A Symbolic Justice Violations Transition System for Unrealizable GR(1) Specifications

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

ABSTRACT

One of the main challenges of reactive synthesis, an automated procedure to obtain a correct-by-construction reactive system, is to deal with unrealizable specifications. Existing approaches to deal with unrealizability, in the context of GR(1), an expressive assume-guarantee fragment of LTL that enables efficient synthesis, include the generation of concrete counter-strategies and the computation of an unrealizable core. Although correct, such approaches produce large and complicated counter-strategies, often containing thousands of states. This hinders their use by engineers.

In this work we present the Justice Violations Transition System (JVTS), a novel symbolic representation of counter-strategies for GR(1). The JVTS is much smaller and simpler than its corresponding concrete counter-strategy. Moreover, it is annotated with invariants that explain how the counter-strategy forces the system to violate the specification. We compute the JVTS symbolically, and thus more efficiently, without the expensive enumeration of concrete states. Finally, we provide the JVTS with an on-demand interactive concrete and symbolic play.

We implemented our work, validated its correctness, and evaluated it on 14 unrealizable specifications of autonomous Lego robots as well as on benchmarks from the literature. The evaluation shows not only that the JVTS is in most cases much smaller than the corresponding concrete counter-strategy, but also that its computation is faster.

CCS CONCEPTS

• **Software and its engineering** → *Formal methods*; *Software verification*;

KEYWORDS

reactive synthesis, GR(1), unrealizability

ACM Reference format:
Aviv Kuvent, Shahar Maoz, Jan Oliver Ringert. 2017. A Symbolic Justice Violations Transition System for Unrealizable GR(1) Specifications. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106240

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106240

1 INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [20]. Rather than manually constructing an implementation and using model checking to verify it against a specification, synthesis offers an approach where a correct implementation of the system is automatically obtained for a given specification, if such an implementation exists. In the case of reactive synthesis, an implementation is typically given as a controller, i.e., an automaton that accepts input from the environment (e.g., from sensors) and produces the system's output (e.g., commands for actuators) to always satisfy the specification. If such a controller exists, the specification is considered realizable. Otherwise, the specification is unrealizable – there exists an environment that can force the system to violate some of its guarantees.

GR(1) is a fragment of LTL, which has an efficient symbolic synthesis algorithm [3, 19] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [7, 13]. GR(1) specifications include assumptions and guarantees about what needs to hold on initial states, on all states (safety), and infinitely often on every run (justice). GR(1) synthesis has been used and extended in different contexts and for different application domains, including robotics [11, 12], scenario-based specifications [16], aspect languages [15], event-based behavior models [6], hybrid systems [9], and device drivers [23], to name a few.

Previous work has shown how the debugging of an unrealizable specification, in the context of GR(1), can be done via the extraction of a counter-strategy (CS), which the engineer may explore in order to analyze the source of unrealizability [10, 18]. First a Rabin game is played over the specification, and then intermediate values saved during the game are used to extract the concrete CS. The extracted concrete CS can be viewed as a labeled transition system (LTS) that represents a deterministic choice for the environment for every choice by the system. The LTS can contain cycles in which all environment assumptions are satisfied while at least one system justice guarantee is violated, states which can force the system to these cycles, and states from which every system choice violates a safety guarantee. Although correct, such concrete CS LTS is often large and complex, making it expensive to compute and difficult to explore effectively.

In this work we introduce the Justice Violations Transition System (JVTS), a new, abstract, symbolic representation of a CS, with three key properties. First, unlike a concrete CS, the JVTS is acyclic and is typically small. Thus, in comparison to the concrete CS, it is much simpler and easier to explore. Second, although it is abstract, it is complete: every infinite and finite play on the concrete CS has a corresponding play on the JVTS. Third, the JVTS states are annotated with invariants that relate them to the specification and

```
env boolean dockRequest;
sys boolean ready;
sys boolean docking;

// don't dock before ready
asm G dockRequest -> ONCE(ready);

gar Ready:
  GF ready;

// respond to dock requests
gar DockingResponse:
  pRespondsToS(dockRequest, docking);

gar G docking -> dockRequest;

gar G docking -> !next(docking);
```

Listing 1: Excerpt of the docking mechanism specification

explain exactly how the CS can force the system to violate the specification. The JVTS is comprised of cycle-states, which represent sets of concrete states that the system can visit infinitely often, and attractor-states, which represent sets of concrete states that the system can visit at most once. We formally define the JVTS, discuss its properties, and present a symbolic algorithm to compute it in Sect. 4.

In addition to statically computing and presenting the JVTS, we introduce an interactive approach to dynamically concretize parts of it and execute it step-by-step while alternating between the concrete and the symbolic representations. We describe the interactive, simultaneous play of concrete and symbolic CS representations in Sect. 5. We further minimize the JVTS by merging of attractor nodes described in Sect. 6.

We have implemented all the above ideas in a GR(1) synthesis framework, and integrated them into an Eclipse-based environment. We present a preliminary evaluation of our work using 14 specifications of autonomous Lego robots (created by students in a project class that we have taught) as well as using benchmarks from the literature. The evaluation provides evidence that in many specifications, the size of the JVTS is much smaller than that of the concrete CS, and that its computation is faster. We present the evaluation in Sect. 7.

Previous works on debugging unrealizable specifications for reactive synthesis (e.g., [1, 4, 10, 18]) have considered the notion of unrealizable core, and the idea of semi-automatic discovery of possible assumptions to repair an unrealizable specification. To the best of our knowledge, all have used concrete CSs and none has considered a symbolic representation such as the JVTS. We discuss related work in Sect. 8.

2 RUNNING EXAMPLE

We start off with a running example of an unrealizable specification of the docking mechanism of a space station. The specification shown in Lst. 1 is deliberately small, to fit and be simple enough to explain in the paper format. The specifications used in our evaluation (see Sect. 7) are larger and more complex.

2.1 Example Specification

The docking mechanism described by the specification in Lst. 1 receives as input a docking request (represented by the Boolean environment variable `dockRequest`) and outputs whether the space station is ready to receive docking requests (the Boolean system

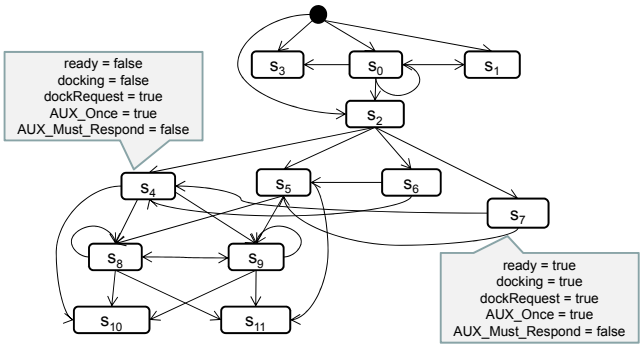

Figure 1: Concrete CS LTS for the docking mechanism specification, as computed by existing tools such as [4, 10, 18].

variable `ready`) and whether the requesting spacecraft should proceed with docking (the Boolean system variable `docking`). The specification contains an assumption that a docking request is only sent if the mechanism was in a `ready` state at some point previously, as expressed in the environment assumption `G dockRequest -> ONCE(ready)`. All systems satisfying the specification will eventually be ready to accept docking requests, expressed by justice guarantee `Ready`: `GF ready`. The guarantee `DockingResponse` ensures that every docking request will eventually receive a response. This is defined using the `pRespondsToS(dockRequest, docking)` pattern, which is translated to `G (dockRequest -> F docking)`[1]. The last two guarantees require that there is a docking request when the mechanism allows docking `G docking -> dockRequest`, and that there should not be two consecutive docking responses `G docking -> !next(docking)`.

This specification is unrealizable. To try to understand the problem using existing tools [4, 10, 18], the engineer can generate a concrete CS, as shown in Fig. 1 (the complete result will list in each state the assignments to all of the variables; to avoid clutter, we show assignments only for states s_4 and s_7).

While the concrete CS in Fig. 1 does not contain many states, one can see that even for such a simple and small specification, the concrete CS as computed by existing tools is already relatively complicated. Moreover, computing an unrealizable core (as suggested by some tools [4, 10]) will not help here, because the core for this example specification contains all guarantees and system variables.

2.2 The Justice Violations Transition System

We present an alternative, the Justice Violations Transition System (JVTS), a symbolic representation of a CS, which is much smaller, simpler (acyclic), and informative. Fig. 2 shows the JVTS we compute for the same example specification.

From the JVTS, the engineer sees that the system can initially choose whether to set `ready` for docking requests or not. If it never sets the variable `ready`, it remains in a cycle-state (q_2) where it will violate the system justice of `Ready`. If it sets `ready` at any point, it moves to an attractor state (q_1) from which the environment can force the system to a second cycle-state (q_3), where the system justice of `DockingResponse` is violated. In addition, in every state the system can choose to set `docking` when the environment sets

[1] This LTL formula is not in GR(1). We use here its equivalent GR(1) translation, see [13].

`!dockRequest`, which will get it to attractor state q_4, representing the violation of the safety `G docking -> dockRequest`.

To better understand the flow when the system chooses to set `ready`, the engineer can begin by selecting the cycle-state q_3 and viewing the invariants over the concrete states it contains, as shown in Fig. 2. `AUX_Must_Respond` is an auxiliary variable indicating that there was previously a docking request to which the system must respond. From these invariants it is clear to the engineer that indeed the system violates the justice guarantee of `DockingResponse`, since previously there was a state where `dockRequest` is set, but the environment does not provide this value again (invariant `!dockRequest`), and the system cannot therefore set `docking` without violating the safety guarantee `G docking -> dockRequest`.

2.3 Concretizing a JVTS State

To better observe how the environment can force the system to q_3 (once the system sets `ready`), the engineer can concretize only the attractor state q_1, resulting in the concrete LTS shown in Fig. 3.

The engineer can choose each of the states in the concretized attractor to view the variable assignments, as she would do in the concrete CS. In Fig. 3, the states are named using the equivalent states in the concrete LTS of Fig 1. The initial state s_2 is the state where `ready & !dockRequest & !docking`. This state has 4 successor states, which depend on the system choice. In all of them, `dockRequest` holds. However, the system can choose whether to respond immediately to this request, resulting in states s_6 or s_7, where `docking` is set, or not to respond immediately, resulting in states s_4 or s_5, where `!docking`. Even if the system responds, the environment sends `dockRequest` immediately and the system cannot respond to it immediately without violating the safety guarantee `G docking -> !next(docking)`. Therefore, in states s_4 and s_5 it also holds that `dockRequest`. Finally, in the next step the environment

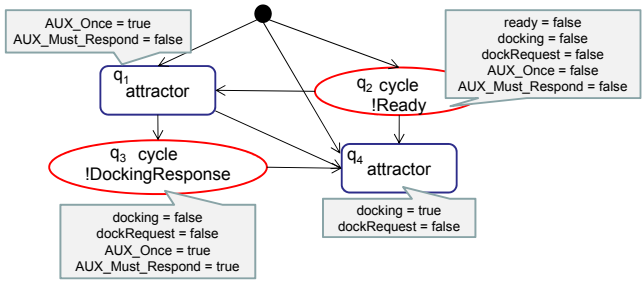

Figure 2: JVTS for the docking mechanism specification. States are labeled with invariants.

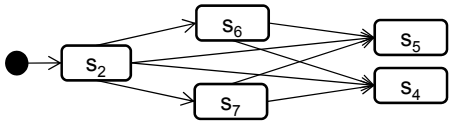

Figure 3: The concretization of attractor q_1 in the JVTS of the space station docking mechanism shown in Fig. 2. The names of concrete states are the ones appearing in Fig. 1.

will not send a docking request, and hence trap the system in the previously described cycle-state q_3.

The above example demonstrates how the JVTS provides the engineer with a high-level view of the CS, and then focus on specific areas of interest for more in-depth examination. Concretization of a JVTS state is done only on demand. The engineer can also choose to start an interactive play from a JVTS state (where the engineer plays the role of the system), moving between concrete states contained in it based on the engineer choices of output. We describe state concretization and interactive play in Sect. 5.

3 PRELIMINARIES

3.1 LTL and GR(1)

We repeat some of the standard definitions of linear temporal logic (LTL), e.g., as found in [3], a modal temporal logic with modalities referring to time. 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).

Definition 3.1. The syntax of LTL formulas over AP is $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$ for $p \in AP$.

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 $u, i \models \varphi_2$;
- $u, i \models \mathbf{X}\varphi$ iff $u, i{+}1 \models \varphi$;
- $u, i \models \varphi_1\mathbf{U}\varphi_2$ 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** and **H** (historically):

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

A 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.

GR(1) synthesis [3] 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 [3]:

- $\mathcal{X}$ input variables controlled by the environment;
- $\mathcal{Y}$ output variables controlled by the system;
- θ^e assertion over $\mathcal{X}$ characterizing initial environment states;

- θ^s assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial system states;
- $\rho^e(\mathcal{X} \cup \mathcal{Y}, \mathcal{X})$ transition relation of the environment;
- $\rho^s(\mathcal{X} \cup \mathcal{Y}, \mathcal{X} \cup \mathcal{Y})$ transition relation of the system;
- $J^e_{i \in 1..n}$ justice goals of the environment;
- $J^s_{j \in 1..m}$ justice goals of the system.

GR(1) synthesis has the following notion of (strict) realizability:

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

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 controller synthesis for φ^{sr} have been presented in [3, 19]. The algorithm of Piterman et al. [19] computes winning states for the system, i.e., states from which the system can ensure satisfaction of φ^{sr}.

3.2 Unrealizability and Rabin Game

A specification φ^{sr} is unrealizable if there is a CS in which the environment can force the system to violate one of its guarantees while satisfying all the environment assumptions. In such a CS, there is an initial environment choice for which the initial states are not winning for the system. A CS can be represented as a labeled transition system (LTS).

Definition 3.2 (CS LTS). Given an unrealizable specification φ^{sr}, a CS is an LTS $\langle Q, T, I, L \rangle$ where:

- Q is a set of states;
- L is a labeling function $L : Q \rightarrow 2^{\mathcal{X} \cup \mathcal{Y}}$;
- $T \subseteq Q \times Q$ is a transition relation where $\forall q \in Q$:
  - q is a deadlock for the system $\nexists q' : T(q, q')$ iff $\exists x \in \mathcal{X} : \rho^e(q, x) \wedge \forall y \in \mathcal{Y} : \neg\rho^s(q, \langle x, y \rangle)$
  - or the transitions from q are environment deterministic and system complete: $\exists_1\, x \in \mathcal{X} : \rho^e(q, x) \wedge \forall y \in \mathcal{Y} : \rho^s(q, \langle x, y \rangle) \rightarrow \exists q' \in Q : L(q') = \langle x, y \rangle \wedge T(q, q')$
- I is a set of initial states such that:
  - the environment initially deadlocks the system $I = \emptyset$ iff $\exists x \in \mathcal{X} : \theta^e(x) \wedge \forall y \in \mathcal{Y} : \neg\theta^s(x, y)$
  - or the initial states are environment deterministic and system complete: $\exists_1 x \in \mathcal{X} : \theta^e(x) \wedge \forall y \in \mathcal{Y} : \theta^s(x, y) \rightarrow \exists q \in I : L(q) = \langle x, y \rangle$

and every infinite path $\pi = q_1 q_2 \ldots$ of the LTS violates φ^{sr}, i.e., $L(q_1)L(q_2)\ldots \not\models \varphi^{sr}$ (unrealizability).

For a CS we say that a state q is *on a cycle* iff the system has a strategy to visit the state infinitely many times.

Definition 3.3 (state on a cycle). Given a CS $TS = \langle Q, T, I, L \rangle$, a state $q \in Q$ is on a cycle in TS if there exists an infinite path $\pi = q_1 q_2 \ldots$ of TS s.t. q repeats infinite times in π.

It is obvious that for every such cycle there exists at least one justice guarantee J^s_i that is not satisfied by any state on the cycle.

Konighofer et al. [10] and Maoz and Sa'ar [18] show how to derive a CS for an unrealizable specification from the intermediate results of a Rabin game. The game computes the environment's winning states, displayed here using μ-calculus notation:

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

where J^e_i is environment justice i, J^s_j is system justice j, and

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

The Rabin game algorithm computes CSs based on cycles violating at least one justice guarantee J^s_i while satisfying all justice assumptions J^e_j. 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.

Note that the above still allows CSs that are "larger" than the ones we compute. Importantly, all the ones we compute satisfy Def. 3.2. Note that a CS can have memory and $|Q| \in O(n|2^{\mathcal{X} \cup \mathcal{Y}}|)$.

During the computation of the Rabin game, the following intermediate results are collected:

- Z - array of sets of concrete states. Cell Z[i] contains concrete states which either violate system justice guarantee $i \pmod m$ or from which the environment can force the system to a cell Z[j] with $j < i$.
- X - A three-dimensional array of sets of concrete states. For indices i from 0 to $|\mathsf{Z}| - 1$ (cell Z[i]), j from 0 to $n - 1$ (for justice assumption J^e_j), and k (maximal number of steps required to satisfy current J^e_j, each step consisting of a concrete state), the cell X[i][j][k] contains a set of concrete states which (1) for $k = 0$ satisfy environment justice assumption J^e_j and have a successor in X[i][j'][k'] for $j' = (j + 1 \mod n)$ and some k' or (2) are a step towards satisfying J^e_j and have a successor in X[i][j][k'] with $k' < k$.

Based on the Rabin game we define the Z-Rank of a state:

Definition 3.4 (Z-Rank). The Z rank of a state s contained in the intermediate results of the Rabin game is:
$ZRank(s) = min\{i \mid s \in \mathsf{Z}[i]\}$.

By construction of the Rabin game, all states on a cycle have the same Z-Rank i and avoid satisfaction of at least the justice guarantee $J^s_{i \bmod m}$. We denote all states of Z-Rank i by $ZRankS(i) \subseteq \mathsf{Z}[i]$. For a set of states S, all of Z-Rank i, we define $ZRank(S) = i$.

4 THE JUSTICE VIOLATIONS TRANSITION SYSTEM (JVTS)

We are now ready to present the main contribution of our work, namely the Justice Violations Transition System (JVTS). We define the JVTS in Sect. 4.1 and describe the symbolic algorithm to compute it in Sect. 4.2.

4.1 Defining the JVTS

A Justice Violations Transition System (JVTS) is an acyclic LTS consisting of two types of states, cycle-states and attractor-states. Each state in the JVTS represents a set of states in some CS, and

each transition in the JVTS represents a set of transitions in the CS. We call this CS a CS of the JVTS.

Definition 4.1 (Justice Violations TS (JVTS)). Given an unrealizable GR(1) specification, a JVTS is an acyclic LTS $TS^j = \langle Q^j, T^j, I^j, L^j \rangle$, s.t. there exists a concrete CS $TS^c = \langle Q^c, T^c, I^c, L^c \rangle$, extracted from the intermediate values of a Rabin game (note that every $q^c \in Q^c$ has a *ZRank* per Def. 3.4), where:

- Q^j is a partition of Q^c where for each $q^j \in Q^j$ either
  - q^j is a cycle-state, i.e., $\forall q^c \in q^j : q^c$ is on a cycle in TS^c (see Def. 3.3) or $\exists C_1, C_2 \subseteq Q^c$ representing cycles in TS^c, s.t. $ZRank(C_1) = ZRank(C_2)$ and q^c is on a path from C_1 to C_2 in TS^c (i.e. in-between cycles with same Z-Rank), or
  - q^j is an attractor-state, i.e., $\forall q^c \in q^j : q^c$ is on a path leading to a deadlock or cycle only through attractor states or $\exists C_1, C_2 \subseteq Q^c$ representing cycles in TS^c, s.t. $ZRank(C_1) > ZRank(C_2)$ and q^c is on a path from C_1 to C_2 in TS^c only through attractor states (i.e. in-between cycles with decreasing Z-Ranks);
- L^j is a labeling function: $\forall q^j \in Q^j : L^j(q^j) = \{L^c(q^c) \mid q^c \in q^j\}$;
- $T^j \subseteq Q^j \times Q^j$ is a transition relation where $T^j(q_1^j, q_2^j)$ iff
$$q_1^j \neq q_2^j \wedge \exists q_1^c \in q_1^j, q_2^c \in q_2^j : T^c(q_1^c, q_2^c); \text{ and}$$
- I^j is a set of initial states: $q^j \in I^j$ iff $(q^j \cap I^c) \neq \emptyset$.

Example 4.2. In our running example (Sect. 2) Fig. 2 shows a JVTS. A CS of the JVTS is shown in Fig. 1. The label of attractor state q_1 is: $L^j(q_1) = \{L^c(s_2), L^c(s_4), L^c(s_5), L^c(s_6), L^c(s_7)\}$, where s_2, s_4, s_5, s_6, and s_7 are the concrete states contained in q_1, as can be seen in Fig. 3. From the definition of L^c in Def. 3.2, the label $L^j(q_1)$ is the set of assignments to input and output variables in the concrete states contained in q_1.

Note that Def. 4.1 allows for multiple JVTSs for the same CS (we look at a minimal JVTS in Sect. 6).

The following states the completeness of the JVTS in terms of paths in its concrete CS computed by the Rabin game algorithm.

Theorem 4.3 (JVTS Completeness). Given a concrete CS LTS TS^c of a JVTS TS^j, the following holds:

- For every infinite path $\pi^c = q_1^c q_2^c ..$ in TS^c, exists a single corresponding finite path $\pi^j = q_1^j .. q_k^j$ in TS^j s.t. q_k^j is a cycle-state for justice J_i^s and $\pi^c \not\models \mathbf{GF}\ J_i^s$;
- For every finite path $\pi^c = q_1^c .. q_r^c$ in TS^c, exists a single corresponding finite path $\pi^j = q_1^j .. q_k^j$ $(k \leq r)$ in TS^j s.t. $L^c(q_r^c) \in L^j(q_k^j)$; and
- For every prefix $\pi_a^c = q_1^c .. q_r^c$ of a finite or infinite path $\pi_b^c = q_1^c .. q_r^c ..$ in TS^c, exist unique corresponding finite paths π_a^j and π_b^j in TS^j s.t. π_a^j is a prefix of π_b^j.

Proof. (sketch) By definition of the JVTS (as a partition of the CS) and by the correctness of the existing algorithm of the Rabin game (for cycles avoiding justice guarantees). □

Example 4.4. In our example (Sect. 2), the infinite path $\pi^c = s_2, s_4, s_8, s_9, s_8, s_9, \ldots$ in the concrete CS has a single finite corresponding path ending in a cycle-state in the JVTS $\pi^j = q_1, q_3$. The finite path $\pi^c = s_2, s_4, s_8, s_9, s_8, s_{10}$ in the concrete CS has a corresponding finite path in the JVTS $\pi^j = q_1, q_3, q_4$.

Algorithm 1 Computing the JVTS

```
1:  if deadendIni(ini) then
2:      attrFromState ← getDeadendIni(ini)
3:      jvts.add(attrFromState)
4:      return ⟨jvts, envChoices⟩
5:  end if
6:  RTS ← compRankingTS
7:  for state in RTS (reverse z-rank order) do
8:      ⟨attrFromCands, envChoices⟩ ← compAttrFromCands
9:      X ← filterX(X[ZRank(state)])
10:     ⟨paths, envChoices⟩ ← compPaths
11:     cycleState ← compCycleState
12:     attrToState ← compAttrToState
13:     attrFromState ← compAttrFromState
14:     jvts.add(cycleState, attrToState, attrFromState)
15: end for
16: return ⟨jvts, envChoices⟩
```

The following theorem states lack of redundant states and transitions in the JVTS.

Theorem 4.5 (JVTS State and Transition Soundness). Given a concrete CS LTS TS^c of a JVTS TS^j, every state $q^j \in Q^j$ and every transition $t^j \in T^j$ appears on at least one path in TS^j corresponding to a concrete path in TS^c.

Example 4.6. In our example, the states q_3 and q_4 and the transition between them in the JVTS appear in the path $\pi^j = q_1, q_3, q_4$, which corresponds to the concrete CS path $\pi^c = s_2, s_4, s_8, s_9, s_8, s_{10}$.

4.2 Computing the JVTS

A naive method to compute the JVTS could have been to first extract a CS using the methods described in [10] and then compute the JVTS from the CS. However, this method would require the enumeration of all states in the CS and is therefore inefficient. Our algorithm, presented below, is purely symbolic, and thus avoids the costly enumeration. The algorithm uses an efficient symbolic representation and manipulation of sets of concrete states. We consider this to be an important part of our contribution.

Alg. 1 presents our symbolic algorithm for computing the JVTS. Its input is the set of initial states `ini` and the intermediate values collected during the Rabin game (Sect. 3.2) - the Z array (`Z`) and the three-dimensional X array (`X`). The output of the algorithm is a JVTS and a set `envChoices`, which contains all possible transitions between states, deterministic for the environment inputs.

We start by checking if the set of initial states `ini` contains a dead-end state for the system (i.e., the environment can force the system to violate a safety guarantee). If such a state exists, it constitutes the CS. It is added to the JVTS (Alg. 1, line 3) and the algorithm ends. Else, we compute the Ranking TS (Alg. 1 line 6), a TS representation of `Z` cells reachable from the initial set of states (Sect. 4.2.1). The algorithm then traverses the Ranking TS states in reverse Z-Rank order. Each Ranking TS state is split into at most 3 JVTS states, a cycle-state and 2 attractor-states.

- In line 8 of Alg. 1, a set of concrete state-candidates to be in one of the attractor-states - the attractor-from-cycle state - are computed, with the relevant environment choices (Sect. 4.2.2);
- In line 9 of Alg. 1, the attractor-from-cycle state candidates are removed from the array `X`;
- In line 10 of Alg. 1, a set of paths along the cells (with the relevant environment choices) in `X` are computed. Each such path

represents a series of steps ending in a cycle which satisfies all environment assumptions while violating a system guarantee (Sect. 4.2.3);

- In lines 11 and 12 of Alg. 1, the set of concrete states to be contained in the cycle-state and in the attractor-to-cycle state are extracted from the computed paths (Sect. 4.2.4 and Sect. 4.2.5);
- In line 13 of Alg. 1, the set of concrete states to be contained in the attractor-from-cycle state are computed using the previously computed attractor-from-cycle state candidates, the attractor-to-cycle state and cycle state (Sect. 4.2.6).

Finally, the algorithm adds the computed states to the JVTS.

As it follows the general steps of the concrete CS extraction described in [10], and relies on the intermediate values of the Rabin game, the result of the algorithm is a valid JVTS, symbolically representing a concrete CS.

Theorem 4.7. *Algorithm 1 outputs a valid JVTS as defined in Def. 4.1, which is minimal in the number of cycle-states.*

4.2.1 Ranking TS. The Ranking TS is an acyclic TS which groups concrete states according to their Z-Rank. Concrete states of Z-Rank i are part of the Ranking TS only if they are reachable from initial states `ini` through other concrete states of Z-Ranks $j \geq i$. The intermediate Ranking TS is non-deterministic, as opposed to the more refined JVTS we later compute from it.

Definition 4.8 (Ranking TS). Given a set of concrete initial states `ini` and an array `Z` of disjoint sets of concrete states, the Ranking TS $\langle Q, T, I\rangle$ is an acyclic TS where:

- $Q \subseteq 2^{2^{\mathcal{X}\cup\mathcal{Y}}}$ is the set of states: $\forall q \in Q, \exists 1 0 \leq i < |\mathtt{Z}|$: $q \subseteq ZRankS(i)$ s.t. all concrete states in q are reachable from `ini` via states of higher Z-Rank:
  $\forall s \in q, \exists \pi = s_1..s_k : s_1 \in \mathtt{ini} \wedge s_k = s \wedge$
  $\forall 1 < j \leq k : \rho(s_{j-1}, s_j) \wedge ZRank(s_{j-1}) \geq ZRank(s_j)$;
- Transition $T(q_1, q_2)$ exists for $q_1 \neq q_2$ iff $\exists s_1 \in q_1, s_2 \in q_2 : \rho(s_1, s_2)$; and
- $I \subseteq Q$ initial states: $q \in I$ iff $q \cap \mathtt{ini} \neq \emptyset$.

The symbolic removal of duplicate concrete states from `Z` (keeping their Z-Rank copy only), ensures that we advance to the lowest Z-Rank for the correctness of the algorithm, as explained in [10].

4.2.2 computeAttrFromCands. After computation of the Ranking TS, we compute the sets of concrete states which are the candidates to be contained in the attractor-from-cycle state. This computation is done for every state `rankState` of the previously computed Ranking TS $\langle Q, T, I\rangle$. We return the set of candidate states (`attrFromCands`) and environment choices collected between the candidate states (`envChoices`). The computation of `attrFromCands` is the standard attractor computation, done to the set of concrete states contained in all Ranking TS states of lower Z-Rank value: $lower = \{s \mid \exists q \in Q : ZRank(\mathtt{rankState}) > ZRank(q) \wedge s \in q\}$. In addition, we store the environment choices taken at each step of the fixed-point computation of the attractor in `envChoices`. The `envChoices` stored here are used in Sect. 4.2.6 to compute the valid reachable (per environment choices taken) set of concrete states contained in the attractor-from-cycle JVTS state.

4.2.3 compPaths. We now describe the symbolic computation of paths along the cells in $\mathtt{X}[Zr]$ array of arrays, where $Zr = ZRank(\mathtt{rankState})$. Each such path will represent a series of steps ending in a cycle which satisfies all of the environment assumptions while violating a system guarantee. The first step of the computation is done in the filtering of $\mathtt{X}[Zr]$, performed in Alg. 1, line 9. The filtered `X` (`filterX`) is constructed thus: $\forall i, j : s \in \mathtt{filterX}[Zr][i][j]$ iff: (1) $s \in \mathtt{X}[Zr][i][j]$, i.e., s computed by Rabin game, and (2) $\forall 0 \leq k < j : s \notin \mathtt{X}[Zr][i][k]$, i.e., s closest to satisfying J_i^e, and (3) $s \notin \mathtt{attrFromCands}$, i.e., paths through s end in a cycle.

After removal of duplicate states in step 2, we can perform an algorithm similar to the one described for concrete CS extraction in [10] in order to compute the paths. The algorithm in [10] enumerates the concrete states beginning with the initial state. For each concrete state reached, the original algorithm locates to which `Z` cell it belongs (its Z-Rank Zr) and to which $\mathtt{X}[Zr][j]$ cell it belongs. It then defines an environment choice for this concrete state based on the following order of priorities: (1) If there exists an environment choice for which all successors of this concrete state are in a `Z` cell of lower Z-Rank than the cell of the current concrete state, this environment choice is taken; (2) Else, the environment choice taken has successors in a cell closest to index 0 in the current $\mathtt{X}[Zr][j]$ array (or in the next array $\mathtt{X}[Zr][(j+1) \mod n]$ if the current concrete state is already in index 0). The concrete successor states are then added to the concrete CS, and are iterated over in turn (if they were not already visited previously as a result of a concrete cycle).

Our symbolic algorithm follows a similar flow, except that instead of checking for successors of concrete states at each step (and iterating over them), it performs a symbolic step between sets of states, composed of two parts:

- In the first part, a valid environment choice which leads to the target set of states `dst` is selected for the source set of states `src` for which an environment choice was not yet taken:
  $newSuccsTrans(\mathtt{src}, \mathtt{dst}, \mathtt{others}, \mathtt{envChoices}) =$
  $$\{\langle s, x\rangle \mid s \in \mathtt{src}, x \in \mathcal{X} \wedge \rho^e(s, x) \wedge \langle s, x\rangle \notin \mathtt{envChoices} \wedge (\forall y \in \mathcal{Y} : \rho^s(s, \langle x, y\rangle) \rightarrow \langle x, y\rangle \in (\mathtt{dst} \cup \mathtt{others}) \wedge (\exists y \in \mathcal{Y} : \rho^s(s, \langle x, y\rangle) \wedge \langle x, y\rangle \in \mathtt{dst})\}$$
  `others` are additional destination states which can be reached via a different system choice for the taken environment choice. The `src` and `dst` sets of states are subsets of states in some $\mathtt{X}[Zr][k]$ cells determined according to the flow in [10].
- In the second part we take all relevant successors in `dst` according to the environment choices taken for states in `src`:
  $succs(\mathtt{src}, \mathtt{dst}, \mathtt{envChoices}) =$
  $$\{s \in \mathtt{dst} \mid \exists s' \in \mathtt{src}, x \in \mathcal{X} : \langle s', x\rangle \in \mathtt{envChoices} \wedge \exists y \in \mathcal{Y} : \langle x, y\rangle = s\}.$$

The successors which are the result of *succs* are then used as the `src` states in the next step.

Despite the similar flow, there are two key differences between our computation and the one described in [10]. First, due to the removal of states that are in `attrFromCands`, done in step 3, the resulting CS represented by the JVTS will attempt to force to states with the lowest Z-Rank possible in one or more steps. In [10], the choice of moving to a state in a lower Z-Rank is only done by

looking one step ahead. Second, during computation of a path, when performing steps over the `filterX`, we consider sets of states and not a single concrete state.

4.2.4 compCycleState. The concrete states contained in a JVTS cycle-state are computed using the previously calculated paths. There are two types of concrete states inside a cycle-state: (1) states which are part of a concrete cycle in a single path, and (2) states which are between concrete cycles in different paths computed from the same Ranking TS state (see Def. 4.1).

We identify states on cycles by iterating over the computed paths, and for each path computing the set of all concrete states which are a part of the cycle. We denote these states on cycles by S. We identify states in-between cycles B using a least fixed-point computation of successors of S within paths: $\mathtt{B} = \mu\mathtt{C}.\mathtt{S} \cup \mathit{succs}(\mathtt{C}, \mathtt{A}, \mathtt{envChoices})$, where A are all the states in the paths for this Ranking TS state.

4.2.5 compAttractorToState. The concrete states contained in the attractor-to-cycle state are computed using the previously computed cycle-state and the paths. The attractor-to-cycle state contains all concrete states in paths that are not in the calculated cycle-state.

4.2.6 compAttractorFromState. The set of concrete states contained in the attractor-from-cycle state are computed from the `attrFromCands` using the Ranking TS, the previously computed `attrTo` and `cycle` states, the concrete states contained in the previously computed JVTS states P and the `envChoices`. The computation begins by finding states $\mathtt{Start} \subseteq \mathtt{attrFromCands}$ reachable in a single step from P:

$$\begin{aligned}\mathtt{Start} = \{s' \in \mathtt{attrFromCands} \mid \\ \exists s \in (\mathtt{attrTo} \cup \mathtt{cycle} \cup \mathtt{P}), x \in \mathcal{X}, y \in \mathcal{Y} : \\ \langle s, x\rangle \in \mathtt{envChoices} \wedge \rho^e(s, x) \wedge \rho^s(s, \langle x, y\rangle) \wedge s' = \langle x, y\rangle\}.\end{aligned}$$

We then compute `attrFrom` as the least fixed-point of states reachable from `start`:

$$\begin{aligned}\mathtt{attrFrom} = \mu\mathtt{A}.\mathtt{Start} \cup \{s' \in \mathtt{attrFromCands} \mid \\ \exists s \in \mathtt{A}, x \in \mathcal{X}, y \in \mathcal{Y} : \langle s, x\rangle \in \mathtt{envChoices} \wedge \\ \rho^e(s, x) \wedge \rho^s(s, \langle x, y\rangle) \wedge s' = \langle x, y\rangle\}.\end{aligned}$$

4.3 Computing JVTS Annotations

On top of the JVTS computation, for its presentation to the engineer, we annotate its states with the following information:

- State type (attractor-state or cycle-state);
- For each cycle-state – the specific violated justice. From the construction of the Z array in the Rabin game, the Z-Rank identifies a system justice guarantee being violated by the states in the relevant Z-array cell, and therefore by the states in the cycle-state constructed from this Z-array cell; and
- Invariants - we annotate each JVTS state with invariants of the form `<var> = <value>`. Computation of the invariants is performed by iterating over the values of each variable and checking if a restriction to the value constitutes an invariant. In the worst-case, the computation of all invariants requires $O(|\mathcal{X} \cup \mathcal{Y}|)$ many efficient symbolic operations. As a performance improvement, we only check for variables that appear in the symbolic representation as only their values might constitute an invariant.

Note that the annotations present aggregated information. They are not the labels from Def. 4.1.

5 CONCRETIZATION AND INTERACTIVE SYMBOLIC AND CONCRETE PLAY

To further assist the engineer in the exploration of the CS, we implemented an interactive play over the JVTS. The engineer can select a JVTS state (cycle or attractor) and can either (1) concretize the state, or (2) perform interactive play starting from concrete states in the JVTS state.

Concretizing a JVTS State. Concretizing a JVTS state results in generating all concrete states contained in it, allowing the engineer to view the concrete cycle violating at least one justice guarantee, or an attractor path towards the next cycle. Generation of concrete states entails enumerating all concrete states contained in the JVTS state. Therefore, this can be (in the worst-case) as expensive as computation of the concrete CS.

Example 5.1. In our running example, Fig. 3 shows the result of concretizing the attractor state q_1. In the figure we see the 5 concrete states it contains, and the transitions between them.

The computation of concrete states contained in a JVTS state is immediate from the JVTS structure returned by Alg. 1 and the `envChoices` accumulated in memory. Each JVTS state consists of a symbolic representation of the set of concrete states it contains, and we simply extract these concrete states by enumerating all assignments to environment and system variables representing a state in this set. We build the transitions between the states using the `envChoices`.

Interactive Symbolic and Concrete Play. The engineer can perform an interactive play starting from a JVTS state. The play consists of 4 steps: (1) The engineer chooses as a starting point a concrete state contained in a JVTS state; (2) The concrete state and all its possible concrete successors, in the same JVTS state and in other JVTS states, are displayed to the engineer; (3) The engineer chooses a concrete successor; (4) We return to step (2).

Example 5.2. In our running example, the engineer chooses to perform interactive play starting from the attractor (symbolic state q_1) in Fig. 2. She selects assignments to environment and system variables in order to pick a concrete state of her interest within JVTS state q_1. She chooses to give the value of `true` to all variables and is shown only the concrete state s_7 and its immediate successors s_4 and s_5 (all of which are in the attractor q_1, as seen in Fig. 3). She then continues the play from state s_4, which will display its immediate successors s_8, s_9, s_{10}, and s_{11} (as they appear in the concrete graph in Fig. 1). Note that these successors are in a different JVTS state q_3; this is indicated to the engineer during the interactive play.

Using the interactive play, the engineer can traverse the concrete states of the CS, on demand, while seeing the context provided by the JVTS states in which they are contained, e.g., the system justice guarantee which the CS attempts to violate or invariants shared by this concrete state and other concrete states in the JVTS state.

Computation of an interactive play is done in the same way a concretization of a JVTS state is performed, except that in each step, the only concrete states extracted are the ones chosen by the

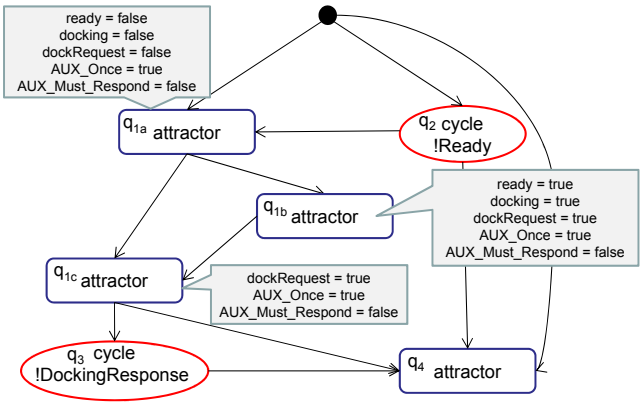

Figure 4: JVTS for docking mechanism specification with unmerged attractors. Attractors q_{1a}, q_{1b}, and q_{1c} can be merged to attractor q_1, as is shown in Fig. 2. The invariants of q_2, q_3, and q_4 are the same as shown in Fig. 2.

engineer and its immediate successors. This allows instantaneous, on-demand computation of concrete states and is thus very efficient.

A set of screenshots demonstrating interactive symbolic and concrete play in our Eclipse-based environment is available from [25].

6 MERGING OF JVTS ATTRACTOR STATES

Since the JVTS computation is based on the intermediate values of the Rabin game, as shown in Sect. 4.2, it may contain sequences of attractor states that were computed from Z cells of different Z-Ranks (Def. 3.4). In our experience, these sequences may be relatively long and do not always provide valuable information to the engineer. Thus, our default implementation merges such sequences of attractor states into one.

Still, in some cases the engineer may be interested in a more refined version of the JVTS, where such attractors are not merged. We therefore provide the engineer with a means to unmerge and merge such sequences of attractor states on demand.

Example 6.1. Using our running example in Sect. 2, running the JVTS algorithm without merging of attractor states results in the JVTS shown in Fig. 4, where state q_{1a}, q_{1b}, and q_{1c} are unmerged. The version of this JVTS shown earlier in Fig. 2 has these attractors merged to a single attractor, q_1.

As can be seen in our example, without merging, the resulting JVTS is not the minimal JVTS which is also acyclic. The algorithm which merges attractors traverses the JVTS and merges neighboring attractors that can be merged without creating a cycle. This merging minimizes the number of attractor states of the JVTS. Thus, the resulting JVTS, after merging, is minimal not only in the number of cycle states (see Sect. 4.2) but also in the number of attractor states.

7 EVALUATION

We have implemented Alg. 1, the interactive play discussed in Sect. 5, and the merging discussed in Sect. 6, in our synthesis environment, based on CUDD [22] as a BDD library, and integrated into Eclipse. Our implementation includes also the computation of concrete CS extraction based on [10, 18]. It is important to note that the concrete CS computed might not be the one represented by the JVTS, i.e., the concrete CS computed by the original algorithm might have a slightly different size than the one represented by the JVTS. Both algorithms work on the same results of the Rabin game but JVTS computation might prefer leaving cycles early as mentioned in Sect. 4.2.3.

We consider the following research questions:

R1 Is the JVTS computation efficient and how does it compare to concrete CS construction?

R2 Is the JVTS smaller than the concrete CS?

Table 1: The 14 unrealizable specifications of Lego robots created by our students in the workshop class and used for the evaluation in this paper. For each specification we report the number of justice assumptions and guarantees (n and m resp.), number of safety assumptions and guarantees (g^e and g^s), number of input and output variables ($|\mathcal{X}|$ and $|\mathcal{Y}|$), and number of auxiliary variables ($|AUX|$) added due to the use of patterns and past LTL formulas.

Name	n	m	$\lvert g^e \rvert$	$\lvert g^s \rvert$	$\lvert \mathcal{X} \rvert$	$\lvert \mathcal{Y} \rvert$	$\lvert AUX \rvert$
Gyro_rev_710	4	2	1	4	3	6	3
Gyro_var2_rev_710	3	2	1	4	3	5	2
Humanoid_rev_458	0	0	0	11	3	10	0
Humanoid_rev_503	0	1	1	16	6	13	2
Humanoid_rev_531	2	1	0	15	1	14	2
Humanoid_rev_741	2	1	3	19	4	19	5
Humanoid_rev_742	0	1	2	24	1	16	2
PCar_rev_769	2	1	2	25	5	13	2
PCar_rev_870	2	1	4	29	5	16	5
PCar_rev_888	1	1	2	19	3	11	2
PCar_un2	1	1	2	19	3	11	2
ColorSort_rev_790	4	2	2	25	11	17	4
ColorSort_rev_791	5	2	2	31	13	22	6
SelfParkingCar_rev_974	3	3	0	42	4	28	5

7.1 Specifications Used

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 unrealizable 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; 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.

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, the students have committed many versions of their specifications to the repository. Most of these were realizable, but some unrealizable. We use here all of the unrealizable specifications from the repository. In total, we have collected 14 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.

Tbl. 1 provides basic information regarding the size of the 14 specifications: number of justice and safety assumptions and guarantees,

number of input and output variables, and number of auxiliary variables. As can be seen, the number of justice guarantees is small, ranging from 0 to 3. The total number of safety guarantees ranges from 5 to 42. The state space (input, output, and auxiliary variables) ranges from 2^{10} to 2^{41}.

In addition to the specifications created by the students, we considered the ARM AMBA AHB Arbiter, which is the most popular GR(1) example in literature, used, e.g., in [1, 3, 4, 10]. We looked at 4 different sizes of AMBA (1 to 4 masters), each in the 3 variants of unrealizability described in [4] (with a justice assumption removed, with a justice guarantee added, and with a safety guarantee added). We have thus run our experiments on 12 AMBA specifications.

All specifications used in our evaluation, the raw results, and the code to reproduce our experiments are available from [25].

7.2 Validation

We have systematically validated the correctness of our implementation by model-checking the symbolic JVTS constructed for the specifications mentioned in this paper and for many more. We first transformed the JVTS into a symbolic controller. The initial states of this symbolic controller are the union of the initial states in the JVTS states. The transitions of this symbolic controller are the transitions accumulated in `envChoices` (see Sect. 4.2). We then used a model-checker to check whether this symbolic controller satisfies the specification φ^{sr} (as described in Sect. 3.1).

In addition, we validated the completeness of JVTS with regard to system choices by ensuring that, given a concrete state and an environment choice represented in the JVTS, every system choice either violates a safety guarantee or leads to a successor which is also represented in the JVTS.

Our validation helped us find a number of bugs in our earlier implementation and to increase our confidence in the correctness of its latest version.

7.3 Results

R1: Computation Time. We run all experiments on an ordinary PC, Intel i7 CPU 3.4GHz, 16GB RAM with Windows 7 64-bit OS, Java 8 64Bit, and CUDD 3 compiled for 64Bit, using only a single core of the CPU. We measured the running time of concrete CS extraction, and of symbolic JVTS computation with and without merging of attractors, for the 14 specifications shown in Tbl. 1, as well as for 12 AMBA AHB specifications from [4]. Times we report are median values of 12 runs per specification measured by Java in milliseconds. Even though the JVTS computation algorithm is deterministic, we performed 12 runs since JVM garbage collection and BDD dynamic-reordering add variance to running times.

As it is well known that BDD-based implementations' performance is sensitive to variable order, it is important to note that in all our experiments we used CUDD's automatic variable reordering for the Rabin game, and no variable reordering for strategy extraction. Our experience shows that this configuration provides the fastest results for both the concrete CS extraction and the JVTS computation, across all the specifications we examined.

Tbl. 2 (left) displays the running time quartiles for concrete CS extraction for the 14 specifications from Tbl. 1 and ratios of JVTS computation times. We see that the JVTS computation is significantly faster for these 14 specifications. In fact, for half of the specifications, the JVTS running time is more than 190 times faster than the concrete CS extraction, and for 25% the running time is more than 1658 times faster.

Table 2: Running time (in ms) quartiles of the concrete CS extraction and ratios of JVTS computation (including annotations) for the 14 specifications from Tbl. 1 and the 12 AMBA specifications. T/O represents running time of over 10 minutes.

	14 Lego Robots		12 AMBA AHB	
Quartile	Concrete	$\frac{\text{Concrete}}{\text{JVTS}}$ ratio	Concrete	$\frac{\text{Concrete}}{\text{JVTS}}$ ratio
MIN	17	1	7	1
Q_1	93	3	11	2
Q_2	8426	190	129	4
Q_3	38794	1658	1292	6
MAX	T/O	∞	26787	23

Table 3: Size quartiles and ratios for the 14 specifications of Tbl. 1 (upper part), and the 12 AMBA specifications (lower part), comparing the concrete CS numbers of states Q^c and transitions T^c and the ratios of JVTS states and transitions with merged attractors (unmerged attractors in parentheses). We use 1 as the size of empty sets of states and transitions to avoid division by 0.

Quartiles	$\lvert Q^c \rvert$	$\lvert T^c \rvert$	$\frac{\lvert Q^c \rvert}{\lvert Q^J \rvert}$ ratio	$\frac{\lvert T^c \rvert}{\lvert T^J \rvert}$ ratio
MIN	8	20	3 (3)	7 (7)
Q_1	167	204	92 (65)	83 (73)
Q_2	1308	1884	878 (468)	1884 (740)
Q_3	3309	608861	1377 (1104)	124270 (101658)
MAX	2124000	66816000	708000 (708000)	22272000 (22272000)
MIN	1	0	1 (1)	1 (1)
Q_1	1	0	1 (1)	1 (1)
Q_2	25	46	10 (8)	20 (15)
Q_3	85	196	21 (14)	34 (24)
MAX	1921	7292	274 (160)	521 (270)

Tbl. 2 (right) displays the running time quartiles and ratios for the 12 AMBA specifications. Here as well we see the that JVTS computation is faster than the concrete CS extraction. For half of the specifications, the running time of the JVTS is faster than the concrete CS by a factor of 4, and for 25% of the specifications, the JVTS computation is faster by a factor of 6.

In all the specifications evaluated, attractors merging added only very small overhead (at most 16 ms) to the JVTS computation time.

To answer R1: Computation time of the JVTS is faster than extraction of a concrete CS, for both the Lego robot (more than 190 times faster for half of the specifications) and the AMBA specifications (more than 4 times faster for half of the specifications).

R2: Size. We have measured the number of states and transitions for the CSs of the 14 specifications listed in Tbl. 1, as well as for the 12 AMBA AHB specifications, for concrete CSs and for JVTSs with/without merging of attractors.

Tbl. 3 (upper part) shows the quartiles of the number of states Q^c and transitions T^c of the concrete CS and the ratios of the JVTS sizes with merged attractors and unmerged attractors, for the 14 specifications listed in Tbl. 1. For all of these specifications, the

number of states and transitions in the JVTS is significantly smaller than the number of states and transitions in the concrete CS. In more than half of the specifications, the number of states in the JVTS is smaller by a factor of at least 878 than in the concrete CS, and in about 25% the factor is more than 1377. For half of the specifications, the number of transitions in the JVTS is smaller by a factor of 1884 than the number of transitions in the concrete, and for 25% of the specifications, the number of transitions is smaller by a factor of 124270. Merging of attractors further reduces the number of states and transitions.

Tbl. 3 (lower part) shows the quartiles of the number of states and transitions of the concrete CS and the ratios of the JVTS sizes (with/without merging of attractors), for the 12 AMBA AHB specifications listed in Sect. 7.1. The results show that in all the AMBA specifications, the size of the JVTS is smaller than or equal to the size of the concrete CS. Half of the JVTSs of these specifications are smaller than the concrete CSs by a factor of 10 or more. Merging of attractors further decreases the size of the JVTS in some cases.

To answer R2: The size of the JVTS, states and transitions, is much smaller than the size of the concrete CS in most the examined specifications.

7.4 Threats to Validity

We briefly discuss threats to the validity of our results.

Internal. The JVTS computation is not trivial and our implementation may have bugs. To mitigate this, we performed a thorough validation (Sect. 7.2) using all specifications available to us.

External. First, we did not perform a user-study, with engineers, to examine how the JVTS achieves its ultimate goal of aiding them in debugging unrealizable specifications. However, the orders of magnitude smaller size of the JVTS and its simplicity (in comparison with the concrete CS, as seen in Sect. 7), as well as the annotations added to the JVTS states (directly referencing elements in the specification), all hint that the JVTS will indeed be easier for engineers to explore and use. Second, we have based most of our evaluation on specifications created by 3rd year CS students with no prior experience in writing LTL specifications. Due to the lack of other real-world unrealizable specification examples, the specifications chosen were all unrealizable specifications available to us.

8 RELATED WORK

GR(1) synthesis was introduced in [19], and has since been used and investigated in many works. To list a few, D'Ippolito et al. [5, 6] used GR(1) to deal with fallible domains and non-anomalous event-based behavior models; Kress-Gazit et al. [11] used GR(1) in robotics; Maoz and Ringert showed GR(1) synthesis for specification patterns [13]. Several tools support GR(1) synthesis, including RATSY [2], TuLiP [24], and Slugs [8]. We give an overview of existing approaches to dealing with unrealizable GR(1) specifications.

Counter-Strategies and Core. Cimatti et al. [4] suggest to use a CS to provide an explanation for unrealizability, and use a core to explain a single specific unrealizability cause. Konighofer et al. [10] compute a core not only for system guarantees but also for output variables. Maoz and Sa'ar [17] present CSs for scenario-based specification. All works use concrete CSs. Our work is the first to suggest a symbolic CS. It can be composed on top of a core calculation.

Interactive Play. Some previous works suggest the use of an interactive play [10, 17, 21], where the engineer explores the CS by playing the role of the system against the winning environment. Again, all these approaches to interactive play rely on a concrete CS. No previous work provides an interactive play which is based on a symbolic representation of the CS, as we have introduced and implemented here.

Repair using Strengthened Assumptions. Finally, Alur et al. [1] proposed a method for semi-automatic strengthening of assumptions. It analyses the concrete CS and suggests candidate assumptions that may solve the cause of unrealizability. The use of our symbolic CS representation in the context of repair is a very interesting future work direction.

9 CONCLUSION

We presented the Justice Violations Transition System (JVTS), a novel symbolic representation of CSs for GR(1). The JVTS is much smaller and simpler than its corresponding concrete CS and is annotated with invariants that explain how the CS forces the system to violate the specification. We compute the JVTS symbolically, and thus efficiently, without expensive enumeration of concrete states. Finally, we provide the JVTS with an on-demand interactive concrete and symbolic play.

We implemented our work, validated its correctness, and evaluated it on 14 unrealizable specifications of autonomous Lego robots and on benchmarks from the literature. The evaluation shows not only that the JVTS is in most cases much smaller than a concrete CS, but also that its computation is significantly faster.

We consider the following future work directions. First, one may propose an analogous symbolic representation of strategies for the realizable case, to allow engineers to efficiently explore a synthesized controller. Second, as we mentioned in Sect. 8, some works investigate the use of concrete CSs to generate candidate assumptions that will repair the unrealizable specification. Based on the JVTS, it may be possible to develop a symbolic and hence much more efficient repair mechanism.

The work is part of a larger project [2] 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 writing and understanding temporal specifications for reactive synthesis (see, e.g., [13, 14]).

ACKNOWLEDGEMENTS

We thank Ofir Fisher and Yoni Wolbe for their help in the implementation. 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).

[2] SYNTECH: http://smlab.cs.tau.ac.il/syntech/

REFERENCES

[1] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *FMCAD*, pages 26–33. IEEE, 2013.

[2] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY - A new requirements analysis tool with synthesis. In *CAV*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

[3] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[4] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *VMCAI*, volume 4905 of *LNCS*, pages 52–67. Springer, 2008.

[5] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models for fallible domains. In *ICSE*, pages 211–220, 2011.

[6] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9, 2013.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. ACM, 1999.

[8] R. Ehlers and V. Raman. Slugs: Extensible GR(1) synthesis. In *CAV*, volume 9780 of *LNCS*, pages 333–339. Springer, 2016.

[9] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray. Control design for hybrid systems with tulip: The temporal logic planning toolbox. In *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*, pages 1030–1041. IEEE, 2016.

[10] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013.

[11] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Trans. Robotics*, 25(6):1370–1381, 2009.

[12] S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit. Reactive high-level behavior synthesis for an atlas humanoid robot. In D. Kragic, A. Bicchi, and A. D. Luca, editors, *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, pages 4192–4199. IEEE, 2016.

[13] S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In *ESEC/FSE*, pages 96–106. ACM, 2015.

[14] S. Maoz and J. O. Ringert. On well-separation of GR(1) specifications. In *FSE*, pages 362–372. ACM, 2016.

[15] S. Maoz and Y. Sa'ar. AspectLTL: an aspect language for LTL specifications. In *AOSD*, pages 19–30. ACM, 2011.

[16] S. Maoz and Y. Sa'ar. Assume-guarantee scenarios: Semantics and synthesis. In *MODELS*, volume 7590 of *LNCS*, pages 335–351. Springer, 2012.

[17] S. Maoz and Y. Sa'ar. Counter play-out: executing unrealizable scenario-based specifications. In *ICSE*, pages 242–251. IEEE / ACM, 2013.

[18] S. Maoz and Y. Sa'ar. Two-way traceability and conflict debugging for aspectltl programs. *T. Aspect-Oriented Software Development*, 10:39–72, 2013.

[19] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[20] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *POPL*, pages 179–190. ACM Press, 1989.

[21] V. Raman and H. Kress-Gazit. Explaining impossible high-level robot behaviors. *IEEE Transactions on Robotics*, 29(1):94–104, 2013.

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

[23] A. Walker and L. Ryzhyk. Predicate abstraction for reactive synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 219–226. IEEE, 2014.

[24] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray. TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 313–314, New York, NY, USA, 2011. ACM.

[25] SYNTECH JVTS website. http://smlab.cs.tau.ac.il/syntech/jvts/.