2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)


# Symbolic Repairs for GR(1) Specifications

Shahar Maoz
Tel Aviv University
Tel Aviv, Israel

Jan Oliver Ringert
University of Leicester
Leicester, UK

Rafi Shalom
Tel Aviv University
Tel Aviv, Israel

***Abstract*—Unrealizability is a major challenge for GR(1), an expressive assume-guarantee fragment of LTL that enables efficient synthesis. Some works attempt to help engineers deal with unrealizability by generating counter-strategies or computing an unrealizable core. Other works propose to repair the unrealizable specification by suggesting repairs in the form of automatically generated assumptions.**

**In this work we present two novel symbolic algorithms for repairing unrealizable GR(1) specifications. The first algorithm infers new assumptions based on the recently introduced JVTS. The second algorithm infers new assumptions directly from the specification. Both algorithms are sound. The first is incomplete but can be used to suggest many different repairs. The second is complete but suggests a single repair. Both are symbolic and therefore efficient.**

**We implemented our work, validated its correctness, and evaluated it on benchmarks from the literature. The evaluation shows the strength of our algorithms, in their ability to suggest repairs and in their performance and scalability compared to previous solutions.**

## I. Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [31]. Instead of using model checking to verify a manually constructed implementation, synthesis offers an approach where a correct implementation of the system is automatically obtained for a given specification, if such an implementation exists. In reactive synthesis, an implementation is given as a controller, i.e., an automaton that accepts input from the environment and produces the system's output to always satisfy the specification (input from the environment may come from sensors, system's output includes commands for actuators). If such a controller exists, the specification is considered realizable. Otherwise, the specification is unrealizable, i.e., there exists an environment that can satisfy all the assumptions while forcing the system to violate some of its guarantees.

While synthesis from Linear Temporal Logic (LTL) specifications is generally considered impractical due its high computational complexity (double exponential in the length of the formula), in this work we focus on GR(1), an assume-guarantee fragment of LTL that has an efficient symbolic synthesis algorithm [3] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [10], [24]. 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).

One of the main challenges of reactive synthesis in general and of GR(1) synthesis in particular is to deal with unrealizable specifications [1], [4], [7], [18], [21]. One way to deal with unrealizability is repairing it, by automatically generating a repair in the form of new, additional assumptions, which would make the specification realizable.

Some existing repair techniques for GR(1) use concrete counter-strategies (CSs) as a starting point. A concrete CS shows how an environment can satisfy all the assumptions while forcing the system to violate some of its guarantees. Since generating a concrete CS is costly, due to the need to enumerate all its states, the efficiency of these repair techniques is limited. Other repair techniques are defined over the more general LTL. As such, they may suggest repairs that are outside the GR(1) fragment, and hence cannot be applied in a GR(1) setting. These limitations hinder the use of existing repair approaches by engineers.

In this work we present two novel, symbolic repair techniques that aim to address these limitations. First, `JVTS-Repair`. The JVTS [21] is a symbolic representation of CSs for GR(1), which is computed symbolically, without the expensive enumeration of concrete states. Given a JVTS, we generate candidate assumptions that eliminate the CS it represents. By construction, the nodes of a JVTS represent symbolically how a CS prevents an implementation. Based on these nodes, `JVTS-Repair` creates assumptions that prevent the CS. By iterating this approach, `JVTS-Repair` finds candidates for repairs. Second, `GLASS`, which does not rely on CSs but addresses unrealizability in a "global" way, directly analyzing assumptions and guarantees. `GLASS` computes a safety assumption that ensures the safety guarantees can be satisfied, a justice assumption for each justice guarantee, and finally an initial assumption to prevent losing initial states.

One strength of `JVTS-Repair` is its efficiency, thanks to its symbolic nature. It is also typically capable of generating many different repairs. It is, however, incomplete; for some unrealizable GR(1) specifications it cannot suggest a repair although such a repair exists. One strength of `GLASS` is that it is not only very efficient but also complete. We prove its completeness in the paper (as well as demonstrate it in our evaluation, see below). Unlike `JVTS-Repair`, `GLASS` is however limited to generating only one repair, and typically generates a repair that involves more variables than the ones suggested by `JVTS-Repair`.

We further extend the two techniques with size minimization using the notion of repair core, roughly, a locally minimal

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00106

```
env boolean r;  // request
env boolean c;  // clear

sys boolean g;  // grant
sys boolean v;  // valid

// sys responds with grant for every request
gar respond: pRespondsToS(r, next(g));

// if cleared or granted no immediate next grant
gar ungrant:  G ((c | g) -> next(!g));

// cleared request means not valid
gar exclude: G (c -> !v);

// infinitely often give valid grants
gar just: GF (g & v);
```

Listing 1
UNREALIZABLE REQUEST-GRANT (RG1) SPECIFICATION, ADOPTED FROM [1], [23]

subset of a repair that is already a repair, and with support for auxiliary variables, which are common in specifications that use patterns. We describe these extensions in Sect. VII.

We have implemented all the above ideas in Spectra Tools [33]. We present an evaluation of our work over benchmarks taken from the literature in terms of ability to repair, computation time, and scalability. The evaluation shows not only that our symbolic repairs are able to repair many specifications that could not be repaired by previous works, but also that their computation is significantly faster. See Sect. VIII.

Program repair is an active research area [13]. While we are partly inspired by works on program repair, we do not deal with repairing imperative programs but rather with repairing GR(1) specifications, which are temporal declarative specifications for reactive systems. Most previous works on repairing unrealizable GR(1) specifications (e.g., [1], [4]) have relied on a concrete CS. Our algorithms are symbolic. We discuss related work in Sect. IX.

## II. RUNNING EXAMPLE

We adopt the request-grant running example RG1 used in previous GR(1) repair papers [1], [23][1]. The environment has two variables, r for requests and c for clearing requests. The system has two variables, g for grants and v for marking grants valid. Four guarantees express the system's requirements. The first states that a request is answered by a grant within a positive finite number of steps[2]. The second prohibits a grant immediately following a grant or a clear. The third states that clear and valid are always mutually exclusive. The last guarantee requires a valid grant infinitely often.

The specification in Lst. 1 is unrealizable, i.e., there exists no implementation of the system that satisfies all guarantees. A simple CS demonstrates this: if the environment cancels (c) all requests (r), it prevents v from becoming true (to satisfy the third guarantee), and thus a valid grant (which violates the last guarantee). How can this unrealizable specification be repaired, i.e., can one add assumptions that will make it realizable?

Our symbolic repair algorithms aim at generating such assumptions.

Consider the following two assumptions: never cancel a request immediately in the next state `G(r -> !next(c))`, and always eventually issue requests that are not immediately canceled `GF(r & !c)`. These two assumptions constitute a repair of the specification, i.e., adding them makes the specification realizable. Such a repair is an example for the output automatically generated by our repair algorithms.

[1]Although GR(1) specifications typically contain assumptions, this example contains only guarantees. Nevertheless, we chose to use this example here because it is simple for presentation in a paper format and has been used in previous closely related works. Note that our repairs handle general GR(1) specifications, which indeed typically include assumptions.

[2]Note that the first guarantee is equivalent to the LTL formula $\mathbf{G}\,(r \rightarrow \mathbf{FX}g)$, which is not a pure GR(1) guarantee. It is implemented with a response pattern (following [24]), which introduces a Boolean auxiliary variable that is implicit in the specification (see also [3] for explicitly introducing the variable). We discuss patterns and auxiliary variables in Sect. VII-B.

## III. PRELIMINARIES

### A. Linear Temporal Logic (LTL)

We will use one of the standard definitions of *linear temporal logic (LTL)*, e.g., as found in [3], over present-future temporal operators **X** (next), **U** (until), **F** (finally), and **G** (globally), and past temporal operator **H** (historically).

For a finite set of Boolean variables $\mathcal{V}$, a *computation* $\sigma = s_0 s_1.. \in (2^{\mathcal{V}})^\omega$ is an infinite sequence of *states*, i.e., of truth assignments $s_i$ to $\mathcal{V}$. We use $\sigma, i \models \psi$ to denote that the LTL formula $\psi$ holds at position $i \geq 0$ of $\sigma$, as defined, e.g., in [3]. We denote $\sigma, 0 \models \psi$ by $\sigma \models \psi$, and say that $\sigma$ satisfies $\psi$.

### B. GR(1) Realizability

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 $\varphi$ prescribes the outputs of a system that from its winning states for all environment choices lead to computations that satisfy $\varphi$. A specification $\varphi$ 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) specification $\mathcal{S}$ consists of the following elements [3]:

- $\mathcal{X}$ input variables controlled by the environment;
- $\mathcal{Y}$ output variables controlled by the system;
- $\mathcal{X}'$ and $\mathcal{Y}'$ copies of input and output variables at next step
- $\theta^e$ assertion over $\mathcal{X}$ characterizing initial environment states;
- $\theta^s$ assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial system states;
- $\rho^e(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}')$ transition relation of the environment;
- $\rho^s(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}')$ 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.

A GR(1) specification is (strictly) realizable iff the following LTL formula is realizable:

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

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 have been presented in [3], [30]. The algorithm of Piterman et al. [30] computes winning states for the system, i.e., states from which the system can realize $\varphi^{sr}$.

Note that the problem of satisfiability, i.e., the existence of a computation that satisfies an LTL formula, is usually easier than the problem of realizability [20].

### C. Counter-Strategies and the JVTS

A specification is unrealizable if the environment can force the system to violate one of its guarantees while satisfying all the environment assumptions, i.e., the environment has a CS. A CS can be given by an LTS that for every state prescribes inputs from the environment that from its winning states for all system choices lead to computations that satisfy $\neg\varphi^{sr}$. A CS can ensure $\neg\varphi^{sr}$ by either forcing the system to a deadlock (violation of an initial guarantee or a safety guarantee) or by satisfying all justice assumptions $J_i^e$ and preventing at least one justice guarantee $J_j^s$ forever.

A Justice Violations Transition System (JVTS) [21] is an acyclic LTS consisting of two types of symbolic states abstracting the CS, attractor-states (forcing to initial or safety guarantee violations or to other symbolic states) and cycle-states (locally preventing at least one justice guarantee). 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 that CS. The JVTS is acyclic, e.g., every play will end in a deadlock or stay in some cycle state forever.

**Definition 1** (Justice Violations TS (JVTS) structure, see [21] for a complete definition)**.** Given an unrealizable GR(1) specification, a JVTS is an acyclic LTS $\langle Q, T, I, L \rangle$, where:

- $Q$ is a set of symbolic states where each $q \in Q$ is either a cycle-state or an attractor-state,
- $T \subseteq Q \times Q$ is an acyclic transition relation,
- $I$ is a set of symbolic initial states, and
- $L$ is a labeling function that labels all states in $Q$ with assertions over $\mathcal{X} \cup \mathcal{Y}$ characterizing the corresponding sets of concrete states in a CS.

Kuvent et al. [21] showed how to compute the JVTS symbolically and thus efficiently, without the expensive enumeration of concrete states of a CS.

### D. Symbolic Algorithm Notation

Symbolic synthesis algorithms in our context operate on sets of states and transitions instead of on their explicit representations. In our algorithms, we use assertions for symbolic representation of sets of states (over variables $\mathcal{X}$, $\mathcal{Y}$) and sets of transitions (over variables $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{X}'$, and $\mathcal{Y}'$). We operate on assertions using the usual Boolean operators, e.g., for an assertion $\xi$ over $\mathcal{X} \cup \mathcal{Y}$ the expression $\xi \wedge \theta^e$ characterizes all states in $\xi$ that are also initial environment states. In addition to standard Boolean operators we also use $prime(\xi)$, which translates an assertion $\xi$ over $\mathcal{X} \cup \mathcal{Y}$ to an equivalent assertion over $\mathcal{X}' \cup \mathcal{Y}'$, and quantification where for $V \subseteq \mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}'$ existential quantification $\xi|_{\exists V}$ yields an assertion without $V$ that holds iff there exists an assignment to variables in $V$ s.t. $\xi$ holds (analogous for universal quantification $\xi|_{\forall V}$).

All operations used in our algorithms have direct implementations using Binary Decision Diagrams with CUDD [32].

## IV. Problem Definition

Intuitively, the repair problem takes as input an unrealizable GR(1) specification $\mathcal{S}$ and produces a set of assumptions that make $\mathcal{S}$ realizable.

A closer look reveals that some (repaired) specifications do not allow assumptions and guarantees to be satisfied together and realizability implies that the system forces the environment to violate assumptions (a case of non-well-separation [25]). We say a specification $\mathcal{S}$ is $\gamma$-sat iff all existing assumptions and guarantees can be satisfied together, i.e., when $\gamma = \theta^e \wedge \theta^s \wedge \mathbf{G}(\rho^e \wedge \rho^s) \wedge \bigwedge_{i \in 1..n} \mathbf{GF}(J_i^e) \wedge \bigwedge_{j \in 1..m} \mathbf{GF}(J_j^s)$ is satisfiable.

**Example 1.** *The specification from Lst. 1 is $\gamma$-sat. Consider adding the assumption $\psi =$* `G(!g -> next(c))` *to always cancel requests if no grant is given. This specification is still unrealizable but not $\gamma$-sat: guarantee* `ungrant` *requires that* `g` *becomes false and together with $\psi$,* `g` *stays false forever and guarantee* `just` *cannot be satisfied.*

*Interestingly, assumption* `GF(r & !c)` *from Sect. II makes the specification realizable. However, after adding this assumption a valid system implementation never grants requests, the environment has to always cancel, and is forced to violate the new assumption.*

We consider repairs leading to $\gamma$-unsat specifications not useful.[3] Adding assumptions to a $\gamma$-unsat specification preserves $\gamma$-unsatisfiability and we thus consider it as unrepairable (indeed we have found several unrealizable and $\gamma$-unsat specifications in our evaluation, see Sect. VIII).

Formally, we define the repair problem as follows: given an unrealizable and $\gamma$-sat specification $\mathcal{S}$, find additional assumptions that will make $\mathcal{S}$ realizable and $\gamma$-sat. We call such a set of assumptions a repair.

## V. JVTS-Based Symbolic Repair

We now present our first main contribution, namely a symbolic repair based on the JVTS. We first describe an algorithm to generate candidate repair assumptions from a JVTS, and then describe the main repair algorithm, which uses the first algorithm as a building block. Finally, we discuss termination and complexity.

[3] Alur et al. [1] and Cavezza and Alrajeh [4] consider weaker formulas for satisfiability of their repairs and allow some repairs we consider not useful.

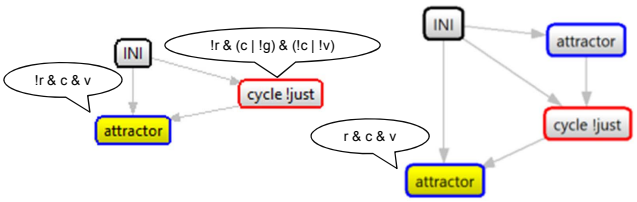


Fig. 1. Two JVTSs representing CSs for the running example `RG1`.

### A. Generating Assumptions from a JVTS

Given a JVTS, we generate candidate repair assumptions based on its cycle nodes, attractor nodes, and edges. We build the assumptions from the symbolic representation of the JVTS, with the goal of eliminating the CS (and additional, similar ones) that the JVTS represents. We present this in Alg. 1.

The first loop (lines 1-3) infers justice assumptions from the cycle nodes of the JVTS by negating the symbolic representation of the label of the node. The generated justice assumption eliminates the ability of the environment to violate the justice of the system represented by the specific JVTS cycle node. With the new assumptions, the system can avoid losing in the scenario of the CS represented by the JVTS by forcing the environment into the cycle.

**Example 2.** *Fig. 1 (left) shows a JVTS computed for our running example specification `RG1` (see Sect. II). The JVTS has a cycle node in which the justice guarantee `just` fails. Negating its label creates the candidate repair justice assumption $\phi_1$:* `GF (r | (!c & g) | (c & v))`

The second loop (lines 4-6) infers safety assumptions from attractor nodes of the JVTS that have no outgoing edges, by negating the symbolic representation of the label of the node. The system can avoid losing by forcing the environment into such attractor nodes. We ignore attractor nodes that have outgoing edges because they must produce stronger assumptions than the ones generated next from outgoing edges in lines 7-9, and stronger assumptions are removed by `removeStrong` in line 10 (see bellow).

**Example 3.** *The JVTS in Fig 1 (left) has an attractor without outgoing edges. Negating its label creates the candidate repair safety assumption $\phi_2$:* `G (r | !c | !v )`

The third loop (lines 7-9) infers safety assumptions from each edge $(v_1, v_2)$ of the JVTS. These assumptions prevent the transition from node $v_1$ to $v_2$ by negating the conjunction of the symbolic representation of $v_1$ with the primed representation of $v_2$, i.e., the equivalent assertion on successor states, that has the system variables quantified out using existential quantification. The system can avoid the CS by forcing the environment into node $v1$, and choosing an action that would make the environment respond by trying to enter into node $v_2$. Note that both $v_1$ and $v_2$ can be either cycle or attractor nodes.

**Example 4.** *The JVTS in Fig 1 (left) has an edge from the cycle node to the attractor. This results in the generation of the candidate repair transition assumption $\phi_3$:*

`G (r | (!c & g) | (c & v) | !next(c) | next(r))`

**Algorithm 1** **InferAssumptionsFromJVTS** infers candidate repair assumptions from a given JVTS.

**Require:** A JVTS $\langle Q, T, I, L\rangle$
**Ensure:** A set of GR(1) assumptions, each of which eliminate the CS
1: **for** each cycle node $v \in Q$ **do**
2: **add** $\{\texttt{GF}\ \neg L(v)\}$ **to** $Cand$
3: **end for**
4: **for** each attractor node $v \in Q$ such that $\forall v' \in Q : \neg T(v, v')$ **do**
5: **add** $\{\texttt{G}\ \neg L(v)\}$ **to** $Cand$
6: **end for**
7: **for** each two nodes $v_1, v_2 \in Q$ such that $T(v_1, v_2)$ **do**
8: **add** $\{\texttt{G}\ \neg(L(v_1) \wedge prime(L(v_2)|_{\exists \mathcal{Y}})\}$ **to** $Cand$
9: **end for**
10: **return** **removeStrong**$(Cand)$

Finally, to avoid redundancy, we return only the weaker assumptions in the set $Cand$. Specifically, the method `removeStrong` eliminates an assumption from the set iff it implies one of the other assumptions in the set. Since we only have justice and safety assumptions, assumption $asm_1$ implies assumption $asm_2$ when the propositional formula of $asm_1$ implies that of $asm_2$, and either both $asm_1$ and $asm_2$ are of the same type, or $asm_1$ is a safety assumption and $asm_2$ is a justice assumption.

Note that by construction, all the generated candidate repair assumptions are safety or justice assumptions, and in the safety assumptions for transitions, only environment variables are primed (since the system variables of $v_2$ are quantified out). Thus, importantly, the candidate repair assumptions we generate respect the restricted structure of individual assumptions in a GR(1) specification.

### B. The JVTS-Repair Algorithm

The assumptions inferred from a single JVTS are typically not enough to ensure realizability, since they only ensure the elimination of a single CS. On the other hand, adding assumptions may render a satisfiable specification $\gamma$-unsat.

Algorithm 2 uses the algorithm **InferAssumptions-FromJVTS** depicted in Alg. 1 for repair generation. It is a breadth first search on assumptions generated from JVTS CSs. For a given unrealizable and $\gamma$-sat specification, it searches for a set of assumptions that makes the specification realizable while keeping it $\gamma$-sat, by adding assumption as long as the specification remains unrealizable and $\gamma$-sat.

`queue` is a queue of sets of assumptions, which represent repair candidates. We begin by inserting an empty set to the queue. While `queue` is not empty, we remove a repair candidate from it, and add the candidate to the specification. If the refined specification is $\gamma$-sat and realizable, the candidate is a repair, thus it is returned, and the search ends. If the refined specification is $\gamma$-sat and not realizable, we generate a JVTS for the refined specification, generate assumptions from it using **InferAssumptionsFromJVTS**, and add the unions of the candidate with each of them to `queue`.

**Algorithm 2 JVTS-Repair** computes a repair, or fails.

**Require:** An unrealizable and $\gamma$-sat $GR(1)$ specification $\mathcal{S}$
**Ensure:** $GR(1)$ assumptions that make $\mathcal{S}$ realizable and $\gamma$-sat, if found

1: $queue.enqueue(\emptyset)$
2: **while** $queue$ is not empty **do**
3: $candidate \leftarrow queue.dequeue$
4: $refined \leftarrow$ **addAssumptions**$(\mathcal{S}, candidate)$
5: **if isSatisfiable**$(refined)$ **then**
6: **if isRealizable**$(refined)$ **then**
7: **return** $candidate$
8: **else**
9: $JVTS \leftarrow$ **computeJVTS**$(refined)$
10: $assumptions \leftarrow$ **InferAssumptionsFromJVTS**$(JVTS)$
11: **for** $asm \in assumptions$ **do**
12: $queue.enqueue(candidate \cup \{asm\})$
13: **end for**
14: **end if**
15: **end if**
16: **end while**

**Algorithm 3 GLASS** computes a repair.

**Require:** An unrealizable and $\gamma$-sat $GR(1)$ specification $\mathcal{S}$
**Ensure:** $GR(1)$ assumptions that make $\mathcal{S}$ realizable and $\gamma$-sat

1: $collSat \leftarrow$ **collWinStates(** $\mathtt{G}(\rho^e \wedge \rho^s) \wedge (\bigwedge_{i\in 1..n} \mathtt{GF} J_i^e \wedge \bigwedge_{j\in 1..m} \mathtt{GF} J_j^s)$ **)**
2: $badEnvTrans \leftarrow collSat \wedge \rho^e \wedge (\rho^s \Rightarrow \neg prime(collSat))|_{\forall\mathcal{Y}'}$
3: $\rho' \leftarrow \mathtt{G}\ \neg badEnvTrans$
4: **if isRealizable(** **addAssumptions(**$\mathcal{S}, \{\rho'\}$**)** **) then**
5: **return** $\{\rho'\}$
6: **end if**
7: **for** $J_j^s \in J^s$ **do**
8: $J \leftarrow$ **envWinStates(** $\mathtt{F}((\mathtt{H}\rho^e) \wedge \neg\rho^s) \vee \bigwedge_{i\in 1..n} \mathtt{GF} J_i^e \wedge \mathtt{G}\neg J_j^s$ **)**
9: **add** $\mathtt{GF}\ \neg J$ **to** $J'$ **unless** $J = \mathtt{F}$
10: **end for**
11: **if isRealizable(** **addAssumptions(**$\mathcal{S}, \{\rho', J'\}$**)** **) then**
12: **return** $\{\rho'\} \cup J'$
13: **end if**
14: $win \leftarrow$ **sysWinStates(** **addAssumptions(**$\mathcal{S}, \{\rho', J'\}$**)** **)**
15: $\theta' \leftarrow (\theta^e \wedge \theta^s \wedge win)|_{\exists\mathcal{Y}}$
16: **return** $\{\theta', \rho'\} \cup J'$

As adding assumptions to a $\gamma$-unsat specification cannot make it $\gamma$-sat, we avoid these search paths. Thus, we never return repair candidates that make the specification $\gamma$-unsat.

**Example 5.** *When we run* `JVTS-Repair` *on our $\gamma$-sat and unrealizable example specification in Lst. 1, it first creates the left hand JVTS in Fig 1. From this JVTS, it generates the three assumptions $\phi_1, \phi_2, \phi_3$. With any of these assumptions the specification remains unrealizable and $\gamma$-sat.*

*The search continues and generates repairs in different ways. For example, after adding the assumption $\phi_2$, Alg. 2 (line 9) generates the right hand JVTS in Fig 1. From the attractor at the bottom of this JVTS, it generates the assumption $\phi_4$:* `G (!r | !c | !v)`*. The set $\{\phi_2, \phi_4\}$ is a valid repair, thus Alg. 2 returns both assumptions to the engineer.*

### C. Termination, Complexity, and Soundness

Algorithm 2 always terminates. The while loop in line 2 must terminate, because queue elements increase in the number of assumptions they contain, each additional assumption causes the refined specification in line 4 to be semantically different from the specification without it (because it eliminates at least one CS that existed before the assumption was introduced), and there is a finite number of specifications for a given set of variables. This forms a bound on the size of queue elements, double exponential in the number of variables (as common to all CS-based repair approaches [4]), and for each size of queue elements there is only a finite number of such elements because the number of assumptions inferred from a JVTS (line 10) is always finite. The computation time for each iteration is composed from a $\gamma$-sat check, a realizability check, a JVTS computation, and a call to **InferAssumptionsFromJVTS**. All these computations are done symbolically. We performed $\gamma$-sat and realizability checks via fixed-point algorithms, polynomial in the number of assumptions, guarantees, and state space [3].

Note in particular that the computation of a JVTS does not require the costly computation of a concrete CS. The JVTS is usually very small, as the number of its cycle nodes is bounded by the number of justice guarantees, and there are at most two attractor nodes per cycle node. Accordingly, the time and space needed for its computation are usually significantly lower than those of concrete CSs [21].

The JVTS based repair is sound (see check in Alg. 2, l. 6) but incomplete; it does not guarantee that a repair would be found. We evaluate its performance in Sect. VIII.

## VI. GLASS Repair

We now present our second main contribution, namely a symbolic global assumption (`GLASS`) repair that computes repairs in a "global" way, considering all causes of unrealizability, instead of a "local" way, considering the causes exhibited by a CS. `GLASS` addresses reasons for unrealizability on the levels of safety guarantees, justice guarantees, and initial states, by computing respective assumptions that ensure realizability. `GLASS` is sound and complete, i.e., it computes a repair for all $\gamma$-sat specifications. In addition, `GLASS` does not suffer from the double exponential complexity of CS-based approaches (see Sect. V-C and [4]).

### A. Algorithm

Algorithm 3 symbolically computes up to three kinds of assumptions (in this order): a safety assumption $\rho'$, a set $J'$ of up to $|J^s|$ justice assumptions, and an initial assumption $\theta'$.

First, Alg. 3 symbolically computes all states $collSat$ from which the system and environment player can collaboratively satisfy all safety and justice assumptions and guarantees. Then, it computes all transitions $badEnvTrans$ that start from $collSat$ and are legal for the environment ($\rho^e$) but force all legal system choices ($\rho^e$) to leave the states $collSat$. The removal of these *bad* environment transitions is the safety assumption $\rho'$. This part is a symbolic version of safety assumption computation from [6], adapted for satisfying all assumptions and guarantees.

**Example 6.** *No safety assumption is necessary for the specification in Lst. 1 ($collSat$ is the set of all states and $\rho'$ in l. 3 introduces no restriction). Consider adding the guarantee* `G(r -> next(v))`*, which contradicts guarantee* `exclude` *when the system grants and the environment cancels in the next step.*

*Now, Alg. 3 would generate the assumption* `G(g -> ((c & v) | next(!c)))`*, which prevents a cancel in the next step after a guarantee.*

If the assumption $\rho'$ does not make the specification realizable, Alg. 3 computes for every justice guarantee $J_j^s$ the states $J$ from which the environment can prevent $J_j^s$ while satisfying all justice assumptions. The negations of these states are the justice assumptions $J'$. The new justice assumptions in $J'$ force the environment to always eventually leave the states where each $J_j^s$ can be prevented. Technically, the LTL formula in Alg. 3, l. 8 is a sub-formula of the negation of $\varphi^{sr}$ evaluated inside algorithms for computing GR(1) CS. Also, the set $J$ contains all states from leaf cycle-nodes in a JVTS.

**Example 7.** *Algorithm 3 computes the justice assumption* `GF((!c -> g & v) & (c -> v))`*. This assumption alone is a repair for the specification from Lst. 1.*

Finally, if the assumptions $\rho'$ and $J'$ are not enough to make the specification realizable, Alg. 3, l. 15 restricts the initial environment states to $\theta'$ where a legal system choice exists to reach a system winning state, i.e., a state to realize the repaired specification. All computed assumptions are in GR(1).

**Example 8.** *No initial assumption is necessary for the specification in Lst. 1. Consider adding the initial guarantee* `v`*. In this case, Alg. 3 computes the initial assumption* `!c` *to avoid violating guarantee* `exclude` *in the initial state.*

### B. Soundness, Completeness, and Minimality

Algorithm 3 is sound and complete as stated in Thm. 1.

**Theorem 1.** *Given an unrealizable and $\gamma$-sat $GR(1)$ specification $\mathcal{S}$, Alg. 3 computes a repair that keeps $\mathcal{S}$ $\gamma$-sat and makes it realizable.*

*Proof.* The proof shows that each section of Alg. 3 allows the system to realize safety, justice, and initial guarantees respectively, and that each new assumption preserves $\gamma$-satisfiability. See supporting materials for details of the proof [37]. □

The time complexity of Alg. 3 is the same as checking GR(1) realizability [3]: the evaluations of LTL formulas in l. 1 and l. 8 require at most three nested fixed-point iterations with at most $2^{|\mathcal{X}\cup\mathcal{Y}|}$ steps, each.

Although the assumptions $\theta'$, $\rho'$, and each $J \in J'$, are minimal (see supporting materials [37]), already the combination of some $J \in J'$ might not be minimal, as these sets of states might not be independent. Importantly, minimality for the assumptions of `GLASS` is defined with respect to realizability from all possible states, whereas minimality for repairs can also be seen in a "local" way as minimal assumptions that allow for realizability from at least one state (see Sect. VII-A and weakness measures in Sect. IX).

## VII. Important Extensions and Variants

### A. Repair Core

Both `JVTS-Repair` and `GLASS` may yield repairs that are not minimal in terms of the set of assumptions required for realizability, i.e., where the set of suggested assumptions includes a strict subset that is already a repair. In general, we would like to reduce the number of assumptions consisting the suggested repair, if possible.[4]

**Example 9.** *As an example, for the specification from Sect. II, the JVTS-based repair suggested the repair consisting on the set of assumptions $\{\phi_1, \psi_2, \psi_3\}$ where $\psi_2$ is:*

```
G (!next(r) | ((r | !c | v) & (!r | c | g)))
```

*and $\psi_3$ is:* `GF ((c | g) & (!c | v))`*, which it obtained by first taking the justice assumption $\phi_1$ at level 1 of the search (see Example 2), and then taking assumptions $\psi_2$ and $\psi_3$ at levels 2 and 3 of the search respectively.*

To try to reduce the number of assumptions in the repair, we apply the delta debugging algorithm DDMin [36] to heuristically compute what we call a *repair core*, a locally minimal subset of the assumptions in the suggested repair that suffices to make the specification realizable. This involves multiple calls to check realizability, each with a different subset of additional assumptions. The correctness of applying delta debugging relies on the following monotonicity: adding an assumption to a realizable specification preserves realizability.

**Example 10.** *Back to our last example above, when we apply DDMin to $\{\phi_1, \psi_2, \psi_3\}$ we find that the subset of assumptions $\{\psi_3\}$ is a repair core: it is sufficient for realizability, and any strict subset of it (in this case, no assumption at all) will not make the specification realizable.*

Finally, although given the suggested repair, some assumptions that appear already in the original specification may become unnecessary for realizability, we choose not to remove any assumptions but only to suggest new ones to add. Suggesting to remove such redundant assumptions is independent of the repair problem.

In Sect. VIII, as part of our evaluation, we report empirical results on the effectiveness of using the repair core to reduce the size of the repairs we found.

### B. Dealing with Auxiliary Variables

Some reactive synthesis tools allow writing specifications that are not pure GR(1), yet reducible to GR(1) specifications, e.g., ones that use LTL specification patterns [24]. In this case, the translation to GR(1) may introduce auxiliary variables, which were not explicitly declared in the original specification.

Though technically, one could use assumptions that include auxiliary variables in a repair, such assumptions may not be desirable because the auxiliary variables do not explicitly appear in the original specification and thus are unknown to the engineer who wrote it. To ensure that we do not generate assumptions with auxiliary variables, we quantify them out from all symbolic representations.

[4]Note that we consider a syntactic notion of minimality here, related to the number of assumptions in the repair, not a semantic one based on implication or weakness (see related work discussion in Sect. IX.)

This quantification is performed as follows. In the JVTS-based repair we existentially quantify auxiliary variables from all node labels before they are used. In GLASS, we existentially quantify auxiliary variables from any suggested assumption when adding it to the specification.

It is important to note that when auxiliary variables are quantified out, some solutions may be lost. This means in particular that in this case, the completeness results in Sect. VI-B will not hold. However, the correctness of repairs that are found is not affected in this case, i.e., GLASS remains sound. In Sect. VIII, as part of our evaluation, we report empirical results on the effect of quantifying out the auxiliary variables on the ability of GLASS to suggest a repair.

## VIII. Evaluation

We have implemented JVTS-Repair and GLASS in Spectra Tools [33], based on CUDD [32] as a BDD library. Our implementation includes also the computation of concrete CS based on [18], [27], the computation of the JVTS based on [21], an implementation of the approach by Alur et al. [1], and the extensions described in Sect. VII.

Means to run our implementation, all specifications used in our evaluation, and all data we report on below, are available in supporting materials for inspection and reproduction [37]. We encourage the interested reader to try them out.

We consider the following two research questions. How do JVTS-Repair and GLASS perform and compare to previous repair work in terms of

**R1** ...the ability to repair?

**R2** ...(a) computation time and (b) scalability?

Below we report on the experiments we have conducted in order to answer the above questions, and on additional observations from the results of these experiments.

### A. Corpus of Specifications

Previous works on repairing unrealizable GR(1) specifications have only used a handful of benchmark specifications for evaluation, from which we used the following in our evaluation: the request-grant specification we used as a running example, RG1, taken from [1], [23]; a similar smaller request-grant specification, RG2, taken from [6]; and the lift specification LIFT used in [1], [4]. We further used 3 different sizes of AMBA (1 to 3 masters), each in the 3 variants of unrealizability described in [7] (with a justice assumption removed, with a justice guarantee added, and with a safety guarantee added), i.e., a total of 9 AMBA specifications we label AMBA-CIMATTI. We have used all these in our evaluation.

On top of these, importantly, we used the benchmark SYNTECH15 [12], which includes a total of 78 specifications of 6 autonomous Lego robots, written by 3rd year undergraduate computer science students in a project class taught by the authors of [12]. Out of the 78 specifications in SYNTECH15, 17 are unrealizable, but 2 of the 17 are $\gamma$-unsat (and thus cannot be repaired by adding assumptions). Therefore below we report on the remaining 15 specifications. We label them SYNTECH15-UNREAL.

From the 61 realizable specifications of SYNTECH15, we produced unrealizable specifications as follows. For each specification, we first found a core of the assumptions that maintains realizability [7]. Recall that removing any assumption from the core makes the specification unrealizable. From each specification with a core of $n$ assumptions, we created $n$ unrealizable variants, each of which missing one of the core assumptions. Some specifications had an empty assumption core and thus produced no unrealizable variants. In this way, we produced a total of 144 unrealizable specifications. Out of these 144 specifications, 8 were $\gamma$-unsat because the original specification was not $\gamma$-sat, leaving 136 unrealizable repairable specifications. We label them SYNTECH15-1UNREAL.

### B. Validation

We have systematically and automatically validated the correctness of our implementation by actually adding the computed additional assumptions to the unrealizable specifications they are supposed to repair, and by independently checking that the repaired specification is indeed satisfiable and realizable, for all the specifications mentioned in this paper. This validation includes not only our two symbolic repair techniques but also our implementation of the algorithm of Alur et al. [1].

The validation increases our confidence in the correctness of our ideas and their implementation.

### C. Experiments Setup

In all cases, we run the three algorithms, AMT13 (our implementation of the algorithm in [1])[5], JVTS-Repair, and GLASS, until termination or timeout, regardless of repairs found or not.

Using AMT13 requires setting several parameters (see Alur et al. [1]). Since we run until termination or timeout, we ignored the $\alpha$ parameter. We chose the $\beta$ parameter according to the heuristic suggested in [1]. The algorithm also requires the engineer to choose environment variables that appear in repairs; we always chose all of them.

Using JVTS-Repair and GLASS requires a decision regarding auxiliary variables. We chose to quantify them out, taking the risk of loosing some solutions (see Sect. VII-B).

We run all experiments on an ordinary PC, Intel Xeon W-2133 CPU 3.6GHz, 32GB RAM with Windows 10 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 to finding first repair. We excluded the first check of satisfiability and unrealizability from the measurements (as they are common and necessary in all approaches).

We used a fixed timeout of 10 minutes. We mark timeouts by TO. Times we report are average values of 10 runs per specification per algorithm, measured by Java in milliseconds. Even though the algorithms we deal with are deterministic, we performed 10 runs since JVM garbage collection and BDD dynamic-reordering add variance to running times.[6]

[5]Note, we count repairs by AMT13 although they might not be $\gamma$-sat.

[6]Since BDD-based implementations' performance is sensitive to variable order, we note that in all our experiments we used CUDD's automatic variable reordering. This is common practice in the literature.

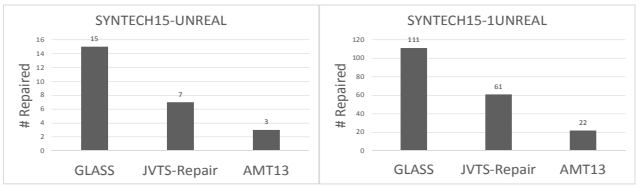


Fig. 2. Total numbers of repaired specifications per algorithm, from SYNTECH15-UNREAL (left) and SYNTECH15-1UNREAL (right).

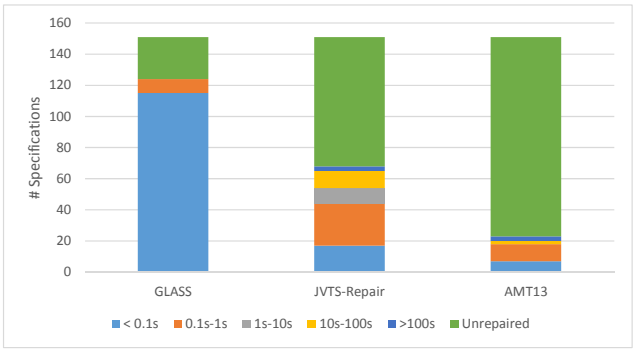


Fig. 3. Computation time to suggesting first repair, per algorithm, for all SYNTECH15-UNREAL and SYNTECH15-1UNREAL specifications, broken down into ranges.

### D. Results: Ability to Repair

Figure 2 presents the results on SYNTECH15-UNREAL and SYNTECH15-1UNREAL. For each of the three algorithms, we show the number of specifications for which at least one repair was found.

The results show that GLASS is able to repair almost all specifications in SYNTECH15-UNREAL and SYNTECH15-1UNREAL. The cases where it fails to suggest a repair are due only to the introduction of auxiliary variables. We verified that GLASS always finds a repair when auxiliary variables are not quantified out. Regarding the two other algorithms, we observe that JVTS-Repair is able to repair more specifications than AMT13.

For the AMBA-CIMATTI specifications, GLASS repaired all, but JVTS-Repair and AMT13 repaired none. For all other specifications, RG1, RG2, and LIFT, all three algorithms found a repair.

To answer R1: GLASS repairs all SYNTECH15-UNREAL and over 80% of SYNTECH15-1UNREAL. JVTS-Repair repairs 45% while AMT13 repairs only 17% of the above specifications. Similar observations hold for the other specifications in the corpus.

### E. Results: Computation Time and Scalability

Figure 3 shows the computation time of all SYNTECH15-UNREAL and SYNTECH15-1UNREAL specifications broken down into several ranges: up to 0.1 seconds, 1 second, 10 seconds, 100 seconds, more than 100 seconds before timeout, and unrepaired either because the search terminated with no repair or because timeout was reached. Evidently, GLASS repairs most of the specifications very quickly, while JVTS-Repair does less well, which is still significantly better than the AMT13 algorithm, both in the number of specifications repaired and in the computation time.

TABLE I
COMPUTATION TIMES TO FIRST REPAIR IN MS, PER ALGORITHM, FOR SELECTED SPECIFICATIONS WITH GROWING NUMBER OF ADDITIONAL SYSTEM AND ENVIRONMENT VARIABLES

| Specification | #vars | Added Environment Variables | | | Added System Variables | | |
|---|---|---|---|---|---|---|---|
| | | GLASS | JVTS | AMT13 | GLASS | JVTS | AMT13 |
| RG1 (Paper example) | 0 | 13 | 78 | 153 | 13 | 78 | 153 |
| | +1 | 12 | 955 | 6110 | 15 | 65 | TO |
| | +2 | 15 | TO | TO | 10 | 68 | TO |
| | +3 | 7 | TO | TO | 12 | 70 | TO |
| | +4 | 9 | TO | TO | 7 | 70 | TO |
| | +5 | 6 | TO | TO | 1 | 62 | TO |
| Lift | 0 | 12 | 53 | 26 | 12 | 53 | 26 |
| | +1 | 12 | 89 | 50 | 12 | 59 | 24 |
| | +2 | 10 | 975 | 203 | 9 | 57 | 39 |
| | +3 | 14 | 125451 | 193564 | 9 | 59 | 370 |
| | +4 | 10 | TO | TO | 12 | 54 | TO |
| | +5 | 12 | TO | TO | 10 | 53 | TO |
| HumanoidLTL_458 | 0 | 6 | 20 | 148 | 6 | 20 | 148 |
| | +1 | 4 | 21 | 249 | 8 | 19 | 348 |
| | +2 | 10 | 29 | 426 | 7 | 20 | 1071 |
| | +3 | 9 | 42 | 775 | 12 | 20 | 3475 |
| | +4 | 6 | 59 | 1558 | 6 | 23 | 11574 |
| | +5 | 4 | 90 | 2900 | 6 | 21 | 47233 |
| Gyro_Var1_710 | 0 | 7 | 93 | 142 | 7 | 93 | 142 |
| | +1 | 9 | 157 | 630 | 7 | 89 | TO |
| | +2 | 9 | 3931 | 31777 | 12 | 99 | TO |
| | +3 | 10 | TO | TO | 6 | 85 | TO |
| | +4 | 6 | TO | TO | 10 | 94 | TO |
| | +5 | 7 | TO | TO | 6 | 84 | TO |

To examine scalability, we conducted the following experiment. Given a specification, we created 5 additional variants with 1 to 5 additional Boolean system variables, and 5 additional variants with 1 to 5 additional Boolean environment variables. We did not constrain the additional variables in any way. This results in an exponential inflation of the state space, while having no effect on the unrealizability and the correctness of suggested repairs. We measured the time required to find the first repair.

Table I shows the running times, for RG1, Lift, and two SYNTECH15-UNREAL specifications HumanoidLTL_458 and Gyro_Var1_710, and their variants. The lines in the table with a zero value for number of added variables refer to the original specification.

The results show that the GLASS repair computation time is almost unchanged. Similarly, JVTS-Repair computation time is almost oblivious to additional system variables, but in most cases goes up rather fast with the addition of environment variables. Finally, they show that the computation time of AMT13, which is based on a concrete CS, scales rather poorly with the addition of either system or environment variables.

Finally, we performed another experiment for scalability using specifications we call $GFcomplete_n$. These specifications have an integer variable kval in the range $1..n$, and $n$ guarantees of the form GF kval=i, for $1 \leq i \leq n$. The state space in these specifications grows linearly in $n$ rather than exponentially.

Figure 4 shows computation time to first repair (log scale) of $GFcomplete_n$ specifications for $2 \leq n \leq 9$. Both GLASS and JVTS-Repair scale well while AMT13 grows quickly,

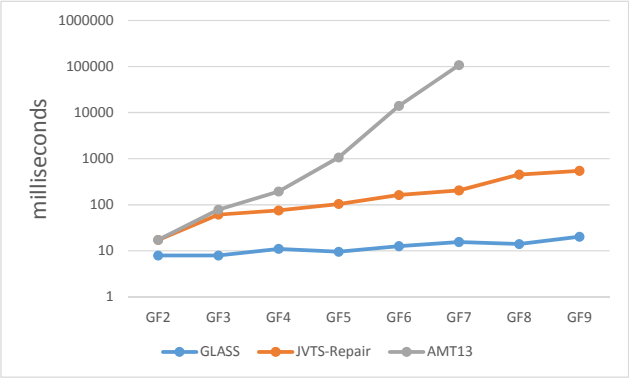


Fig. 4. Computation times to first repair for instances of the $GFcomplete_n$ specifications. Note that the graph is in logarithmic scale. Missing graph points for `AMT13` are due to the algorithm reaching the timeout.

and fails to find a repair within the timeout for the last two specifications.

To answer R2: `GLASS` is able to repair almost all specifications within 1s. `JVTS-Repair` and `AMT13` are slower. `GLASS` scales well and seems to be indifferent to additional variables. `JVTS-Repair` scales well only for additional system variables. `AMT13` does not scale well.

### F. Results: Additional Observations

**Number of variables.** In general, one may prefer repairs that are simpler to understand. As a proxy for simplicity, we report the number of variables in the repairs we found. We consider repairs for which all assumptions have up to 5 variables to be *small*. We count a variable `v` if it appears in an assumption. The algorithms differ significantly in this regard. `GLASS` has small repairs only for 47% (of 136 repaired specifications), `JVTS-Repair` has small repairs for 57% (of 71), and `AMT13` has small repairs for all (of 28) corpus specifications it repairs.

**Effectiveness of repair core.** We found that the repair core (see Sect. VII-A) is effective in reducing the size of the repairs. For `GLASS`, the repair core is strictly smaller than the initially suggested repair for 68% of the repaired specifications. For `JVTS-Repair`, the repair core is strictly smaller than the initially suggested repair for 53% of the repaired specifications. Interestingly, without repair core computation, `GLASS` offered small repairs only for 25% of the repaired specifications (as opposed to 47% reported above), while for `JVTS-Repair`, the difference is insignificant.

Additional observations appear in [37].

### G. Threats to Validity

We briefly discuss threats to the validity of our results.

**Internal.** The symbolic computations are not trivial and our implementation may have bugs. To mitigate this, we performed a thorough validation using all specifications available to us, see Sect. VIII-B.

**External.** First, one of the main threats is the fit of the generated assumptions to the real world. Note that this threat equally applies to the previous solutions by Alur et al. and by Cavezza and Alrajeh [1], [4].

Second, we did not perform a user-study, with engineers, to examine whether users will find the repairs useful, understand their meaning, and indeed add them to their specifications.

Third, we have based most of our evaluation on specifications from the SYNTECH15 set [12], which were created by 3rd year undergraduate computer science students with no prior experience in writing LTL specifications. Due to the lack of other real-world unrealizable specification examples, the specifications we used were all unrealizable specifications available to us, and ones we have systematically created from the realizable ones. Still, note that the scope of our evaluation is much larger than that of similar and competing works on reactive synthesis in general and on repair in particular.

## IX. Related Work

### A. Automatic Program Repair

Program repair is an active research area [13], with two main classes of approaches, both of which start with a faulty program and a test suite that reveals a defect. Generate-and-validate approaches explore candidate programs in a search space until a program that passes all tests is found (see, e.g., [22], [15], [17], [34]). Semantics-driven approaches infer program specifications, translate the repair problem into constraints, and use solvers to find patches (see, e.g., [16], [28], [29], [35]). A recent TSE survey [13] discusses 108 program repair papers, more than 50 of which published between 2013 and 2016. Almost all focus on imperative languages such as Java and C.

We are partly inspired by this body of work. However, we do not deal with repairing imperative programs but rather with repairing GR(1) specifications, which are temporal declarative specifications for reactive systems. We focus on repair of unrealizability based on the symbolic generation of new assumptions. Both our JVTS-based and `GLASS` techniques can be viewed as semantics-driven repairs, as they rely on dedicated symbolic algorithms to solve what can be viewed as a set of constraints. The iterative nature of the JVTS-based technique is somewhat similar to a generate-and-validate approach.

### B. Repair of GR(1) Specifications

GR(1) synthesis was introduced in [30]. It has since been used and investigated by many, including, e.g., Kress-Gazit et al. [19], who used GR(1) in robotics; Maoz and Ringert [24], who showed GR(1) synthesis for specification patterns; D'Ippolito et al. [8], [9], who used GR(1) to deal with fallible domains and non-anomalous event-based behavior models; and Gritzner and Greenyer [14], who used scenario-based GR(1) specifications to synthesize executable PLC code. Several tools support GR(1) synthesis [2], [11], [33].

Some works deal with unrealizable GR(1) specifications using CSs and cores, but do not consider repair [7], [18], [26]. Below we discuss other works that are most closely related to ours.

Li et al. [23] suggested mining assumptions from *concrete* CS of unrealizable GR(1) specifications. Strengthening the specification with these assumptions may make the specification realizable. Their algorithm enumerates constraints following three patterns for GR(1) safety and justice assumptions. If the CS satisfies the enumerated constraint, its negation is added as an assumption that rules out the CS. Li et al. [23] also suggest using traces of correct behaviors, if available, to validate candidates. The generation of constraints is unguided and each requires model-checking against an automaton of the CS.

Alur et al. [1] proposed another method for semi-automatic strengthening of assumptions. Again, they analyze a *concrete* CS, but their instantiation of template-based candidate assumptions is guided by the CS. The work provides limited evaluation over three specifications. In contrast to [1] and [23], our `JVTS-Repair` uses a symbolic CS representation. We have re-implemented the approach from [1] in order to be able to compare it to ours. Our evaluation showed that our symbolic repairs are able to repair many more specifications that [1] cannot, and that they are typically much faster.

Cavezza and Alrajeh [4] proposed the generation of interpolants as a means to compute new assumptions. Similar to [1], [23], this work uses a concrete CS. The work provides evaluation over two benchmarks, Lift and AMBA. Unfortunately, we were unable to compile and run an implementation of this approach that we can use to directly compare with our symbolic repairs (as we have done with [1]). However, our evaluation includes a comparison of results against the same benchmark specifications reported on in [4] (Lift and AMBA).

Kuvent et al. [21] presented the JVTS, a symbolic representation of CSs for GR(1). The JVTS is computed symbolically, without the expensive enumeration of concrete states, it is much smaller and simpler than its corresponding concrete CS, and it is annotated with invariants that explain how the CS forces the system to violate the specification. Using the JVTS for repair was suggested as future work in [21]. Our JVTS-based repair follows this suggestion.

Chatterjee et al. [6] presented algorithms for computing minimal assumptions that repair LTL specifications. The safety assumption generation of `GLASS` repair is inspired by their algorithm, but is adapted for the $\gamma$-sat specifications and is formulated and implemented symbolically. Unfortunately, first, the computation of minimal liveness assumptions of [6] is NP-hard, and second, even if provided with these assumptions, they cannot be expressed in GR(1).

Most recently, Cavezza et al. [5] presented a weakness measure for GR(1) formulas, which is based on the Hausdorff dimension, a concept that captures the notion of size of the omega-language satisfying an LTL formula. The measure provides a means to quantify the quality of a GR(1) specification, by measuring how permissive are its assumptions. As such, it may be useful in evaluating and selecting between suggested repairs. We leave the evaluation and selection between suggested repairs of our symbolic repairs, based on this measure and on other criteria (see Sect. X), to future work.

## X. Conclusion and Future Work

We presented two symbolic repair techniques for unrealizable GR(1) specifications. The first algorithm infers new assumptions based on the recently introduced JVTS. The second algorithm infers new assumptions directly from the specification. We further extended our work with repair core, and with support for specifications that have auxiliary variables.

We implemented our work, validated its correctness, and evaluated it on benchmarks from the literature, including 151 unrealizable specifications of autonomous Lego robots (SYNTECH15 [12]). The evaluation shows not only that our symbolic repairs are able to repair many specifications that could not be repaired by previous works, but also that their computation is significantly faster and scales better against growing number of variables. Both algorithms are sound. `GLASS` is also complete but generates a single repair. `JVTS-Repair` typically generates many suggested repairs.

We consider the following future research. The fast computation time allows our JVTS-based repair to effectively generate many rather than only one candidate repairs. This is a strength of the JVTS-based approach that is not available in `GLASS`, and creates an opportunity to select between or prioritize the different candidate repairs based on some criteria. Such criteria may include semantic criteria (e.g., implication or weakness [5]) and other criteria that may affect the readability and usefulness of the suggested repairs, e.g., size in terms of number of assumptions and number of variables used in them (arguably, an engineer will hesitate to use a repair she cannot understand). Above all, a suggested repair will not be used if it does not correctly characterize the behavior of the real environment in which the system should run, and thus the theoretically weakest or the smallest repairs may not be the best in practice. We leave this interesting investigation of multiple repairs presentation and criteria for selection and prioritization to future work.

The work is part of a larger project[7] 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., [24], [25]).


## Acknowledgements

We thank Yuval Moskovitch and Slava Novgorodov for implementing the algorithm of [1] in our synthesis environment. 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).


[7] 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] D. G. Cavezza and D. Alrajeh. Interpolation-based GR(1) assumptions refinement. In *TACAS*, volume 10205 of *LNCS*, pages 281–297, 2017.

[5] D. G. Cavezza, D. Alrajeh, and A. György. A weakness measure for GR(1) formulae. In *FM*, volume 10951 of *LNCS*, pages 110–128. Springer, 2018.

[6] K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In F. van Breugel and M. Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *LNCS*, pages 147–161. Springer, 2008.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] E. Firman, S. Maoz, and J. O. Ringert. Performance heuristics for GR(1) synthesis and related algorithms. In D. Fisman and S. Jacobs, editors, *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017.*, volume 260 of *EPTCS*, pages 62–80, 2017.

[13] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Trans. Software Eng.*, 45(1):34–67, 2019.

[14] D. Gritzner and J. Greenyer. Synthesizing executable PLC code for robots from scenario-based GR(1) specifications. In M. Seidl and S. Zschaler, editors, *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers*, volume 10748 of *LNCS*, pages 247–262. Springer, 2017.

[15] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Towards practical program repair with on-demand candidate generation. In *ICSE*, pages 12–23, 2018.

[23] W. Li, L. Dworkin, and S. A. Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*, pages 43–50, 2011.

[16] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (T). In *ASE*, pages 295–306, 2015.

[17] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811, 2013.

[18] 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.

[19] 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.

[20] O. Kupferman. Automata theory and model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 107–151. Springer, 2018.

[21] A. Kuvent, S. Maoz, and J. O. Ringert. A symbolic justice violations transition system for unrealizable GR(1) specifications. In *ESEC/FSE*, pages 362–372, 2017.

[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

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

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

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

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

[28] S. Mechtaev, M. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *ICSE*, pages 129–139, 2018.

[29] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE*, pages 448–458, 2015.

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

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

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

[33] Spectra Website. http://smlab.cs.tau.ac.il/syntech/spectra/.

[34] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE*, pages 416–426, 2017.

[35] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Software Eng.*, 43(1):34–55, 2017.

[36] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[37] Supporting Materials Website. http://smlab.cs.tau.ac.il/syntech/repair/.