Unrealizable Cores for Reactive Systems Specifications

Shahar Maoz
Tel Aviv University
Tel Aviv, Israel

Rafi Shalom
Tel Aviv University
Tel Aviv, Israel

***Abstract*—One of the main challenges of reactive synthesis, an automated procedure to obtain a correct-by-construction reactive system, is to deal with unrealizable specifications. One means to deal with unrealizability, in the context of GR(1), an expressive assume-guarantee fragment of LTL that enables efficient synthesis, is the computation of an unrealizable core, which can be viewed as a fault-localization approach. Existing solutions, however, are computationally costly, are limited to computing a single core, and do not correctly support specifications with constructs beyond pure GR(1) elements.**

In this work we address these limitations. First, we present QuickCore, a novel algorithm that accelerates unrealizable core computations by relying on the monotonicity of unrealizability, on an incremental computation, and on additional properties of GR(1) specifications. Second, we present Punch, a novel algorithm to efficiently compute all unrealizable cores of a specification. Finally, we present means to correctly handle specifications that include higher-level constructs beyond pure GR(1) elements.

We implemented our ideas on top of Spectra, an open-source language and synthesis environment. Our evaluation over benchmarks from the literature shows that QuickCore is in most cases faster than previous algorithms, and that its relative advantage grows with scale. Moreover, we found that most specifications include more than one core, and that Punch finds all the cores significantly faster than a competing naive algorithm.

I. Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [26]. GR(1) is an assume-guarantee fragment of Linear Temporal Logic (LTL) that has an efficient symbolic synthesis algorithm [7]. GR(1) specifications include assumptions and guarantees that specify what should hold in all initial states, in all states and transitions (safeties), and infinitely often on every computation (justices). The expressive power of GR(1) covers many well-known LTL specification patterns [12], [19], and it has been recently applied in several domains, including robotics (see, e.g., [18]).

One of the main challenges of reactive synthesis in general and of GR(1) synthesis in particular is to deal with unrealizable specifications [2], [8], [11], [15], [16], [21]. To help engineers debug unrealizable specifications, several works have suggested the computation and use of an *unrealizable core*, a locally minimal subset of guarantees that is sufficient for unrealizability. Computing the core may be viewed as a fault-localization approach to unrealizability.

However, existing solutions to computing an unrealizable core suffer from three main limitations. First, core computation in existing solutions is costly, as it requires many invocations of realizability checking. Second, existing solutions are limited to finding a single core and thus provide only partial information about the realizability problems in the specification at hand. Third, existing solutions are limited to pure GR(1) specifications, and do not correctly handle specifications with richer language constructs. All these limit the applicability of core computations as an effective means to dealing with unrealizability.

In this work we address these three limitations. Our first contribution is `QuickCore`, a novel algorithm that accelerates unrealizable core computations. The effectiveness and correctness of `QuickCore` are based on four observations. First, that unrealizability is monotonic. Second, that core computations can be incremental. Third, that checking the realizability of specifications with fewer justices is typically significantly faster than of those with more justices. And fourth, that minimizing initial guarantees requires only one fixed-point computation.

Our second contribution is `Punch`, an algorithm to efficiently compute all unrealizable cores of a specification. Moreover, `Punch` computes the intersection of all cores without having to compute all of them. In particular, `Punch` is able to quickly check whether a core that was found is the only one that may be found.

Finally, our third contribution is the extension of core computations to correctly handle specifications that include, beyond pure GR(1) elements, higher-level constructs such as patterns, past LTL operators, monitors, and counters, as supported, e.g., in Spectra [20]. This is important in order to correctly apply core computations to these more compact and readable specifications, and to present the results not at the level of the internal representation but at the abstraction level used by the engineer who wrote the specification.

It is important to note that while `QuickCore` is specific to GR(1) unrealizability, `Punch` is in fact a generic algorithm for computing all cores. Indeed, the definition and correctness of `Punch` in Alg. 3 is independent of realizability checking or GR(1). Thus, `Punch` can be used, e.g., to compute all vacuity cores [23], all realizable cores (subsets of assumptions that suffice for realizability), and more generally, all locally minimal subsets for any monotonic criterion (e.g., unsatisfiability). Similarly, our third contribution on correctly dealing with higher-level constructs beyond pure GR(1) applies not

1558-1225/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00016

```
1  env boolean b1;
2  env boolean b2;
3  env boolean b3;
4
5  sys Int(1..3) f;
6
7  // No buttons are initially pressed
8  asm !b1 and !b2 and !b3;
9
10 // Request is removed when satisfied
11 asm G ( (b1 and f=1) -> next(!b1));
12 asm G ( (b2 and f=2) -> next(!b2));
13 asm G ( (b3 and f=3) -> next(!b3));
14
15 // Request must remain while unsatisfied
16 asm G ( (b1 and f!=1) -> next(b1));
17 asm G ( (b2 and f!=2) -> next(b2));
18 asm G ( (b3 and f!=3) -> next(b3));
19
20 // Lift is initially at lowest floor
21 gar f=1;
22
23 // Always stay at same floor or move to adjacent floor
24 gar G (f>=next(f)-1 and f<=next(f)+1);
25
26 // Do not move up when there are no requests
27 gar G (f<next(f)) ->(b1 or b2 or b3);
28
29 // Eventually grant each request
30 gar GF (b1 -> f=1);
31 gar GF (b2 -> f=2);
32 gar GF (b3 -> f=3);
33
34 // Visit every floor infinitely often
35 gar GF f=1;
36 gar GF f=2;
37 gar GF f=3;
```

Listing 1
LIFT CONTROLLER SPECIFICATION, ADOPTED FROM [23]

only to unrealizable cores, but, in principle, to any similar analysis of specifications with higher-level constructs that are reduced to GR(1). We consider this generality a nice advantage of our work, with possible future applications beyond the focus of the present paper.

We implemented our ideas on top of Spectra, a rich specification language and open source tool set for reactive synthesis [20]. All our algorithms are implemented on top of, and compared to, recently suggested heuristics for realizability and core computations for GR(1) [13]. We validated and evaluated our work on benchmarks from the literature. The evaluation shows that `QuickCore` is in almost all cases faster than previous algorithms and that many specifications indeed include more than one core. It further shows that `Punch` is much faster than a competing naive algorithm, and that it is able to compute almost all cores for the SYNTECH benchmarks in reasonable times.

Means to deal with unrealizability of temporal specifications have been studied in the literature. Beyond unrealizable cores [15], these also include different approaches to counter-strategy generation [16], [22] and assumption refinement or repair [2], [8], [21]. We discuss related work in Sect. IX.

II. Running Example

As a running example, we use a lift specification (see Lst. 1), taken from [23], which has appeared in several variants in previous GR(1)-related papers [2], [7], [8], [25]. The specification is written in Spectra format [20], [28]. It is small and simple, to fit the paper presentation. In our evaluation we have used larger and more complex specifications, taken from benchmarks.

The specification models a controller for a three floors lift. The lift has three request buttons, one on each floor. Requests are represented by environment variables `b1`, `b2`, and `b3`, which may be independently true or false. The current floor of the lift is represented by the system variable `f`. The environment is required to initially have no requests (l. 8), turn off any granted request at the next step (ll. 11-13), and keep ungranted requests (ll. 16-18). The system is required to start the lift on the first floor (l. 21), and to disallow the lift to move more than one floor at a time (l. 24). The system is also required not to move up when there are no requests (l. 27), to eventually grant every request (ll. 30-32), and to make sure every floor is visited infinitely often (ll. 35-37).

The specification is unrealizable but it is not easy to see why and debug it. So, the engineer may want to employ a fault-localization approach and find an unrealizable core, a locally minimal subset of guarantees that is sufficient for unrealizability. A modified specification with only these guarantees is already unrealizable; removing any one guarantee from that specification, will render it realizable.

By running our new algorithm `QuickCore` the engineer finds that the set of guarantees in lines {21,27,36} is a core.

However, an unrealizable core is not necessarily unique, i.e., the specification may induce additional unrealizable cores. Our new algorithm `Punch`, first finds the above core, then finds that the guarantee in line 27 is the intersection of all the cores, and finally reports all five remaining cores, namely {21,27,37}, {27,35,36}, {27,35,37}, {27,36,37}, and {24,27,30,37}. As our evaluation shows, having more than one core is indeed rather common.

Note that the early detection of the intersection of all the cores indicates whether additional cores exist. Moreover, the intersection is of interest, since making the specification realizable by removing or weakening only one guarantee is only possible with the guarantees in the intersection. Finally, the size of the intersection serves as a lower bound on the size of any of the cores induced by the specification.

III. Preliminaries

A. LTL, GR(1), and Realizability

We use a standard definition of *linear temporal logic (LTL)*, e.g., as found in [7], 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 $\mathcal{V}$. 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.

GR(1) is a fragment of LTL. A GR(1) specification contains 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.

We use the following abstract syntax definition of a GR(1) specification taken from [23].

Definition 1 (Abstract syntax of a specification)**.** *A GR(1) specification is a tuple* $Spec = \langle V_e, V_s, D, M_e, M_s \rangle$*, where* V_e *and* V_s *are sets of environment and system variables respectively,* $D : V_e \cup V_s \rightarrow Doms$ *assigns a finite domain to each variable*[1]*, and* M_e *and* M_s *are the environment and system modules. A module is a triplet* $M = \langle I, T, J \rangle$ *that contains sets of initial assertions* $I = \{I_n\}_{n=1}^{i}$*, safety assertions* $T = \{T_n\}_{n=1}^{t}$*, and justice assertions* $J = \{J_n\}_{n=1}^{j}$ *of the module, where* $i = |I|, t = |T|$ *and* $j = |J|$*. The set of elements of module* $M = \langle I, T, J \rangle$ *is* $B_M = I \cup \{\mathbf{G}\, T_i\}_{i=1}^{t} \cup \{\mathbf{GF}\, J_i\}_{i=1}^{j}$.

Given a set $\mathcal{Z}$ of variables, $\mathcal{Z}' = \{\mathbf{X}v | v \in \mathcal{Z}\}$ contains a copy of its variables at the next state. Let $M_e = \langle I_e, T_e, J_e \rangle$, $M_s = \langle I_s, T_s, J_s \rangle$, and $\mathcal{V} = V_e \cup V_s$. Then, the elements of I_e, T_e, J_e, I_s, T_s and J_s are propositional logic expressions over $V_e, \mathcal{V} \cup V_e', \mathcal{V}, \mathcal{V}, \mathcal{V} \cup \mathcal{V}'$ and $\mathcal{V}$ respectively.

GR(1) has efficient symbolic algorithms for realizability checking and controller synthesis, presented in [7], [25]. For this a game structure of a two-player game $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ is defined. The GR(1) game has a set of variables $\mathcal{V} = V_e \cup V_s$, environment and system variables ($\mathcal{X} = V_e$ and $\mathcal{Y} = V_s$ resp.), environment and system initial states ($\theta_e = \wedge_{d \in I_e} d$ and $\theta_s = \wedge_{d \in I_s} d$ resp.), environment and system transitions ($\rho_e = \wedge_{t \in T_e} t$ and $\rho_s = \wedge_{t \in T_s} t$ resp.), and a winning condition $\varphi = \bigwedge_{j \in J_e} \mathbf{GF} j \rightarrow \bigwedge_{j \in J_s} \mathbf{GF} j$.

A GR(1) specification is realizable, i.e., allows an implementation, iff the system wins the game. Roughly, this means that if the environment keeps all initial assumptions then the system should keep all initial guarantees, as long as the environment keeps all safety assumptions the system should keep all safety guarantees, and in all infinite plays, if the environment keeps all justice assumptions the system should keep all justice guarantees.

For this the algorithm of [7], [25] computes a winning region which is a set of winning states from which the system has a winning strategy. A winning strategy prescribes the outputs of a system for all possible environment choices that allows the system to win. The winning region is computed according to a fixed-point computation over transitions and justices alone. GR(1) realizability checks if for all initial environment choices the system can enter a winning state. GR(1) synthesis computes a winning strategy, if one exists.

B. Monotonic Criteria and Cores

Given a set E, and a monotonic criterion on subsets of E, a core is a local minimum that satisfies the criterion. Formally:

Definition 2 (Monotonic criterion)**.** *A Boolean criterion over subsets of* E *is monotonic iff for any two sets* A, B *such that* $A \subseteq B \subseteq E$*, if* A *satisfies the criterion then* B *satisfies the criterion.*

[1]The use of any finite domain rather than only Boolean variables is straightforward and supported by many tools, including Spectra.

Definition 3 (Core)**.** *Given a set* E *and a monotonic criterion over its subsets, a set* $C \subseteq E$ *is a core of* E *iff* C *satisfies the criterion, and all its proper subsets* $C' \subset C$ *do not satisfy the criterion.*

Unrealizability is monotonic w.r.t. subsets of guarantees, i.e., adding guarantees to an unrealizable specification keeps it unrealizable. Intuitively, this is so because adding guarantees strengthens the constraints on the system, and does not change the constraints on the environment. Formally:

Proposition 1 (Unrealizability is monotonic)**.** *Given two specifications,* $Spec_1 = \langle V_e, V_s, D, M_e, M_s^1 \rangle$*, and* $Spec_2 = \langle V_e, V_s, D, M_e, M_s^2 \rangle$*, such that* $B_{M_s^1} \subseteq B_{M_s^2}$*. Then, if* $Spec_1$ *is unrealizable,* $Spec_2$ *is also unrealizable.*

C. Existing Domain-Agnostic Minimization Algorithms

We recall three existing domain-agnostic minimization algorithms from the literature, namely delta debugging [29] (`DDMin`), `QuickXplain` [14], [24], and linear minimization, which we denote by `LinearMin`. All three algorithms find a core of a set E, given a monotonic criterion `check`.

Algorithm 1 The delta debugging algorithm `DDMin` from [29] as a recursive method that minimizes a set of elements E by partitioning it into n parts (initial value $n = 2$)

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

We show a pseudo-code for `DDMin` in Alg. 1. The inputs for the algorithm are a set E and the number n of parts of E to check. The algorithm starts with $n = 2$ and refines E and n in recursive calls according to different cases (ll. 3, 8, and 14). The computation starts by partitioning E into n subsets and evaluating `check` on each subset $part$ and its complement. If `check` holds (l. 2 or l. 7), the search is continued recursively on the subset $part$ or on its complement respectively. If `check` holds neither on any subset $part$ nor on the complements, the algorithm increases the granularity of the partitioning to $2n$ (l. 14) and recurs, or terminates when the granularity is not smaller than the size of E (l. 12). `DDMin` has quadratic worst-case complexity and logarithmic best-case complexity in terms of $|E|$.

`QuickXplain` is a recursive divide and conquer algorithm that minimizes each half, one after the other, in an incremental way (see Sect. III-D). It has a worst-case complexity of $O(k + klog(\frac{|E|}{k}))$, where k is the size of the largest core. To the best of our knowledge, `QuickXplain` was never previously

applied to computing unrealizable cores for reactive systems specifications.

Finally, `LinearMin` was originally suggested in [11], and compared with `DDMin` in [15]. `LinearMin` goes over elements of the input set one by one, and removes an element iff the criterion holds for the set without the element. An example of `LinearMin` can be found in ll. 11-16 of Alg. 2. The complexity of `LinearMin` is linear in $|E|$.

D. Minimization With a Base and Incremental Core Computation

We define a notion of minimization with a base as follows.

Definition 4 (Minimization with a base)**.** *Assume a set E of elements, two disjoint subsets $Base, A \subseteq E$, and a minimization algorithm `Alg` that detects cores according to a monotonic criterion `check`. Assume also that the set $Base \cup A$ satisfies the criterion. We denote by $MinWBase(Alg, E, Base, A, check)$ an algorithm that computes a locally minimal $A' \subseteq A$ s.t. $Base \cup A'$ satisfies the criterion, by applying `Alg` to A, and replacing every $\texttt{check}(X)$ operation with $\texttt{check}(Base \cup X)$.*

Monotonicity is ensured for minimization with a base because $A_1 \subseteq A_2$ implies $Base \cup A_1 \subseteq Base \cup A_2$.

Note that $Base$ does not have to be a subset of a core of E, and it may contain a core. $Base$ only has to satisfy $A \cap Base = \emptyset$, and that $Base \cup A$ satisfies the criterion. The following holds trivially.

Proposition 2. *Given the notations of Definition 4,*

(i) If $Base$ is a subset of all cores of E, then $Base \cup A'$ is a core of E.

(ii) If $Base$ contains a core of E, then $A' = \emptyset$.

Finally, we use the idea of incremental minimization in some of our algorithms. Lemma 1 states that incrementally minimizing two subsets that partition a set produces a core.

Lemma 1 (Incremental Core Computation)**.** *Let A and B be disjoint sets such that $E = A \cup B$ satisfies a monotonic criterion. Let A' be a locally minimal subset of A such that $A' \cup B$ satisfies the criterion, and B' be a locally minimal subset of B such that $A' \cup B'$ satisfies the criterion. Then $A' \cup B'$ is a core of E.*

Proof. By definition $A' \cup B'$ satisfies the criterion. A' exists because E satisfies the criterion, and B' exists because $A' \cup B$ satisfies the criterion.

To prove that $A' \cup B' \subseteq E$ is locally minimal, let $x \in A' \cup B'$. If $x \in A'$ then by definition of A', $(A' \setminus \{x\}) \cup B$ does not satisfy the criterion, thus by monotonicity $(A' \setminus \{x\}) \cup B'$ does not satisfy the criterion either. Otherwise $x \in B'$, and by definition of B', $A' \cup (B' \setminus \{x\})$ does not satisfy the criterion. □

IV. QUICKCORE

We are now ready to present our first contribution, the `QuickCore` algorithm, which aims to accelerate unrealizable core computations. The correctness and efficiency of `QuickCore` rely on the following observations. First, that unrealizability is monotonic (see Prop. 1). Second, that it is possible to compute a core incrementally (see Lemma 1). Third, that checking the realizability of specifications with fewer justices is typically much faster. Fourth, that removing justice assumptions from an unrealizable specification that has no justice guarantees preserves unrealizability and preserves winning regions. And finally, that minimizing the initial assertions requires only a single computation of the winning region of the system, plus a small constant number of symbolic operations.

A. `QuickCore` Algorithm Overview

Roughly, `QuickCore` begins by trying to remove as many justices as possible. Therefore, if no justice guarantees are required for a core it removes all of them, and all justice assumptions. Otherwise, it minimizes justice guarantees alone using `DDMin` with a base. Later, `QuickCore` minimizes safety guarantees using `DDMin` with a base. Finally, `QuickCore` minimizes all initial guarantees using `LinearMin`.

Algorithm 2 presents `QuickCore` in pseudo-code. `QuickCore` minimizes guarantees, group by group, according to their type; first justices, then safeties, and ending with initial assertions. It begins with a realizability check of the specification without justice guarantees in line 1. If the specification without these is realizable, at least one such justice guarantee is required for unrealizability, and so it minimizes justice guarantees alone (i.e., keeping all initial assertions and safeties) using `DDMin` with a base in line 2. Otherwise, it determines that the core it computes has no justice guarantees (line 4) and removes all environment justices in line 5 before it continues to look for system safeties and initial assertions.

With a specification that has a minimized set of justice guarantees and maybe no environment justices, `QuickCore` uses `DDMin` with a base in order to minimize the set of safeties alone (line 7). It keeps initial assertions and the minimized justice guarantees unchanged.

At the last stage of `QuickCore`, it computes the winning region of the system in line 8 for the specification with the minimized set of justices and safeties. It now uses linear minimization in the loop in lines 11-16, going over initial assertions one by one, and checking if the system wins without each. If so, we keep this initial assertion because we want to maintain unrealizability. Checking this is done with $SysWin(\theta_e, \theta_s, w)$, which determines if for every possible environment choice satisfying θ_e, there is a choice for the system from θ_s that is inside the winning region w. This is computed with a small, constant number of symbolic operations.

Remark 1. *Realizability checks in our implementation use the performance heuristics suggested in [13]. `QuickCore` must disable two of these heuristics, namely, fixed-point recycling and early detection of unrealizability, before line 8, in order to avoid an incomplete winning region computation. For example, the latter heuristics may over-approximate the winning region by halting the outer greatest fixed-point when the winning*

region becomes small enough to ensure that the system loses. This is good enough for realizability checks, so we use it safely before line 8, yet using it from that line on may wrongly keep some initial guarantees.

Example 1. *For the specification in Lst. 1 (see Sect. II),* `QuickCore` *finds that at least one justice is required for a core in line 1, and detects the justice in line 36 of the specification with the minimization in line 2. Given this justice, the safety in line 27 of the specification is detected in line 7. Lines 11-16 decide that given the above two guarantees, the only initial guarantee of the specification (line 21) is required for the core. The resulting core is thus the set of guarantees in lines {21,27,36}.*

Algorithm 2 QuickCore Given an unrealizable specification find a locally minimal subset of guarantees that keeps it unrealizable

Require: An unrealizable specification $S = \langle V_e, V_s, D, M_e, M_s \rangle$, where $M_e = \langle I_e, T_e, J_e \rangle$ and $M_s = \langle I_s, T_s, J_s \rangle$
Ensure: An unrealizability core of S
▷ Begin by minimizing only justice guarantees
1: **if Realizable**$(\langle V_e, V_s, D, M_e, \langle I_s, T_s, \emptyset \rangle \rangle)$ **then**
2: $J_c \leftarrow$ **MinWBase**$(\texttt{DDMin}, B_{M_s}, I_s \cup T_s, J_s, \neg$**Realizable**$(S))$
3: **else**
4: $J_c \leftarrow \emptyset$
5: $M_e \leftarrow \langle I_e, T_e, \emptyset \rangle$
6: **end if**
▷ Continue with minimizing only safety guarantees
7: $T_c \leftarrow$ **MinWBase**$(\texttt{DDMin}, B_{M_s}, I_s \cup J_c, T_s, \neg$**Realizable**$(S))$
▷ End with minimizing only initial guarantees
8: $w \leftarrow$ **ComputeWinRegion**$(\langle V_e, V_s, D, M_e, \langle I_s, T_c, J_c \rangle \rangle)$
9: $I_c \leftarrow I_s$
10: $envIni \leftarrow \wedge_{d \in I_e} d$
11: **for** $i \in I_c$ **do** ▷ Use linear minimization for initial guarantees
12: $I_c \leftarrow I_c \setminus \{i\}$
13: **if SysWin**$(envIni, \wedge_{d \in I_c} d, w)$ **then**
14: $I_c \leftarrow I_c \cup \{i\}$
15: **end if**
16: **end for**
17: return $I_c \cup T_c \cup J_c$

B. Correctness of `QuickCore`

`QuickCore` is correct in the sense that it detects an unrealizable core of the guarantees. To prove correctness we will show that (1) Finding cores of subsets of guarantees one by one yields a valid core; (2) Removing all justice assumptions from a specification with no justice guarantees impact neither the realizability of any subset of guarantees, nor the winning region of the system; and (3) It is enough to compute the winning region of the system once when minimizing initial guarantees.

(1) A core can be computed in a compositional manner, by minimizing pairwise disjoint subsets one by one, using an extension by induction of Lemma 1 to any number of finite pairwise disjoint sets. Specifically, `QuickCore` partitions the set of guarantees into three sets, and minimizes each set with established algorithms for local minimum, namely, `DDMin` and `LinearMin`.

(2) Removing all justice assumptions when no justice guarantees are needed for a core does not affect the overall correctness of `QuickCore`, because realizability checks and computation of winning regions of the system are unchanged.

Intuitively, in an unrealizable specification with no justice guarantees, the environment can (and must) win with *finite* plays of the game. Thus, even though generally removing assumptions may turn a realizable specification into an unrealizable one, this does not happen when removing justice assumptions from a specification with no justice guarantees. For such specifications, when the system does not lose finitely, it wins the infinite games, regardless of any justice assumptions. Accordingly, all environment justices in this case are unhelpful in the sense defined in [11], i.e., assumptions whose removal does not change the realizability induced by any subset of the guarantees. Note that having several assumptions, each unhelpful alone, does not mean that they are unhelpful as a set, yet justice assumptions in this case are unhelpful as a set.

Formally, the game structure of the GR(1) game [7] for both specifications is the same, because in GR(1) games, justices appear only in the winning condition $\bigwedge_{j \in J_e} \mathbf{GF} j \rightarrow \bigwedge_{j \in J_s} \mathbf{GF} j$. When there are no justice guarantees, this condition is $\bigwedge_{j \in J_e} \mathbf{GF} j \rightarrow \top$, which is an LTL tautology regardless of the justice assumptions. This ensures both the equirealizability, which is needed for the correctness of the minimization of safeties in line 7, and the correctness of the winning region computation in line 8.

(3) Finally, the correctness of the minimization of initial guarantees follows from the fact that checking realizability has two parts. We first compute the winning region of the system in the GR(1) game, and then check if the system can reach it given all possible environment initial choices. Since the winning region depends only on safeties and justices, and we keep them unchanged at this phase of `QuickCore`, it is enough to compute the winning region only once.

C. Complexity of `QuickCore`

Consider a specification with n guarantees. The worst-case complexity of `DDMin` is $O(n^2)$ realizability checks [29]. The same holds for `QuickCore`.

V. Punch

We now present our second contribution, the `Punch` algorithm for computing all cores.

`Punch` finds all cores of a set E according to a monotonic criterion. The algorithm is generic in the sense that it requires a `check` that evaluates the monotonic criterion, and a `computeCore`(E', B) that provides a core in $E' \subseteq E$, given E' that satisfies the criterion, and a set $B \subseteq E'$ that is a subset of all the cores in E'. In particular, `Punch` detects all minimal size cores. As a byproduct, `Punch` provides an efficient way to find the intersection of all the cores, without having to compute all of them. This intersection provides an early estimate of the size of the smallest cores, and, in particular, an early verdict on the existence of additional cores.

Algorithm 3 presents `Punch` in pseudo-code, as a recursive algorithm that takes two parameters as input, a set E that has at least one core, and a set K that is a subset of all the cores

Algorithm 3 **Punch** Find all cores according to a monotonic **check**

Require: A set E of elements such that **check**(E)=$\top$
Require: A set $K \subseteq E$ that is a subset of all cores in E
Ensure: All cores in E

1: $C_0 \leftarrow$ **computeCore**(E, K)
2: $AllCores \leftarrow \{C_0\}$
3: $CI \leftarrow \emptyset$
4: $Cont \leftarrow \emptyset$
5: **for** $x \in C_0 \setminus K$ **do**
6: **if check**$(E \setminus \{x\})$ **then**
7: add x to $Cont$
8: **else**
9: add x to CI
10: **end if**
11: **end for**
12: **for** $x \in Cont$ **do**
13: $AllCores \leftarrow AllCores \cup$ **Punch**$(E \setminus \{x\}, K \cup CI)$
14: **end for**
15: return $AllCores$

in E. The recursive algorithm finds all the cores in E. Thus, `Punch` $(E, \emptyset)$ returns all the cores in E.

The algorithm finds its first core $C_0 \subseteq E$ by applying `computeCore` in line 1. Later, in lines 5-11, it splits all the elements $x \in C_0 \setminus K$ into two sets according to whether $E \setminus \{x\}$ satisfies the criterion. $Cont$ gets all positives and CI gets all negatives. Finally, in lines 12-14, for all elements $x \in Cont$, it considers the punched sets $E \setminus \{x\}$ (hence the name `Punch`), and recursively looks for all cores inside them, while adding CI to K. By collecting all such cores into $AllCores$ the computation ends.

Example 2. *Applying* `Punch` *to the example in Lst. 1 finds the six cores of the run described in Sect. II. Specifically, that run is of* `PQC`*($B_{M_s}, \emptyset$) (see Sect. V-C).*

Two cores consist of the guarantees in lines 21, 27, and one of the guarantees in lines 36 and 37. Indeed, if the lift starts at the first floor, and moves up only when there are requests, it may never be able to reach the other two floors. Three other cores consist of the guarantee in line 27, and pairs of the guarantees in lines 35-37, which require the elevator to visit two different floors infinitely. Indeed, the system can be forced to visit the lower of the two floors, and not to go up (line 27). Finally, the guarantees in lines 24, 27, 30, and 37 allow the environment to force the lift to the first floor (line 30), the system then may or may not move up to the second floor (line 24), which allows the environment to keep the lift bellow the third floor (line 27), and thus fail the guarantee to visit that floor (line 37).

A. Correctness of `Punch`

First we prove the following Lemma.

Lemma 2. *In running* `Punch` $(E, \emptyset)$*, after the loop in lines 5-11, the set CI contains the intersection of all cores in E.*

Proof. $K = \emptyset$, thus $CI = \{x \in C_0 | check(E \setminus \{x\}) = \bot\}$. Moreover, $check(E \setminus \{x\}) = \top$ iff $E \setminus \{x\}$ contains a core iff x does not belong to all the cores in E. □

We now show that the preconditions hold in all recursive calls, and that `Punch` is sound and complete.

The Preconditions of `Punch` *are Met in All Recursive Calls:* `Punch` requires that E satisfies the criterion, and that K is a subset of all the cores in E. The two preconditions must hold for the recursive calls in line 13.

According to line 6, $\forall x \in Cont$ `check`$(E \setminus \{x\}) = \top$, which satisfies the first precondition. According to lines 5 and 9, for all $x \in Cont$, $K \cup CI \subseteq E \setminus \{x\}$. Now, by assumption, K is a subset of all the cores in E, and by similar reasoning to that of Lemma 2, CI is also a subset of all the cores in E. In particular, for all $x \in Cont$, $K \cup CI$ must be a subset of all the cores in $E \setminus \{x\}$. This satisfies the second precondition.

`Punch` is Sound: A set is added to our list of cores when it is detected by `computeCore` in recursive calls at line 1 of `Punch`. The preconditions of `computeCore` are met because they match the preconditions of `Punch`.

`Punch` is Complete: This follows from Thm.1 below.

Theorem 1. *Let C be a core of a set E. In running* `Punch` (E, K) *such that K is a subset of all the cores in E, we will have $C \in AllCores$ in line 15 of Alg. 3.*

Proof. By induction on $n = |E|$, notice that for $n = 1$ there must be exactly one core of size 1, and the algorithm is correct for both possible choices of K.

Assume by induction that the claim is correct for all sets strictly smaller than n, and fix a set $|E| = n$ with $C \subseteq E$ a core of E. If C is detected in line 1, $C = C_0$ and we are done. Otherwise, $C \neq C_0$. Since C is a core, and C_0 satisfies the criterion, we know that $C_0 \not\subseteq C$, thus there is an $x \in C_0$ such that $x \notin C$. Moreover, $x \notin K$ because $K \subseteq C$, so $x \in C_0 \setminus K$. Since C is a core of E and $x \notin C$, C is a core of $E \setminus \{x\}$. This means (1) that $E \setminus \{x\}$ satisfies the criterion, which in turn means that $x \in Cont$; and (2) since $|E \setminus \{x\}| < n$, by induction C is found as a core by the call to `Punch` in line 13, when the x of the loop coincides with the x in the proof. This completes the proof.

□

B. Complexity of `Punch`

In general, the number of cores of a set according to a monotonic criterion may be exponential in the size of the set [4]. Thus, the worst-case complexity of `Punch` and of any other algorithm that would enumerate all cores, is exponential.

Lemma 2 shows that obtaining the intersection of all the cores requires only one core computation plus a number of realizability checks the size of the core that we found.

C. Employing `Punch` to Unrealizable Cores

Employing `Punch` to compute all unrealizable cores requires an implementation of `check` to check for unrealizability. We created two implementations of `Punch`, which we label `PUD` and `PQC`. In `PUD`, we implemented `computeCore` using `DDMin` with a base. Specifically, $computeCore(E, K)$ in line 1 of `Punch` is implemented with $K \cup MinWBase(\texttt{DDMin}, E, K, E \setminus K, check)$. In `PQC`, we implemented `computeCore` with an extended version

of QuickCore that supports minimization with a base. The version of QuickCore minimizes justices, safeties, and initial assertions that do not belong to a given base set. Prop. 2(i) ensures the correctness of both implementations of computeCore.

VI. Memoization

We implemented memoization for our algorithms, which allows us to avoid checking the criterion whenever the check is redundant, based on the results of prior checks of the criterion and on the criterion's monotonicity.

Basically, we keep a set of prior positive checks and a set of prior negative checks of the criterion. Whenever a check for a set is about to occur, if we already have a positive criterion subset, then we know the set is positive and we avoid actually checking it. Similarly, if we already have a negative superset then we know the set is negative. Only if memoization fails we perform an actual check of the criterion and store the result as positive or negative accordingly.

Two additional features accelerate the required subset checks. First, we keep each memoized set sorted, which enables linear time subset checks. Second, we keep the collections of positive and negative results sorted according to the size of the memoized sets, and look for subsets and supersets of relevant size only (this is correct because a larger set cannot be a subset of a smaller set, i.e., if $|B| > |A|$ then $B \not\subseteq A$).

A. Memoization in QuickCore

Realizability checks are the most computationally expensive parts of QuickCore. Almost all these checks occur as a part of DDMin runs within it. We use the DDMin implementation in Spectra. In this generic implementation of DDMin, sets for which the criterion failed are recorded, and we avoid checking them and their subsets because monotonicity ensures that they must fail as well. This heuristics was already mentioned in [29], and implemented in [15] and in [13] for unrealizable cores. On top of it, we use the memoization mechanism we described at the beginning of this section.

B. Memoization in Punch

We seek to avoid as many as possible calls to check and to computeCore.

For check, we use the memoization mechanism we described in the beginning of this section, and add all cores found by computeCore to the set of positive sets. In PQC and PUD (see Sect. V-C) all check operations share the same memoization, whether the ones invoked at line 6 of Punch or at their particular implementations of computeCore.

For computeCore, when we look for a core of a set E (see line 1 of Punch), we use the first core that is a subset of E in previously found cores, if one exists. This is important because, for example, having two disjoint cores means that without memoization, we would run a core computation to unnecessarily seek the second core for every element of the first core. Memoization ensures that the number of times we run an actual core computation in Punch is equal to the number of cores in E.

```
1 sys boolean b;
2
3 monitor boolean a {
4   !a; // initially false
5   G a->next(a); // once true, remain true forever
6 }
7
8 gar b;
9 gar G b iff a;
```

Listing 2
An unrealizable specification with a monitor

VII. Beyond Pure GR(1) Specifications

We now present our third contribution, correct and efficient core computations for specifications that are reducible to GR(1), yet include language constructs beyond pure GR(1), such as patterns, monitors, and past LTL formulas.

A. Reducing Higher-Level Constructs into Pure GR(1)

Recall that many higher-level language constructs can be reduced to pure GR(1) form by replacing them with additional auxiliary variables as well as new guarantees or assumptions. See, e.g., [19], [20].

A pattern (e.g., the response pattern $\mathbf{G}\ (p \rightarrow \mathbf{F}\ q)$, which is not in pure GR(1) form) is reduced according to a deterministic Buchi automaton that represents it. The states of the automaton are encoded using new auxiliary variables, its initial state is encoded using an auxiliary assertion about the initial values of the auxiliary variables, its transitions are encoded using an auxiliary safety, and its acceptance condition is encoded into a justice (assumption or guarantee).

Monitors and counters are constructs that track a certain value. They are reduced by adding auxiliary variables that encode that value, and optional auxiliary elements that are assertions about its initial value, and its current and next values (safeties). For example, the monitor in Lst. 2 adds one Boolean auxiliary variable a, one auxiliary initial assertion (line 4), and one auxiliary safety (line 5).

The reduction is completed by considering auxiliary variables and elements as system variables and guarantees respectively. This allows one to apply GR(1) realizability checks and synthesis.

B. A Simple but Incorrect Approach

One may suggest that core computations would minimize the GR(1) system module elements, and then trace back to the elements that induced them in the original specification. This approach, however, is incorrect. As an example, for the unrealizable specification in Lst. 2, the (incorrect) core computed by this approach contains only lines 8 and 9. The reason is that only system elements are minimized. The incorrect computation ignores the auxiliary monitor initial assertion at line 4 although without this assertion, the specification is realizable! If unrealizability is a result of auxiliary elements alone, we may even incorrectly get an empty core (see Prop. 2(ii)). This means that we must also consider auxiliary elements for minimization, and at the same time avoid redundant ones that unnecessarily complicate the reduced specification and inflate the state space.

C. Our Approach

We have implemented a framework to correctly handle specifications that include high-level constructs. The framework relies on two-way traceability between the high-level language construct and the GR(1) elements it reduces to.

Specifically, each distinguishable specification construct, as written by the engineer, is assigned an ID that represents all of the GR(1) elements it reduces to. Thus, all elements induced from patterns and past LTL operators are assigned the ID of the high-level element they belong to, while each assertion inside monitors and counters has its own ID.

Our implementation builds system and environment modules according to subsets of IDs. Core computation begins with a set of all environment IDs for the assumptions, and a set of auxiliary and system IDs for the guarantees. It performs realizability checks given subsets of IDs. For example, `QuickCore` may eliminate all justice assumptions (line 5). If any of these justice assumptions were induced by patterns, the produced environment module avoids not only these justices but also their matching pattern-induced initial and safety assertions, as well as the pattern-induced auxiliary variables that encoded the states of this pattern's automaton. These are the exact sets of elements and variables that match the subset of high-level assumptions.

In Lst. 2, the correct core we compute in this way includes lines 4, 8, and 9. Together, they are sufficient for unrealizability, and each of them is necessary.

VIII. Evaluation

We have implemented our ideas on top of Spectra [20], [28], with the performance heuristics from [13]. Our implementation includes `QuickCore` and `Punch`. For the purpose of evaluation, it also includes an instance of the `DDMin` algorithm implemented in Spectra, an implementation of `QuickXplain` [14], [24], and a naive top down algorithm for computing all cores we label `TD` (see below). All the above implementations take advantage of the memoization described in Sect. VI.

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 [1]. We encourage the interested reader to try them out.

The following research questions guide our evaluation.

R0 Which existing domain-agnostic minimization algorithm is the most efficient in our setup?

R1 Is `QuickCore` efficient, in terms of the number of realizability checks and running time, in comparison to previous algorithms?

R2 Is `QuickCore` effective, in terms of the size of the core it finds, in comparison to previous algorithms?

R3 Are specifications with multiple unrealizable cores common and how many such cores do most specifications have?

R4 Is `Punch` efficient in detecting all cores?

Below we report on the experiments we have conducted in order to answer the above questions.

Table I
Efficiency of DDMin vs. QuickXplain

Spec set	$\leq 0.1s$	0.1-1s	1-10s	10-100s	$\geq 100s$
SYN15U	0.10	0.48	-	-	-
SYN17U	0.08	0.28	0.41	0.52	0.84
AM+GN	0.10	0.40	0.42	0.80	0.99

A. Corpus of Specifications

We use the benchmarks `SYNTECH15` and `SYNTECH17` [13], [20], [28], which include a total of 227 specifications of 10 autonomous Lego robots, written by 3rd year undergraduate computer science students in a project class taught by the authors of [13]. We use all the unrealizable GR(1) specifications from these, i.e., 14 unrealizable specifications from `SYNTECH15` (which we label `SYN15U`) and 26 unrealizable specifications from `SYNTECH17` (which we label `SYN17U`).

In addition, we used 5 different sizes of `AMBA` [5] and of `GENBUF` [6] (1 to 5 masters, 5 to 40 senders resp.), from each of the 3 variants of unrealizability described in [11]. We label these 30 specifications by `AM+GN`. Note that these specifications are structurally synthetic and artificially inflated. We therefore report on their performance (in R0, R1, and R4) but not on their more qualitative aspects (R2 and R3). Still, the supporting materials [1] include all the data we collected.

B. Validation

We have implemented an automatic test that checks that every core that we found is indeed a locally minimal subset of the guarantees that maintains unrealizability. We run this check over logs of cores produced by our algorithms, independent of their original detection. We also verified that the different algorithms that compute all the cores of a specification (i.e., ones that terminated before the timeout was reached) found the same number of cores.

C. Experiments Setup

We ran 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.

Times we report are average values of 10 runs, measured by Java in milliseconds. Although the algorithms we deal with are deterministic, we performed 10 runs since JVM garbage collection and BDD dynamic-reordering add variance to running times.

We used a timeout of 10 minutes for the algorithms that compute all cores, and no timeout for the algorithms that find a single core.

D. Results: Existing Domain-Agnostic Algorithms in Our Setup

In Sect. III-C we discussed three existing domain-agnostic algorithms for finding a local minimum given a monotonic criterion. We also noted that `DDMin` was compared and found superior to `LinearMin` in [15].

Table II
EFFICIENCY OF QUICKCORE VS. DDMIN

Spec set	$\leq 0.1s$	0.1-1s	1-10s	10-100s	$\geq 100s$
SYN15U	0.79	0.51	-	-	-
SYN17U	1.38	0.37	0.54	0.06	0.015
AM+GN	1.53	0.63	0.71	0.30	0.20

Table I presents the performance of DDMin versus QuickXplain. The columns show the geometric mean of the ratio of the running times (namely, the running time of DDMin divided by that of QuickXplain), dissected by the running time range obtained for DDMin. We use '-' to mark cases in which no specifications had DDMin running time within the corresponding range. For example, the number 0.28 in the second row means that for SYN17U specifications for which a core was found by DDMin in between 0.1 and 1 seconds, the geometric mean indicates that DDMin was more than three times faster than QuickXplain.

The results show that DDMin is more efficient than QuickXplain on all our corpus (although the gap lessens with scale). This justifies our choice of DDMin as the domain-agnostic minimization algorithm inside QuickCore and PUD. It also justifies our choice to use DDMin as the baseline algorithm for examining the efficiency of QuickCore (see R1 below).

To answer R0: DDMin is the most efficient domain-agnostic single core computation algorithm in our setup.

E. Results: Efficiency of QuickCore versus DDMin

Table II presents the performance of QuickCore versus DDMin. We chose to compare QuickCore with DDMin because DDMin is a well known and widely used algorithm for core computation over a monotonic criteria, and because it was previously used in the context of unrealizable cores for GR(1) specifications. Moreover, in R0 above we showed that DDMin is the most efficient domain-agnostic single core algorithm on our corpus.

The columns show the geometric mean of the ratio of the running times (namely, the running time of QuickCore divided by that of DDMin), dissected by the running time range obtained for DDMin, for all specifications in each set. We use '-' to mark cases in which no specifications had DDMin running time within the corresponding range. For example, the number 0.30 in the third row means that for AM+GN specifications for which a core was found by DDMin in between 10 and 100 seconds, the geometric mean indicates that QuickCore was more than three times faster than DDMin.

The results show that QuickCore is in most cases much faster than DDMin. This improvement gets better with scale, i.e., almost consistently, the slower DDMin, the faster QuickCore becomes relative to it. The acceleration is most noticeable for SYN17U specifications for which DDMin require over 10 seconds. For those specifications QuickCore was faster than DDMin by well over an order of magnitude.

Table III
CORE SIZES

Spec set	Core Size	QC	DDMin	QX	Global
SYN15U	19%	5.14	5.07	5.21	5.07
SYN17U	18%	4.5	4.38	4.5	3.92

Table IV
NUMBER OF CORES

Spec set	S	M	≥ 5	≥ 10	≥ 50	≥ 100
SYN15U	6	8	5	2	1	0
SYN17U	6	20	9	7	4	4

The only two categories in which the running time of QuickCore is worse than that of DDMin is for SYN17U and AM+GN specifications whose DDMin running time is at most 100 milliseconds. Since for this range running times are very small, we do not consider it to be a major weakness of QuickCore.

We also computed the actual number of realizability checks (i.e., without realizability checks avoided by memoization, see Sect. VI) of QuickCore and DDMin (not shown in the table). We found that the median reduction in the number of actual realizability checks of QuickCore over DDMin was 11.3%, 19.5%, and 15.4%, and over QuickXplain was 3.3%, 10.2%, and 27.6%, for SYN15U, SYN17U, and AM+GN, respectively.

To answer R1: QuickCore typically performs fewer realizability checks than DDMin and QuickXplain, it is in most cases much faster than DDMin, and the running time improvement seems to become better with scale.

F. Results: Effectiveness of QuickCore

Table III presents core size results. Column Core size shows the median ratio between the size of cores found by QuickCore and the total number of guarantees in the specification. Columns QC, DDMin, QX, and Global show the average absolute size of the cores found by QuickCore, DDMin, QuickXplain, and the size of the smallest core found until the timeout by Punch, respectively.

The results show that in the SYNTECH specifications, most of the cores found by QuickCore are over five times smaller than the total number of guarantees in the specification. They further show that the cores found by QuickCore have a slightly different size than the cores found by the other algorithms, and that all algorithms output cores that are close in size to the size of the globally minimal core.

To answer R2: QuickCore, DDMin, and QuickXplain are all effective in localizing unrealizability.

G. Results: Number of Cores in Specifications

Table IV presents the number of cores in the SYNTECH specifications. Columns 'S' and 'M' show how many specifications have a single core and multiple cores, resp. The remaining four columns show how many of these specifications have at least 5, 10, 50, and 100 cores. For example, the

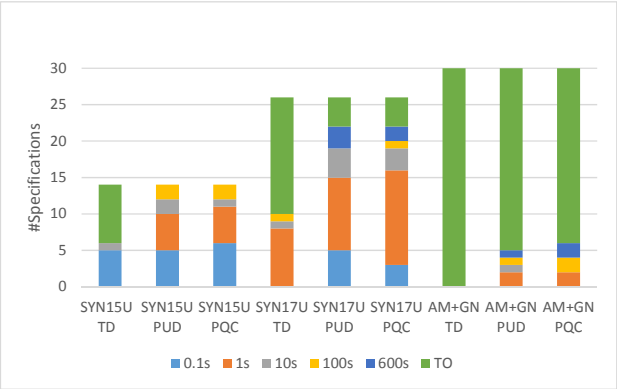

Figure 1. Running times to compute all cores using TD (left columns) PUD (center columns), and PQC (right columns), for the SYNTECH and the AM+GN sets, divided by increasing ranges, in seconds.

number 1 in the first row under '≥ 50' means that exactly one specification of SYN15U has at least 50 cores.

To answer R3: Most SYNTECH specifications have multiple cores. Specifications with over 50 cores exist in each set of specifications. These results motivate the need to compute more than one core per specification.

H. Results: Running Times of All Cores Algorithms

In order to evaluate the performance of Punch for computing all unrealizable cores, we use PQC and PUD (see Sect. V-C).

Since Punch is the first algorithm employed to compute all unrealizable cores for temporal specifications reducible to GR(1), as a comparison, we use a rather naive baseline we label TD (we discuss alternative algorithms in Sect. IX).

TD is a naive top down search for all cores. For a given subset of guarantees, if check fails, TD knows that the subset and all its subsets are not cores. Otherwise, it continues recursively to all subsets that exclude exactly one element. It detects the set as a core iff all of these subsets fail check. It memoizes subsets we already finished computing all cores for, to avoid unnecessary recursive calls (in addition to the memoization described in Sect. VI). To detect unrealizable cores, check is implemented as an unrealizability check.

Figure 1 shows the running times for SYN15U, SYN17U, and AM+GN specifications. Each specification set has three columns, one for TD, one for PUD, and one for PQC. Each column shows a breakdown of how many specifications completed running within 0.1, 1, 10, 100, and 600 seconds. Finally, we mark the number of specifications that timed out (did not complete all 10 runs within 10 minutes) with TO.

The results show that the differences between the performance of PUD and PQC are minor, with mostly a slight advantage to PQC. Regardless of their minor differences, the two instances of Punch perform significantly better than TD. The two instances of Punch were able to find all cores of SYN15U specifications within 100 seconds, and 22 out of the 26 of the SYN17U specifications before the timeout was reached. This is much better than TD, which was able to find all cores within the timeout for less than half of the specifications. On AM+GN specifications, PUD and PQC found all cores for 5 and 6 specifications respectively, but TD did not complete the computation on any of the specifications.

To answer R4: Both instances of Punch are significantly faster than TD in finding all the cores of a specification. PQC seems slightly better than PUD.

I. Additional Results

Recall that Punch provides early detection of the intersection of all the cores (see Sect. V-B). The size of the intersection is a lower bound on the size of the global minimum. In practice, it provides a good early estimate of the size of the global minimum for SYNTECH specifications. The results show that the core intersection size is on average 76.8% and 52.5% of the smallest core found by Punch within the timeout for SYN15U and SYN17U respectively.

J. Threats to Validity

We discuss threats to the validity of our results. First, symbolic computations are not trivial and our implementation of QuickCore, DDMin, QuickXplain, and Punch may contain bugs. To mitigate, we performed a thorough validation using all specifications available to us, see Sect. VIII-B. Second, we have based most of our evaluation on the SYNTECH specifications, which were created by 3rd year undergraduate CS students with no prior experience in writing LTL specifications (collected by the authors of [20] in classes they have taught). We further examined specifications from the AM+GN set. We do not know if these are representative of specifications engineers would write in practice. Third, although we have found that cores are typically much smaller than the complete set of guarantees (see Tbl. III, roughly 5 guarantees instead of 25), we did not perform a user-study, with engineers, to examine whether users will find the reported cores useful for understanding the reasons for the unrealizability of their specifications.

IX. Related Work

A. A Single Unrealizable Core for GR(1)

Previous works have considered the computation of an unrealizable core for GR(1) specifications. Cimatti et al. [11] have used LinearMin (see Sect. III-C). Konighofer et al. [15] have used DDMin and implemented it in the RATSY synthesizer. Their comparison of DDMin with LinearMin shows that DDMin is almost always much faster than LinearMin, with a greater advantage on larger specifications. Firman et al. [13] have used DDMin with several performance heuristics, including memoization, and implemented it in the Spectra synthesizer. All these were limited to computing a single core and did not correctly handle specifications with constructs beyond pure GR(1). We present QuickCore to be used instead of DDMin. We further show how to correctly handle specifications with constructs beyond pure GR(1). We compare

QuickCore to Spectra's implementation of DDMin, i.e., with the heuristics from [13], and our evaluation provides evidence that QuickCore is faster and scales better than DDMin.

Our choice of DDMin for both the algorithm we compare to, and the algorithm used within QuickCore for the incremental minimization, is based both on DDMin being a well known and widely used domain-agnostic minimization algorithm, and on the fact that it was the choice in previous work on GR(1) unrealizability.

There are other domain-agnostic minimization algorithms over monotonic criteria for single cores, e.g., QuickXplain [14], [24]. Compared to DDMin, QuickXplain has a better asymptotic complexity in terms of the number of checks (see Sect. III-C). Nevertheless, as we show in our evaluation (Sect. VIII-D), QuickXplain performs worse on our corpus.

For temporal specifications, Schuppan [27] presented LTL unsatisfiability cores by weakening LTL formulas in a way that ignores sub-formulas not required for unsatisfiability. He further presented a similar approach for GR(1) unrealizability cores. To the best of our knowledge, these ideas have only been explored theoretically. Moreover, the work neither handles all cores nor deals with extensions of GR(1).

B. All Unrealizable Cores for GR(1)

We present Punch as the first efficient algorithm to compute all unrealizable cores of specifications reducible to GR(1). However, Punch is a domain-agnostic algorithm. Other domain-agnostic algorithms for all cores computations over monotonic criteria appear in the literature, see, e.g., [17]. In [3] there is a comparison of several such algorithms, which concludes that none of the known algorithms is better than the others in all domains. Recently, MUST [4] was proposed as an algorithm and tool that outperforms previous ones.

Punch is intended as a first algorithm for the computation of all cores of unrealizable GR(1) specifications. We consider its comparison against domain-agnostic algorithms, as well as its specialization for GR(1) as future work, see Sect. X.

C. Other Approaches to Dealing with GR(1) Unrealizability

Beyond core computations, other approaches have been suggested to dealing with unrealizability of GR(1) specifications. Maoz and Sa'ar [22] have presented the computation of counter-strategies, which show how the environment can prevent any system from satisfying the specification. Kuvent et al. [16] have presented the JVTS, a symbolic, more succinct and simple representation of a GR(1) counter-strategy.

Other works have considered means to repair unrealizable specifications by automatically suggesting additional assumptions that will make the specification realizable, see, e.g., [2], [8], [9], [10], [21]. It may be possible to combine the computation of a core or of all cores with a repair approach. See our discussion of future work in Sect. X.

X. Conclusion and Future Work

We presented three contributions related to the computation of unrealizable cores of GR(1) specifications, including faster algorithms for computing an unrealizable core and for computing all cores. We further presented means to correctly compute the core when specifications include high-level constructs.

We implemented our work, validated its correctness, and evaluated it on benchmarks from the literature. The evaluation shows that QuickCore is usually faster than previous algorithms, with a relative advantage that improves with scale. Moreover, we found that most specifications have multiple cores, and that Punch finds all the cores significantly faster than a competing naive algorithm.

Our work has important implications to anyone using GR(1) specifications and their extensions for synthesis and related analyses. First, core computations are now faster, and computing more than one core promotes a more comprehensive view of unrealizability. Moreover, we handle higher-level constructs correctly and efficiently, and our algorithm for finding all cores extends to cores for any monotonic criterion.

We consider the following concrete future work directions. First, as Punch in its raw form is domain-agnostic, it is important to compare its performance with recent domain-agnostic all cores minimization algorithms over monotonic criteria such as MUST [4].

We have already presented a variant of Punch, namely PQC, which employs a domain-specific algorithm (a variant of QuickCore) to compute a single core, and compared it with another variant, namely PUD, which uses DDMin, a domain-agnostic algorithm for a single core computation. The results showed that PQC performs slightly better than PUD. It may be possible to improve the performance of detecting all unrealizable cores by taking advantage of GR(1) specific properties in other ways.

Second, we consider means to combine the computation of a core or of all cores with a repair approach. For example, a repair of unrealizability that is based on weakening a small as possible subset of the guarantees could rely on the fact that it cannot succeed without weakening at least one guarantee from every core, or in particular weakening one guarantee from a non-empty intersection of all cores.

Finally, the ability to compute all cores raises questions as to their presentation to the engineer. Should all cores be computed and presented? Perhaps an on demand approach should be used? In which order should we present the cores? These questions call for further investigation and evaluation.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We thank Roee Sinai for implementing the QuickXplain algorithm in the Spectra environment. We thank Inbar Shulman for comments about an earlier draft of the paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

REFERENCES

[1] Supporting materials website. http://smlab.cs.tau.ac.il/syntech/unrealcores/.

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

[3] J. Bendík and I. Cerna. Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 131–142, 2018.

[4] J. Bendík and I. Cerná. MUST: minimal unsatisfiable subsets enumeration tool. In A. Biere and D. Parker, editors, *TACAS*, volume 12078 of *LNCS*, pages 135–152. Springer, 2020.

[5] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. pages 1188–1193. EDA Consortium, San Jose, CA, USA, 2007.

[6] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

[7] 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.

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

[9] 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.

[10] D. G. Cavezza, D. Alrajeh, and A. György. Minimal assumptions refinement for realizable specifications. In *FormaliSE@ICSE 2020: 8th Int. Conf. on Formal Methods in Software Engineering*, pages 66–76. ACM, 2020.

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

[12] 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.

[13] E. Firman, S. Maoz, and J. O. Ringert. Performance heuristics for GR(1) synthesis and related algorithms. *Acta Inf.*, 57(1-2):37–79, 2020.

[14] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In D. L. McGuinness and G. Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

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

[16] 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.

[17] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.

[18] S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit. Reactive high-level behavior synthesis for an atlas humanoid robot. In *ICRA*, pages 4192–4199, 2016.

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

[20] S. Maoz and J. O. Ringert. Spectra: A specification language for reactive systems. *Software and Systems Modeling*, 2021. To appear.

[21] S. Maoz, J. O. Ringert, and R. Shalom. Symbolic repairs for GR(1) specifications. In *ICSE*, pages 1016–1026, 2019.

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

[23] S. Maoz and R. Shalom. Inherent vacuity for GR(1) specifications. In *ESEC/FSE*, pages 99–110. ACM, 2020.

[24] J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, pages 592–607. Springer, 2013.

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

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

[27] V. Schuppan. Towards a notion of unsatisfiable and unrealizable cores for LTL. *Sci. Comput. Program.*, 77(7-8):908–939, 2012.

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

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