Performance Heuristics for GR(1) Synthesis and Related Algorithms

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

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

1 Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [27]. Rather than manually constructing a system and using model checking to verify its compliance with its specification, synthesis offers an approach where a correct implementation of the system is automatically obtained, if such an implementation exists.

GR(1) is a fragment of LTL, which has an efficient symbolic synthesis algorithm [1, 26] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [6, 19]. GR(1) synthesis has been used and extended in different contexts and for different application domains, including robotics [16], scenario-based specifications [23], aspect languages [22], event-based behavior models [5], and device drivers [29], to name a few.

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

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

We evaluate the presented heuristics on two sets of specifications. The first set, SYNTECH15, consists of 78 specifications of 6 autonomous Lego robots, written by 3rd year undergraduate computer science students in a project class we have taught. The second set consists of specifications for the ARM AMBA AHB Arbiter (AMBA) and a Generalized Buffer from an IBM tutorial (GenBuf), which are the most popular GR(1) examples in literature, used, e.g., in [1, 4, 14, 30]. Our evaluation addresses the

D. Fisman, S. Jacobs (Eds.): Sixth Workshop on Synthesis (SYNT 2017)
EPTCS 260, 2017, pp. 62–80, doi:10.4204/EPTCS.260.7

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

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

2 Preliminaries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Specifications for GR(1) synthesis have to be expressible in the above structure and thus do not cover the complete LTL. Efficient symbolic algorithms for GR(1) realizability checking and strategy synthesis for φ^{sr} have been presented in [1, 26]. The algorithm of Piterman et al. [26] computes winning states for the system, i.e., states from which the system can ensure satisfaction of φ^{sr}. We denote the states from which the system can force the environment to visit a state in R by $\bigcirc\!\!\!\!\diamond(R)$ defined as:

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

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

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

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

Unrealizability and Rabin(1) Game A specification φ is unrealizable if there is a counter-strategy in which the environment can force the system to violate at least one of its guarantees while satisfying all the environment assumptions. Maoz and Sa'ar [24] show how to compute the fixed-point algorithm given by Könighofer et al. [14] by playing a generalized Rabin game with one acceptance pair (Rabin(1) game[1]). The algorithm computes the set of the winning states for the environment by calculating cycles

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

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

1: $Z = \texttt{true}$
2: **while** not reached fixed-point of Z **do**
3: **for** $j = 1$ **to** $|J^s|$ **do**
4: $Y = \texttt{false}; cy = 0$
5: **while** not reached fixed-point of Y **do**
6: $start = J^s_j \wedge \bigcirc Z \vee \bigcirc Y$
7: $Y = \texttt{false}$
8: **for** $i = 1$ **to** $|J^e|$ **do**
9: $X = Z$ // better approx. than `true`, see [1]
10: **while** not reached fixed-point of X **do**
11: $X = start \vee (\neg J^e_i \wedge \bigcirc X)$
12: **end while**
13: $Y = Y \vee X$
14: `X[`j`][`i`][`cy`++]` $\leftarrow X$
15: **end for**
16: `Y[`j`][`cy`++]` $\leftarrow Y$
17: **end while**
18: $Z = Y$
19: `Z[`j`]` = Y
20: **end for**
21: **end while**
22: **return** Z

Algorithm 2 Rabin(1) game algorithm from [24, 28] to compute environment winning states Z

1: $Z = \texttt{false}; cz = 0$
2: **while** not reached fixed-point of Z **do**
3: **for** $j = 1$ **to** $|J^s|$ **do**
4: $Y = \texttt{true}$
5: **while** not reached fixed-point of Y **do**
6: $start = \neg J^s_j \wedge \boxdot Y$
7: $Y = \texttt{true}$
8: **for** $i = 1$ **to** $|J^e|$ **do**
9: $pre = \boxdot Z \vee J^e_i \wedge start$
10: $X = \texttt{false}; cx = 0$
11: **while** not reached fixed-point of X **do**
12: $X = pre \vee (\neg J^s_j \wedge \boxdot X)$
13: `X[`cz`][`i`][`cx`++]` $\leftarrow X$
14: **end while**
15: $Y = Y \wedge X$
16: **end for**
17: **end while**
18: $Z = Z \vee Y$
19: `Z[`cz`++]` $\leftarrow Y$
20: **end for**
21: **end while**
22: **return** Z

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

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

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

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

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

The algorithm from [24] (extended to handle J^e as implemented in JTLV [28]) for computing the set W_{env} is shown in Alg. 2. Again, the algorithm already implements some optimizations over the naive implementation of Eqn. (2), e.g., the early update of Z in L. 18. Algorithm 2 stores intermediate computation results in arrays `Z[]` (L. 19) and `X[][][]` (L. 13) for strategy construction.

Delta Debugging (DDMin) The Delta Debugging algorithm [34] (DDMin) finds a locally minimal subset of a set E for a given monotonic criterion `check`. We show the DDMin algorithm in Alg. 3. The input of the algorithm are a set E and the number n of partitions of E to check. The algorithm starts with $n = 2$ and refines E and n in recursive calls according to different cases (L. 6, L. 11, and L. 14). The computation starts by partitioning E into n subsets and evaluating `check` on each subset *part* (L. 4) and its complement (L. 10). If `check` holds (L. 6 or L. 11), the search is continued recursively on the subset

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

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

Algorithm 3 Delta Debugging algorithm DDMin from [34] as a recursive method that minimizes a set of elements E by partitioning it into n partitions (initial value $n = 2$)

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

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

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

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

3 Suggested Performance Heuristics

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

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

3.1.1 Early detection of fixed-point

Rationale. The GR(1) game and the Rabin(1) game iterate over the justice guarantees in the outermost fixed-point. Each iteration refines the set of winning states based on the justice guarantee and the calculated set from the previous iteration (`for`-loop in Alg. 1, L. 3 and Alg. 2, L. 3). Computing a fixed-point for the same justice guarantee J^s_j and the same set Z always yields the same set of winning states. We can exploit the equality to detect if we will reach a fixed-point without completing the `for`-loop, i.e., without computing the fixed-points for all justice guarantees. We found this heuristics implemented in the Rabin(1) game in JTLV [28]. We have not seen a similar implementation for the GR(1) game.

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

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

3.1.2 Early detection of unrealizability

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

Examples: Early Detection of Fixed-Point

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

Listing 1: Heuristics very effective

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

Listing 2: Heuristics does not yield improvement

Examples: Early Detection of Unrealizability

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

Listing 3: Heuristics very effective

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

Listing 4: Heuristics does not yield improvement

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

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

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

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

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

3.1.3 Fixed-point recycling

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

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

Examples: Fixed-Point Recycling

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

Listing 5: Heuristics very effective

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

Listing 6: Heuristics does not yield improvement

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

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

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

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

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

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

Examples: Contained Sets in DDMin

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

Listing 7: Heuristics very effective

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

Listing 8: Heuristics does not yield improvement

3.2 Unrealizable Core Calculation

3.2.1 Contained sets

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

This heuristics was mentioned in [35] and also implemented for unrealizable core calculation in [14].

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

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

3.2.2 Incremental GR(1) for similar candidates

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

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

Examples: Incremental GR(1) in DDMin

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

Listing 9: Heuristics very effective

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

Listing 10: Heuristics does not yield improvement

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

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

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

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

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

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

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

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

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

Listing 11: Heuristics very effective

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

Listing 12: Heuristics does not yield improvement

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

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

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

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

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

4 Evaluation

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

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

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

4.1 Procedure

We used the GR(1) game and Rabin(1) game implementations shown in Alg. 1 and Alg. 2 as reference (recall that these algorithms already contain performance improvements over naive implementations following the fixed-point formulation, see Sect. 2). We have implemented these two algorithms and all our suggested heuristics in C using CUDD 3.0 [31]. We measure running-times in nanoseconds using C

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

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

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

4.2 Evaluation Materials

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

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

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

In addition to the specifications created by the students, we considered the ARM AMBA AHB Arbiter (AMBA) and a Generalized Buffer from an IBM tutorial (GenBuf), which are the most popular GR(1) examples in literature, used, e.g., in [1, 4, 14, 30]. We included 5 different sizes of AMBA (1 to 5 masters) and 5 different sizes of GenBuf (5 to 40 requests), each in its original version plus the 3 variants of unrealizability described in [4] (justice assumption removed, justice guarantee added, and safety guarantee added). We have thus run our experiments also on 20 AMBA and 20 GenBuf specifications.

All specifications used in our evaluation, the raw data recorded from all runs, and the program to reproduce our experiments are available from [36].

4.3 Evaluation Results

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

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

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

SYNTECH15 realizable

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

AMBA/GenBuf realizable

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

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

SYNTECH15 unrealizable

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

AMBA/GenBuf unrealizable

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

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

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

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

SYNTECH15 unrealizable

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

AMBA/GenBuf unrealizable

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

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

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

4.3.2 Results for Unrealizable Core Calculation

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

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

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

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

4.4 Validation of Heuristics' Correctness

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

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

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

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

4.5 Threats to Validity

We discuss threats to the validity of our results.

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

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

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

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

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

5 Related Work

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

Others have focused on strategy construction for GR(1). Strategies are constructed from the memory stored in the `X`, `Y`, and `Z` arrays in Alg. 1 and Alg. 2. Schlaipfer et al. [30] suggest synthesis of separate strategies for each justice guarantee to avoid a blow-up of the BDD representation. Bloem et al. [1] discuss different minimization of synthesized strategies that do not necessarily minimize their BDDs. We consider space and time related heuristics for strategy construction an interesting next step.

It is well-known that the order of BDD variables heavily influences the performance of BDD-based algorithms [11, 33]. The GR(1) implementation of Slugs [8] uses the default dynamic variable reordering of CUDD [31] (as we do). Slugs turns off reordering during strategy construction. Filippidis et al. [9] reported better performance with reordering during strategy construction. We are not aware of any GR(1) specific heuristics for (dynamic) BDD variable ordering.

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

Finally, a number of heuristics for BDD-based safety game solvers have been reported as outcome of the SYNTCOMP reactive synthesis competitions [11, 12, 13]. Most of these optimizations are on the level of predecessor computations (operators $\Diamond$ in Alg. 1 and $\bigcirc$ in Alg. 2), while the heuristics we implemented are on the level of fixed-points and repeated computations. It seems possible to combine these heuristics. Notably, an approach for predicate abstraction for predecessor computation has already been implemented for GR(1) synthesis [29, 32].

6 Conclusion

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

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

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

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

References

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

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

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

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

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

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

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

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

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

[10] Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors (2002): *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. *Lecture Notes in Computer Science* 2500, Springer, DOI: 10.1007/3-540-36387-4.

[11] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2017): *The first reactive synthesis competition (SYNTCOMP 2014)*. *STTT* 19(3), pp. 367–390, DOI: 10.1007/s10009-016-0416-3.

[12] Swen Jacobs, Roderick Bloem, Romain Brenguier, Ayrat Khalimov, Felix Klein, Robert Könighofer, Jens Kreber, Alexander Legg, Nina Narodytska, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan

Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2016): *The 3rd Reactive Synthesis Competition (SYNTCOMP 2016): Benchmarks, Participants & Results*. In Piskac & Dimitrova [25], pp. 149–177, DOI: 10.4204/EPTCS.229.12.

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

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

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

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

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

[18] Shahar Maoz, Or Pistiner & Jan Oliver Ringert (2016): *Symbolic BDD and ADD Algorithms for Energy Games*. In Piskac & Dimitrova [25], pp. 35–54, DOI: 10.4204/EPTCS.229.5.

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

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

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

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

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

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

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

[26] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) Designs*. In: *VMCAI*, pp. 364–380, DOI: 10.1007/11609773_24.

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

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

[29] Leonid Ryzhyk & Adam Walker (2016): *Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned*. In Piskac & Dimitrova [25], pp. 84–99, DOI: 10.4204/EPTCS.229.8.

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

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

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

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

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

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

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