Synthesis from Infinite-State Generalized Reactivity(1) Specifications *

Benedikt Maderbacher, Felix Windisch, and Roderick Bloem

Graz University of Technology, Graz, Austria
{benedikt.maderbacher,roderick.bloem}@iaik.tugraz.at
felix.windisch@tugraz.at

Abstract. Reactive synthesis is used to automatically generate circuits or programs from temporal logic specifications. In propositional reactive synthesis, generalized reactivity(1) (GR(1)) has proven to be a good compromise between expressiveness and performance.
In domains like cyber-physical systems, however, programs must operate on data such as integers or reals, resulting in an infinite state space. This paper shows how to achieve the advantages of GR(1) to infinite-state reactive synthesis. We show how to use an SMT solver to solve the synthesis problem and present techniques to efficiently compute the enforceable predecessor and optimize the fixpoint computation.
We show how to generate efficient programs as the result of the synthesis procedure, using techniques that are different from the circuit generation methods used in the propositional domain. We have implemented our method as a prototype to show its efficiency on several existing and new benchmarks.

1 Introduction

Reactive synthesis [5] aims to automatically create circuits or programs that interact with an environment from temporal logic specifications. Traditionally this has been done for inputs and outputs represented as vectors of boolean variables. This is a natural representation of hardware circuits. Software, however, operates on more complex data such as integers or reals, in the case of cyber-physical systems. In recent years, reactive synthesis of potentially infinite state systems using arithmetic has gained popularity [16, 27, 20, 18, 15, 12]. These focus on either Büchi and co-Büchi conditions or full LTL specifications.

In this paper, we show how to solve GR(1) games in the setting of infinite-state systems and demonstrate its advantages over methods relying on deterministic Büchi automata. Generalized reactivity of rank 1 (GR(1)) [24, 9] is a subset of LTL that has been shown to provide an excellent trade-off between expressivity and performance in propositional reactive synthesis.

Whereas most reactive synthesis tools use BDDs [21, 22, 14], SAT solvers have been applied to reactive synthesis in the Boolean domain [7, 8]. In the

* This project has received funding from the Austrian research promotion agency FFG under project FATE (№ 894789).

infinite domain, this is obviously not an option and we use SMT solvers instead. This allows us to handle theories with decidable quantifier elimination such as real arithmetic or linear integer arithmetic. The problem of infinite state reactive synthesis is known to be undecidable [16] this also applies to infinite state GR(1).

We show that a straightforward implementation of the GR(1) fixpoint using an SMT solver can be very inefficient in terms of computation speed and size of the resulting program. We present several optimizations that can improve both the speed of the computation and the size of the resulting program by two orders of magnitude, depending on the characteristics of the specification.

The core operation in the computation of the winning region is the *enforceable preimage* computation. Given a set of target states, enforceable preimage computes a set of source states from which the system can guarantee that the target is reached, no matter what the environment does. This computation relies on the elimination of a universal and an existential quantifier and is thus expensive. We present and compare different techniques for implementing this step. Most notably, we use a formula simplification technique from the software verification community [13] to significantly reduce the size of the representation of the intermediate results, leading to a large speedup.

Furthermore, we show how early termination and reusing the intermediate results of the fixpoint computation [11, 9, 17] lead to further improvements in computation time of up to a factor two. Together, these methods enable us to solve problems from the cyber-physical systems domain that have been intractable before.

A naive method to extract a program from the results of the fixpoint computation may result in an extremely large nondeterministic program. We show how to construct programs that are much smaller and goal-directed in the sense that they always select the update that results in the most progress toward the current goal. (Cf. [6, 17].) Additionally, we simplify strategies to obtain a short case distinction to decide which update the system takes. For our case studies, these representations are easily understandable by human experts.

Example of Infinite State GR(1) Synthesis

We explain the problem of infinite state GR(1) synthesis using an example. The specification consists of two parts: an arena and a winning condition. The arena defines the state and input variables as well as the actions that the environment and the system can take. The winning condition sets out what system has to do to satisfy the specification. Our example consists of one state variable $x \in \mathbb{R}$ that is controlled by the system and one input variable $d \in \mathbb{R}$ controlled by the environment. When it is the system's turn, it can choose one out of three actions:

$$\begin{aligned} a_1 &= (x \geq 0 \wedge x \leq 6, x := x + 1 + d), \\ a_2 &= (x \geq 0 \wedge x \leq 6, x := x - 1 + d), \text{ or} \\ a_3 &= (x \geq 0 \wedge x \leq 6, x := x + d). \end{aligned} \tag{1}$$

Each action consists of a guard and an update to every state variable. The system chooses one action where the guard is satisfied and changes its state using the

chosen updates, which are deterministic assignments to the state variables. If no guard is satisfied, the specification is violated. This way, guards can be used to express a safety invariant for the system. In this example, the value of x has to remain between 0 and 6. The updates can refer to the old values of the state variables as well as the inputs provided by the environment. The options for the system are to either increase or decrease x by one, or to leave it unchanged. In either case, a disturbance d is added to x.

The moves of the environment are defined by a relation σ_e between state and input variables that restricts how the next input can be selected. In contrast to the system, the environment can have an infinite number of choices. In our example, the value d can be any value between -1 and 1:

$$\sigma_e = -1 \leq d' \leq 1. \tag{2}$$

The last part of the arena is a predicate over the state variables that specifies a set of initial states:

$$\mathit{init} = 1 \leq x \leq 5. \tag{3}$$

Whereas the arena defines the safety part of the specification, the liveness part is defined using a winning condition. We give this condition using LTL syntax, where $\square\lozenge$ means infinitely often:

$$\phi = (\square\lozenge(d < 0) \wedge \square\lozenge(d > 0)) \implies (\square\lozenge(x < 1) \wedge \square\lozenge(x > 5)). \tag{4}$$

A GR(1) condition consists of assumptions and guarantees. Both consist of a conjunction of propositions that must be satisfied infinitely often. The specification above reads as follows. Assuming that the environment chooses both $d < 0$ and $d > 0$ infinitely often, the system has to guarantee that x is smaller than one infinitely often and also that x is greater than five infinitely often.

Our algorithm combines the classic formula for solving GR(1) games [9] with SMT solvers and quantifier elimination to solve fixpoints for infinite state games [26]. The GR(1) formula is a triply nested fixpoint. Intuitively, greatest fixpoints are used to achieve infinitely repetitive, cyclical behavior, whereas least fixpoints are used to achieve a goal in a finite number of steps. Roughly speaking, the outer greatest fixpoint makes sure that the system guarantees are achieved infinitely often. Nested within the greatest fixpoint is a conjunction of least fixpoints that assure that each of the system guarantees in turn is reached in finitely many steps. Finally, nested within each least fixpoint is a disjunction of least fixpoints that allow the system not to make progress as long as the environment does not fulfill the assumptions.

Given this specification, our algorithm constructs a *strategy* consisting of a list of conditions stating the conditions under which each update is taken. To achieve this, the system uses an additional integer variable k to track which guarantee it is currently working towards.

Listing 1 shows the strategy for our example. This strategy consists of one sub-strategy for each goal and uses the variable g to track which sub-strategy it is currently using. First, it checks if the goal of the current sub-strategy is satisfied

Listing 1: Example system

```
if((k = 1 ∧ x < 1) ∨ (k = 2 ∧ x > 5)) {
    k := k ⊕ 1
}
if ( k = 1 ) {
    if(x + d >= 1) {
        x := x − 1 + d
    } else {
        x := x + d
    }
}
else if ( k = 2 ) {
    if(x + d <= 5) {
        x := x + 1 + d
    } else {
        x := x + d
    }
}
```

in which case it changes to the next sub-strategy wrapping around from the last to the first one. Next, it uses the current sub-strategy to decide which update to execute. The goal of the sub-strategy $k = 1$ ($k = 2$) is to make progress towards $x < 1$ ($x > 5$, resp.). If $k = 1$ the system decreases x unless the disturbance leads to an underflow, in which case it does not change x. Note that the environment can prevent the system from moving towards the goal by picking $d = -1$, but it cannot do this forever as this would violate the assumptions. The second sub-strategy for reaching $x > 5$ works analogously.

2 Related Work

Methods for solving infinite state reactive synthesis problems also referred to as reactive synthesis modulo theories, can be grouped into three categories: abstraction based, fixpoint solvers, and game solving using SMT. Abstraction-based methods have been proposed for temporal stream logic[16] and work by translating the infinite state problem to a propositional LTL synthesis problem. Any solution to the propositional synthesis problem can be translated back into the original infinite state setting. However, the translation is an underapproximation in the sense that the propositional formulation may be unrealizable although there exists a system that fulfills the original specification. Thus, the translation may need to be refined. This approach can be extended from uninterpreted functions to theories and arithmetic by computing the abstraction ahead of time using syntax-guided synthesis [12] or by using lazy abstraction and refinement [20]. Abstraction-based methods have been used for the related problem of synthesis of LTL modulo theories [25], which uses infinite inputs and outputs, but cannot express assignments or other relations between variables over successive time steps. Another approach is to encode the synthesis problem into a constraint solver. This can be versions of constrained Horn clauses [3, 15] or solvers for fixpoint logic [30, 29].

The third category is to implement iterative fixpoint loops and use SMT solvers to represent state sets. This has been done for safety games [26] as well as for games with Büchi and LTL winning conditions [27]. Acceleration for fixpoint computations has been proposed to handle unbounded strategy loops [18]. This is done by representing the specification as a reactive program game that

combines finite states with infinite-state variables. During game solving acceleration lemmas are learned to express properties such as if this transition is taken infinitely often the guard of another transition will eventually be true.

The method presented in this paper also implements an iterative fixpoint loop, using an SMT solver. It presents improvements to the computation of enforceable predecessor and the minimization of strategies that the aforementioned papers do not consider. These techniques can be combined with existing techniques such as acceleration.

The use of different decision procedures for synthesis for Boolean safety specifications was studied in detail in [7, 8]. It was shown that careful implementations of SAT-based methods are competitive with BDD-based methods. Nevertheless, these approaches have not been widely adopted in the Boolean domain.

Minimizing controllers produced by synthesis tools has been done by dtControl [1] which is based on decision tree learning. Our tool on the other hand is based on simplification.

3 Synthesis Method

3.1 Problem Definition

The definition of our problem builds on a theory such as real arithmetic in which satisfiability is decidable and there are effective means to perform quantifier elimination.

A synthesis problem has a set $\mathbb{V}$ of variables partitioned into a set $\mathbb{X}$ of state variables and a set $\mathbb{I}$ of input variables. We will not define the output variables separately but will rather assume that the environment can observe (a subset of) the state variables. Let V, X, and I be the sets of all valuations of $\mathbb{V}$, $\mathbb{X}$, and $\mathbb{I}$, respectively. A valuation $x \in X$ is also called a *state*. We will represent systems and specifications symbolically. For a set of variables $\mathbb{V}$, we will denote a set of primed copies of the variables by $\mathbb{V}'$. We will use primed copies to denote the values of the variables in the "next state".

An *update* u assigns to every variable $\mathbf{x} \in \mathbb{X}$ an expression $u(\mathbf{x})$ over $\mathbb{V}$ from the theory. For an expression e, let $e[x]$ be the value of e in state x. We write $u[x] \in X$ for the state that results from executing u in state x, i.e., for each $\mathbf{x} \in \mathbb{V}$, we assign $e[x]$ to $\mathbf{x}$, where e is the expression $u(\mathbf{x})$. We will represent an update u symbolically as $\mathbf{u}$, which we define as $\bigwedge_{\mathbf{x} \in \mathbb{X}} \mathbf{x}' = u(\mathbf{x})$.

A *guard* is a predicate over $\mathbb{V}$ from the theory. An *action* a is a pair (g, u), where g is a guard and u is an update. We will also represent an action $a = (g, u)$ symbolically as $\mathbf{a} = g \wedge \mathbf{u}$.

A *system* is a tuple $M = (\mathbb{X}, \mathbb{I}, init, A)$, where $\mathbb{X}$ are the state variables, $\mathbb{I}$ are the input variables, $init \subseteq X$ is the set of initial states, and A is the set of system actions. Let $\sigma_s = \bigvee_{a \in A} \mathbf{a}$ be the transition relation formed by the actions in A. A sequence $\pi = x_0, i_0, x_1, i_1, \ldots$ with states $x_t \in X$ and inputs $i_t \in I$ is a *trace* of M if $x_0 \in init$ and for all t there is an action $(g, u) \in A$ such that $x_t \models g$ and $x_{t+1} = u(x_t)$.

Algorithm 1 GR(1) fixpoint algorithm from [9]

```
1: Z = True
2: while Z changed do
3:     for j ∈ [1, n] do
4:         Y = False
5:         r = 0
6:         while Y changed do
7:             start = (J_j^s ∧ pre(Z)) ∨ pre(Y)
8:             Y = False
9:             for i ∈ [1, m] do
10:                X = Z
11:                while X changed do
12:                    X = start ∨ (¬J_i^e ∧ pre(X))
13:                end while
14:                Y = Y ∨ X
15:                mX[j][r][i] = X
16:            end for
17:            mY[j][r] = Y
18:            r += 1
19:        end while
20:        Z = Y
21:    end for
22: end while
23: return Z
```

An infinite-state GR(1) specification is a 6-tuple $\mathcal{G} = (\mathbb{X}, \mathbb{I}, init, \sigma_e, A, \phi)$, where $\mathbb{X}$, $\mathbb{I}$, $init$, and A are as before; $\sigma_e \subseteq X \times I$ is the environment transition relation, and ϕ is a GR(1) winning condition of the form $\bigwedge_i \square\lozenge J_i^e \implies \bigwedge_j \square\lozenge J_j^s$, where J_j^s and J_i^e are sets of states, i.e., subsets of X.

Intuitively speaking, the synthesis problem for an infinite-state GR(1) specification $\mathcal{G} = (\mathbb{X}, \mathbb{I}, init, \sigma_e, A, \phi)$ is to find a system $M = (\mathbb{X}', \mathbb{I}, init', A')$ that fulfills it. We let $\mathbb{X}' \supseteq \mathbb{X}$ and require that $init' \restriction \mathbb{X} = init$. Thus, we allow additional state variables. However, we require that the variables in $\mathbb{X}$ must be changed according to the set A defined in the specification. That is, for any trace $x_0, i_0, \ldots$ of M and for all t we have that there is an action $(g, u) \in A$ such that $x_t \restriction \mathbb{X} \models g$ and $x_{t+1} \restriction \mathbb{X} = u(x_t \restriction \mathbb{X})$. As a second restriction, we require that the system never deadlocks. That is, for any sequence $x_0, i_0, \ldots, x_k, i_k$ where for all t we have $(x_t, i_t) \in \sigma_e$ there must be at least one action $(g, u) \in A'$ such that $x_k, i_k \models g$. Finally, for any trace $x_0, i_0, \ldots$ of M we require that if for each t, $(x_t, i_t) \in \sigma_e$ and for all i there are infinitely many t such that $x_t \models J_i^e$, then for all j there are infinitely many t such that $x_t \models J_j^s$.

3.2 Game Solving

In this section, we describe the GR(1) fixpoint. Except for the enforceable preimage computation, it is the same as in the Boolean case [9]. We describe how to implement the enforceable predecessor in the next section.

The algorithm relies on the repeated application of the enforceable predecessor operation

$$pre(Y) = \forall i \in I : (\sigma_e(x, i) \implies \exists x' \in X : \sigma_s(x, i, x') \wedge x' \in Y').$$

For a set of states $Y \subseteq X$, the enforceable preimage operator returns all states for which the system can ensure that a state in Y is visited in one step, i.e., all states such that for every environment input from σ_e the system has a response in σ_s that results in a state in Y.

Algorithm 1 describes the fixpoint algorithm for GR(1) games. The algorithm consists of three nested fixpoints. The outermost loop (Line 2) computes the greatest fixpoint that computes the states in the winning region, i.e., the states for which the system can guarantee that the specification is satisfied. In the next layer, a least fixpoint (Line 6) for each justice guarantee (Line 3) is used to compute how the system can reach the next guarantee while remaining inside the winning region. The iterates used to reach each justice guarantee are stored in mY, which is used when constructing a winning strategy.

The innermost layer handles the justice assumptions. Instead of reaching the next goal, the system can also win if the system does not fulfill one of the justice assumptions indefinitely. For each assumption (Line 9), a greatest fixpoint (Line 11) is computed that encodes the states for which the environment does not fulfill some assumption. The results of this computation are stored in mX and used as part of the strategy construction.

If the algorithm terminates, the set Z contains the winning region for which the system player can guarantee to satisfy the specification.

3.3 Enforceable Predecessor

We present several ways to compute the enforceable predecessor operator. In the experimental section, we will evaluate their relative performance. Propositional methods use BDDs to encode sets of states. BDDs provide relatively efficient symbolic operations, especially for the mixed quantifiers used in the enforceable predecessor. However, BDDs can not represent infinite sets. Instead, we use an SMT solver and store the state sets as terms of quantifier-free theories such as linear integer arithmetic or linear real arithmetic [26]. To compute the enforceable predecessor, we must eliminate the quantifiers. This is the main bottleneck of the algorithm.

In Algorithm 2, we present six ways to implement the enforceable predecessor. Our algorithm uses two external functions. First, qe is a function that eliminates the outer quantifiers from a formula, returning an equivalent formula. We use QSAT-based quantifier elimination [4] as implemented in Z3 [23].

Second, simplifyDDA is the simplification algorithm introduced in [13] for software verification. It is a recursive algorithm that simplifies quantifier-free formulas as follows: it takes a formula ϕ to be simplified and a context ψ and returns a formula ϕ' such that $\phi \wedge \psi$ and $\phi' \wedge \psi$ are equivalent. It does this by considering each node in the parse tree in a bottom-up fashion. For a node

Algorithm 2 Implementation choices for enforceable predecessor

```
1: pre_1(Y) =
2:     qe(∀i ∈ I : (σ_e(x, i) ⟹ ∃x' ∈ X : σ_s(x, i, x') ∧ x' ∈ Y))
3:
4: pre_2(Y) =
5:     simplifyDDA(qe(∀i ∈ I : (σ_e(x, i) ⟹ ∃x' ∈ X : σ_s(x, i, x') ∧ x' ∈ Y)))
6:
7: pre_3(Y) =
8:     e = qe(∃x' ∈ X : σ_s(x, i, x') ∧ x' ∈ Y)
9:     qe(∀i ∈ I : (σ_e(x, i) ⟹ e))
10:
11: pre_4(Y) =
12:     e = simplifyDDA(qe(∃x' ∈ X : σ_s(x, i, x') ∧ x' ∈ Y));
13:     simplifyDDA(qe(∀i ∈ I : (σ_e(x, i) ⟹ e)))
14:
15: pre_5(Y) =
16:     e = ⋁_{a∈A} wp(Y, a);
17:     qe(∀i ∈ I : (σ_e(x, i) ⟹ e))
18:
19: pre_6(Y) =
20:     e = simplifyDDA(⋁_{a∈A} wp(Y, a));
21:     simplifyDDA(qe(∀i ∈ I : (σ_e(x, i) ⟹ e)))
```

$\phi = \phi_1 \wedge \phi_2$, it will check whether ϕ is equivalent to either ϕ_1 or ϕ_2. If so, it removes the other subexpression. If not, simplifies ϕ_1 with context $\psi \wedge \phi_2$ and after it finishes simplification of ϕ_1, it similarly simplifies ϕ_2. The case of disjunction is similar, negation is handled by first converting the expression to negation normal form.

We define some versions of the enforceable predecessor using the weakest precondition. Recall that we write actions (g, u) symbolically as $g \wedge \mathbf{u}$. Given an action a and a set of states Y that is represented as a formula, we define the weakest precondition as $\mathrm{wp}(Y, a) = g \wedge Y'$ where Y' is obtained from Y by simultaneously substituting the expression $u(\mathbf{x})$ for $\mathbf{x}$ for every $\mathbf{x} \in \mathbb{V}$. Thus, state $y \in Y$ can be reached from x by action a if $y \in \mathrm{wp}(Y, a)$ and the set of states that can reach any $y \in Y$ using some action is $\bigcup_{a \in A} wp(Y, a)$.

The implementation choices that we consider differ in three dimensions:

1. Are universal and existential quantifiers eliminated in one step or separately?
2. How is the existential elimination performed?
3. Is simplification applied to the intermediate results?

Given these options, we study six variants of the enforceable predecessor.

1. Version pre_1 is the simplest version. It uses a single call to qe to remove the universal and the existential quantifier in one step.

2. Version pre_2 applies simplification on the result of pre_1.
3. Version pre_3 calls the quantifier elimination procedure twice: first to eliminate the existential quantifier and then to eliminate the universal quantifier elimination. This uses QSAT quantifier elimination twice instead of using the recursive QSAT quantifier elimination once as in pre_1.
4. The fourth version is similar to pre_3, but applies simplification after each quantifier elimination.
5. Variant pre_5 uses the fact that there is only a small number of actions from which the system can choose and explicitly constructs the disjunction over the weakest precondition of each action.
6. The final version is like pre_5, but does simplify the formula after expansion and quantifier elimination.

4 Strategy Extraction

The game-solving algorithm from the previous section describes how to compute the winning region for the system. Once a winning region has been obtained, we must construct a strategy that ensures that the system remains within the winning region.

Propositional methods typically construct a symbolic transition relation that corresponds closely to a circuit computing each of the Boolean outputs [9]. In our case, we have a relatively small number of possible actions and thus we construct strategies that make a case distinction between the actions, where we try to minimize the size of the conditions that are used to select the action. We present two versions: a simple and an eager strategy. The simple strategy attempts to find the smallest program, whereas the eager strategy prioritizes actions that get to the next goal faster.

Our method creates strategies in the form of programs that follow the template shown in Listing 2. Angle brackets are used to indicate template parameters that are filled in according to the extracted strategy

Our GR(1) strategy satisfies the system justice guarantees in a round-robin fashion. We use a variable k to keep track of the next goal and we use a substrategy for each goal. (Cf. [28]) The main part of the strategy consists of an infinite loop consisting of three parts. First, the strategy reads the values of the input variables from the environment. Second, the strategy checks whether the current goal g has been reached. We use a while loop to iterate over goals to find the next one that is not currently satisfied. (We use $\oplus$ to denote addition modulo the number of justice guarantees). If all goals are satisfied, any strategy can be chosen; the template picks the predecessor of the current strategy.

Each sub-strategy consists of a cascading if statement with one branch for each action. A *condition* $\gamma(j, l)$ defines if action l can be executed when in substrategy j. The definition of γ depends on whether a simple or eager strategy is used. We describe how γ is constructed from the results of the fixpoint algorithm in the next section.

Listing 2: Strategy Template

```
k := 0
while(True) {
    readInputs()
    k' := k
    while( <⋁_j (k = j ∧ J_j^s)> ∧
           k' ≠ k ⊕ 1 ) {
        k := k ⊕ 1
    }
    ...
    if( k ==< j >) {
        if(<γ(j,1)>) {
            <α_1>
        } ...

        else if (<γ(j,l)>) {
            <α_l>
        } ...
        else {
            <α_k>
        }
    }
    ...
    sendOutputs()
}
```

4.1 Simple Strategy

To define the conditions $\gamma(j,l)$, we introduce some notation. Recall that $mY[j][r]$ is the rth iteration of the least fixpoint loop leading to J_j^s. That is, J_j^s can be reached from a state in $mY[j][r]$ in r steps. There are inputs for which the r steps are necessary, but there may also be inputs for which the goal can be reached in fewer steps. We define $mY[j][< r] = \bigvee_{0 \leq r' < r} mY[j][r']$ to be set of all states from which J_j^s can be reached in fewer than r steps and $mY[j][= r] = mY[j][r] \wedge \neg \bigvee_{0 \leq r' < r} mY[j][r']$.

Intuitively, $mX[j][r][i]$ is the set of states where the system is r steps away from reaching J_j^s and is waiting for the environment to fulfill J_i^e to take another step. Let $mX[j][\prec (r,i)]$ be equal to $\bigvee_{r'} \bigvee_{i'} mX[j][r'][i']$, where (r',i') is lexicographically smaller than (r,i) i.e. $r' < r$ or $r' = r$ and $i' < i$. Intuitively, this is the set of states that are either fewer than r steps from the next goal J_j^s, or at least are waiting for a smaller environment condition. Let $mX[j][= (r,i)] = mX[j][r][i] \wedge \neg mX[j][\prec (r,i)]$

For each action a_l, we will define a set of rules of the form $S_o \wedge \sigma_e \wedge \mathrm{wp}(S_t, a_l)$. Given a set of origin states S_o and a set of target states S_t, these rules compute the subset S of S_o such that for all $s \in S$, a_l reaches target states, assuming that the environment selects an input according to its transition relation σ_e. In the following, S_o and S_t are constructed from the specification and the iterates computed by the fixpoint.

For a goal J_l^s and an action a_l, we construct the condition

$$\gamma(j,l) = \gamma_1(l) \vee \gamma_2(j,l) \vee \gamma_3(j,l) \tag{5}$$

from three parts γ_1, γ_2, and γ_3.

The first part is defined as

$$\gamma_1(l) = (\bigwedge_j J_j^s) \wedge \rho_e \wedge \mathrm{wp}(Z, a_l).$$

This part concerns the case where the system simultaneously satisfies all guarantees. In this case, the system does not need to make progress towards any guarantee and can produce any output as long as it remains in the winning region Z.

Next,

$$\gamma_2(j,r,l) = mY[j][=r] \wedge \rho_e \wedge \mathrm{wp}(mY[j][<r], a_l) \text{ and}$$
$$\gamma_2(j,l) = \bigvee_{r \geq 1} \gamma_2(j,r,l).$$

These equations define when action l can make progress towards the goal of sub-strategy j. The function $\gamma_2(j,r,l)$ defines that when system is in a state that was added in the r-th iteration of the Y fixpoint, it has to move into one of the state sets from an earlier iteration ($mY[j][<r]$) to ensure progress towards J^s_j. Equation $\gamma_2(j,l)$ combines the conditions for all iterations r for a specific sub-strategy. Note that if $r = 0$, the goal is reached and the strategy template will start progress towards the next goal.

Finally,

$$\gamma_3(j,r,i,l) = mX[j][=(r,i)] \wedge \neg J^e_i \wedge \rho_e \wedge \mathrm{wp}(mX[j][r][i], a_l) \text{ and}$$
$$\gamma_3(j,l) = \bigvee_r \bigvee_i \gamma_3(j,r,i,l).$$

This equation handles the situation where the environment does not satisfy one of its assumptions. The function $\gamma_3(j,r,i,l)$ defines that if the system is in $mX[j][r][i]$ but not in a set with a smaller index, and the assumption J^e_i is violated, it can use an action that does not undo progress by remaining in $mX[j][r][i]$. Combining the $\gamma_3(j,r,i,l)$ conditions for all iterations r and assumptions i results in $\gamma_3(j,l)$.

4.2 Eager Strategy

The strategy template is biased towards actions with low indices. For instance, action a_1 will be chosen whenever possible, even if another action makes more progress towards the next justice constraint. In the following, we adapt the construction of the conditions to be *eager*, i.e., to always choose the action that moves closest to the current goal. Such strategies are often preferable because they force actions to be taken as soon as possible and not to be delayed unnecessarily. The drawback of this strategy is that it allows for less nondeterminism and thus leaves less room for optimization.

The constructions for γ_1 and γ_3 are unchanged as they are not related to making progress. The condition γ_2 is changed to only allow the action with the

most progress:

$$\begin{aligned}
\gamma_2'(j,r,r',l) &= mY[j][=r] \wedge \rho_e \wedge \mathrm{wp}(mY[j][<r'], a_l),\\
\gamma_2'(j,r,r') &= \bigvee_l \gamma_2'(j,r,r',l),\\
\gamma_2'(j,r,l) &= \bigvee_{r'\le r} \gamma_2'(j,r,r',l) \wedge \neg\gamma_2'(j,r,r'-1), \text{ and}\\
\gamma_2'(j) &= \bigvee_{r\ge 1} \bigvee_l \gamma_2'(j,r,l).
\end{aligned} \tag{6}$$

The new function $\gamma_2'(j,r,r',l)$ encodes when a_l can move from the state set in iteration r to the states in iteration r' for $r' \le r$. We define $\gamma_2'(j,r,r')$ as the set of states from which the system can use some action to move to iteration r'. Using these definitions, $\gamma'(j,r,l)$ defines the states from which a_l makes the most progress for given j and r, by requiring that it makes progress to r' and no action can make more progress. The function $\gamma_2'(j)$ combines all conditions from $\gamma_2'(j,r,l)$ and encodes the set of states where progress toward the goal of sub-strategy j can be made.

Finally, we modify γ to use γ_3 only if no condition in γ_2 is satisfied:

$$\gamma'(j,l) = \gamma_1(l) \vee \Big(\bigvee_{r\ge 1} \gamma_2'(j,r,l)\Big) \vee \Big(\neg\gamma_2'(j) \wedge \bigvee_r \gamma_3(j,r,l)\Big).$$

The resulting strategy always selects an action that makes the most progress. The non-determinism that results when multiple actions are equally good is resolved by the ordering in the if-else statement in the template.

4.3 Simplifying Strategies

The strategies defined above can result in very large conditions with many unnecessary checks. We will here describe our minimization procedure. The procedure is comparatively simple but, as we will see, it leads to a much smaller, readable program. The algorithm is shown in Algorithm 3. It uses the winning region Z, the environment transition relation σ_e, and the set $\gamma(j,l)$ of conditions as inputs, and computes a new set $\hat{\gamma}(j,l)$ of optimized conditions.

The strategies defined above can have overlapping conditions. If a condition is implied by the union of all other conditions, it can be set to false and the corresponding update can be removed. This optimization is done in Lines 7–10, where SAT denotes a call to a decision procedure to decide satisfiability of a theory formula.

If the environment does not fulfill its transition relation σ_e, the system trivially fulfills the specification, no matter what its future actions are. Thus, the truth values of the conditions are irrelevant if σ_e is false. Similarly, assuming that the environment fulfills its transition relation, the system is guaranteed to stay in the winning region and the truth values of the conditions are irrelevant outside of Z. We can thus use $Z \wedge \sigma_e$ as a context for optimization. In this way,

Algorithm 3 Simplify strategy conditions

```
1: for j ∈ [1, n] do
2:     simplifyConditions(γ, j, Z ∧ σ_e)
3: end for
4:
5: procedure simplifyConditions(γ, j, ctx)
6:     for l ∈ [1, k] do γ̄(j, l) = γ(j, l) end for
7:     for l ∈ [1, k] do
8:         if ¬SAT(γ̄(j, l) ∧ ¬⋁_{l'≠l} γ̄(j, l')) then
9:             γ̄(j, l) = False
10:        end if
11:    end for
12:    for l ∈ [1, k] do
13:        γ̂(j, l) = simplifyDDA(γ̄(j, l), ctx ∧ ⋀_{l'∈[1,l−1]} ¬γ̂(j, l'))
14:    end for
15: end procedure
```

we can remove unnecessary checks from the conditions, in particular, the explicit checks for σ_e and the implied checks for Z in Eq. (5) and Eq. (6).

Similarly, we can use the order in which the conditions are checked to optimize them, because for any condition in an else-if statement, all previous if conditions are false. We use this fact in Lines 12–13 to provide a minimization context for a call to simplifyDDA.

As an additional post-processing step (not shown), we remove all branches for which the condition is false. Similarly, we can remove any checks that read "**if** true" and all subsequent checks.

5 Evaluation

We implemented our approach as a prototype in Python using the Z3 SMT solver [23] to represent the state space. Quantifier elimination is done by using Z3's QSAT-based quantifier elimination [4] (tactic `qe2`). For the nested quantifiers in pre_1 and pre_2 we use the recursive version of the same algorithm (tactic `qe_rec`). Afterward, Z3's simplifier (tactic `ctx-solver-simplify`) is used to reduce the size of the formula. As this operation is fast it is always enabled. For simplifyDDA [13] we use our own Python implementation which uses Z3 to check for redundancies. Our prototype implementation gr1mT is available online[1].

To evaluate our implementation we use benchmarks from the tools ConSynth [3] and Raboniel [20] as well as new benchmarks. We compare our implementation against gensys-ltl [27], rpgsolve [18], and Raboniel [20] state-of-the-art tools for infinite-state reactive synthesis. We do not compare against ConSynth as this tool is no longer available. The evaluation is split into three parts, first, we give a short description of the used benchmarks, then we evaluate the runtime

[1] `https://doi.org/10.5281/zenodo.13132527`

for solving the games, and finally, we evaluate the runtime and resulting code size for complete synthesis including strategy extraction and simplification.

5.1 Benchmarks

In the Cinderella game [10, 19] five buckets with a maximum capacity of C are arranged in a circle. On the environment's turn, it can add one unit of water distributed over the buckets however it chooses. On the system's turn, it picks two buckets next to each other and completely empties them. The goal is for all buckets to remain below the maximum capacity. Two variations with more complex winning conditions have been proposed in [3]. One of them uses a Büchi winning condition where the capacity is smaller but only has to be met infinitely often instead of always. The other is similar but uses a parity condition which we could not translate into GR(1). The Repair-Critical benchmark is used in the version from [3]. The system has to find a scheduling for two programs that fulfills four properties of the form $\square(\varphi \implies \lozenge\psi)$. Properties of this shape cannot directly be expressed in GR(1), but one can write a simple two-state deterministic Büchi automaton for it. We introduce an additional state variable for each of these automata, encode the automaton transitions as part of the environment, and add a guarantee to require that the accepting state of the automaton is visited infinitely often.

The next set of benchmarks was originally presented with the Raboniel tool[20]. The elevator benchmarks consist of an elevator serving n floors the system can move up, down, or remain stationary. Each floor has to be visited infinitely often. In the elevator signal benchmarks the environment chooses the target floor. The water tank benchmarks are adapted from an example in [2]. The safety benchmark consists of two coupled tanks with a safety property. The liveness variant consists only of one tank and liveness property. In the sort benchmark, a fixed number of integers has to be sorted by swapping neighboring values.

Finally, we include new benchmarks with more complex dynamics and GR(1) specifications. The example presented in the introduction is included as intro example. Tanks noise and tanks tap are more advanced versions of the watertank benchmark. Both consist of two coupled tanks, a liveness property. In the noise version, the environment can manipulate the water level in both tanks. In the tanks tap the environment controls the outflow valve. The disturbances of the environment are bounded by d. The last benchmark is called car-1D it models a vehicle on a straight road with position, velocity, and acceleration. Similar to the example in the introduction the goal is to drive between the two ends of the arena without overshooting. The system has to select an acceleration value, velocity and position are computed from that. The environment can apply bounded disturbances to the acceleration input of the system, both positive and negative disturbances have to be used infinitely often. In Table 1 all benchmarks are listed with: the type of state and input variables, the number of assumptions, the number of guarantees, and whether it is realizable.

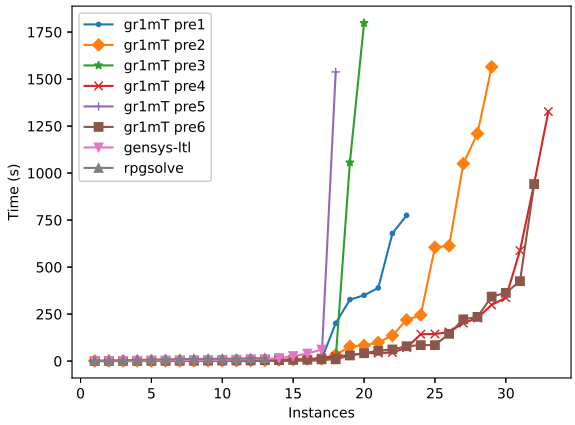


Fig. 1: Experimental results for computing the winning region. The timeout is 30 minutes. The plot shows how many instances can be solved in a given time.

5.2 Game Solving

In the first experiment, we evaluate the game-solving performance of different enforceable predecessor implementations in our tool and compare it to gensys-ltl and rpgsolve. For this task, a tool has to determine which player wins the game and in case the system wins it has to produce its winning region. The runtime for all tools is without extracting a strategy and creating an executable program. For gensys-ltl we report the runtime of its best configuration. In cases where we were unable to translate a benchmark to a certain tool, the result is given as a dash. Raboniel is not included in this table, because it always generates a system and cannot be used for realizability checking only. The results are shown in Fig. 1 and Table 1.

We observe that for safety specifications without assumptions or guarantees, gensys-ltl is the fastest. This is to be expected as gensys-ltl uses a specialized safety game solver that consists of a single greatest fixpoint. Nevertheless, for the two realizable safety benchmarks our best configuration is only half a second slower which is still very competitive. The exception is the unrealizable Cinderella example which is only solved by gensys-ltl. Our approach diverges in this case and does not find a fixpoint even after twice as many fixpoint iterations as gensys-ltl uses. This is also true for much lower capacities. We suspect that this divergence is because our tool uses the GR(1) fixpoint loop instead of a simple safety game solver.

For all benchmarks that have at least one assumption or guarantee a configuration of our tool is the fastest. Using pre_5 which uses expansion and universal quantifier elimination without simplification is the fastest configuration for benchmarks that can be solved in less than five seconds. The baseline configu-

ration of our tool pre_1 is only slightly slower. These benchmarks have relatively simple transition functions and the overhead of using the simplifier does not payoff. Note however that even the configurations using simplify are often faster than the other two tools. The elevator benchmark highlights the benefits of using a GR(1) solver, it has one guarantee for each floor. Our tool can handle these properties directly where gensys-ltl and rpgsolve have to encode them as a single Büchi condition. For an elevator with 10 floors, our tool is 10x faster than gensys-ltl and 6x faster than rpgsolve. Another huge difference can be seen in the Cinderella GF benchmark where our tool is 50x faster than gensys-ltl and 4200x faster than rpgsolve. As this is a simple Büchi property this result has to be attributed to the more efficient computation of the predecessor function.

One outlier is the benchmark repair-critical which is only solved by our tool using pre_5. It consists of complicated transition functions that use mostly equalities and a complex temporal specification. This is the only benchmark where a large amount of runtime is spent doing equality checks (45% instead of 1%).

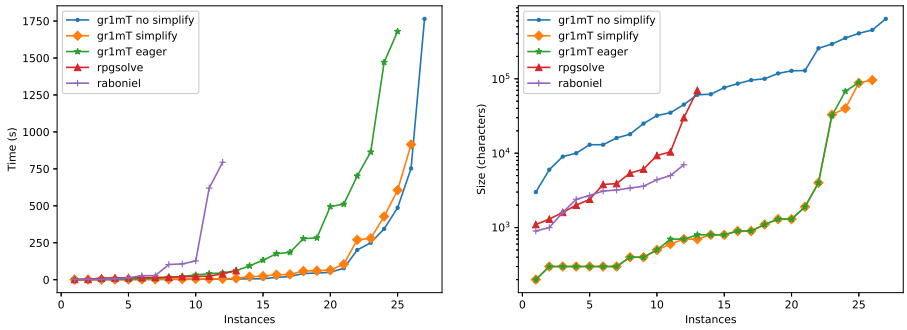
For the new cyber-physical system (CPS) examples the simplifyDDA procedure provides a large benefit. Many instances are only solved by pre_4 and pre_6, which one of them is faster depends on the concrete benchmark. Neither gensys-ltl nor rpgsolve were able to solve any of the new CPS benchmarks in the allocated time.

We can conclude that our GR(1) solver performs very well on both old and new benchmarks. It is only outperformed on safety games by a dedicated safety game solver. In terms of configurations pre_5 performs best on smaller examples and complex integer benchmarks, whereas pre_4 and pre_6 perform best on complex CPS examples.

5.3 Strategy Extraction

In the second experiment, we evaluate the runtime for complete synthesis and the size of the produced programs. We use our tool with predecessor function pre_4 and three strategy extraction methods: simple strategies without simplification, simple strategies with simplification, and eager strategies with simplification. This is compared to rpgsolve and Raboniel. We cannot compare with gensys-ltl as it does not implement strategy extraction. Both rpgsolve and Raboniel are used with strategy simplification enabled. The results are shown in Fig. 2 and Table 2. For each tool and configuration we list the runtime in seconds and the code size as a multiple of 1000 characters. We opted to compare program size in terms of characters as this way we can compare different output formats. To get results that are as comparable as possible we only count the transition function without any headers that are the same for all programs. However, differences in the encoding can still have some influence on the code size.

We can observe that for our GR(1) tool exporting a raw strategy has only minimal overhead compared to computing the winning region. However, these strategies are huge. Applying our simplification algorithm moderately increases the runtime and reduces the code size up to 2900x. In most cases, each substrategy can be simplified to one if statement with a simple condition. This



(a) run time (b) code size

Fig. 2: Experimental results for synthesis including strategy extraction. The time-out is 30 minutes. The plots show how many instances can be solved in a given time and how many produced programs are less than or equal to a given size.

results in very small and human-readable programs. For example, the strategy shown in the introduction was reduced to around 400 characters from originally over 33000. Simplifying eager strategies takes significantly more time than simplifying simple strategies, because the unsimplified eager strategies are even larger. The results are often the same as simple strategies and sometimes larger. When comparing with rpgsolve and Raboniel our tool is both significantly faster and produces smaller results.

6 Conclusion

We presented a method for the synthesis of infinite-state generalized reactivity(1) specifications. It combines the GR(1) fixpoint loop for propositional synthesis with an SMT-based state representation for infinite-state games. To improve the performance of this approach we introduce new methods for implementing the enforceable predecessor, including the usage of advanced simplification techniques. A novel application of simplification techniques allows us to synthesize human-readable programs that are only a few lines long.

In this paper, we only considered benchmarks that can be solved in a finite amount of fixpoint iterations. A promising avenue for future work would be to combine the advantages of GR(1) and simplification with the ability to handle unbounded strategy loops using rpgsolve's acceleration.

Our evaluation shows that all but one of the benchmarks used in the literature could be translated to GR(1). We also show that our GR(1) solver outperforms other state-of-the-art tools in both runtime and code size. This demonstrates that GR(1) can provide similar benefits for infinite-state synthesis as for finite-state synthesis.

A Experimental Results

Benchmark	T.	$\lvert J^e \rvert$	$\lvert J^s \rvert$	R.	pre_1	pre_2	pre_3	pre_4	pre_5	pre_6	gensys	rpg
Cinderella C=2	$\mathbb{R}$	0	0^*	R	5.7	11.7	1.0	2.2	25.5	1.7	0.5	1.9
Cinderella C=1.9(20)	$\mathbb{R}$	0	0^*	U	TO	TO	TO	TO	TO	TO	40.2	TO
Cinderella GF C=1.4	$\mathbb{R}$	0	1	R	TO	33.4	4.7	6.4	1.7	7.3	61.1	TO
Repair-Critical	$\mathbb{Z}$	0	4	R	TO	TO	TO	TO	1538	TO	TO	-
Elevator 3	$\mathbb{Z}$	0	3	R	0.4	0.7	0.6	1.1	0.5	0.8	4.6	6.5
Elevator 4	$\mathbb{Z}$	0	4	R	0.5	0.9	0.8	1.5	0.6	1.2	6.7	7.8
Elevator 5	$\mathbb{Z}$	0	5	R	0.6	1.1	0.9	1.8	0.7	1.4	10.5	8.9
Elevator 8	$\mathbb{Z}$	0	8	R	0.9	1.9	1.6	3.5	1.1	2.8	15.9	12.3
Elevator 10	$\mathbb{Z}$	0	10	R	1.2	2.4	2.2	4.3	1.5	3.5	26.7	15.7
Elevator Signal 3	$\mathbb{Z}$	0	1	R	0.8	1.4	0.9	3.8	0.7	1.9	3.3	11
Elevator Signal 4	$\mathbb{Z}$	0	1	R	0.9	1.7	1.2	5.1	0.8	2.8	4.1	11
Elevator Signal 5	$\mathbb{Z}$	0	1	R	1.1	3.0	1.5	7.6	1.0	3.4	4.9	12
Elevator Signal 8	$\mathbb{Z}$	0	1	R	2.0	5.5	3.0	16.8	1.5	6.5	7.8	13
Elevator Signal 10	$\mathbb{Z}$	0	1	R	3.6	12.7	4.7	25.7	1.7	9.5	10	15
Watertanks-safety	$\mathbb{R}$	0	0^*	R	0.5	0.7	0.5	0.8	0.5	0.7	0.2	0.9
Watertank-liveness	$\mathbb{R}$	0	1	R	0.6	0.9	0.7	1.3	0.6	1.1	3.3	3.2
Sort 4	$\mathbb{Z}$	1	$0^\dagger$	R	0.6	0.9	0.7	1.4	0.6	1.2	1.3	-
Sort 5	$\mathbb{Z}$	1	$0^\dagger$	R	1.3	3.3	1.8	5.0	0.9	5.2	1.4	-
Intro example	$\mathbb{R}$	2	2	R	1.5	2.6	2.3	4.9	1.6	5.0	TO	-
Tanks noise d=0.01	$\mathbb{R}$	0	1	R	201	136	1056	46	TO	62	TO	TO
Tanks noise d=0.011	$\mathbb{R}$	0	1	R	389	219	TO	71	TO	85	TO	TO
Tanks noise d=0.0115	$\mathbb{R}$	0	1	R	TO	1210	TO	229	TO	235	TO	TO
Tanks noise d=0.015	$\mathbb{R}$	0	1	U	TO	TO	TO	945	TO	TO	TO	TO
Tanks noise d=0.02	$\mathbb{R}$	0	1	U	TO	605	TO	202	TO	343	TO	TO
Tanks tap d=0.05	$\mathbb{R}$	2	1	R	327	83	1798	42	TO	42	TO	-
Tanks tap d=0.1	$\mathbb{R}$	2	1	R	350	78	TO	43	TO	55	TO	-
Tanks tap d=0.155	$\mathbb{R}$	2	1	R	TO	1565	TO	299	TO	941	TO	-
Tanks tap d=0.16	$\mathbb{R}$	2	1	U	679	245	TO	143	TO	363	TO	-
Tanks tap d=0.2	$\mathbb{R}$	2	1	U	9.1	8.3	12.6	8.2	TO	9.3	TO	-
Car-1D d=0.1	$\mathbb{R}$	2	2	R	TO	1050	TO	144	TO	85	TO	-
Car-1D d=0.2	$\mathbb{R}$	2	2	R	TO	TO	TO	337	TO	145	TO	-
Car-1D d=0.3	$\mathbb{R}$	2	2	R	TO	TO	TO	588	TO	222	TO	-
Car-1D d=0.352	$\mathbb{R}$	2	2	R	TO	TO	TO	1327	TO	425	TO	-
Car-1D d=0.353	$\mathbb{R}$	2	2	U	TO	613	TO	155	TO	79	TO	-
Car-1D d=0.4	$\mathbb{R}$	2	2	U	775	99	TO	31	TO	30	TO	-

Table 1: Experimental results for computing the winning region. For each problem, we indicate the type of the variables, the number of assumptions, the number of guarantees, and whether it is realizable. The guarantee column uses 0^* for a single guarantee *true* and $0^\dagger$ for *false*. These are required to get syntactically correct GR(1) conditions. The runtime is given in seconds and the timeout is 30 minutes.

Benchmark	no simplify		simplify		eager		rpgsolve		raboniel	
	time	size	time	size	time	size	time	size	time	size
Cinderella C=2	2.1	13	2.6	0.7	2.3	0.7	2.6	6.1	TO	-
Cinderella GF C=1.4	6.1	293	20.7	33	176	89	TO	-	TO	-
Elevator 3	0.7	6	0.8	0.3	0.9	0.3	14.1	1.1	2.2	1.0
Elevator 4	0.8	10	1.1	0.4	1.7	0.4	16.9	1.6	2.3	1.6
Elevator 5	1.1	16	1.5	0.5	3.1	0.5	22.0	2.0	5.8	2.4
Elevator 8	2.0	45	3.4	0.8	16.2	0.8	39.7	3.9	27.9	3.6
Elevator 10	2.5	76	5.5	0.9	41	0.9	61.3	5.4	103.0	4.4
Elevator Signal 3	3.2	18	4.2	0.7	8.9	0.7	11.9	2.4	27.3	3.4
Elevator Signal 4	4.3	25	6.1	0.8	21.0	0.8	13.5	3.8	127	5
Elevator Signal 5	6.3	32	8.9	0.9	46	0.9	15.2	9.3	795	7
Elevator Signal 8	15.5	61	24.2	1.1	283	1.1	20.5	30	TO	-
Elevator Signal 10	21.6	86	36.0	1.3	702	1.3	24.4	70	TO	-
Watertanks-safety	0.7	3	0.8	0.3	0.7	0.3	1.4	1.3	12.7	3.1
Watertank-liveness	0.9	9	1.8	0.2	7.3	0.2	11.6	10.4	106.8	2.7
Sort 4	1.0	13	3.5	0.4	61	0.4	-	-	2.1	0.9
Sort 5	4.5	62	33.4	0.6	1680	0.8	-	-	620	3.2
Intro example	3.1	35	5.8	0.3	32	0.3	-	-	TO	-
Tanks noise d=0.01	50	96	65	1.3	185	1.3	TO	-	TO	-
Tanks noise d=0.011	78	100	105	1.9	278	1.9	TO	-	TO	-
Tanks noise d=0.0115	250	118	270	4	511	4	TO	-	TO	-
Tanks tap d=0.05	45	128	59	0.3	94	0.3	-	-	TO	-
Tanks tap d=0.1	42	129	61	0.3	132	0.3	-	-	TO	-
Tanks tap d=0.155	344	353	428	0.3	495	0.3	-	-	TO	-
Car-1D d=0.1	201	257	280	40	864	32	-	-	TO	-
Car-1D d=0.2	487	408	606	88	1471	68	-	-	TO	-
Car-1D d=0.3	753	453	915	96	TO	-	-	-	TO	-
Car-1D d=0.352	1765	639	TO	-	TO	-	-	-	TO	-

Table 2: Experimental results for synthesis including strategy simplification. Our GR(1) tool uses configuration pre_4 and the three configurations simple strategy without simplify, simple strategy with simplify, and eager strategy with simplify. The runtime is given in seconds (timeout 30 minutes) and the code size is in multiples of 1000 characters.

References

[1] Pranav Ashok et al. "dtControl: decision tree learning algorithms for controller representation". In: *HSCC*. ACM, 2020, 17:1–17:7. DOI: 10.1145/3365365.3382220.

[2] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017. DOI: 10.1007/978-3-319-50763-7.

[3] Tewodros A. Beyene et al. "A constraint-based approach to solving games on infinite graphs". In: *POPL*. ACM, 2014, pp. 221–234. DOI: 10.1145/2535838.2535860.

[4] Nikolaj S. Bjørner and Mikolás Janota. "Playing with Quantified Satisfaction". In: *LPAR (short papers)*. Vol. 35. EPiC Series in Computing. EasyChair, 2015, pp. 15–27. DOI: 10.29007/VV21.

[5] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. "Graph Games and Reactive Synthesis". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 921–962. DOI: 10.1007/978-3-319-10575-8_27.

[6] Roderick Bloem et al. "Reduction of Resolution Refutations and Interpolants via Subsumption". In: *Haifa Verification Conference*. Vol. 8855. Lecture Notes in Computer Science. Springer, 2014, pp. 188–203. DOI: 10.1007/978-3-319-13338-6_15.

[7] Roderick Bloem et al. "SAT-based methods for circuit synthesis". In: *FMCAD*. IEEE, 2014, pp. 31–34. DOI: 10.1109/FMCAD.2014.6987592.

[8] Roderick Bloem et al. "Satisfiability-Based Methods for Reactive Synthesis from Safety Specifications". In: *CoRR* abs/1604.06204 (2016). arXiv: 1604.06204.

[9] Roderick Bloem et al. "Synthesis of Reactive(1) designs". In: *J. Comput. Syst. Sci.* 78.3 (2012), pp. 911–938. DOI: 10.1016/J.JCSS.2011.08.007.

[10] Marijke H. L. Bodlaender et al. "Cinderella versus the Wicked Stepmother". In: *IFIP TCS*. Vol. 7604. Lecture Notes in Computer Science. Springer, 2012, pp. 57–71. DOI: 10.1007/978-3-642-33475-7_5.

[11] Anca Browne et al. "An Improved Algorithm for the Evaluation of Fixpoint Expressions". In: *Theor. Comput. Sci.* 178.1-2 (1997), pp. 237–255. DOI: 10.1016/S0304-3975(96)00228-9.

[12] Wonhyuk Choi et al. "Can reactive synthesis and syntax-guided synthesis be friends?" In: *PLDI*. ACM, 2022, pp. 229–243. DOI: 10.1145/3519939.3523429.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. "Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis". In: *SAS*. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 236–252. DOI: 10.1007/978-3-642-15769-1_15.

[14] Rüdiger Ehlers and Vasumathi Raman. "Slugs: Extensible GR(1) Synthesis". In: *CAV (2)*. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 333–339. DOI: 10.1007/978-3-319-41540-6_18.

[15] Marco Faella and Gennaro Parlato. "Reachability Games Modulo Theories with a Bounded Safety Player". In: *AAAI*. AAAI Press, 2023, pp. 6330–6337. DOI: 10.1609/AAAI.V37I5.25779.

[16] Bernd Finkbeiner et al. "Temporal Stream Logic: Synthesis Beyond the Bools". In: *CAV (1)*. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 609–629. DOI: 10.1007/978-3-030-25540-4_35.

[17] Elizabeth Firman, Shahar Maoz, and Jan Oliver Ringert. "Performance heuristics for GR(1) synthesis and related algorithms". In: *Acta Informatica* 57.1-2 (2020), pp. 37–79. DOI: 10.1007/S00236-019-00351-9.

[18] Philippe Heim and Rayna Dimitrova. "Solving Infinite-State Games via Acceleration". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 1696–1726. DOI: 10.1145/3632899.

[19] Antonius JC Hurkens, Cor AJ Hurkens, and Gerhard J Woeginger. "How Cinderella won the bucket game (and lived happily ever after)". In: *Mathematics Magazine* 84.4 (2011), pp. 278–283. DOI: 10.4169/math.mag.84.4.278.

[20] Benedikt Maderbacher and Roderick Bloem. "Reactive Synthesis Modulo Theories using Abstraction Refinement". In: *FMCAD*. IEEE, 2022, pp. 315–324. DOI: 10.34727/2022/ISBN.978-3-85448-053-2_38.

[21] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. "Strix: Explicit Reactive Synthesis Strikes Back!" In: *CAV (1)*. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 578–586. DOI: 10.1007/978-3-319-96145-3_31.

[22] Thibaud Michaud and Maximilien Colange. "Reactive synthesis from LTL specification with Spot". In: *Proceedings of the 7th Workshop on Synthesis, SYNT@CAV 2018*. 2018.

[23] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[24] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) Designs". In: *VMCAI*. Vol. 3855. Lecture Notes in Computer Science. Springer, 2006, pp. 364–380. DOI: 10.1007/11609773_24.

[25] Andoni Rodríguez and César Sánchez. "Boolean Abstractions for Realizability Modulo Theories". In: *CAV (3)*. Vol. 13966. Lecture Notes in Computer Science. Springer, 2023, pp. 305–328. DOI: 10.1007/978-3-031-37709-9_15.

[26] Stanly Samuel, Deepak D'Souza, and Raghavan Komondoor. "GenSys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces". In: *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 1585–1589. DOI: 10.1145/3468264.3473126.

[27] Stanly Samuel, Deepak D'Souza, and Raghavan Komondoor. "Symbolic Fixpoint Algorithms for Logical LTL Games". In: *ASE*. IEEE, 2023, pp. 698–709. DOI: 10.1109/ASE56229.2023.00212.

[28] Matthias Schlaipfer, Georg Hofferek, and Roderick Bloem. "Generalized Reactivity(1) Synthesis without a Monolithic Strategy". In: *Haifa Verification Conference*. Vol. 7261. Lecture Notes in Computer Science. Springer, 2011, pp. 20–34. DOI: 10.1007/978-3-642-34188-5_6.

[29] Hiroshi Unno et al. "Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 2111–2140. DOI: 10.1145/3571265.

[30] Hiroshi Unno et al. "Program Verification via Predicate Constraint Satisfiability Modulo Theories". In: *CoRR* abs/2007.03656 (2020). arXiv: 2007.03656.