Evolution-Aware Heuristics for GR(1) Realizability Checking

Dor Ma'ayan
Tel Aviv University
Israel

Shahar Maoz
Tel Aviv University
Israel

Jan Oliver Ringert
Bauhaus University Weimar
Germany

***Abstract*—Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. Despite significant research progress over the past few decades, reactive synthesis is still in its early stages of practical adoption. One significant barrier to using reactive synthesis outside academia is the long realizability checking and synthesis time of specifications.**

This paper introduces a novel, evolution-aware approach for realizability checking. Our approach leverages the key observation that realizability checking is an operation that developers frequently perform during iterative specification development; therefore, utilizing intermediate data from previous realizability checks can substantially improve running times. Our approach computes a local semantic diff between previous and current versions of the specification, and, based on the diff and the previous realizability checking result, applies a set of sound heuristics. These heuristics reuse intermediate data collected during the previous specification's realizability checking to accelerate the current specification's realizability checking. Our evaluation demonstrates that these heuristics are applicable in 70% of cases from a real-world dataset containing thousands of specifications, and that their application significantly improves the running time of realizability checking.

I. INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [30]. Rather than manually constructing an implementation of a reactive controller and using model checking to verify it against a specification, synthesis offers an approach where a correct implementation is automatically obtained for a given specification, if such an implementation exists. Given the promise of correct-by-construction systems, much research progress has been achieved over the last two decades on reactive synthesis theory, algorithms, tools, and applications [4], [6], [16], [20], [21].

Despite all this research progress, one of the remaining significant bottlenecks of reactive synthesis, even for fragments of LTL such as GR(1) [5], is the long synthesis and realizability checking time for large specifications. Previous studies show that as the specification grows, running times can be considerable [18], making reactive synthesis challenging for real-world cases. This finding is not surprising considering that the fixed-point algorithm for the realizability checking of GR(1) specifications can be solved in time of $O(N^3)$ where N is the size of the state space of the specification.

A key observation from previous studies [18], [19], [32] is that specifications should be developed in an incremental, iterative process, involving frequent realizability checking for synthesizing intermediate versions of the specification and for conducting various analyses that involve realizability checks. This observation raises the question of **whether using data retrieved from the previously analyzed version of the specification can lead to performance gains in the running time of realizability checking for the current version**.

This paper proposes novel, sound evolution-aware heuristics for faster realizability checking of GR(1) reactive synthesis specifications. The heuristics are based on the realizability checking results of the previous version of the specification, the diff between the previous version of the specification and its current version, and intermediate data that was retained during the realizability checking of the previous specification. It is important to note that our approach is sound and never gives a wrong answer. Nevertheless, we use the term heuristics because (1) our approach is not always applicable and (2) is not always better than the default approach.

An essential component for evolution-aware heuristics is the ability to compute the diff between two versions of a specification. Therefore, we introduce a novel, local semantic differencing algorithm that can detect strengthening and weakening at the granularity of assumptions and guarantees.

We have implemented and evaluated our heuristics on a large corpus of specifications that includes more than 3,000 pairs of specifications (v_1 and v_2), that were recorded in a real-world specification development process by users. The results show that our proposed heuristics are applicable in 70% of the cases and that in most of the cases, the heuristics improve realizability checking running times significantly.

In summary, this paper makes the following contributions:

- **Local Semantic Specification Diff.** We present an algorithm that computes a local semantic diff between two GR(1) specifications.
- **Evolution-Aware Heuristics.** We introduce novel evolution-aware heuristics for realizability checks of GR(1) specifications and implement them on top of Spectra. We show that the cases where the heuristics are applicable are prevalent and that they reduce realizability checking times significantly.
- **Reproducibility.** We make the implementation, dataset, evaluation scripts, and analysis available as an artifact.

II. Preliminaries and Related Work

We provide necessary background on GR(1) and Rabin(1) synthesis and on Spectra, with references to relevant papers for details and complete formal definitions. We also discuss related work on performance heuristics for GR(1) and on diff-based and incremental analyses.

A. GR(1)

LTL formulas can be used as specifications of reactive systems, representing a 2-player game between an environment player and a system player [5]. Atomic propositions in such games are interpreted as environment (input) variables $\mathcal{X}$ and system (output) variables $\mathcal{Y}$. An assignment to all variables is called a state, which can be represented as a tuple $\langle X, Y \rangle$, where $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$ are the assignments of environment and system variables resp. in that state. A transition between states occurs when the environment player assigns values to the input variables, which is then answered by the system player assigning values to the output variables. A play is a sequence of states, which can be infinite or finite, if it reaches a deadlock for one of the players, i.e., a state from which the player has no legal assignment to its variables.

Since LTL synthesis is computationally expensive (2EXPTIME-complete [30]), authors have suggested LTL fragments with more efficient synthesis algorithms. One such fragment is GR(1) [5], whose expressive power covers most of the well-known LTL specification patterns [8], [21]. GR(1) specifications include assumptions and guarantees about what needs to hold on all initial states, on all states and transitions (safety), and infinitely often on every run (justice). A GR(1) specification consists of the following elements [5]:

- $\mathcal{X}$ and $\mathcal{Y}$ are disjoint sets of input and output variables controlled by the environment and the system, resp.;
- θ^e is an assertion, i.e., a propositional logic formula, over $\mathcal{X}$ characterizing initial environment states;
- θ^s is an assertion over $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$ characterizing initial system states;
- ρ^e is an assertion over $\mathcal{V} \cup \mathcal{X}'$, with $\mathcal{X}'$ a primed copy of variables $\mathcal{X}$; given a state, ρ^e restricts the next input;
- ρ^s is an assertion over $\mathcal{V} \cup \mathcal{V}'$, with $\mathcal{V}'$ a primed copy of variables $\mathcal{V}$; given a state and input, ρ^s restricts the next output;
- $J^e_{i \in 1..n}$ is a set of assertions over $\mathcal{V}$ for the environment to satisfy infinitely often (called justice assumptions);
- $J^s_{j \in 1..m}$ is a set of assertions over $\mathcal{V}$ for the system to satisfy infinitely often (called justice guarantees).

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

$$\begin{aligned}\phi_{sr} =& (\theta_e \rightarrow \theta_s) \wedge (\theta_e \rightarrow G((H\rho_e) \rightarrow \rho_s)) \\ & \wedge (\theta_e \wedge G\rho_e \rightarrow (\bigwedge_{i=1}^{n} GFJ^e_i \rightarrow \bigwedge_{j=1}^{m} GFJ^s_j))\end{aligned} \quad (1)$$

GR(1) realizability can be checked using the 3-nested fixed-point algorithm shown in Alg. 1 from [5]. The algorithm utilizes μ calculus [15], by evaluating a formula with 3-nested fixed-point computations. These computations evaluate to the set of states from which the system can always eventually satisfy its justice guarantees, or from which the system can eventually always prevent the satisfaction of the environment's justice assumptions. It relies on the controlled-predecessor operator $\bigcirc\!\!\!\!\bigcirc S$ computing states from which the system ρ^s can force the environment ρ^e to states in S. The realizability check of a GR(1) specification computes the system winning states Z (Alg. 1) and checks whether for all initial environment states (θ^e) there exists an initial system state (θ^s) in Z.

Algorithm 1 GR(1) game algorithm [5] to compute system winning states Z

```
1:  Z = TRUE
2:  while not reached fixed-point of Z do
3:      for j = 1 to |J^s| do
4:          Y = FALSE
5:          while not reached fixed-point of Y do
6:              start = J^s_j ∧ ⦶Z ∨ ⦶Y
7:              Y = FALSE
8:              for i = 1 to |J^e| do
9:                  X = Z  //better approx. than TRUE, see [5]
10:                 while not reached fixed-point of X do
11:                     X = start ∨ (¬J^e_i ∧ ⦶X)
12:                 end while
13:                 Y = Y ∨ X
14:             end for
15:         end while
16:         Z = Y
17:     end for
18: end while
    return Z
```

B. Rabin(1) Game

Some of our heuristics require playing a Rabin game [31] instead of a GR(1) game. Rabin(1) game is the dual game for GR(1), as it computes the environment winning states instead of the system winning states; therefore, the realizability checking algorithm of Rabin(1) game is symmetric to Alg. 1. As such, it also includes 3-nested fixed-point computations. However, while the GR(1) algorithm is initialized with a system winning states set of **TRUE** (all the states), the Rabin(1) game realizability checking algorithm is initialized with an empty environment winning states set **FALSE**. A detailed algorithm can be found in [14], [25].

C. Spectra

We implemented our heuristics over Spectra[1]. Spectra [23] is a specification language and a synthesis environment that includes a GR(1) synthesizer. It extends GR(1)'s Boolean variables to finite-type variables, e.g., enumerations and bounded integers. Beyond GR(1) synthesis with several performance heuristics [10], [28], it includes language extensions that are reduced into GR(1), e.g., patterns [21] and triggers [2], as well as several analyses beyond realizability checking, e.g., well-separation detection [11], [22] and cores and repairs for unrealizable specifications [24], [27]. **Although our diff algorithm and heuristics are generic for every GR(1)**

[1]Spectra is available from https://github.com/spectrasynthesizer

specification language, we chose to implement them over Spectra since it is a mature reactive synthesizer and since it comes with a large corpus of specifications for evaluation. Since 2015, Spectra has been used by hundreds of final-year CS undergrads in semester-long project classes, specifying and executing robotic and other systems, resulting in the SYNTECH collection of specifications.

D. Performance Heuristics for GR(1)

Previous work has suggested performance heuristics for GR(1). Firman et al. [10] presented a list of heuristics to potentially accelerate GR(1) synthesis and related algorithms. The list includes several heuristics for controlled predecessor computation and BDDs (Binary Decision Diagrams), early detection of fixed-points and unrealizability, fixed-point recycling, and several heuristics for unrealizable core computations. These heuristics are implemented in Spectra.

Recently, Yatskan et al. [35] showed additional performance heuristics for GR(1), including discarding intermediate memory not required for many GR(1) algorithms, and improving the embedding of finite automata into GR(1) when supporting advanced language constructs like patterns and triggers.

Our work is very different than the above two, since our heuristics are evolution-aware. Our work complements these prior efforts, and does not aim to replace them.

E. Diff-based and Incremental Analyses

The formal methods literature includes work on incremental analyses. Most prominently, incremental SAT/SMT solving leverages previously learned information to efficiently solve a sequence of related SAT/SMT problems. Instead of solving each problem independently, an incremental solver retains state and reuses insights from earlier problem-solving stages. This approach has been applied to automated planning, lazy SMT solving, symbolic execution, and bounded model checking, and is supported by many solvers, see, e.g., [7], [9]. As an example, Beyer et al. [3] store and reuse abstraction precisions between consecutive model checking runs. To evaluate their approach, they created a dataset comprising industrial verification problems derived from 62 Linux kernel device drivers, with 1,119 revisions at the granularity of a Git commit.

Li et al. [17] proposed incremental analysis for consecutive Alloy models. They collected data on consecutive versions by instrumenting the Alloy analyzer. They exploit incremental SAT solving and the delta between two model versions. An interesting related work suggests incremental analysis of evolving Alloy Models [34]. Like our work, this work does not rely on the incremental analyses enabled by SAT or SMT solvers. The work statically analyses the impact of the change and uses memoization to enable solution reuse. It is evaluated on 24 mutated models and on 36 faulty models that have been repaired, demonstrating reductions in analysis times.

In contrast to the above, our work focuses on GR(1) realizability checking. Therefore, our heuristics address elements not present in SAT/SMT or symbolic model checking, i.e., temporal constraints of system and environment players, and GR(1)-specific semantics — captured in our local diff algorithm. Moreover, another novelty of our work is in relying on the granularity of tool interaction of specification development for detecting that a computation from scratch is not necessary for GR(1) realizability checking. We further provide an empirical evaluation based on thousands of GR(1) specifications with real, fine-grained version data. To the best of our knowledge, no existing LTL or GR(1) synthesizer supports any form of incremental synthesis.

III. Problem Statement and Solution Overview

This section describes the problem we aim to solve and provides an overview of our approach.

Consider an original specification (denoted as v_1) and an evolution specification (denoted as v_2). The evolution specification is based on v_1, with some modifications made by the developer, which are represented by a diff. **Our goal is to compute the realizability of v_2 more efficiently than the traditional realizability checking algorithm by leveraging both the diff between the specifications and intermediate data collected during the realizability checking of v_1.**

To achieve this, we introduce a set of evolution-aware heuristics. These heuristics are based on three key elements: (1) the realizability checking result of v_1, (2) the diff between v_1 and v_2, and (3) intermediate data retrieved during the realizability checking of v_1.

Fig. 1 illustrates an overview of our approach. Initially, the developer performs a realizability check on v_1; we store both the result and relevant intermediate data from this computation on disk. Subsequently, when the developer performs a realizability checking on an evolution of v_1 with some modifications, we first compute the difference between v_1 and v_2. Based on this diff and the realizability checking result of v_1, identify and apply an appropriate heuristic for checking the realizability of v_2. This heuristic may either return a realizability checking result immediately, or utilize the intermediate data retrieved during the realizability checking of v_1. Finally, we store the realizability checking result of v_2 on the disk, together with relevant data for the realizability checking of the next evolution.

The remainder of this paper is organized as follows. Sect. IV presents how we compute a diff between two versions of the specification, Sect. V describes our heuristics, Sect. VI presents the evaluation, and Sect. VII concludes.

IV. Local Semantic Diff

Since our heuristics are based on the evolution of specifications over time, it is essential to properly define and compute the diff between two versions of the specifications. A simple diff might be syntactic, e.g., describing certain assumptions or guarantees that were added or removed. However, such syntactic diff has many limitations that make it not a good fit for our purpose. We therefore present a semantic diff.

The following subsection presents our definition of a semantic diff and an algorithm for computing it. Subsection IV-C presents an extension of our approach to handle language features that introduce auxiliary variables.

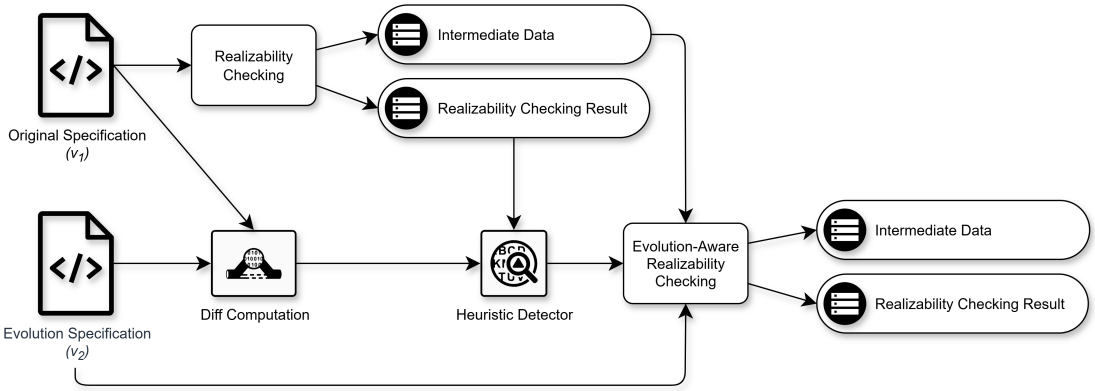


Fig. 1: Evolution-Aware Realizability Checking - Overview

A. Basic Local Semantic Diff

We define PlayerModule as the respective assertions $\langle \theta, \rho, J = \{J_n\}_{n=1}^{j} \rangle$ of the environment player, e.g., θ^e, and the system player, e.g., θ^s, as defined in each GR(1) specification (see II-A).

We define a semantic diff between two PlayerModules to be a vector of Boolean values $\langle \uparrow(\theta), \downarrow(\theta), \uparrow(\rho), \downarrow(\rho), \uparrow(J), \downarrow(J) \rangle$ that describes whether initial, safety, and justice assertions in the first PlayerModule were strengthened (denoted as $\uparrow()$) or weakened (denoted as $\downarrow()$) in the second PlayerModule.

Algorithm 2 computes a local semantic diff of two PlayerModules. It takes as input two PlayerModules, one from the original specification (denoted as v_1) and one from the evolution (denoted as v_2), and outputs the diff between them.

The algorithm checks whether there are constraints in initial and safety assertions of the v_1 PlayerModule that are not in the v_2 PlayerModule, and vice-versa (Alg. 2, lines 1-4). Then, the algorithm checks whether any justice assertion in v_1 is not implied by a justice assertion in v_2 (ll. 7-19), i.e., whether justice assertions were weakened. Notice that weakened could mean removal or that a modified, but not strengthened version exists in v_2 (not satisfying l. 10). Finally, it checks whether any justice assertion in v_2 has no justice assertion in v_1 that it implies (ll. 20-32), i.e., whether justice assertions were strengthened (added or modified and not weakened). Intuitively, for $J_{v_2} = J_{v_1}$ all justices are implied and the algorithm reports $\neg\downarrow(J)$ and $\neg\uparrow(J)$.

Note that in our definitions, the diff may indicate that initial assertions were both strengthened and weakened (and similarly for safety and justice assertions). Also, "not strengthened" means "weakened or not changed", and so "not strengthened" and "not weakened" together mean "not changed".

All the logical operations in the algorithm are performed symbolically using BDDs. Note that these include only basic logical operations and none that require quantification. Also note that the algorithm should run twice to compare two specifications: once to compare the v_1 and v_2 system PlayerModules, and once to compare their environment PlayerModules.

Algorithm 2 Computing GR(1) PlayerModule Diff

Input: $\langle \theta_{v_1}, \rho_{v_1}, J_{v_1} = \{J_n^{v_1}\}_{n=1}^{|J_{v_1}|} \rangle, \langle \theta_{v_2}, \rho_{v_2}, J_{v_2} = \{J_n^{v_2}\}_{n=1}^{|J_{v_2}|} \rangle$
Output: $\langle \uparrow(\theta), \downarrow(\theta), \uparrow(\rho), \downarrow(\rho), \uparrow(J), \downarrow(J) \rangle$

```
1:  ↑(θ) = (θ_{v1} ∧ ¬θ_{v2} ≠ FALSE)  // Initial assertion strengthened?
2:  ↓(θ) = (θ_{v2} ∧ ¬θ_{v1} ≠ FALSE)  // Initial assertion weakened?
3:  ↑(ρ) = (ρ_{v1} ∧ ¬ρ_{v2} ≠ FALSE)  // Safety assertion strengthened?
4:  ↓(ρ) = (ρ_{v2} ∧ ¬ρ_{v1} ≠ FALSE)  // Safety assertion weakened?
5:  ↑(J) = false
6:  ↓(J) = false
7:  for i = 1 to |J_{v1}| do
8:      implied = false
9:      for j = 1 to |J_{v2}| do
10:         if (J_j^{v2} → J_i^{v1}) = TRUE) then
11:             implied = true // satisfying J_j^{v2} satisfies J_i^{v1}
12:             break
13:         end if
14:     end for
15:     if ¬implied then // J_i^{v1} not covered by any justice assertion in v2
16:         ↓(J) = true // Justice assertions weakened
17:         break
18:     end if
19: end for
20: for i = 1 to |J_{v2}| do
21:     implied = false
22:     for j = 1 to |J_{v1}| do
23:         if (J_j^{v1} → J_i^{v2}) = TRUE then
24:             implied = true // satisfying J_j^{v1} satisfies J_i^{v2}
25:             break
26:         end if
27:     end for
28:     if ¬implied then // J_i^{v2} not covered by any justice assertion in v1
29:         ↑(J) = true // Justice assertions strengthened
30:         break
31:     end if
32: end for
```

B. Soundness, Locality, and Complexity

Alg. 2 is sound in the sense that it returns a correct diff for every input of two PlayerModules. However, our diff technique operates with inherent locality — it examines initial, safety, and justice assertions of a given PlayerModule independently, without considering the overall semantics of the specification. This locality has important implications for the algorithm's completeness: while it identifies all direct syntactic and single-element semantic relationships, it may not capture complex semantic implications that arise from interactions between multiple elements. For instance, a new justice assertion might

be implied not by any single original justice assertion but by the combined effect of two or more justice and some of the safety assertions (this would be a case of inherent vacuity as defined for GR(1) in [26]). Thus, the local nature of the algorithm may result in the identification of stricter diffs than what would be observed when analyzing the complete specification semantics. Importantly however, this strictness does not compromise the correctness of our heuristics (which we will present in Sect. V) but only may, in some cases, lead to scenarios where applicable heuristics go undetected.

While it is theoretically possible to compute a diff that captures the complete semantics of the entire specification (e.g., checking whether all system winning strategies for v_1 are also winning for v_2), this would be at least as hard as checking GR(1) realizability and require a different representation. Greimel et al. [12] presented a cubic algorithm (same number of fixed-points as GR(1) realizability, but a larger state space (union of variables)) for a global refinement relation. So while it might be possible to develop further heuristics, a complete handling without incurring prohibitive overhead is impossible. Indeed, under the common assumption that basic BDD logical operations take a constant time, the complexity of computing the diff using Alg. 2 is only $O(|J_{v_1}| * |J_{v_2}|)$ where $|J_{v_1}|$ and $|J_{v_2}|$ are the number of justice assertions in specifications v_1 and v_2, respectively.

C. Computing Local Semantic Diff between Specifications with Advanced Language Features

While Alg. 2 functions effectively for pure GR(1) specifications, specification languages such as Spectra incorporate advanced language features, including patterns and triggers, whose (automatic) translation into GR(1) introduces auxiliary variables. These variables are not explicitly defined by the specification developer but nevertheless significantly influence the overall specification semantics.

In order to correctly compare these language features at the BDD level in Alg. 2, we need to establish a mapping between auxiliary variables in v_1 and their corresponding variables in v_2, allowing us to replace auxiliary variables in v_1 with their v_2 counterparts in the assertions compared by Alg. 2. Our mapping approach examines the behavioral characteristics of each variable pair by comparing their associated BDDs across initial, safety, and justice assertions. When these behavioral signatures match, we replace corresponding BDD variables in all assertions of the PlayerModule $\langle \theta_{v_2}, \rho_{v_2}, J_{v_2} = \{J_n^{v_2}\}_{n=1}^{|J_{v_2}|} \rangle$ with those of specification v_1. This mapping technique enables our diff algorithm to handle advanced language features while maintaining its semantic analysis capabilities. The matching of auxiliary variables is correct as their values are completely and deterministically defined by exactly one language element (the one introducing that variable). Even if two different language elements translate to equivalent initial, safety, and justice assertions, these would be identified at the BDD level. Note that the BDD structure and ordering of variables are not an issue for the matching, as the BDDs of v_1 and v_2 are loaded into the same BDD engine instance.

V. Evolution-Aware Heuristics

This section presents our evolution-aware heuristics. Each heuristics gets as input whether v_1 was realizable or not, the winning states Z of v_1, and the local diff between v_1 and v_2 that was computed using the algorithm described in Sect. IV.

We present two categories of heuristics: *simple* and *advanced*. The simple heuristics (Sect. V-B) are immediate and do not require any fixed-point computation, while the advanced heuristics (Sect. V-C) reduce the fixed-point problem of GR(1) synthesis into a smaller one.

A. What Do We Keep on the Disk?

Our heuristics leverage data from prior realizability checks that are persisted to disk. For each realizability checking operation, we store the following essential data:

- **Realizability checking result.** We persist a Boolean value indicating whether the specification is realizable or unrealizable.
- **Copy of $\mathbf{v_1}$.** We maintain a complete textual representation of the original specification v_1. This enables us to load it during evolution-aware realizability checking and compute the semantic diff between v_1 and the evolution specification.
- **The value of Z for $\mathbf{v_1}$.** Z represents the set of winning states for the system (see Alg. 1) — specifically, states from which the system has a strategy to ensure all runs will satisfy the specification. Since Z is represented symbolically using a BDD, we serialize and store this BDD structure on disk for subsequent retrieval and analysis.

This persistent storage strategy allows our evolution-aware approach to access critical intermediate data from previous realizability checks without recomputing it.

B. Simple Heuristics

The simple heuristics do not require running the fixed-point algorithm and can immediately return whether the evolution specification v_2 is realizable based solely on the realizability checking result of v_1 and the diff between v_1 and v_2. The design rationale for these heuristics was guided by monotonicity of the GR(1) winning states computation based on strengthening and weakening of assumptions and guarantees.

Table I presents the simple heuristics cases. Each row describes a heuristics case; the columns describe whether the original specification was required to be realizable, the required diff between the original and evolution specifications, and the realizability result for such a case.

The first simple heuristics (S1) applies when there is no semantic difference between v_1 and v_2. It includes cases where the two specifications are syntactically the same and, more interestingly, cases where the specifications are not syntactically the same but based on our diff are semantically the same. In these cases, there is no need to perform realizability checking again, and the realizability of v_2 is defined to be the realizability of v_1.

TABLE I: Simple Heuristics — avoid running the GR(1) fixed-point algorithm

No.	v_1 Real?	Required Local Semantic Diff	v_2 Real?
S1	✗ ∨ ✓	$\neg(\uparrow(\theta^s) \vee \uparrow(\theta^e) \vee \downarrow(\theta^s) \vee \downarrow(\theta^e) \vee \uparrow(\rho^s) \vee \uparrow(\rho^e) \vee \downarrow(\rho^s) \vee \downarrow(\rho^e) \vee \uparrow(J^s) \vee \uparrow(J^e) \vee \downarrow(J^s) \vee \downarrow(J^e))$ No Diff	$= v_1$ Real
S2	✗	$\neg(\uparrow(\theta^e) \vee \downarrow(\theta^s) \vee \uparrow(\rho^e) \vee \downarrow(\rho^s) \vee \uparrow(J^e) \vee \downarrow(J^s))$ Gar strengthened and/or asm weakened	✗
S3	✓	$\neg(\downarrow(\theta^e) \vee \uparrow(\theta^s) \vee \downarrow(\rho^e) \vee \uparrow(\rho^s) \vee \downarrow(J^e) \vee \uparrow(J^s))$ Asm strengthened and/or gar weakened	✓

Theorem 1 (Correctness S1)**.** *Given specifications v_1 and v_2 where the output of Alg. 2 matches the required diff of heuristics S1, the result in Table I is correct.*

Proof. Alg. 2 determined for the equivalence of initial and safety assertions of v_1 and v_2 (Alg. 2, ll. 1-4) and found a complete pairwise matching of identical justice assertions (Alg. 2, l. 10 and l. 23). The result of Alg. 1 on v_2 must correspond to the result of Alg. 1 on v_1. □

The second simple heuristics (S2) applies when v_1 is unrealizable and v_2 only strengthens guarantees and/or weakens assumptions. In such cases, v_2 would be unrealizable since the relationship between the system and environment has not changed in a way that would affect its realizability.

Theorem 2 (Correctness S2)**.** *Given specifications v_1 and v_2 where the output of Alg. 2 matches the required diff of heuristics S2, the result in Table I is correct.*

Proof. Realizability requires that for all initial environment states (θ^e), there exists an initial system state (θ^s) in Z. Due to $\neg\uparrow(\theta^e)$, i.e., more or equal initial environment choices, and $\neg\downarrow(\theta^s)$, i.e., less or equal system choices, v_2 remains unrealizable for equal or smaller Z values computed by Alg. 1. Due to $\neg\uparrow(\rho^e)$ and $\neg\downarrow(\rho^s)$ the application of $\textcircled{\scriptsize\diamond}S$ to any S always yields a smaller or equal set of states in v_2 than in v_1, i.e., X (union in Alg. 1, l. 11) is smaller in v_2 making Y (union in Alg. 1, l. 13) and thus Z (Alg. 1, l. 16) smaller in v_2. Due to $\neg\downarrow(J^s)$, the loop in Alg. 1, ll. 3-17 may be repeated with an additional or a strengthened justice guarantee reducing the states $start$ in Alg. 1, l. 6 and thus Z. Due to $\neg\uparrow(J^e)$, the loop in Alg. 1, ll. 8-14 may be reduced, reducing the union of Xs in Alg. 1, l. 13 and thus Y and Z; or in case a justice assumption is weakened, its negation represents a smaller set of states in Alg. 1, l. 11 reducing also X, Y, and Z. □

The third simple heuristics (S3) is dual to the second and applies when v_1 is realizable and v_2 only weakens guarantees and/or strengthens assumptions. In such cases, v_2 would be realizable since the relationship between the system and environment has not changed in a way that would affect its realizability.

Theorem 3 (Correctness S3)**.** *Given specifications v_1 and v_2 where the output of Alg. 2 matches the required diff of heuristics S3, the result in Table I is correct.*

Proof. Analogous to the proof for S2 in Thm. 2. □

C. Advanced Heuristics

The goal of the advanced heuristics is to reduce the realizability checking problem to a simpler problem by reusing data retrieved from the realizability checking of v_1 as a starting point for the realizability checking of v_2. Since the advanced heuristics involve heavier computations compared to the simple heuristics, they should be used only when no simple heuristics is applicable.

Table II presents the more advanced heuristics. Each row represents a heuristics; the columns describe whether the original specification was required to be realizable, the required diff between the original and evolution specifications, and a brief description of the applied heuristics for such a case.

Again, our design rationale was to exploit the monotonicity of the winning state computation given strengthened and weakened assumptions and guarantees. The duality of GR(1) and Rabin(1) provided additional cases.

The first advanced heuristics (A1) applies when v_1 is realizable and where justice guarantees were not weakened, justice assumptions were not strengthened, safety guarantees were not weakened, and safety assumptions were not strengthened. In such a case, we initialize the greatest fixed-point Z in Alg. 1 with Z_{v_1} (i.e., the system's winning states for v_1) instead of **TRUE**, and return the realizability checking result of the algorithm. Intuitively, this is correct because when we have not weakened system constraints or strengthened environment constraints, the winning states of v_2 must be a subset of the winning states of v_1 (the size of Z being a greatest fixed-point is monotonically decreasing). By initializing the algorithm with Z_{v_1} instead of **TRUE**, we expect to reduce the number of fixed-point iterations required to reach convergence.

Theorem 4 (Correctness A1)**.** *Given specifications v_1 and v_2 where the output of Alg. 2 matches the required diff of heuristics A1, then the result of Alg. 1 for v_2 initialized with $Z = Z_{v_1}$ is identical to the result of Alg. 1 for v_2.*

Proof. The system winning states Z in Alg. 1 are determined by computing the greatest fixed-point Z via the least fixed-point Y and the greatest fixed-point X. We need to show that Z_{v_1} includes the winning states computed by Alg. 1 for v_2, i.e., that it is safe to start the fixed-point iteration from Z_{v_1} instead of **TRUE**. We do so by showing that the diff reduces

TABLE II: Advanced Heuristics — initialize GR(1)/Rabin(1) fixed-point algorithm from previous results

No.	v_1 Real?	Required Local Semantic Diff	Heuristics
A1	✓	$\neg\downarrow(J^s) \wedge \neg\uparrow(J^e) \wedge \neg\downarrow(\rho^s) \wedge \neg\uparrow(\rho^e)$ Justice gar not weakened and justice asm not strengthened and safety gar not weakened and safety asm not strengthened	Initialize the Z BDD of Alg. 1 with $\boldsymbol{Z_{v_1}}$ instead of **TRUE**
A2	✗	$\neg\uparrow(J^s) \wedge \neg\downarrow(J^e) \wedge \neg\uparrow(\rho^s) \wedge \neg\downarrow(\rho^e)$ Justice gar not strengthened and justice asm not weakened and safety gar not strengthened and safety asm not weakened	Initialize the Y BDD of a Rabin(1) Game with $\neg Z_{v_1}$, return negation of Rabin(1) result
A3	✓	$\neg(\uparrow(\theta^e) \vee \downarrow(\theta^s) \vee \uparrow(\rho^s) \vee \uparrow(\rho^e) \vee \downarrow(\rho^s) \vee \downarrow(\rho^e) \vee \uparrow(J^s) \vee \uparrow(J^e) \vee \downarrow(J^s) \vee \downarrow(J^e))$ All gar unchanged except ini strengthened (or equals) and All asm unchanged except ini weakened (or equals)	Check whether the system wins from all initial states in Z_{v_1}
A4	✗	$\neg(\downarrow(\theta^e) \vee \uparrow(\theta^s) \vee \uparrow(\rho^s) \vee \uparrow(\rho^e) \vee \downarrow(\rho^s) \vee \downarrow(\rho^e) \vee \uparrow(J^s) \vee \uparrow(J^e) \vee \downarrow(J^s) \vee \downarrow(J^e))$ All gar unchanged except ini weakened (or equals) and All asm unchanged except ini strengthened (or equals)	Check whether environment wins from all initial states in $\neg Z_{v_1}$

Z. The remaining arguments are those of the second part of the proof of Thm. 2. □

The second heuristics (A2) applies when v_1 is unrealizable and where justice guarantees were not strengthened, justice assumptions were not weakened, safety guarantees were not strengthened, and safety assumptions were not weakened. In such a case, we initialize the Y BDD of the second nested fixed-point computation in the Rabin(1) game with $\neg Z_{v_1}$ (i.e., the environment's winning states for v_1) and return the negation of the Rabin(1) game result. This works because when system constraints have not been strengthened and environment constraints have not been weakened, by starting the greatest fixed-point Y from $\neg Z_{v_1}$, we can more efficiently determine if the environment still maintains its winning states and potentially reduce the number of fixed-point iterations.

Theorem 5 (Correctness A2)**.** *Given specifications v_1 and v_2 where the output of Alg. 2 matches the required diff of heuristics A2, the result of Rabin(1) for v_2 initialized with $Y = \neg Z_{v_1}$ is identical to the result of Rabin(1) for v_2.*

Proof. Analogous to the proof of Thm. 4 for Rabin(1). □

The third heuristics (A3) applies when v_1 is realizable and all guarantees are unchanged except that initial guarantees may be strengthened and all assumptions are unchanged except that initial assumptions may be weakened. Here, rather than running the full fixed-point algorithm, we check whether the system wins from all initial states of v_2 in the winning states Z_{v_1}. This suffices as the modifications to the specification only affect the initial states, while the winning region computation yields the same result as for v_1. The fourth heuristics (A4) applies when v_1 is unrealizable and all guarantees are unchanged except that initial guarantees may be weakened, and all assumptions are unchanged except that initial assumptions may be strengthened. Dual to A3, we check whether the environment wins from all initial states of v_2 in the environment's winning states $\neg Z_{v_1}$. This suffices as the modifications to the specification only affect the initial states, while the winning region computation yields the same result as for v_1.

Theorem 6 (Correctness A3 and A4)**.** *Given specifications v_1 and v_2 where the output of Alg. 2 matches the required diff of heuristics A3 or A4, the result in Table II is correct.*

Proof. As established in the proof of Thm. 1, given the diff patterns for A3 and A4, Alg. 1 will compute the same values of Z for v_1 and v_2. The check for v_2, whether for all initial environment states (θ^e) there exists an initial system state (θ^s) in Z_{v_1} is thus sufficient for A3 (analogous for A4). □

Note that similar to how A1 and A2 are duals, so are A3 and A4. The distinction is important for the computation of A1 and A2 while it is insignificant for A3 and A4 (both checks are simple BDD operations). Also note that the restrictions of initial assertions in A3 and A4 are not necessary for their correctness while apparently limiting their completeness. However, the "missing" cases of A3 are captured by S3 and those of A4 are captured by S2.

D. Incompleteness

One source of incompleteness of our heuristics is the local semantic diff (see discussion and example of locality in Sect. IV-B). Due to this locality, the diff may fail to detect an opportunity for applying a heuristics, resulting in no heuristics being used although one could have been used, or a less effective heuristics being chosen (e.g., selecting an *advanced* heuristics instead of a *simple* one). Yet, this does not affect correctness — it only impacts potential performance gains.

Another source of incompleteness is the coverage of cases S1-S3 and A1-A4. As an example, no heuristics match a diff of specifications where a justice guarantee is added ($\uparrow(J^s)$) and a justice assumption is added ($\uparrow(J^e)$). That said, the heuristics never produce incorrect results.

VI. Evaluation

We describe an evaluation of our heuristics. We first present the research questions, then describe our evaluation dataset and

validation technique, and then present the results. Finally, we present threats to the validity of our evaluation.

A. Research Questions

We explore the following research questions:

- **RQ1: Applicability of the heuristics.** What is the prevalence of the heuristics' applicability in a representative specification corpus?
- **RQ2: Performance gain.** What is the performance gain of the heuristics?
- **RQ3: Overheads.** What are the overheads of computing the heuristics?

B. Specification Corpus, Implementation, and Validation

Metric	Patrolling		Deliveries		Esc. Guard		Cat. Int.		All
	MD	SD	MD	SD	MD	SD	MD	SD	MD
#Lines	84.4	22.5	76.5	21.8	59.7	25.8	71.1	17.6	72.9
#Asm	16.6	5.9	8.9	3.4	6.9	3.8	9.8	4.6	10.6
#Gar	11.2	5.3	19.6	8.7	12.4	4.7	13.9	6.1	14.3
#Sys-V	1.9	0.3	5.1	2.3	1.9	0.7	4.6	1.2	3.4
#Env-V	9.1	3.4	4.2	1.5	1.8	1.1	2.4	0.5	4.4

TABLE III: Reporting median (MD) and standard deviation (SD) size measurements of the final specifications. The number of lines excludes comment lines; all the data was computed syntactically.

1) Corpus: To evaluate our approach, we used the SPECVER25 specification benchmark, a corpus of specifications developed by 11 teams of third-year computer science undergraduates as part of a semester-long software engineering project course [19]. The dataset includes 3,100 version pairs, representing a complete set of snapshots taken during the development of reactive system specifications by users. 59.16% of the snapshots were realizable specifications, 40% were unrealizable, and 0.84% of the snapshots resulted in a timeout of 10 minutes while trying to compute their realizability.

SPECVER25 is based on four different development tasks; in total, it includes 44 different projects. The tasks are relatively complex and include several interdependent features, combining non-trivial interplay between the system and the environment and requiring the use of both safety and justice assertions. The tasks are inspired, in part, by common specification examples from the literature [1], [29]. Table III reports the median and standard deviation of the number of lines, the number of assumptions, the number of guarantees, the number of system variables, and the number of environment variables of the final 44 specifications. The snapshots we collected showed high diversity across the 44 projects, reflecting the varied ways teams approached their tasks and engaged with the tool. The distribution ranged from 25.5 snapshots at the 25th percentile (Q1), to 56.0 at the median (Q2), and 103.5 at the 75th percentile (Q3).

The following paragraphs include a short description of the development tasks:

Patrolling A robot should patrol between random target locations on a grid under several constraints about its movement. For example, each target location may be blocked for a finite number of steps and then eventually unblocked; when a target is blocked, the robot should not visit it.

Deliveries Two robots should use an elevator to travel between floors and turn off randomly located lights in some floors. Only one robot can use the elevator at a time, and the robots should never collide with each other. When a robot turns off a light, a new light turns on in a different location.

Escaping Guard A robot should travel between the corners of a grid with obstacles while ensuring that a guard that chases it never catches it. While the guard has a size advantage compared to the robot, the robot has the advantage to move two steps for each step of the guard.

Catching Intruder Two robots should collaborate to catch an intruder that tries to escape from them. The intruder enters the grid from a random cell in the bottom row, and the robots should block it together so that it cannot move. When the intruder is blocked by the robots (forced to place) for 4 steps, it disappears, and then a new intruder appears in a random cell in the bottom row.

Although these specifications were developed by students, their use does not compromise the validity of our evaluation. First, the specifications are non-trivial: participants developed the specifications in teams of two over multiple weeks and were required to develop correct and realizable specifications for complex systems. Second, our heuristics address diffs in specification evolution and realizability behavior, which arise regardless of developers' level of expertise. Thus, we consider this corpus sufficiently representative for our purposes.

SPECVER25 includes intermediate versions of specifications collected during the development process. It was created by adding a logging mechanism to the Spectra IDE, which tracked all user interactions with the synthesizer. These logs capture a fine-grained history of each specification's evolution, including realizability checks and other analyses that require solving the underlying synthesis game. We constructed version pairs by matching consecutive specification files according to their timestamps, as recorded by the Spectra logging mechanism, within the same project. **This makes SPECVER25 unique among other available datasets, as it enables the reconstruction of the exact sequence of changes made to each specification over time.**

Notice that we reached 3,100 unique pairs of specifications after we filtered out version pairs that were binary identical, which accounted for a non-negligible portion of the data. If these had been included, the applicability rate of heuristic S1 would have been significantly higher. By focusing on meaningful changes, we ensure that our evaluation is conservative.

Finally, note that we could not have used any other collection of GR(1) specifications for our purposes. Previous SYNTECH benchmarks (see [18], [23]) and the SYNTCOMP benchmarks (see [13]) could not be used because they include only a single version of each specification or sometimes a few versions of a specification but not at a fine-grained, true granularity recorded from a development process.

2) Implementation: We have implemented our local semantic diff algorithm and heuristics on top of the Spectra GR(1) synthesizer [23], with a modified version of the CUDD BDD library, implemented in C [33]. The heuristics were

implemented in both Java and C and integrated into Spectra's existing analysis and synthesis pipeline.

To enable the reuse of intermediate data from one realizability checking to another, we persist BDD structures on the disk. Specifically, we serialize BDDs in a compact binary format using CUDD's storing mechanism. This allows us to reload and initialize the algorithm with prior results (Z_{v_1} or $\neg Z_{v_1}$) without recomputation.

3) Validation: To validate the correctness of our implementation of the heuristics, we conducted extensive testing. Specifically, we ran over all the pairs of specifications available in SPECVER25. We compared the results between the public version of Spectra (i.e., without our heuristics), and our evolution-aware heuristics, to check that they agree.

C. Experiment Setup and Reporting

We ran all experiments on an AWS t3.2xlarge instance (representing an ordinary laptop with up to 3.1GHz CPU and 32GB RAM) with Windows 10 OS, Java 11, and CUDD 3, using one CPU core. Times we use are average values of 3 runs, measured in milliseconds. Although the algorithms we deal with are deterministic, we performed 3 runs as JVM garbage collection and the CUDD implementation of dynamic reordering add variance to running times. We used a 10 min timeout for each realizability checking run.

D. Results

1) RQ1: Prevalence of the heuristics: Table IV reports the prevalence of the heuristics' applicability in the SPECVER25 dataset. **We found that our heuristics are frequent in a real-world dataset, and applicable in nearly 70% of the cases**. The results show that 24.25% of the cases in the dataset fall under the "simple" heuristics category, meaning that it is possible to determine whether the evolution specification is realizable or not without any fixed-point computation. In addition, 44.96% of the cases in the dataset fall under the "advanced" category, meaning that it is possible to leverage the intermediate data from the realizability checking of the original specification as a starting point for the computation of the evolution realizability checking.

The last two columns in Table IV show the first (Q1) and third (Q3) quartiles of heuristic prevalence across projects. These quartiles indicate the middle 50% of projects. For instance, for A1, half of the projects have prevalence between 18.7% and 32.2%, while for S1 the range is 10.9%–18.6%. Overall, the quartile ranges are fairly tight, which suggests that the heuristics are consistently useful across projects rather than being dominated by just one or two cases.

As mentioned in Sect. IV-C, we applied a special treatment for cases where the specifications include auxiliary variables implicitly added by advanced language features. We found that 56.30% of the specifications in our dataset included auxiliary variables, indicating that our handling of auxiliary variables in the local semantic diff contributes significantly to the prevalence of the heuristics' applicability.

TABLE IV: Heuristics Application Statistics

Heuristics	Count	%	Q1/Project (%)	Q3/Project (%)
None	959	30.94	21.54	38.36
A1	740	23.87	18.68	32.15
A2	560	18.06	12.57	22.39
S1	444	14.32	10.94	18.58
S3	214	6.90	4.65	11.11
S2	95	3.06	1.96	4.64
A3	45	1.45	1.35	3.82
A4	43	1.39	1.50	3.08
Total	3100	100.00		

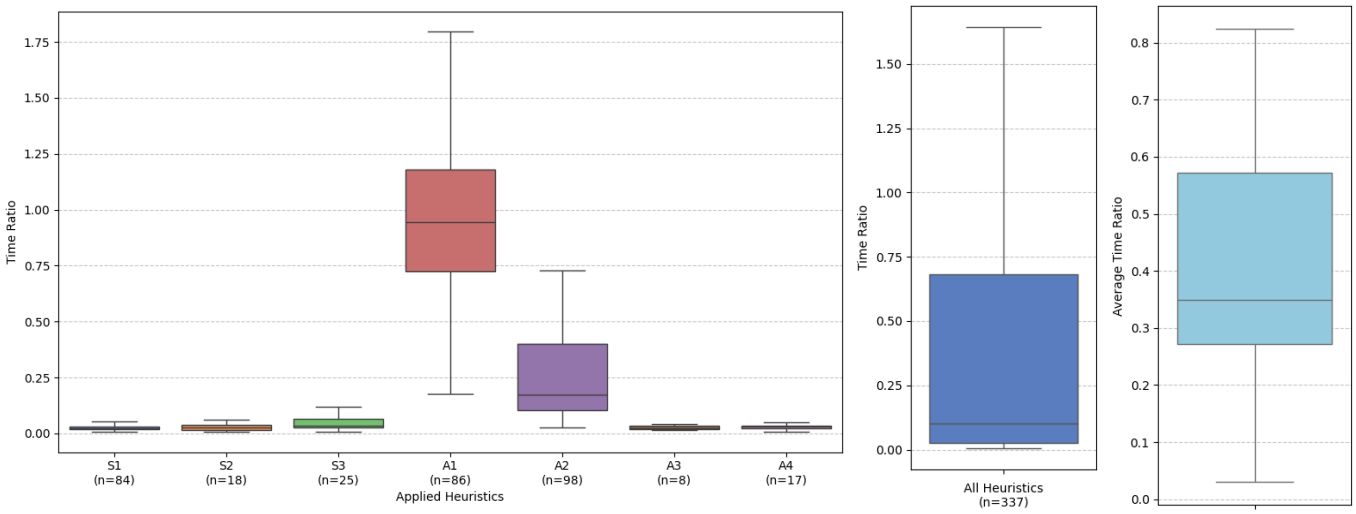
2) RQ2: Performance gain of the heuristics: We report on the performance gain of the heuristics through several measures: the overall effect on absolute running times, the ratio between the running time of the evolution-aware realizability checking and the baseline realizability checking for the same specifications (when considering the slowest cases, where speed-up is more critical), and the effect on the total number of timeouts.

Our analysis shows that the heuristics significantly improve the **absolute running time** of realizability checking. **While the average computation time without heuristics was 34.6 sec. (median of 4.03 sec.)**, the average computation time using the heuristics was **23.4 sec. (median of 3.55 sec.)**.

As another measure of performance gain, we calculate the **ratio between the running time of the evolution-aware realizability checking and the baseline realizability checking, for the same specification**. For instance, a time ratio of 0.5 indicates that the heuristic-based approach is twice as fast. Since speed-up is more critical when the running time is high, we focused on cases where the baseline realizability check took over 60 seconds. Fig. 2 (left) shows box-plots of the time ratios for these cases, grouped by the specific heuristics applied. The n values reflect the number of such cases for each heuristics (a 30 seconds threshold shows similar trends).

As can be seen, the heuristics are effective across the board, with varying degrees of performance gains. S1, S2, and S3 achieved geometric mean time ratios of 0.0251, 0.0241, and 0.0384 resp. (with medians 0.0217, 0.0269, and 0.0352), making them consistently over an order of magnitude faster than baseline realizability checking. A Wilcoxon signed-rank test for S1, S2, and S3 confirmed that their speedup is statistically significant with ($p < 0.0001$) and Cohen's effect-size $d > 1$. The advanced heuristics A3 and A4 show similar improvements, with geometric means of 0.0238 and 0.0282. A2 is also beneficial (geometric mean of 0.2111), though to a lesser extent - a factor of about 5, since A2 also involves fixed-point computation as part of the Rabin(1) game realizability checking. A Wilcoxon signed-rank test for A2, A3, and A4 confirmed that the speedup of these heuristics is also statistically significant with ($p < 0.0001$, except $p < 0.005$ for A3) and Cohen's effect-size $d > 1$.

A1 showed more modest gains: its geometric mean time ratio was 0.8880, and its median was 0.9462, meaning it improved running time in many cases, but not drastically: on average, A1 reduced realizability checking time by about 11%,



(a) Grouped by the applied heuristics

(b) All cases

(c) Per-project

Fig. 2: Box-plots of the time ratio between evolution-aware running time and baseline running time for cases where baseline running time was above 60 seconds. Outliers are excluded from the plots for readability.

and in over half of the cases, it provided a speed-up of 5% or more. A Wilcoxon signed-rank test showed that the speedup of A1 is not statistically significant. We investigated the reason behind A1's lower performance and found a modest correlation between higher time ratios and diffs that involved $\downarrow(\theta^e)$ and $\downarrow(\rho^e)$. We found out that 26.74% of the A1 cases involved either of these cases and when filtering them out, the geometric mean time ratio dropped to 0.8154 (with median of 0.8811).

To further explore the reasons for the relatively modest performance of A1, we looked at the number of fixed-point iterations (in A1 and A2), over the entire dataset. While in 856 cases (66.46%) there was a reduction in the number of fixed-point iterations for A1 and A2, in only 38 cases (2.95%) there was an increase in the number of fixed-point iterations, and the increases where only in A1, not in A2. This data partly explains the performance results we observed. We further analyzed the distribution of the slow A1 cases (Time Ratio > 1, with runs longer than 60 seconds) across projects. Such cases occurred in several teams and tasks, though with varying frequencies: for example, Team 2 had 2 of 3 Deliveries and 3 of 4 Patrolling cases that were slower, while Team 6 had 5 of 8 Patrolling cases. Other teams showed fewer occurrences, such as Team 4 Patrolling (6 of 16) and Team 8 Catching Intruder (2 of 8). Notably, one of the four tasks (Escaping Guard) had no slow instances at all. A complete breakdown is included in the artifact. Overall, these observations suggest that the effect is not systematic, but may depend on specific teams or patterns of changes.

Despite the more modest performance gains of A1 compared to the other heuristics, as we show in Fig. 2 (middle), **across all instances where realizability checking took more than 60 seconds and a heuristics was applied, the geometric mean time ratio was 0.1212, that is, an average speed-up factor of about 8, indicating a substantial improvement when using the heuristics.**

Fig. 2 (right) shows the average performance gain per project across all instances where realizability checking took more than 60 seconds. The median ratio is around 0.35, with most projects falling between 0.25 and 0.6, indicating consistent performance improvements between different projects.

Finally, the performance gain of the heuristics is also visible in the **total number of timeouts**. In the entire dataset, 28 cases of baseline realizability checking reached timeout and there was a relevant heuristics to apply. Out of these, 19 cases (73.08%) didn't reach the timeout using the heuristics. Moreover, in the entire dataset we had only 2 cases where baseline realizability checking didn't reach the timeout but the evolution-aware heuristics did.

3) RQ3: Overhead of the heuristics: While our evolution-aware heuristics offer significant performance benefits for the overall realizability checking time, they also, naturally, introduce certain overheads that should be considered. These overheads include: storing the intermediate data on the disk, and in particular, storing the Z BDD on the disk, loading the Z BDD from the disk, loading the previous game model from the disk, and computing the local semantic diff between v_1 and v_2. In order to measure these overheads, we computed and logged the running times of each of these operations and analyzed the overheads they introduce over the entire dataset. As we will detail next, **our findings show that the use of the heuristics introduces a minimal, negligible overhead to the overall performance of the realizability checking.**

The loading of the previous Z BDD from the disk is extremely lightweight, with an average of 10.63 ms, a median of 3.00 ms, and **95% of the cases taking under 33.93 ms**.

Loading the previous game model, which involves parsing the earlier version of the specification and rebuilding its game model again, has an average time of 2620.56 ms and a median of 2135.50 ms. **95% of the executions complete within 3120.47 ms**, indicating a consistent and bounded overhead.

The local semantic game comparison is also efficient in practice: it has an average running time of 272.29 ms, a median of 19.33 ms, and **95% of comparisons finish in under 496.10 ms**. This low overhead is expected since the local semantic diff algorithm does not involve fixed-point computations nor quantifications, and consists of a constant number of Boolean operations, plus $\mathcal{O}(n^2)$ additional Boolean operations, where n is the number of justice assertions. Since n is typically small, the complexity remains low, and the heuristics remain scalable and fast even as specifications evolve and grow.

Finally, storing the Z BDD introduces the highest overhead among the measured steps, with an average of 2863.46 ms and a median of 85.33 ms. **As such, 95% of the cases complete in under 15172.40 ms**. Importantly, this operation executes only after the realizability checking has been completed and its result has been presented to the developer, as it is used only to persist intermediate data for potential reuse. Therefore, it does not interfere with the speed at which the checking result is produced, and does not have any influence on the user experience of the specification developer.

In summary, the heuristics' overhead is small across all measured steps. Most operations complete within a few hundred milliseconds, and even in the rare more expensive cases, 95% of overheads out of more than 3,000 samples remain low. This confirms that the heuristics are suitable for practical use without introducing noticeable delays.

E. Threats to Validity

We discuss threats to the validity of our results. First, the implementation of the diff algorithm, the heuristic detector, and the heuristics as part of the realizability checking algorithm is nontrivial, and may have bugs. To mitigate this, we performed validation of our implementation using unit tests for each component. In addition, the design of our evaluation also validates our implementation, as for each pair of specifications, we ensure that the realizability checks of the baseline and evolution-aware algorithms agree. See Sect. VI-B3.

Second, although our algorithms are deterministic, the garbage collection of the JVM and CUDD introduces variance to the running times. To mitigate this, we repeated each experiment 3 times, and we report average values.

Third, although the results were obtained from a large corpus with more than 3,000 unique pairs of specifications, it is possible that different results could be obtained when running on a different dataset. In particular, our dataset comprises four robotic systems developed by 11 student teams each, which may limit the diversity of domains, complexity, and development practices represented. While the specifications were created iteratively by real users in realistic settings, results may not fully generalize to industrial projects, other domains, or more complex specifications. We partly mitigated this by analyzing results per project and observing consistent trends across all four tasks. Further, since our local semantic diff operates at the granularity of small changes, the applicability of heuristics depends more on the structure of edits than on the overall system domain or project size, partially reducing this threat. Still, further studies on industrial or heterogeneous datasets are needed to strengthen external validity.

VII. Conclusion, Implications, and Future Work

We introduced evolution-aware heuristics to enhance the performance of GR(1) realizability checking. Our heuristics are based on the observation that the development process of specifications for reactive synthesis involves frequent realizability checks, and that intermediate data from previous realizability checks can be used to make realizability checking faster. Our evaluation shows that the applicability of such heuristics is high and that they have the potential to significantly improve the running times of realizability checking.

The heuristics target the developer feedback loop in the synthesis environment. Our results show that many iterative edits yield diffs that can enable faster realizability checks and, as a result, more responsive tooling. This improvement is not only measurable but meaningful: it supports new interaction techniques, such as a continuous "realizability monitoring" that provides continuous realizability status to the developer as the specification evolves, or a real-time realizability debugger.

We consider the following future work directions. First, developing evolution-aware heuristics for other analyses, e.g., inherent vacuity [26], non-well-separation [11], [22], and the computation of cores [27]. For example, if a certain assumption or guarantee is found to be inherently vacuous, and the diff consists of its removal from the specification, the realizability of the new specification is identical to that of the original specification. As another example, if the original specification is realizable, the diff consists of the addition of some guarantees, and the new specification is unrealizable, computing an unrealizable core for the new specification can be accelerated by a heuristics that exploits the fact that a core must include at least one of the newly added guarantees (avoid checking subsets of guarantees that are disjoint to the diff). All these create further opportunities to use an evolution-aware heuristics and significantly improve developers' experience.

Another future direction is to combine our semantic diff with a syntactic one, to provide user friendly insights about the diff between versions of specifications and use them, e.g., to suggest commit messages or assist in specification review.

Artifact availability

The artifact includes (1) the implementation of our heuristics, (2) the dataset used for evaluation, (3) scripts to reproduce the performance measurements of the heuristics, (4) a description of the four development tasks, and (5) a Python notebook containing the statistical analysis and plots presented in Sect. VI. The artifact is publicly available at: https://doi.org/10.5281/zenodo.17069809.

Acknowledgments

This research was supported by the Israel Science Foundation (grant No. 1954/23, SPECTACKLE) as well as by Len Blavatnik and the Blavatnik Family Foundation. The authors thank the anonymous reviewers for helpful comments.

References

[1] J. Alonso-Mora, J. A. DeCastro, V. Raman, D. Rus, and H. Kress-Gazit. Reactive mission and motion planning with deadlock resolution avoiding dynamic obstacles. *Auton. Robots*, 42(4):801–824, 2018.

[2] G. Amram, D. Ma'ayan, S. Maoz, O. Pistiner, and J. O. Ringert. Triggers for reactive synthesis specifications. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023*, pages 729–741. IEEE, 2023.

[3] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 389–399. ACM, 2013.

[4] 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 *Computer Aided Verification*, pages 425–429. Springer Berlin Heidelberg, 2010.

[5] 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.

[6] D. G. Cavezza, D. Alrajeh, and A. György. Minimal assumptions refinement for realizable specifications. In K. Bae, D. Bianculli, S. Gnesi, and N. Plat, editors, *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020*, pages 66–76. ACM, 2020.

[7] L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 411–420, New York, NY, USA, 1999. Association for Computing Machinery.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Selected Revised Papers*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[10] E. Firman, S. Maoz, and J. O. Ringert. Performance heuristics for GR(1) synthesis and related algorithms. *Acta informatica*, 57(1):37—79, 2020.

[11] A. Gorenstein, S. Maoz, and J. O. Ringert. Kind controllers and fast heuristics for non-well-separated GR(1) specifications. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*, pages 28:1–28:12. ACM, 2024.

[12] K. Greimel, R. Bloem, B. Jobstmann, and M. Y. Vardi. Open implication. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *LNCS*, pages 361–372. Springer, 2008.

[13] S. Jacobs, G. A. Perez, R. Abraham, V. Bruyere, M. Cadilhac, M. Colange, C. Delfosse, T. van Dijk, A. Duret-Lutz, P. Faymonville, et al. The Reactive Synthesis Competition (SYNTCOMP): 2018-2021. *arXiv preprint arXiv:2206.00251*, 2022.

[14] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *International journal on software tools for technology transfer*, 15(5):563–583, 2013.

[15] D. Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.

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

[17] X. Li, D. Shannon, J. Walker, S. Khurshid, and D. Marinov. Analyzing the uses of a software modeling tool. In *Proceedings of the Sixth Workshop on Language Descriptions, Tools, and Applications, LDTA@ETAPS 2006*, volume 164 of *Electronic Notes in Theoretical Computer Science*, pages 3–18. Elsevier, 2006.

[18] D. Ma'ayan and S. Maoz. Using reactive synthesis: An end-to-end exploratory case study. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 742–754. IEEE, 2023.

[19] D. Ma'ayan, S. Maoz, and J. O. Ringert. Exploring development methods for reactive synthesis specifications. *ACM Trans. Softw. Eng. Methodol.*, Sept. 2025.

[20] S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit. Reactive high-level behavior synthesis for an atlas humanoid robot. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4192–4199, 2016.

[21] S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 96–106. ACM, 2015.

[22] S. Maoz and J. O. Ringert. On well-separation of GR(1) specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 362–372, New York, NY, USA, 2016. Association for Computing Machinery.

[23] S. Maoz and J. O. Ringert. Spectra: a specification language for reactive systems. *Softw. Syst. Model.*, 20(5):1553–1586, 2021.

[24] S. Maoz, J. O. Ringert, and R. Shalom. Symbolic repairs for GR(1) specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1016–1026, 2019.

[25] S. Maoz and Y. Sa'ar. Two-Way Traceability and Conflict Debugging for AspectLTL Programs. *LNCS Trans. Aspect Oriented Softw. Dev.*, 10:39–72, 2013.

[26] S. Maoz and R. Shalom. Inherent vacuity for GR(1) specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 99–110. ACM, 2020.

[27] S. Maoz and R. Shalom. Unrealizable cores for reactive systems specifications. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, page 25–36. IEEE Press, 2021.

[28] S. Maoz and I. Shevrin. Just-in-time reactive synthesis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, page 635–646. ACM, 2020.

[29] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10):2208–2224, 2021.

[30] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190. ACM, 1989.

[31] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[32] L. Ryzhyk and A. Walker. Developing a practical reactive synthesis tool: Experience and lessons learned. In R. Piskac and R. Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, volume 229 of *EPTCS*, pages 84–99, 2016.

[33] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.3. 0. *University of Colorado at Boulder*, 621, 1998.

[34] W. Wang, K. Wang, M. Gligoric, and S. Khurshid. Incremental Analysis of Evolving Alloy Models. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019*, volume 11427 of *LNCS*, pages 174–191. Springer, 2019.

[35] R. Yatskan, I. Shevrin, and S. Maoz. Performance heuristics for GR(1) realizability checking and related analyses. In *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025*, volume 15696 of *LNCS*, pages 40–59. Springer, 2025.