (or Boolean variables) AP then $|S| = 2^{|AP|}$, i.e., the number of iterations is exponential in AP.

Because the least (greatest) fixed-point is unique and ψ is monotonic we can safely start the iteration from under-approximations (over-approximations). Good approximations can reduce the number of iterations to reach the fixed-point.

GR(1) Synthesis GR(1) synthesis [1] handles a fragment of LTL where specifications contain initial assumptions and guarantees over initial states, safety assumptions and guarantees relating the current and next state, and justice assumptions and guarantees requiring that an assertion holds infinitely many times during a computation. A GR(1) synthesis problem consists of the following elements [1]:

- $\mathscr{X}$ input variables controlled by the environment;
- $\mathscr{Y}$ output variables controlled by the system;
- θ^e assertion over $\mathscr{X}$ characterizing initial environment states;
- θ^s assertion over $\mathscr{X} \cup \mathscr{Y}$ characterizing initial system states;
- $\rho^e(\mathscr{X} \cup \mathscr{Y}, \mathscr{X})$ transition relation of the environment;
- $\rho^s(\mathscr{X} \cup \mathscr{Y}, \mathscr{X} \cup \mathscr{Y})$ transition relation of the system;
- $J^e_{i \in 1..n}$ justice constraints of the environment to satisfy infinitely often;
- $J^s_{j \in 1..m}$ justice constraints of the system to satisfy infinitely often.

GR(1) synthesis has the following notion of (strict) realizability [1] defined by the LTL formula:

$$\varphi^{sr} = (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathtt{G}((\mathtt{H}\rho^e) \rightarrow \rho^s)) \wedge (\theta^e \wedge \mathtt{G}\rho^e \rightarrow (\bigwedge_{i \in 1..n} \mathtt{GF}J^e_i \rightarrow \bigwedge_{j \in 1..m} \mathtt{GF}J^s_j)).$$

Specifications for GR(1) synthesis have to be expressible in the above structure and thus do not cover the complete LTL. Efficient symbolic algorithms for GR(1) realizability checking and strategy synthesis for φ^{sr} have been presented in [1, 27]. The algorithm of Piterman et al. [27] computes winning states for the system, i.e., states from which the system can ensure satisfaction of φ^{sr}. We denote the states from which the system can force the environment to visit a state in R by $\diamondsuit(R)$ defined as:

$$\diamondsuit(R) = \{q \in 2^{\mathscr{X} \cup \mathscr{Y}} \mid \forall x \in 2^{\mathscr{X}} : \neg\rho^e(q,x) \vee \exists y \in 2^{\mathscr{Y}} : (\rho^s(q, \langle x,y \rangle) \wedge \langle x,y \rangle \in R)\}.$$

The system winning states are given by the following formula using μ-calculus notation:

$$W_{sys} = \nu Z. \bigcap_{j=1}^{m} \mu Y. \bigcup_{i=1}^{n} \nu X. (J^s_j \cap \diamondsuit(Z)) \cup \diamondsuit(Y) \cup (\neg J^e_i \cap \diamondsuit(X)) \tag{1}$$

The algorithm from [1] for computing the set W_{sys} is shown in Alg. 1. Note that this algorithm already contains some performance improvements over the naive evaluation of Eqn. (1), e.g., the nested fixed-points Y are not computed independently for each J^s_j and Z; instead the value of Z is updated before computing J^s_{j+1}. Algorithm 1 stores intermediate computation results in arrays `Z[]` (L. 19), `Y[][]` (L. 16), and `X[][][]` (L. 14). This memory is used for strategy construction [1].

Unrealizability and Rabin(1) Game A specification φ is unrealizable if there is a counter-strategy in which the environment can force the system to violate at least one of its guarantees while satisfying all the environment assumptions. Maoz and Sa'ar [25] show how to compute the fixed-point algorithm given by Könighofer et al. [14] by playing a generalized Rabin game with one acceptance pair (Rabin(1) game[1]). The algorithm computes the set of the winning states for the environment by calculating cycles

[1] We use Rabin(1) to refer to the dual of GR(1) to avoid confusion with "Generalized Rabin(1) synthesis" as defined by Ehlers [7], where assumptions and guarantees are expressed by generalized Rabin(1) conditions.

Algorithm 1 GR(1) game algorithm from [1] to compute system winning states Z

```
 1: Z = true
 2: while not reached fixed-point of Z do
 3:    for j = 1 to |J^s| do
 4:       Y = false; cy = 0
 5:       while not reached fixed-point of Y do
 6:          start = J_j^s ∧ ◇Z ∨ ◇Y
 7:          Y = false
 8:          for i = 1 to |J^e| do
 9:             X = Z // better approx. than true, see [1]
10:             while not reached fixed-point of X do
11:                X = start ∨ (¬J_i^e ∧ ◇X)
12:             end while
13:             Y = Y ∨ X
14:             X[j][i][cy] ← X
15:          end for
16:          Y[j][cy++] ← Y
17:       end while
18:       Z = Y
19:       Z[j] = Y
20:    end for
21: end while
22: return Z
```

Algorithm 2 Rabin(1) game algorithm from [25, 29] to compute environment winning states Z

```
 1: Z = false; cz = 0
 2: while not reached fixed-point of Z do
 3:    for j = 1 to |J^s| do
 4:       Y = true
 5:       while not reached fixed-point of Y do
 6:          start = ¬J_j^s ∧ □Y
 7:          Y = true
 8:          for i = 1 to |J^e| do
 9:             pre = □Z ∨ J_i^e ∧ start
10:             X = false; cx = 0
11:             while not reached fixed-point of X do
12:                X = pre ∨ (¬J_j^s ∧ □X)
13:                X[cz][i][cx++] ← X
14:             end while
15:             Y = Y ∧ X
16:          end for
17:       end while
18:       Z = Z ∨ Y
19:       Z[cz++] ← Y
20:    end for
21: end while
22: return Z
```

violating at least one justice guarantee J_i^s while satisfying all justice assumptions J_j^e. Cycles can be left by the system iff the environment can force it to a future cycle (ensures termination) or to a safety guarantee violation.

We denote the states from which the environment can force the system to visit a state in R by $\boxdot(R)$ defined as:

$$\boxdot(R) = \{q \in 2^{\mathscr{X}\cup\mathscr{Y}} \mid \exists x \in 2^{\mathscr{X}} : \rho^e(q,x) \wedge \forall y \in 2^{\mathscr{Y}} : (\neg\rho^s(q,\langle x,y\rangle) \vee \langle x,y\rangle \in R)\}.$$

The set of environment wining states is given by the following formula using μ-calculus notation:

$$W_{env} = \mu Z.\bigcup_{j=1}^{m} \nu Y.\bigcap_{i=1}^{n} \mu X.(\neg J_j^s \cup \boxdot(Z)) \cap \boxdot(Y) \cap (J_i^e \cup \boxdot(X)) \tag{2}$$

The algorithm from [25] (extended to handle J^e as implemented in JTLV [29]) for computing the set W_{env} is shown in Alg. 2. Again, the algorithm already implements some optimizations over the naive implementation of Eqn. (2), e.g., the early update of Z in L. 18. Algorithm 2 stores intermediate computation results in arrays `Z[]` (L. 19) and `X[][][]` (L. 13) for strategy construction.

Delta Debugging (DDMin) The Delta Debugging algorithm [35] (DDMin) finds a locally minimal subset of a set E for a given monotonic criterion `check`. We show the DDMin algorithm in Alg. 3. The input of the algorithm are a set E and the number n of partitions of E to check. The algorithm starts with $n = 2$ and refines E and n in recursive calls according to different cases (L. 6, L. 11, and L. 14). The computation starts by partitioning E into n subsets and evaluating `check` on each subset *part* (L. 4) and its complement (L. 10). If `check` holds (L. 6 or L. 11), the search is continued recursively on the subset

part (or its complement), until *part* (or its complement) has no subsets that satisfy `check`. If `check` neither holds on any subset *part* nor on the complements the algorithm increases the granularity of the partitioning to $2n$ (L. 14) and restarts.

One application of DDMin is to find an unrealizable core, a locally minimal subset of system guarantees for which a specification is unrealizable. To compute an unrealizable core the method `check` performs a realizability check for the given subset *part* of system guarantees.

Algorithm 3 Delta Debugging algorithm DDMin from [35] as a recursive method that minimizes a set of elements E by partitioning it into n partitions (initial value $n = 2$)

```
1:  if n > |E| then
2:      return E
3:  end if
4:  for part ∈ partition(E,n) do
5:      if check(part) then
6:          return ddmin(part, 2)
7:      end if
8:  end for
9:  for part ∈ partition(E,n) do
10:     if check(E \ part) then
11:         return ddmin(E \ part, n − 1)
12:     end if
13: end for
14: return ddmin(E, min(|E|,2n))
```

Syntax in Examples Throughout the paper we present listings with example specifications that describe GR(1) synthesis problems. We use the following syntax in these specifications:

- $\mathscr{X},\mathscr{Y}$: variables are either environment controlled ($\mathscr{X}$) and introduced by the keyword `env` or system controlled ($\mathscr{Y}$) and introduced by the keyword `sys`; variables have a type and a name, e.g., `sys boolean[4] button` declares a system variable of `boolean` array type of size 4 with the name `button`.
- θ^e,ρ^e,J^e: assumptions are introduced by the keyword `asm`; initial assumptions, i.e., conjuncts of θ^e, are propositional expressions over $\mathscr{X}$, safety assumptions, i.e., conjuncts of ρ^e, start with the temporal operator `G` and are propositional expressions over $\mathscr{X}$ and $\mathscr{Y}$ that may contain the operator `next` to refer to successor values of variables in $\mathscr{X}$, and justice assumptions, i.e., elements J^e_i, start with the temporal operators `GF` and are propositional expressions over $\mathscr{X}$ and $\mathscr{Y}$.
- θ^s,ρ^s,J^s: guarantees are introduced by the keyword `gar`; guarantees are defined analogously to assumptions with the difference that θ^s may also refer to variables in $\mathscr{Y}$ and ρ^s may apply the operator `next` also to variables in $\mathscr{Y}$.

We denote propositional operators by standard symbols, i.e, conjunction ($\wedge$) by `&`, disjunction ($\vee$) by `|`, and negation ($\neg$) by `!`.

3 Suggested Performance Heuristics

We now present a list of heuristics for optimizing running times. The first list applies to the GR(1) and Rabin(1) fixed-point algorithms (Sect. 3.1). The second list applies to computing unrealizable cores (Sect. 3.2). For each heuristics we present a **rationale** including a source of the heuristics, the **heuristics** and how we implemented it in Alg. 1-3, and two **examples** for specifications where (1) the heuristics is effective and where (2) it does not yield an improvement.

3.1 GR(1) and Rabin(1) Fixed-Point Algorithm

3.1.1 Early detection of fixed-point

Rationale. The GR(1) game and the Rabin(1) game iterate over the justice guarantees in the outermost fixed-point. Each iteration refines the set of winning states based on the justice guarantee and the calculated set from the previous iteration (`for`-loop in Alg. 1, L. 3 and Alg. 2, L. 3). Computing a fixed-point for the same justice guarantee J^s_j and the same set Z always yields the same set of winning states. We can exploit the equality to detect if we will reach a fixed-point without completing the `for`-loop, i.e., without computing the fixed-points for all justice guarantees. We found this heuristics implemented in the Rabin(1) game in JTLV [29]. We have not seen a similar implementation for the GR(1) game.

Heuristics. For each iteration of the justice guarantees J^s we save the resulting set of winning states for justice J^s_j as `Z[`j`]` (Rabin(1), `Z[`cz`]`). Starting in the second iteration of the outermost fixed-point we compare for each justice J^s_j the resulting Z of its iteration to the previously computed `Z[`j`]` (Rabin(1), `Z[`$cz-|J^s|$`]`). If the sets are equal the algorithm reached a fixed-point with winning states Z. The heuristics is correct since the next iteration of justice $J^s_{j\oplus 1}$ will start from the set `Z[`j`]` (Rabin(1), `Z[`$cz-|J^s|$`]`), which is the same set it started from when it was previously computed. Hence, $\forall k > j$: `Z[`k`]=Z[`j`]` (`Z[`$cz-|J^s|$`]=Z[`$cz-|J^s|+k$`]`), so by definition we reached a fixed-point for $k = n$ (all justice guarantees).

Examples. Given the realizable GR(1) specification in Listing 1, the standard GR(1) algorithm computes the set of winning states in two iterations of the outer-most loop (Alg. 1, L. 2). The value of Z becomes $a[0] \wedge a[1] \wedge a[2] \wedge a[3]$ after the first step of the loop over the justice guarantees J^s (L. 3). Early fixed-point detection allows the algorithm to stop after checking J^s_1 for the second time ($|J^s|+1$ executions of body of loop in L. 3) instead of going over all justice guarantees again ($2 \cdot |J^s|$ executions of body of loop in L. 3). For the similar specification in Listing 2 with a different order of justices early fixed-point detection does not yield any improvement ($2 \cdot |J^s|$ executions of body of loop in L. 3 are required) because the last justice guarantee changed the fixed-point.

3.1.2 Early detection of unrealizability

Rationale. The GR(1) game and the Rabin(1) game compute all winning states of the system and environment. When running GR(1) synthesis or checking realizability we are interested whether there exists a winning system output for all initial inputs from the environment. When running Rabin(1) synthesis or checking unrealizability we are interested whether there is one initial environment input

Examples: Early Detection of Fixed-Point

```
1 sys boolean[4] a;
2 gar G (a[0] = next(a[0])) &
3       (a[1] = next(a[1])) &
4       (a[2] = next(a[2])) &
5       (a[3] = next(a[3]));
6 gar GF a[0] & a[1] & a[2] & a[3];
7 gar GF a[0];
8 gar GF a[1];
9 gar GF a[2];
```

Listing 1: Heuristics very effective

```
1 sys boolean[4] a;
2 gar G (a[0] = next(a[0])) &
3       (a[1] = next(a[1])) &
4       (a[2] = next(a[2])) &
5       (a[3] = next(a[3]));
6 gar GF a[0];
7 gar GF a[1];
8 gar GF a[2];
9 gar GF a[0] & a[1] & a[2] & a[3];
```

Listing 2: Heuristics does not yield improvement

Examples: Early Detection of Unrealizability

```
1 sys Int(0..10000) c;
2 gar c=10000;
3 gar G next(c)=c+1;
4 gar GF (c mod 2 = 1);
```

Listing 3: Heuristics very effective

```
1 sys Int(0..10000) c;
2 gar c=0; // only difference
3 gar G next(c)=c+1;
4 gar GF (c mod 2 = 1);
```

Listing 4: Heuristics does not yield improvement

such that the environment wins for all system outputs. Thus, in both cases it is not necessary to compute all winning states, instead we can stop computation once we can determine the outcome for the initial states.

Heuristics. The outermost fixed-point in the GR(1) game is a greatest fixed-point. The game starts from the set of all states and refines it to the winning states. Thus, after the computation of the winning states for a justice guarantee we check whether the system still wins from all initial inputs. We implemented this check in Alg. 1 after L. 19. If the system loses for at least one initial environment input we stop the computation of winning states.

The outermost fixed-point in the Rabin(1) game is a least fixed-point. The game starts from an empty set of states and extends it to the winning states. Thus, after the computation of the winning states for a justice guarantee we check whether the environment now wins from some initial input. We implemented this check in Alg. 2 after L. 19. If the environment wins for at least one initial input we stop the computation of winning states.

Examples. Given the unrealizable GR(1) specification in Listing 3, the standard GR(1) algorithm computes the system winning states starting with all possible values of `c`. In every iteration of the Z fixed-point (see Alg. 1, L. 2) two states are removed (the states with largest uneven and even value of `c`). For an integer domain `0..`n (n=10000 in Listing 3) the GR(1) algorithm will compute $n/2$ justice guarantee iterations. Our heuristics will compute only 2 justice guarantee iterations for the example shown in Listing 3. The heuristics will not yield an improvement over the regular GR(1) implementation for the example shown in Listing 4. Here the losing initial state is only detected in iteration $n/2$.

The same examples are also effective and non-effective examples for the Rabin(1) game algorithm.

3.1.3 Fixed-point recycling

Rationale. The GR(1) game and the Rabin(1) game are solved by computing nested fixed-points of monotonic functions (see Eqn. (1) and Eqn. (2)). The time complexity of a straightforward implementation of the fixed-point computation is cubic in the state space and can be reduced to quadratic time [2], as mentioned in [1]. This method can also be applied to the Rabin(1) game. Interestingly, although fixed-point recycling is used to obtain quadratic instead of cubic time complexity of the GR(1) algorithm [1], to the best of our knowledge no GR(1) tool has implemented it following [2] and it has never been systematically evaluated.

Heuristics. Fixed-points are usually computed by fixed-point iteration starting from $\bot$ (least fixed-points) or $\top$ (greatest fixed-points) until a fixed point is reached. The same principle works for the evaluation of nested fixed-points where for each iteration step of the outer fixed-point, the inner fixed-point is computed from scratch. The main idea of [2] is to exploit the monotonicity of fixed-point

Examples: Fixed-Point Recycling

```
sys Int(0..10000) c;
sys boolean two;
gar G two; // force two Z-iterations
gar G (next(c) = c+1) |
      (c=10000 & next(c) = 0);
gar GF c = 0;
asm GF c = 10000;
```

Listing 5: Heuristics very effective

```
sys Int(0..10000) c;
sys boolean two;
gar G two; // force two Z-iterations
gar G (next(c) = c+1) |
      (c=10000 & next(c) = 0);
gar GF c = 0;
asm GF c = 0; // only difference
```

Listing 6: Heuristics does not yield improvement

computations and start nested fixed-point calculations from approximations computed in earlier nested computations. Consider the formula $\mu Z.\nu Y.\mu X.\psi(Z,Y,X)$, iteration $k+1$ of Z, and iteration l of Y: due to monotonicity $Z_k \subseteq Z_{k+1}$ and $Y_l^{of\ Z_k} \subseteq Y_l^{of\ Z_{k+1}}$. Thus, the fixed-point X for Z_k and $Y_l^{of\ Z_k}$ is an under-approximation of the fixed-point X for Z_{k+1} and $Y_l^{of\ Z_{k+1}}$ (see [2] for more details).

In both, the GR(1) algorithm and the Rabin(1) algorithm, the fixed-point computations also depend on justice assumptions J_i^e and justice guarantees J_j^s. This dependence does not interfere with monotonicity of the computation. However, the algorithms compute $|J^e| \cdot |J^s|$ values of the fixed-point X for each iteration of Y (stored in array `X[][][]` in Alg. 1, L. 14).

We implemented this heuristics in the GR(1) game Alg. 1 with a modified start value for the fixed-point computation of X in L. 9. Unless the algorithm computes the first iteration of Z the value of X is set to the previously computed result for the same justice assumption J_i^e and justice guarantee J_j^s and same iteration cy of Y, i.e., X is set to memory cell `X[`j`][`i`][`cy`]` intersected with Z. This value is an over-approximation of the greatest fixed-point X and its computation likely terminates after fewer iterations.

Similarly, we implemented the fixed-point recycling heuristics in the Rabin(1) game Alg. 2 with a modified start value for the fixed-point computation of X in L. 10. Unless the algorithm computes the first iteration of Z the value of X is set to the previously computed result for the same justice assumption J_i^e and justice guarantee J_j^s for the same iteration of Y. This value is an under-approximation of the least fixed-point X and its computation likely terminates after fewer iterations. Note that in Alg. 2 the fixed point value of X is only stored for the last iteration of Y (L. 13). We had to change the implementation to store X for all iterations of Y to use fixed-point recycling as described in [2].

It is important to note that this heuristics changes the worst-case running time of both algorithms from $O(|J^e| \cdot |J^s| \cdot |N|^3)$ to $O(|J^e| \cdot |J^s| \cdot |N|^2)$ [1, 2].

Examples. Consider the realizable GR(1) specification in Listing 5. The variable `c` models a counter from 0 to 10,000 that increases and resets to 0 when reaching 10,000. The second variable `two` serves only the purpose of ensuring two iterations of the Z fixed-point (recycling cannot happen in the first iteration). In the first iteration of Z and Y the nested computation of the X fixed-point requires 10,000 iterations (in each iteration losing one state to end with `two & x=0`). In the second Z and first Y iteration the same computation repeats. Here, the fixed-point recycling heuristics starts from `two & x=0` and finishes after one iteration instead of additional 10,000. It is important to note that on the same specification without variable `two` the heuristics would not yield an improvement because a single Z iteration is enough to detect that all states are winning states. As another example for no improvement, consider the slightly modified specification from Listing 6. Here the single justice guarantee and justice assumption coincide and each nested computation of the X fixed-point requires two iterations with and without recycling.

Examples: Contained Sets in DDMin

```
1 sys boolean x;
2 gar g1: x;
3 gar g2: G TRUE;
4 gar g3: G TRUE;
5 gar g4: G !x;
```

Listing 7: Heuristics very effective

```
1 sys boolean x;
2 gar g1: FALSE;
3 gar g2: G TRUE;
4 gar g3: G TRUE;
5 gar g4: G TRUE;
```

Listing 8: Heuristics does not yield improvement

3.2 Unrealizable Core Calculation

3.2.1 Contained sets

Rationale. The delta debugging algorithm DDMin shown in Alg. 3 might check subsets of guarantees which are contained in previously checked realizable subsets (e.g., after increasing the number of partitions to $2n$ when all other checks failed). In these cases we don't have to execute the costly realizability check: a subset *part* of a realizable set E (failure of `check(`E`)`) is also realizable.

This heuristics was mentioned in [36] and also implemented for unrealizable core calculation in [14].

Heuristics. We extend the generic DDMin algorithm shown in Alg. 3. Before checking a candidate set E', i.e., executing `check(`E'`)`, we look up whether E' is a subset of any previously checked set E with negative evaluation of `check(`E`)`.

Examples. Given the unrealizable GR(1) specification in Listing 7, the computation of an unrealizable core based on DDMin from Alg. 3 calls the method `check` with the following subsets of guarantees (positive results of `check` are underlined): $\{\texttt{g1},\texttt{g2}\}$ (L. 5, $n=2$), $\{\texttt{g3},\texttt{g4}\}$ (L. 5, $n=2$), $\{\texttt{g3},\texttt{g4}\}^*$ (L. 10, $n=2$), $\{\texttt{g1},\texttt{g2}\}^*$ (L. 10, $n=2$), $\{\texttt{g1}\}^*$ (L. 5, $n=4$), $\{\texttt{g2}\}^*$ (L. 5, $n=4$), $\{\texttt{g3}\}^*$ (L. 5, $n=4$), $\{\texttt{g4}\}^*$ (L. 5, $n=4$), $\{\texttt{g2},\texttt{g3},\texttt{g4}\}$ (L. 10, $n=4$), $\underline{\{\texttt{g1},\texttt{g3},\texttt{g4}\}}$ (L. 10, $n=4$), $\{\texttt{g1}\}^*$ (L. 5, $n=3$), $\{\texttt{g3}\}^*$ (L. 5, $n=3$), $\{\texttt{g4}\}^*$ (L. 5, $n=3$), $\{\texttt{g3},\texttt{g4}\}^*$ (L. 10, $n=3$), $\underline{\{\texttt{g1},\texttt{g4}\}}$ (L. 10, $n=3$), $\{\texttt{g1}\}^*$ (L. 5, $n=2$), $\{\texttt{g4}\}^*$ (L. 5, $n=2$), $\{\texttt{g4}\}^*$ (L. 10, $n=2$), and $\{\texttt{g1}\}^*$ (L. 10, $n=2$). Out of these 19 calls to `check` the described heuristics will avoid running the realizability check in the 13 cases marked with a star (*). Given the similar unrealizable specification in Listing 8, the described heuristics does not yield any improvement. The method `check` is never invoked on a subset that it failed on. It is invoked on: $\underline{\{\texttt{g1},\texttt{g2}\}}$ (L. 5, $n=2$) and $\underline{\{\texttt{g1}\}}$ (L. 5, $n=2$).

3.2.2 Incremental GR(1) for similar candidates

Rationale. Due to the nature of the DDMin algorithm (Alg. 3), there are multiple calls to check realizability of subsets of guarantees. Some of the subsets share elements. We can try to reuse computation results from previous calls to `check` for related subsets of guarantees to speed up the computation of fixed-points, both in Rabin(1) and GR(1) games.

Heuristics. The main idea is to reuse results of previous computations of the GR(1) game (Alg. 1) or the Rabin(1) game (Alg. 2). We identified three cases in DDMin (Alg. 3). In each case we use different methods to reuse the computations from previous rounds.

Examples: Incremental GR(1) in DDMin

```
1 sys boolean x;
2 sys boolean y;
3 gar g1: G !y;
4 gar g2: G !x;
5 gar g3: GF !y;
6 gar g4: G x;
```

Listing 9: Heuristics very effective

```
1 sys boolean x;
2 sys boolean y;
3 gar g1: G !y;
4 gar g2: G next(!x); // changed
5 gar g3: GF !y;
6 gar g4: GF x; // changed
```

Listing 10: Heuristics does not yield improvement

Case 1: An unrealizable subset *parent* was found (the set *part* in Alg. 3, L. 5) and DDMin descends to perform the search on subsets of *parent*, starting with $n = 2$. We examine the differences between *parent* and its current subset of guarantees to check. We have the following scenarios:

1. Only initial guarantees were removed from *parent*: In both the GR(1) and Rabin(1) games we can reuse the winning states (Z in Alg. 1 and Alg. 2) that were computed for *parent*, and preform only a simple check for realizability. For GR(1) we check if the system can win from all its initial states. For Rabin(1) we check if the environment can win for some of its initial state.

2. Only safety guarantees were removed from *parent*: Since there are less constraints the attractors Y are larger, hence the set of winning states Z can be larger. In GR(1) we compute Z using greatest fixed-point, so we cannot reuse the previously computed Z_{prev} to initialize Z. However, Z_{prev} is equivalent to the values Y stored as $\mathtt{Z}[j]$ in Alg. 1, L. 19 in the last fixed-point iteration of Z. Thus, Z_{prev} is a safe under-approximation of the least fixed-point Y and we change the initialization of Y in line 4 to $Y = Z_{prev}$.

3. Only justice guarantees were removed from *parent*: We can reuse all information of the previous computation up to the first removed justice guarantee. We reuse the memory $\mathtt{Z}_{prev}$, $\mathtt{Y}_{prev}$, and $\mathtt{X}_{prev}$ from the first iteration of Z on *parent* up to the first removed justice guarantee. Then we continue the computation.

Case 2: All subsets *part* of *parent* are realizable and DDMin continues with complements in Alg. 3, L. 9: In this case and for $n > 2$ the candidates $E \setminus part$ contain previously checked and realizable candidates. Our main observation is that the system winning states for guarantees $E \setminus part$ cannot be more than for any of its subsets. We can check realizability of a GR(1) game by initializing its greatest fixed-point Z to the intersection of system winning states Z_{prev} of previously computed subsets. Alternatively, we can check realizability with a Rabin(1) game by initializing its least fixed point Z to the union of environment winning states Z_{prev} of previously computed subsets.

Case 3: All subsets and complements are realizable and DDMin increases search granularity in Alg. 3, L. 14: For the new run Case 1 applies (with the previous parent) and Case 2 applies when checking complements of the sets with higher granularity.

Examples. The specification in Listing 9 is unrealizable because the system cannot satisfy `g2` and `g4` together. The first set that includes both guarantees in a check of DDMin (Alg. 3, L. 9) is $\{\mathtt{g2}, \mathtt{g3}, \mathtt{g4}\}$. Previously computed winning states are states with `x=true` for $\{\mathtt{g2}\}$ and `x=false` for $\{\mathtt{g3},\ \mathtt{g4}\}$. Their intersection is empty and determines that $\{\mathtt{g2}, \mathtt{g3}, \mathtt{g4}\}$ is unrealizable without even playing a game. The second specification in Listing 10 is very similar. Again the reason for unrealizability are guarantees `g2` and `g4`. However, at the same DDMin step as before the previously computed winning states for subsets of $\{\mathtt{g2}, \mathtt{g3}, \mathtt{g4}\}$ are all states for $\{\mathtt{g2}\}$ and all states for $\{\mathtt{g3},\ \mathtt{g4}\}$. The intersection of these winning states is still the set of all states. In this case our incremental heuristics does not yield improvement.

Examples: GR(1) game vs. Rabin(1) game

```
1 env boolean y;
2 sys Int(0..127) x;
3 asm GF !y;
4 gar G y -> next(x)=x+1;
```

Listing 11: Heuristics very effective

```
1 env boolean y;
2 sys Int(0..127) x;
3
4 gar G y -> next(x)=x+1;
```

Listing 12: Heuristics does not yield improvement

3.2.3 GR(1) game vs. Rabin(1) game

Rationale. GR(1) games and Rabin(1) games are determined: each game is either unrealizable for the system player or unrealizable for the environment player. To check for unrealizability, it is thus equally possible to play the Rabin(1) game or GR(1) game.

The implementations of Könighofer et al. [14] and Cimatti et al. [4] use the GR(1) game for checking realizability during unrealizable core computation.

Heuristics. We replace the implementation of `check`. Instead of playing the GR(1) game we play the Rabin(1) game and negate the result.

Examples. The specification in Listing 11 is unrealizable because the environment can force the system to a deadlock state: the states `x = 127` have no successor for environment input `y = true`. Both the Rabin(1) game and the GR(1) game require $O(n)$ (here $n = 127$) Z iterations to compute the Z fixed-point. Each Z iteration requires two Y iterations. In the Rabin(1) game, each Y iteration requires two X iterations. However, in the GR(1) game[2] another $O(n)$ X iterations are required for each Y iteration. For the similar specification in Listing 12 the numbers of fixed-point iterations of the Rabin(1) game are the same and here also coincide with the number of iterations of the GR(1) game and the heuristics does not contribute.

4 Evaluation

Our evaluation is divided into two parts following the division of heuristics into performance heuristics for the GR(1) and the Rabin(1) algorithm from Sect. 3.1 and performance heuristics for calculating unrealizable cores from Sect. 3.2. For both, we address the following two research questions:

RQ1 What is the effectiveness of each of the heuristics individually and together?

RQ2 Is there a difference in effectiveness with regard to different sets of specifications?

4.1 Procedure

We used the GR(1) game and Rabin(1) game implementations shown in Alg. 1 and Alg. 2 as reference (recall that these algorithms already contain performance improvements over naive implementations following the fixed-point formulation, see Sect. 2). We have implemented these two algorithms and all our suggested heuristics in C using CUDD 3.0 [32]. We measure running-times in nanoseconds using C

[2]For this example we assume initialization of $X = \texttt{true}$ in Alg. 1, L. 9 instead of Z. Note that the optimization of $X = Z$ from [1], that we used in all our experiments as base case, achieves fewer X iterations.

APIs. Our implementation starts with the BDD variable order as it appears in the specification. We use the default dynamic variable reordering of CUDD.

We have executed each realizability check for every specification 50 times (see Sect. 4.5). We aggregated the 50 runs of each specification as a median. The ratios we report are ratios of medians of each heuristics compared to a base case (original implementations of algorithms as shown in Alg. 1-3) for the same specification.

4.2 Evaluation Materials

Only few GR(1) specifications are available and these were usually created by authors of synthesis algorithms or extensions thereof.

For the purpose of evaluation, we have used specifications created by 3rd year CS students in a workshop project class that we have taught. Over the course of a semester, the students have created specifications for the following systems, which they actually built and run: ColorSort – a robot sorting Lego pieces by color; Elevator – an elevator servicing different floors; Humanoid – a mobile robot of humanoid shape; PCar – a self parking car; Gyro – a robot with self-balancing capabilities; and SelfParkingCar - a second version of a self parking car. We call this set of specifications SYNTECH15.

The specifications were *not* created specifically for the evaluation in our paper but as part of the ordinary work of the students in the workshop class. During their work spanning one semester, the students have committed many versions of their specifications to the repository. In total, we have collected 78 specifications. We consider these GR(1) specifications to be the most realistic and relevant examples one could find for the purpose of evaluating our work.

In addition to the specifications created by the students, we considered the ARM AMBA AHB Arbiter (AMBA) and a Generalized Buffer from an IBM tutorial (GenBuf), which are the most popular GR(1) examples in literature, used, e.g., in [1, 4, 14, 31]. We included 5 different sizes of AMBA (1 to 5 masters) and 5 different sizes of GenBuf (5 to 40 requests), each in its original version plus the 3 variants of unrealizability described in [4] (justice assumption removed, justice guarantee added, and safety guarantee added). We have thus run our experiments also on 20 AMBA and 20 GenBuf specifications.

All specifications used in our evaluation, the raw data recorded from all runs, and the program to reproduce our experiments are available from [37].

4.3 Evaluation Results

We now present aggregated data from all runs on all specifications with different heuristics and their combination. We decided to present for all experiments minimum, maximum, and quartiles of ratios.

4.3.1 Results for GR(1)/Rabin(1) Fixed-Point Algorithms

We present the ratios of running times for heuristics from Sect. 3.1 separately for realizable and unrealizable specifications from the set SYNTECH15 and AMBA and GenBuf. The different heuristics are abbreviated as follows: *efp* is the early fixed point detection from Sect. 3.1.1, *eun* is the early unrealizability detection from Sect. 3.1.2, and *fpr* is the fixed-point recycling from Sect. 3.1.3. By *all* we refer to the use of all heuristics together. All results are rounded to two decimals. Tbl. 1 shows the ratios of running times for 61 realizable SYNTECH15 specifications (top) and for 10 realizable AMBA and GenBuf specifications (bottom). Tbl. 2 shows the ratios of running times for 17 unrealizable SYNTECH15 specifications (top) and for 30 unrealizable AMBA and GenBuf specifications (bottom). All tables show

SYNTECH15 realizable

	GR(1) algorithm				Rabin(1) algorithm				Rabin(1) / GR(1)	
Quartile	efp	eun	fpr	all	efp	eun	fpr	all	orig	all
MIN	0.61	0.94	0.6	0.53	0.59	0.92	0.6	0.52	0.52	0.46
Q_1	0.95	1	0.93	0.9	0.94	0.99	0.94	0.9	0.84	0.85
Q_2	0.99	1	0.96	0.95	0.98	1	0.96	0.95	0.91	0.91
Q_3	1	1.02	1	0.98	1	1	0.99	0.99	0.96	0.97
MAX	1.09	1.11	1.1	1.12	1.04	1.08	1.04	1.05	1.29	1.34

AMBA/GenBuf realizable

	GR(1) algorithm				Rabin(1) algorithm				Rabin(1) / GR(1)	
Quartile	efp	eun	fpr	all	efp	eun	fpr	all	orig	all
MIN	0.83	0.97	0.74	0.66	0.84	0.99	0.6	0.58	0.83	0.82
Q_1	0.93	0.99	0.83	0.82	0.91	1	0.86	0.84	0.88	0.88
Q_2	0.99	1	0.92	0.9	0.99	1	0.92	0.91	0.92	0.92
Q_3	1	1	0.95	0.94	1	1	0.96	0.96	0.95	0.94
MAX	1	1.01	0.96	0.96	1.01	1.02	0.99	0.97	1.05	1.04

Table 1: Ratios of the heuristics to the original GR(1) and Rabin(1) running times for realizable specifications.

SYNTECH15 unrealizable

	GR(1) algorithm				Rabin(1) algorithm				Rabin(1) / GR(1)	
Quartile	efp	eun	fpr	all	efp	eun	fpr	all	orig	all
MIN	0.94	0.36	0.87	0.36	0.92	0.61	0.9	0.61	0.51	0.5
Q_1	0.98	0.73	0.97	0.74	0.96	0.84	0.96	0.87	0.84	0.96
Q_2	1	0.88	0.99	0.88	0.99	0.92	0.98	0.92	0.9	1
Q_3	1.02	0.91	1.01	0.91	1	0.95	1	0.94	0.96	1.04
MAX	1.13	0.95	1.15	0.96	1.01	0.97	1.12	0.98	1.04	1.48

AMBA/GenBuf unrealizable

	GR(1) algorithm				Rabin(1) algorithm				Rabin(1) / GR(1)	
Quartile	efp	eun	fpr	all	efp	eun	fpr	all	orig	all
MIN	0.85	0.001	0.93	0.001	0.71	0.001	0.89	0.001	0.68	0.69
Q_1	0.99	0.1	0.99	0.1	0.96	0.09	0.98	0.09	0.87	0.93
Q_2	1	0.54	1	0.52	1	0.62	0.99	0.57	0.93	0.97
Q_3	1	0.97	1.02	0.97	1	0.98	1	0.98	1	1.01
MAX	1.33	1.07	1.06	1.07	1.3	1.01	1.03	1.01	1.85	1.86

Table 2: Ratios of the heuristics to the original GR(1) and Rabin(1) running times for unrealizable specifications.

first ratios of running times for the GR(1) algorithm, then ratios for the Rabin(1) algorithm, and finally a comparison between the Rabin(1) and GR(1) algorithms.

RQ1: Effectiveness of heuristics The heuristics of early fixed-point detection reduces running times by at least 5% on 25% of the realizable specifications (Tbl. 1, *efp*), but seems even less effective on unrealizable specifications (Tbl. 2, *efp*). As expected, the early detection of unrealizability has no notable effect on realizable specifications (Tbl. 1, *eun*), but on unrealizable specifications reduces running times of 50% of the specifications by at least 12%/46% for GR(1) and more than 8%/38% for Rabin(1) (Tbl. 1, *eun*). The heuristics of fixed-point recycling appears ineffective for unrealizable specifications (Tbl. 2), but reduces running times of 25% of the realizable specifications by at least 7%/17% for GR(1) and at least 6%/14% for Rabin(1) (Tbl. 1, *fpr*). As good news, the combination of all heuristics usually improves over each heuristics separately (column *all*). Another interesting observation is that the Rabin(1) algorithm determines realizability faster than the GR(1) algorithm for almost all specifications.

SYNTECH15 unrealizable	DDmin with GR(1)				DDmin with Rabin(1)				Rabin(1) / GR(1)	
Quartile	sets	opt	inc	all	sets	opt	inc	all	orig	all
MIN	0.47	0.66	0.79	0.3	0.44	0.75	0.73	0.35	0.85	0.89
Q_1	0.56	0.94	1.19	0.5	0.59	0.92	1.32	0.51	1.03	1.04
Q_2	0.6	0.96	1.32	0.56	0.65	0.95	1.49	0.55	1.05	1.09
Q_3	0.73	0.97	1.62	0.6	0.74	0.98	1.65	0.65	1.19	1.28
MAX	0.75	0.98	2	0.71	0.78	1.03	2.11	0.78	1.38	1.85

AMBA/GenBuf unrealizable	DDmin with GR(1)				DDmin with Rabin(1)				Rabin(1) / GR(1)	
Quartile	sets	opt	inc	all	sets	opt	inc	all	orig	all
MIN	0.46	0.05	0.91	0.02	0.46	0.04	0.69	0.02	0.66	0.81
Q_1	0.61	0.71	1.08	0.45	0.61	0.72	1.09	0.45	0.93	0.93
Q_2	0.69	0.9	1.35	0.57	0.7	0.94	1.28	0.56	1.02	1.01
Q_3	0.91	0.97	1.46	0.66	0.83	0.97	1.64	0.65	1.13	1.18
MAX	1.2	1.12	2.23	1.09	3.08	1.06	2.38	0.91	1.69	1.41

Table 3: Ratios of the heuristics to the original DDMin running times for unrealizable specifications.

RQ2: Difference between specification sets For realizable specifications, we see that the suggested heuristics perform better on the AMBA and GenBuf set than on SYNTECH15, i.e., all heuristics (columns *all*) decreases running times on 50% of the AMBA and GenBuf specifications by at least 10% and for SYNTECH15 specifications by at least 5%. A more significant difference between the specification sets is revealed by Tbl. 2 of unrealizable specifications. Here the speedup for 50% of the specifications, mainly obtained by *eun*, is at least around 10% for SYNTECH15 but at least around 50% for AMBA and GenBuf. We believe that this difference is due to the systematic and synthetic reasons for unrealizability added by Cimatti et al. [4].

4.3.2 Results for Unrealizable Core Calculation

We present the ratios of running times for heuristics from Sect. 3.2 for unrealizable specifications from the sets SYNTECH15 and AMBA and GenBuf. The different heuristics are abbreviated as follows: *sets* is the contained sets in the core calculation from Sect. 3.2.1, *opt* uses the optimized GR(1) and Rabin(1) algorithms from Sect. 3.1, and *inc* is the incremental algorithm for similar candidates from Sect. 3.2.2. Here, by *all* we refer to the combination of *sets* and *opt* but not *inc*, because only the first two seem to improve running times. All the results are rounded to two decimals (or more if otherwise 0). Tbl. 3 shows the ratios of running times for 17 unrealizable SYNTECH15 specifications (top) and for 30 unrealizable AMBA and GenBuf specifications (bottom). All tables show first ratios of running times for DDMin with the GR(1) algorithm, then ratios for DDMin with the Rabin(1) algorithm, and finally a comparison between the Rabin(1) and GR(1) algorithms.

RQ1: Effectiveness of heuristics The heuristics of contained sets appears very effective on all specifications and reduces running times of 50% of the specifications by at least 40%/31% for DDMin with GR(1) and at least 35%/30% for DDMin with Rabin(1) (Tbl. 3, *sets*). Using the GR(1) and Rabin(1) algorithms with all heuristics again improves running times for 50% of the specifications by at least 4% (columns *opt*). Contrary to our expectation the reuse of previous BDDs for incremental game solving slows down running times on almost all specifications (columns *inc*) with a maximum factor of 2.38x. We believe that this increase in running times is due to increased BDD variable reordering times. We use the automatic reorder of CUDD in all our tests, and the overall reordering time is directly affected by

keeping many BDDs of previous runs. As good news again, the combination of the heuristics *sets* and *opt* usually improves running times even further and roughly obtains a speedup of at least 2x for 50% of the specifications (column *all*).

RQ2: Difference between specification sets The combination of all heuristics similarly improves running times for the SYNTECH15 and the AMBA and GenBuf specifications (columns *all*). The heuristics *sets* consistently performs a few percent better on SYNTECH15 than on AMBA and GenBuf. The heuristics *opt* performs better on the first quartile of AMBA and GenBuf specifications. This is consistent with the observed behavior in Tbl. 2.

4.4 Validation of Heuristics' Correctness

Our implementation of the different heuristics might have bugs, so to ensure correctness of the code we performed the following validation. We have computed the complete set of winning states using the original algorithm and compared the result to the winning states computed by the modified algorithms employing each of the three heuristics separately. As expected, only for unrealizable specifications the heuristics for detecting unrealizability early computed less winning states.

To further ensure that the game memory allows for strategy construction (memory is different for fixed-point recycling), we have synthesized strategies from the game memory produced when using our heuristics. We have verified the correctness of the strategies by LTL model checking against the LTL specifications for strict realizability of the original GR(1) and Rabin(1) specifications.

For the DDMin heuristics, we have compared the resulting core of each heuristics to the original one. Since the heuristics are not on the DDMin itself but on the `check`, the core was never different. Furthermore, we executed DDMin again on the core, to validate the local minimum.

Validation was successful on all 118 specifications used in this paper.

4.5 Threats to Validity

We discuss threats to the validity of our results.

Internal. The implementation of the different heuristics might have bugs, so to ensure correctness of the code we performed validations as described in Sect. 4.4.

Another threat is the variation of the running times of the same test. Different runs of the same algorithm may result in slightly different running times, so the ratios we showed in Sect. 4.3 might not be accurate if we run each test only once. We mitigate it by performing 50 runs of each algorithm and reporting medians as described in Sect. 4.1.

External. The results of the different heuristics might not be generalizable due to the limited number of specifications used in our evaluation. We divided our evaluation into two sets: (1) SYNTECH15, which are realistic specifications created by students for different robotic systems, and (2) the AMBA and GenBuf specifications, which were created by researchers and systematically scaled to larger sizes. The total number of the specifications might be insufficient. The set SYNTECH15 consists of 78 specifications (17 unrealizable). The set AMBA and GenBuf consists of 40 specifications (30 unrealizable).

We share some observations on the sets of specifications that might have an influence of generalizability of the results. First, the AMBA and GenBuf specifications used in literature were generated systematically for growing parameters (number of AMBA arbiters and GenBuf requests). Thus the 40 AMBA and GenBuf specifications essentially describe only two systems. Furthermore, the reasons for unrealizability of AMBA and GenBuf were systematically introduced [4] and consist of a single change

each. Second, the running times of checking realizability of the SYNTECH15 specifications are rather low and range from 1.5ms to 1300ms, with median around 30ms. In this set the specifications are biased based on the numbers of revisions committed by students: the Humanoid has 21 specifications (8 unrealizable), the Gyro has 11 specifications (2 unrealizable), and the SelfParkingCar has only 4 specifications in total. Furthermore, none of the specifications were written by engineers, so we cannot evaluate how our results may generalize to large scale real-world specifications.

5 Related Work

Könighofer et al. [14] presented diagnoses for unrealizable GR(1) specifications. They also implemented the heuristics for DDMin mentioned in Sect. 3.2.1. They suggest further heuristics that approximate the set of system winning states. These heuristics are different from the ones we presented as they are riskier: in case they fail the computation reverts to the original GR(1) algorithm. An analysis of the speed-up obtained from their heuristics for DDMin alone was not reported.

Others have focused on strategy construction for GR(1). Strategies are constructed from the memory stored in the `X`, `Y`, and `Z` arrays in Alg. 1 and Alg. 2. Schlaipfer et al. [31] suggest synthesis of separate strategies for each justice guarantee to avoid a blow-up of the BDD representation. Bloem et al. [1] discuss different minimization of synthesized strategies that do not necessarily minimize their BDDs. We consider space and time related heuristics for strategy construction an interesting next step.

It is well-known that the order of BDD variables heavily influences the performance of BDD-based algorithms [11, 34]. The GR(1) implementation of Slugs [8] uses the default dynamic variable reordering of CUDD [32] (as we do). Slugs turns off reordering during strategy construction. Filippidis et al. [9] reported better performance with reordering during strategy construction. We are not aware of any GR(1) specific heuristics for (dynamic) BDD variable ordering.

As a very different and complementary approach to ours, one can consider rewriting the GR(1) specification to speed up realizability checking and synthesis. Filippidis et al. [9] report on obtaining a speedup of factor 100 for synthesizing AMBA by manually changing the AMBA specification of [1] to use less variables and weaker assumptions. We have not focused on these very specific optimizations of single specifications. Our work presents and evaluates specification agnostic heuristics.

Finally, a number of heuristics for BDD-based safety game solvers have been reported as outcome of the SYNTCOMP reactive synthesis competitions [11, 12, 13]. Most of these optimizations are on the level of predecessor computations (operators $\diamondsuit$ in Alg. 1 and $\boxdot$ in Alg. 2), while the heuristics we implemented are on the level of fixed-points and repeated computations. It seems possible to combine these heuristics. Notably, an approach for predicate abstraction for predecessor computation has already been implemented for GR(1) synthesis [30, 33].

6 Conclusion

We presented a list of heuristics to potentially reduce running times for GR(1) synthesis and related algorithms. The list includes early detection of fixed-points and unrealizability, fixed-point recycling, and heuristics for unrealizable core computations. We implemented and evaluated the heuristics and their combination on two sets of benchmarks, first SYNTECH15, a set of 78 specifications created by 3rd year undergraduate computer science students in a project class of one semester, and second on the two systems AMBA and GenBuf available and well-studied in GR(1) literature.

Our evaluation shows that most heuristics have a positive effect on running times for checking realizability of a specification and for unrealizable core calculation. Most importantly, their combination outperforms the individual heuristics and even in the worst-case has no or a very low overhead. In addition, the heuristics similarly improve running times for both sets of specifications whereas the synthetic reasons for unrealizability in AMBA and GenBuf lead to faster computations.

The work is part of a larger project on bridging the gap between the theory and algorithms of reactive synthesis on the one hand and software engineering practice on the other. As part of this project we are building engineer-friendly tools for reactive synthesis, see, e.g., [18, 19, 20, 21].

Acknowledgments This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

References

[1] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2012): *Synthesis of Reactive(1) Designs*. *J. Comput. Syst. Sci.* 78(3), pp. 911–938. Available at `http://dx.doi.org/10.1016/j.jcss.2011.08.007`.

[2] Anca Browne, Edmund M. Clarke, Somesh Jha, David E. Long & Wilfredo R. Marrero (1997): *An Improved Algorithm for the Evaluation of Fixpoint Expressions*. *Theor. Comput. Sci.* 178(1-2), pp. 237–255, DOI: `10.1016/S0304-3975(96)00228-9`. Available at `http://dx.doi.org/10.1016/S0304-3975(96)00228-9`.

[3] Pavol Cerný, Viktor Kuncak & Parthasarathy Madhusudan, editors (2016): *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015*. *EPTCS* 202, DOI: `10.4204/EPTCS.202`. Available at `https://doi.org/10.4204/EPTCS.202`.

[4] Alessandro Cimatti, Marco Roveri, Viktor Schuppan & Andrei Tchaltsev (2008): *Diagnostic Information for Realizability*. In: *VMCAI*, *LNCS* 4905, Springer, pp. 52–67, DOI: `10.1007/978-3-540-78163-9_9`. Available at `http://dx.doi.org/10.1007/978-3-540-78163-9_9`.

[5] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman & Sebastián Uchitel (2013): *Synthesizing nonanomalous event-based controllers for liveness goals*. *ACM Trans. Softw. Eng. Methodol.* 22(1), p. 9. Available at `http://doi.acm.org/10.1145/2430536.2430543`.

[6] Matthew B. Dwyer, George S. Avrunin & James C. Corbett (1999): *Patterns in Property Specifications for Finite-State Verification*. In: *ICSE*, ACM, pp. 411–420. Available at `http://doi.acm.org/10.1145/302405.302672`.

[7] Rüdiger Ehlers (2011): *Generalized Rabin(1) Synthesis with Applications to Robust System Synthesis*. In: *NASA Formal Methods*, *LNCS* 6617, Springer, pp. 101–115. Available at `http://dx.doi.org/10.1007/978-3-642-20398-5_9`.

[8] Rüdiger Ehlers & Vasumathi Raman (2016): *Slugs: Extensible GR(1) Synthesis*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, *Lecture Notes in Computer Science* 9780, Springer, pp. 333–339, DOI: `10.1007/978-3-319-41540-6_18`. Available at `http://dx.doi.org/10.1007/978-3-319-41540-6_18`.

[9] Ioannis Filippidis, Richard M. Murray & Gerard J. Holzmann (2015): *A multi-paradigm language for reactive synthesis*. In Cerný et al. [3], pp. 73–97, DOI: `10.4204/EPTCS.202.6`. Available at `http://dx.doi.org/10.4204/EPTCS.202.6`.

[10] Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors (2002): *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Lecture Notes in Computer Science 2500, Springer.

[11] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2015): *The First Reactive Synthesis Competition (SYNTCOMP 2014)*. CoRR abs/1506.08726. Available at `http://arxiv.org/abs/1506.08726`.

[12] Swen Jacobs, Roderick Bloem, Romain Brenguier, Ayrat Khalimov, Felix Klein, Robert Könighofer, Jens Kreber, Alexander Legg, Nina Narodytska, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2016): *The 3rd Reactive Synthesis Competition (SYNTCOMP 2016): Benchmarks, Participants & Results*. In Piskac & Dimitrova [26], pp. 149–177, DOI: `10.4204/EPTCS.229.12`. Available at `http://dx.doi.org/10.4204/EPTCS.229.12`.

[13] Swen Jacobs, Roderick Bloem, Romain Brenguier, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2015): *The Second Reactive Synthesis Competition (SYNTCOMP 2015)*. In Cerný et al. [3], pp. 27–57, DOI: `10.4204/EPTCS.202.4`. Available at `https://doi.org/10.4204/EPTCS.202.4`.

[14] Robert Könighofer, Georg Hofferek & Roderick Bloem (2013): *Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies*. STTT 15(5-6), pp. 563–583, DOI: `10.1007/s10009-011-0221-y`. Available at `http://dx.doi.org/10.1007/s10009-011-0221-y`.

[15] Dexter Kozen (1983): *Results on the Propositional mu-Calculus*. Theor. Comput. Sci. 27, pp. 333–354, DOI: `10.1016/0304-3975(82)90125-6`. Available at `http://dx.doi.org/10.1016/0304-3975(82)90125-6`.

[16] Hadas Kress-Gazit, Georgios E. Fainekos & George J. Pappas (2009): *Temporal-Logic-Based Reactive Mission and Motion Planning*. IEEE Trans. Robotics 25(6), pp. 1370–1381. Available at `http://dx.doi.org/10.1109/TRO.2009.2030225`.

[17] Gary T. Leavens, Shigeru Chiba & Éric Tanter, editors (2013): *Transactions on Aspect-Oriented Software Development X*. Lecture Notes in Computer Science 7800, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-36964-3`.

[18] Shahar Maoz, Or Pistiner & Jan Oliver Ringert (2016): *Symbolic BDD and ADD Algorithms for Energy Games*. In Piskac & Dimitrova [26], pp. 35–54, DOI: `10.4204/EPTCS.229.5`. Available at `http://dx.doi.org/10.4204/EPTCS.229.5`.

[19] Shahar Maoz & Jan Oliver Ringert (2015): *GR(1) synthesis for LTL specification patterns*. In Elisabetta Di Nitto, Mark Harman & Patrick Heymans, editors: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, ACM, pp. 96–106, DOI: `10.1145/2786805.2786824`. Available at `http://doi.acm.org/10.1145/2786805.2786824`.

[20] Shahar Maoz & Jan Oliver Ringert (2015): *Synthesizing a Lego Forklift Controller in GR(1): A Case Study*. In: Proc. 4th Workshop on Synthesis, SYNT 2015 colocated with CAV 2015, EPTCS 202, pp. 58–72, DOI: `10.4204/EPTCS.202.5`.

[21] Shahar Maoz & Jan Oliver Ringert (2016): *On well-separation of GR(1) specifications*. In Thomas Zimmermann, Jane Cleland-Huang & Zhendong Su, editors: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, ACM, pp. 362–372, DOI: `10.1145/2950290.2950300`. Available at `http://doi.acm.org/10.1145/2950290.2950300`.

[22] Shahar Maoz & Yaniv Sa'ar (2011): *AspectLTL: an aspect language for LTL specifications*. In Paulo Borba & Shigeru Chiba, editors: AOSD, ACM, pp. 19–30. Available at `http://doi.acm.org/10.1145/1960275.1960280`.

[23] Shahar Maoz & Yaniv Sa'ar (2012): *Assume-Guarantee Scenarios: Semantics and Synthesis*. In: *MODELS*, *LNCS* 7590, Springer, pp. 335–351. Available at http://dx.doi.org/10.1007/978-3-642-33666-9_22.

[24] Shahar Maoz & Yaniv Sa'ar (2013): *Two-Way Traceability and Conflict Debugging for AspectLTL Programs*. In *T. Aspect-Oriented Software Development* [17], pp. 39–72. Available at http://dx.doi.org/10.1007/978-3-642-36964-3_2.

[25] Shahar Maoz & Yaniv Sa'ar (2013): *Two-Way Traceability and Conflict Debugging for AspectLTL Programs*. In *T. Aspect-Oriented Software Development* [17], pp. 39–72, DOI: 10.1007/978-3-642-36964-3_2. Available at http://dx.doi.org/10.1007/978-3-642-36964-3_2.

[26] Ruzica Piskac & Rayna Dimitrova, editors (2016): *Proceedings Fifth Workshop on Synthesis, SYNT at CAV 2016, Toronto, Canada, July 17-18, 2016*. *EPTCS* 229, DOI: 10.4204/EPTCS.229. Available at http://dx.doi.org/10.4204/EPTCS.229.

[27] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) Designs*. In: *VMCAI*, *LNCS* 3855, Springer, pp. 364–380. Available at http://dx.doi.org/10.1007/11609773_24.

[28] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, ACM Press, pp. 179–190, DOI: 10.1145/75277.75293.

[29] Amir Pnueli, Yaniv Sa'ar & Lenore D. Zuck (2010): *JTLV: A Framework for Developing Verification Algorithms*. In: *CAV*, *LNCS* 6174, Springer, pp. 171–174, DOI: 10.1007/978-3-642-14295-6_18.

[30] Leonid Ryzhyk & Adam Walker (2016): *Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned*. In Piskac & Dimitrova [26], pp. 84–99, DOI: 10.4204/EPTCS.229.8. Available at http://dx.doi.org/10.4204/EPTCS.229.8.

[31] Matthias Schlaipfer, Georg Hofferek & Roderick Bloem (2011): *Generalized Reactivity(1) Synthesis without a Monolithic Strategy*. In Kerstin Eder, João Lourenço & Onn Shehory, editors: *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, *Lecture Notes in Computer Science* 7261, Springer, pp. 20–34, DOI: 10.1007/978-3-642-34188-5_6. Available at http://dx.doi.org/10.1007/978-3-642-34188-5_6.

[32] Fabio Somenzi: *CUDD: BDD package, University of Colorado, Boulder.* http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf.

[33] Adam Walker & Leonid Ryzhyk (2014): *Predicate abstraction for reactive synthesis*. In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, IEEE, pp. 219–226, DOI: 10.1109/FMCAD.2014.6987617. Available at http://dx.doi.org/10.1109/FMCAD.2014.6987617.

[34] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan & Fabio Somenzi (1998): *A Performance Study of BDD-Based Model Checking*. In Ganesh Gopalakrishnan & Phillip J. Windley, editors: *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings*, *Lecture Notes in Computer Science* 1522, Springer, pp. 255–289, DOI: 10.1007/3-540-49519-3_18. Available at http://dx.doi.org/10.1007/3-540-49519-3_18.

[35] Andreas Zeller (1999): *Yesterday, My Program Worked. Today, It Does Not. Why?* In: *ESEC/FSE*, *LNCS* 1687, Springer, pp. 253–267, DOI: 10.1007/3-540-48166-4_16. Available at http://dx.doi.org/10.1007/3-540-48166-4_16.

[36] Andreas Zeller & Ralf Hildebrandt (2002): *Simplifying and Isolating Failure-Inducing Input*. *IEEE Trans. Software Eng.* 28(2), pp. 183–200, DOI: 10.1109/32.988498. Available at http://dx.doi.org/10.1109/32.988498.

[37] *SYNTECH GR(1) Performance Website*. http://smlab.cs.tau.ac.il/syntech/performance/.