Generalized Reactivity(1) Synthesis without a Monolithic Strategy*

Matthias Schlaipfer, Georg Hofferek, and Roderick Bloem

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Austria.

Abstract. We present a new approach to synthesizing systems from Generalized Reactivity(1) specifications. Our method does not require a monolithic strategy, which can be prohibitively large. Instead, our approach constructs a circuit directly from the iterates of the fixpoint computation that computes the winning region. We build the overall system by combining these circuit parts. Our approach has generally lower memory requirements than previous GR(1) synthesis approaches, and is also faster. In addition to that, the circuits we build are eager, in the sense that they typically fulfill system guarantees faster than the circuits obtained with previous approaches, as experiments show.

1 Introduction

Formal methods have recently seen increased attention focused on synthesis techniques in which programs are created automatically from specifications. Such techniques may create full systems from a specification given, for instance, in temporal logic [7, 17, 16, 10, 11, 4, 19, 15, 8, 18], or they may synthesize pieces of a program that are difficult to implement or were previously implemented incorrectly [20, 22, 14, 23, 12]. Synthesis promises to remove an important burden from the programmer, who only has to think about the specification and not about implementation details. The main drawback currently is the lack of capacity of synthesis tools — they are only applicable to small examples. Time and especially memory use of current tools are often prohibitive, keeping many realistic examples outside of the realm of synthesis.

In this paper we consider an alternative method to generate systems in GR(1) synthesis [16]. Our synthesis method is implemented in RATSY [4], a tool for synthesis of GR(1) properties. GR(1) properties can be viewed as an implication of deterministic Büchi automata. Let A_1 through A_n be the assumptions on the environment and let G_1 through G_m be the guarantees that the system has to fulfill. If $A_1, \ldots, A_n$ and $G_1, \ldots, G_m$ are expressed as deterministic Büchi automata, then the formula

$$\bigwedge_i A_i \implies \bigwedge_j G_j$$

* This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613).

is a GR(1) specification. Previous work [5, 6] has shown that GR(1) is expressive, easy to use, and that it allows for a relatively efficient implementation of synthesis.

The time and memory use of RATSY is high. Moreover, it has the drawback of producing systems that are not very eager. The general strategy used by RATSY is to satisfy the guarantees one at a time in a round-robin fashion. Each of the guarantees is fulfilled using an attractor strategy in which the approach to the guarantee is enforced whenever all of the assumptions have been fulfilled. This formulation of the strategy is very general and allows significant freedom to choose a small implementation. However, since the attractor strategy only requires *some* progress to be made, this formulation allows for very lazy implementations that may take a long time to fulfill a goal.

In this paper we present an alternative to the round-robin strategy, which is *eager*: each goal is achieved as soon as possible. This gives us more desirable, responsive systems, which achieve some robustness against failure by the environment to achieve its liveness goals, at a fraction of the cost of other methods [2, 3]. At the same time, our experimental results show that our new approach reduces time and memory use significantly at the expense of generating larger circuits.

The rest of the paper is organized as follows. In Section 2, we revisit some preliminaries necessary for our method and establish notation. In Section 3, we present our synthesis approach. Section 4 presents specific aspects of our implementation. In Section 5, we summarize our experimental results, and Section 6 concludes the paper.

2 Preliminaries

2.1 Generalized Reactivity

Generalized Reactivity(1) [16], GR(1) for short, is a syntactic restriction of Linear Temporal Logic (LTL). Let $\mathcal{I}$ and $\mathcal{O}$ be sets of (propositional) variables. A GR(1) specification φ is required to be of the form $\varphi = \varphi^e \rightarrow \varphi^s$ where φ^e and φ^s can be written as a conjunction of the following three parts [16]:

- A Boolean formula φ_i^e (φ_i^s, respectively) over $\mathcal{I}$ and $\mathcal{O}$, characterizing the initial states.
- An LTL formula φ_t^e (φ_t^s, respectively), characterizing the transitions of the environment (system, respectively). φ_t^e is of the form $\bigwedge_i \mathsf{G}(B_i)$, where each B_i is a Boolean combination of variables from $\mathcal{I}$ and $\mathcal{O}$, and expressions of the form $\mathsf{X}(v)$, where v is a variable from $\mathcal{I}$ for φ_t^e and from $\mathcal{I} \cup \mathcal{O}$ for φ_t^s.
- An LTL formula φ_f^e (φ_f^s, respectively), characterizing the fairness states for the environment (system, respectively). φ_f^e and φ_f^s are both of the form $\bigwedge_i \mathsf{G}\,\mathsf{F}\,B_i$ for a Boolean formulae B_i over $\mathcal{I} \cup \mathcal{O}$.

It is easiest to think about φ^e and φ^s as encoding the symbolic representation of the product of several Büchi automata. (Extra variables for encoding automata

states can be modeled as extra state variables.) The goal of synthesis is to generate a Mealy machine that satisfies the GR(1) specification. The Mealy machine has inputs $\mathcal{I}$, state variables $\mathcal{I} \cup \mathcal{O}$, and the outputs coincide with (the next state values of) the state variables [16]. A Mealy machine satisfies a GR(1) specification if (1) φ_i^e and φ_i^s are satisfied initially, (2) as long as the environment fulfills φ_t^e, the system fulfills φ_t^s, and (3) if the environment fulfills φ_f^e, the system fulfills φ_f^s.

Synthesizing a correct system from its specification corresponds to finding a strategy in a *game* between the system (protagonist) and the environment (antagonist). Following [16], we define a *game structure* $\langle \mathcal{I}, \mathcal{O}, \Theta, \rho_e, \rho_s, \varphi \rangle$. Here, $\mathcal{I}$ is a set of (Boolean) input variables, which are under the environment's control. Similarly, $\mathcal{O}$ is a set of output variables, under the system's control. We define a *state* to be an interpretation of $\mathcal{I} \cup \mathcal{O}$, assigning to each variable either true or false. We will denote the set of all states with Q. Furthermore, Θ is the initial condition; in our case $\Theta = \varphi_i^e \wedge \varphi_i^s$. The transition relation of the environment is denoted by $\rho_e(\mathcal{I}, \mathcal{O}, \mathcal{I}')$, relating a present state $q \in Q$ to possible values for next inputs $I' \in \mathcal{I}'$, where $\mathcal{I}'$ contains primed copies of the elements in $\mathcal{I}$. In our case, $\rho_e = \varphi_t^e$, where we replace each occurrence of $\mathsf{X}\,v$ with v', representing the "next state" of v. Similarly, $\rho_s(\mathcal{I}, \mathcal{O}, \mathcal{I}', \mathcal{O}')$ is the transition relation of the system, relating a present state $q \in Q$ and a next input I' to possible values for next outputs $O' \in \mathcal{O}'$. We set $\rho_s = \varphi_t^s$, again replacing all occurrences of $\mathsf{X}\,v$ with v'. Finally, φ is the winning condition of the game. We use $\varphi = \varphi^e \rightarrow \varphi^s$. Furthermore, let J_j^G (for $j \in \{1, \ldots, n\}$) and J_i^A (for $i \in \{1, \ldots, m\}$) be the sets of fair states of the system and the environment, characterized by the conjuncts B_i in φ_f^s and φ_f^e, respectively. We will also refer to them as "guarantee states" and "assumption states" respectively.

One step of the game consists of the environment choosing values for the next inputs I', after which the system must choose values for the next outputs O'. The game is won by the system iff the resulting sequence of states in the game graph satisfies the winning condition φ. Informally speaking (and slightly simplified), the system wins the game if it is either able to ensure infinitely many visits to all sets of guarantee states, or if it can prevent the environment from visiting at least one of the sets of assumption states infinitely often.

A *strategy* is a (partial) function that maps the present state and next inputs to next outputs. It is called a *winning strategy* if every play that adheres to it is won by the system. A winning strategy can easily be turned into a correct implementation for the system [16, 6].

2.2 μ-Calculus

For solving games, we use the propositional μ-calculus [13]. Formulae of the μ-calculus are defined recursively. Let Q be the set of all states of a game structure. Furthermore, let $\mathcal{V}$ be a set of variables. Every subset $S \subseteq Q$ and every variable $V \in \mathcal{V}$ is a μ-calculus formula. Let A, B be μ-calculus formulae. Then also $\neg A$, $A \cup B$, and $A \cap B$ are μ-calculus formulae, with the obvious semantics. Moreover, the μ-calculus comprises least and greatest fixpoint formulae, defined as follows.

For a μ-calculus formula P with a free variable $V \in \mathcal{V}$ the following are μ-calculus formulae:

$$\mu V \,.\, P(V) = \bigcup_i V_i, \quad \text{where } V_0 = \emptyset \text{ and } V_{i+1} = P(V_i) \text{ and} \tag{1}$$

$$\nu V \,.\, P(V) = \bigcap_i V_i, \quad \text{where } V_0 = Q \text{ and } V_{i+1} = P(V_i). \tag{2}$$

We extend the classical μ-calculus with a mixed-preimage operator MX, defined as follows:

$$\mathsf{MX}(V) = \{(i, o) \in Q \mid \forall i' \,.\, \exists o' \,.((i, o, i') \models \rho_e \rightarrow (i, o, i', o') \models \rho_s) \wedge (i', o') \in V\}.$$

Semantically, $\mathsf{MX}(V)$ denotes the set of all states from which the system can force the play into a state in V, irrespective of the input i' chosen by the environment.

2.3 Computing the Winning Region of GR(1) Games

Piterman et al. [16] presented a μ-calculus formula for computing the *winning region* of a GR(1) game, i.e., the set of states from which the system can win the game by adhering to a winning strategy. The winning region of such a specification is given by the following triply-nested fixpoint formula [16]:

$$\text{Win} = \nu Z \,.\, \bigwedge_{j=1}^{n} \mu Y \,.\, \bigvee_{i=1}^{m} \nu X \,.(J_j^G \wedge \mathsf{MX}(Z)) \vee \mathsf{MX}(Y) \vee (\neg J_i^A \wedge \mathsf{MX}(X)). \tag{3}$$

Piterman et al. [16] also show how to use the intermediate values of the fixpoint computations in Equation 3 to construct a strategy, consisting of the disjunction of three sub-strategies. These three sub-strategies correspond to the three disjuncts in Equation 3. Sub-strategy ρ_1 is applied when the game has reached a guarantee state in J_j^G. In this case, a counter jx that stores which guarantee should be fulfilled next is incremented (modulo n, the number of guarantees). Sub-strategy ρ_2 takes the play at least one step closer to a state in J_j^G. Sub-strategy ρ_3 ensures that the play stays in a region in which at least one of the sets of assumption states J_i^A is not visited.

Although systems that adhere to these sub-strategies are correct, they are not necessarily *eager*. The strategies only ensure that each step takes the play at least one step closer to a guarantee state, or even stay where they are as long as an assumption state is not reached. However, in many situations the system might have a choice of getting not only one, but several steps closer to a guarantee state. An eager system would always make the choice that takes it as close to the guarantee states as possible. It might even fulfil multiple guarantees at once.

2.4 Relation Determinization

The strategies computed according to [16] are relations, mapping a tuple (i, o, i') of present state inputs i, present state outputs o, and next state inputs i' to possible next state outputs o'. These relations can be represented by a characteristic function, usually in form of a Binary Decision Diagram (BDD). In order to build circuits, we need to extract completely specified functions for each of the Boolean output signals $o_i \in \mathcal{O}$ from this relation.

There are several ways to find functions compatible with a given relation [24, 1, 9, 6, 5]. We will use the approach presented in [6, 5], as it integrates seamlessly with the symbolic algorithms we use. When given a BDD $b(\mathcal{I}, \mathcal{O}, \mathcal{I}', \mathcal{O}')$ we will write $\mathsf{FN}(b)$ to denote a circuit with inputs $\mathcal{I}, \mathcal{O}, \mathcal{I}'$ and outputs $\mathcal{O}'$, such that $\mathsf{FN}(b)(i, o, i') = o'$ only if $b(i, o, i', o') = \mathsf{true}$ or $\neg\exists o' . b(i, o, i', o')$. To emphasize that $\mathsf{FN}(b)$ is a (completely specified) function, we will sometimes write $\mathsf{FN}(b)(\mathcal{I}, \mathcal{O}, \mathcal{I}' \rightarrow \mathcal{O}')$.

3 Onion Rings Approach

Previous work [6, 5] shows that tools implementing GR(1) synthesis spend a lot of time computing the strategy relation. While computing the winning region is comparatively fast, combining the intermediate results of the fixpoint computation to form a monolithic strategy requires a lot of CPU time and memory. We will demonstrate how to build a correct-by-construction circuit directly from the intermediate results of the winning region computation, without having to build a monolithic strategy relation first.

Our new synthesis approach is based on the intermediate results of fixpoint computations, which we call *onion rings*. The name stems from the form of the intermediate results of an attractor computation. Like in an onion, each iteration of the computation adds a layer of states "around" the previous results. We will show how to build two kinds of circuits. The first are *enable circuits* which detect whether we can reach a particular onion ring. The second are circuits that provide correct outputs for the case that we move to the particular enabled onion ring. Before that, however, we will introduce some simple auxiliary circuits that we will need to combine the other parts. In the following, we will denote circuits and combinational gates by upper-case sans-serif letters (e.g., circuit A, B, $\mathsf{C}, \ldots$, gates AND, XOR). For "standard gates" such as AND and XOR, we will use infix notation (e.g. A AND B). BDDs will be denoted by lowercase letters (e.g. $a, b, c, \ldots$). Operations on BDDs will be denoted using the common logic operators such as $\wedge$ and $\exists$. Furthermore, we will write $\mathcal{C}(b)$ to denote a circuit equivalent to b. That is, a circuit having the variables of b as its inputs, and one output that is true if and only if the inputs are in the on-set of b.[1]

[1] Note that it is trivial to construct such a circuit by using one multiplexer for each BDD node.

3.1 Auxiliary Circuits

The first auxiliary circuit we need is called SELECT. It has n one-bit selector inputs $\mathsf{S_1}, \ldots, \mathsf{S_n}$, and n data inputs $\mathsf{F_1}, \ldots, \mathsf{F_n}$, each m bits wide. Furthermore, $\mathsf{SELECT}((\mathsf{S_1}, \ldots, \mathsf{S_n}),\ (\mathsf{F_1}, \ldots, \mathsf{F_n}))$ has an m bits wide data output, which is equal to $\mathsf{F_1}$ if $\mathsf{S_1}$ is true, equal to $\mathsf{F_2}$ if $\mathsf{S_1}$ is false and $\mathsf{S_2}$ is true, equal to $\mathsf{F_3}$ if $\mathsf{S_1}$ and $\mathsf{S_2}$ are false and $\mathsf{S_3}$ is true, etc. I.e., the output equals the input with the lowest index for which the corresponding selector bit is true. If all selector signals $\mathsf{S_1}, \ldots, \mathsf{S_n}$ are false, the output of the SELECT circuit is arbitrary.

The second auxiliary circuit is a comparator. $\mathsf{COMP}(\mathsf{A}, \mathsf{B})$ outputs true if and only if A and B are (bitwise) equal.

3.2 Enable Circuits for Onion Rings

From the computation of the winning region of a GR(1) game, we get a set of BDDs $x[j, r, i]$, each referring to variables in $\mathcal{I} \cup \mathcal{O}$ [16]. The indices i and j range over all the sets of assumption and guarantee states of the specification, respectively. The index r denotes iterations in the computation of the least fix-point over Y in Equation 3. Each of these BDDs symbolically represents a set of states in the GR(1) game. The set represented by $x[j, r, i]$ denotes the set of states from which the system can either (1) enforce moving one step closer to one of the states in the j-th guarantee set (i.e., a state in $x[j, r', i']$ for $r' < r$ and arbitrary i'), or (2) stay in that part of $x[j, r, i]$, which does not share any states with the i-th set of assumption states. We will use primes to denote BDDs that represent sets of "next states" (i.e., referring to variables in $\mathcal{I}' \cup \mathcal{O}'$). For each BDD $x'[j, r, i]$, we build an enable circuit that detects whether a state described by $x'[j, r, i]$ is reachable by obeying the system transition relation ρ_s for some next state output O':

$$\mathsf{EN}[\mathsf{j}, \mathsf{r}, \mathsf{i}](\mathsf{I}, \mathsf{O}, \mathsf{I}') = \mathcal{C}(\exists O' \,.\, \rho_s \wedge x'[j, r, i]) \tag{4}$$

A circuit $\mathsf{EN}[\mathsf{j}, \mathsf{r}, \mathsf{i}]$ outputs true if and only if the tuple $(\mathsf{I}, \mathsf{O}, \mathsf{I}')$ at its input satisfies the system transition relation and gets the system into the set of states represented by $x'[j, r, i]$ under some output O'.

3.3 Function Circuits for Onion Rings

For each onion ring, as described in the previous section, we build a corresponding function circuit $\mathsf{FN}(\mathsf{I}, \mathsf{O}, \mathsf{I}' \rightarrow \mathsf{O}')$ that computes the system's outputs for this particular case:

$$\mathsf{FN}[\mathsf{j}, \mathsf{r}, \mathsf{i}](\mathsf{I}, \mathsf{O}, \mathsf{I}' \rightarrow \mathsf{O}') = \mathsf{FN}(\rho_s \wedge x'[j, r, i]) \tag{5}$$

The enable and function circuits are combined in a way to make maximum progress when approaching a guarantee state J_j^G. That means we want to choose the function corresponding to the minimal (r, i) whose enable circuit outputs

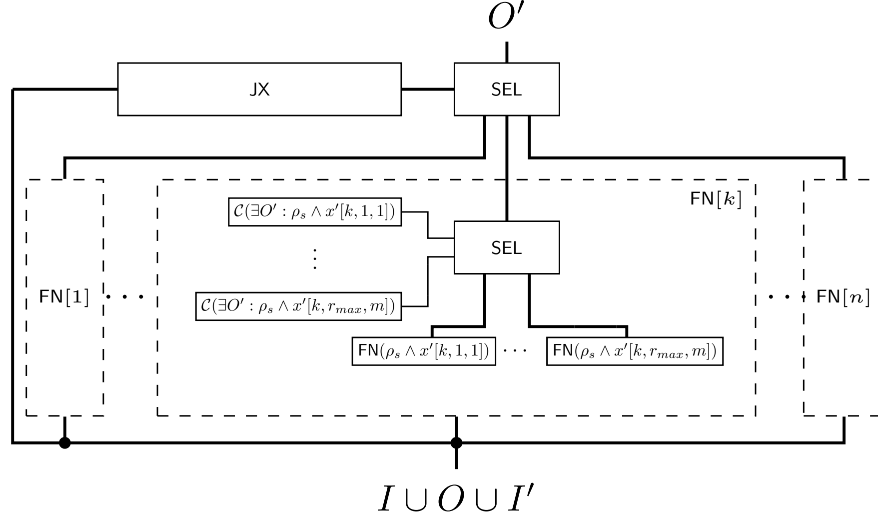


Fig. 1: Diagram of the whole circuit. Dashed boxes symbolize parts of the circuit that are built analogously to the neighboring parts drawn in detail.

true. I.e., we step to the innermost onion ring possible. We use SELECT circuits, as described in Section 3.1, to achieve this by ordering the selector (enable signals) and data inputs (function signals) lexicographically according to (r, i).

$$\begin{aligned} \mathsf{FN[j]} \;=\; \mathsf{SELECT}((&\mathsf{EN[j,1,1]},\ldots,\mathsf{EN[j,1,m]}, \\ &\mathsf{EN[j,2,1]},\ldots,\mathsf{EN[j,2,m]},\ldots, \\ &\mathsf{EN[j,r_{max},1]},\ldots,\mathsf{EN[j,r_{max},m]}), \\ &(\mathsf{FN[j,1,1]},\ldots,\mathsf{FN[j,1,m]}, \\ &\mathsf{FN[j,2,1]},\ldots,\mathsf{FN[j,2,m]},\ldots, \\ &\mathsf{FN[j,r_{max},1]},\ldots,\mathsf{FN[j,r_{max},m]})) \end{aligned} \tag{6}$$

This gives us circuits for approaching each of the guarantee states eagerly. Each $\mathsf{FN[j]}$ corresponds to one of the dashed boxes in Fig. 1.

3.4 Bookkeeping Circuit for Guarantee Selection

Finally we have to choose which guarantee to approach next. In [16], this was done in a round-robin fashion using a modular counter jx. We present a new approach that satisfies each guarantee as quickly as possible without having to wait for a counter to match a guarantee's index j. We will first informally describe the principal idea behind the bookkeeping circuit we employ, and afterwards define it formally.

Our approach uses one bit of memory for each guarantee ($\mathsf{JX}[1], \ldots, \mathsf{JX}[n]$), plus one *master* bit (master). Initially, the master bit and all JX bits are all set to the same (arbitrary) value. The semantics of these bits is as follows: JX[j] XOR master is true if and only if guarantee j has already been satisfied in the current round. A guarantee j will be satisfied when the play is about enter a state in the set represented by ${J'}_j^G$. When this happens, the corresponding bit $\mathsf{JX}[j]$ is flipped (if it was not already different from the master bit). As soon as all guarantees were satisfied in one round (i.e., all JX bits are different from the master bit), the master bit is flipped and the procedure starts another round. Thus, the JX bits together with the master bit allow us to determine which guarantees still have to be pursued at a given time.

Note that the sets J_j^G are not necessarily disjunct. Empirical evidence suggests that there are often states which belong to several (or even all) sets J_j^G. Imagine, for example, an arbiter with N request and grant signals, and N guarantees stating that every request must eventually be granted. Then the state in which no requests are made (and thus no grants are given) fulfills *all* N guarantees and thus belongs to all sets J_j^G. Our bookkeeping circuit can take advantage of that by flipping all the JX bits corresponding to (yet unfulfilled) guarantees that are fulfilled in a particular state. This leads to a much more eager systems compared to systems using a modular counter for bookkeeping. We will illustrate this with an example in Section 3.6.

We will now provide a formal definition of our bookkeeping circuit.

JX flip signal. We need *flip signals* to determine whether a guarantee is being satisfied or not. Guarantee j is being satisfied whenever we move to the states represented by ${J'}_j^G$.

$$\mathsf{JX_{flip}[j]} \;=\; \mathcal{C}({J'}_j^G) \tag{7}$$

JX update. The value of a JX bit is updated (flipped) whenever the corresponding $\mathsf{JX_{flip}}$ signal is true and the JX bit is still equal to the master bit. A diagram of this circuit is shown in Figure 2.

$$\mathsf{JX'[j]} \;=\; \mathsf{JX[j]\ XOR\ (JX_{flip}[j]\ AND\ COMP(JX[j],\ master))} \tag{8}$$

Master update. The update of the master bit works as follows: The master bit is flipped when it is unequal to all next JX bits. I.e., the flip happens if and only if all guarantees have been satisfied in a round. A diagram of this circuit is shown in Figure 3.

$$\mathsf{master'} \;=\; \mathsf{master\ XOR\ (AND_j\ (NOT\ COMP(JX'[j],\ master)))} \tag{9}$$

Guarantee selection. Finally, we need to select a guarantee that should be pursued at the moment. Candidates are all those guarantees whose JX bit equals

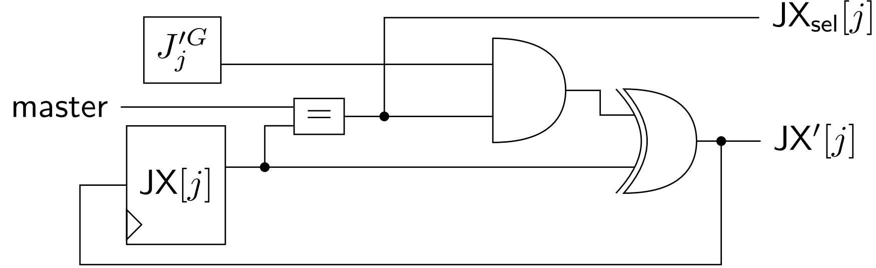


Fig. 2: Circuit for updating a JX bit with the signal for selecting a guarantee.

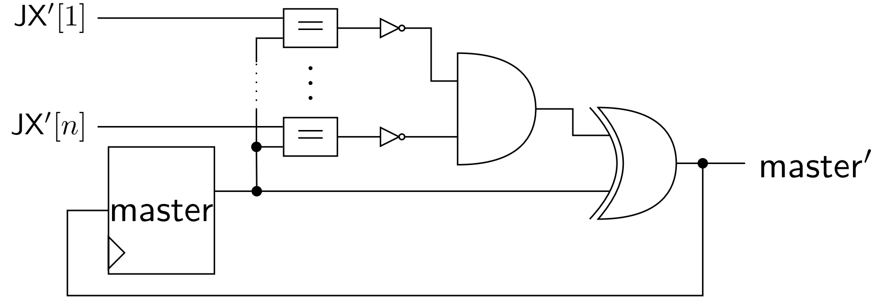


Fig. 3: Circuit for updating the master bit.

the master bit, as these are the guarantees not yet fulfilled in the current round. The signal $\mathsf{JX_{sel}}[\mathsf{j}]$ tells whether or not guarantee j is a candidate for selection.

$$\mathsf{JX_{sel}}[\mathsf{j}] \;=\; \mathsf{COMP}(\mathsf{master}, \mathsf{JX}[\mathsf{j}]) \tag{10}$$

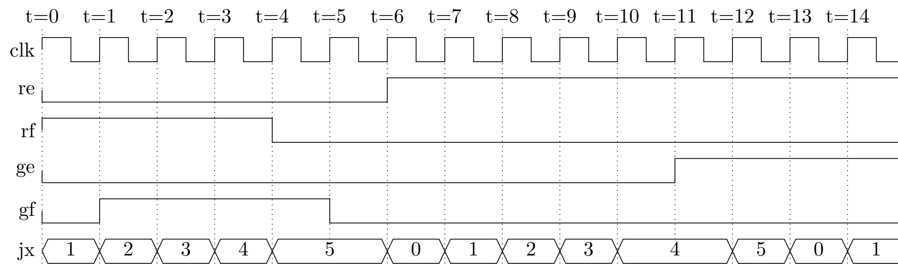
3.5 Combining Functions with Guarantee Selection

To achieve eagerness, we use the signals $\mathsf{JX_{sel}}[\mathsf{j}]$ selector signals for the topmost SELECT circuit in Figure 1. I.e., we choose to make progress towards the lowest-numbered unsatisfied guarantee by choosing the corresponding function.[2]

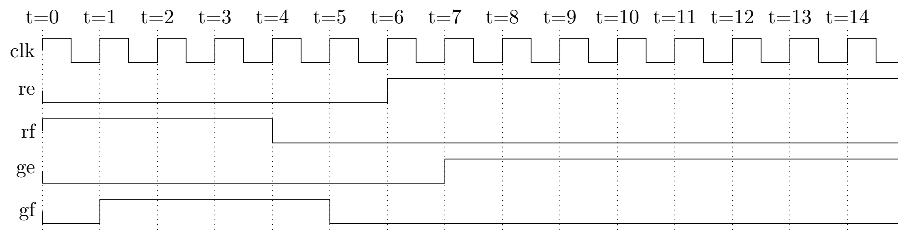
$$\mathsf{FN}(\mathsf{I}, \mathsf{O}, \mathsf{I}' \rightarrow \mathsf{O}') = \mathsf{SELECT}((\mathsf{JX_{sel}}[1], \ldots, \mathsf{JX_{sel}}[\mathsf{n}]), (\mathsf{FN}[1], \ldots, \mathsf{FN}[\mathsf{n}])) \tag{11}$$

Note that the outputs of the circuit in Equation 11 are also the primary outputs (O') of the overall system.

[2] This gives designers the possibility to prioritize guarantees by reordering them in the specification.



(a) Modular counter jx as in [16]



(b) Bookkeeping as Section 3.4.

Fig. 4: Timing diagram of a 6-input arbiter. The request and grant signals with indices $a, \dots, d$ have been omitted, as they are all zero for the entire time shown.

3.6 Demonstration of Eagerness

To illustrate how the circuits synthesized according to our new method are more eager, we make the following comparison. We take a specification for a full-handshake arbiter with six request lines (ra to rf) and six corresponding grant lines (ga to gf). We synthesize circuits for this specification, once according to [16], and once according to the method presented in this paper. We simulate both circuits with the same input values. We set the request rf to 1 in clock cycles 0 to 4 and we set request re to 1 from cycle 6 on. The timing diagrams of our simulations are shown in Figures 4a and 4b. For increased readability, we have omitted some of the signals which are 0 the entire time in the waveform.

We observe a difference in the behavior of granting the requests by the two circuits in comparison:

1. **Round-robin strategy:** We see that the first request (rf) is immediately answered with grant gf. The second request (re), however, is only granted in cycle 11 (i.e., with a delay of 5 cycles), because the modular counter jx has to loop around first, in order to reach the value 4, which corresponds to ge. In each of the "wasted" 5 cycles, the implementation discovers that there is no request on one particular request line, and thus, the corresponding guarantee is fulfilled.

2. **Guarantee selection with bookkeeping:** In this case the first request is granted in the same clock cycle as in the circuit with the round-robin strategy. The grant answering request *re* is given in cycle 7, though. This is a delay of only 1 cycle; a 4-cycle improvement: the bookkeeping approach discovers immediately that the guarantees corresponding to those request lines where no request is made are all fulfilled. Thus, the system can immediately fulfils the remaining guarantee concerning request *re*.

4 Implementation

We implemented the proposed synthesis method as an extension to RATSY (Requirements Analysis Tool with Synthesis) [4]. Computation of the winning region (and the necessary intermediate results) had already been implemented in this tool. We start with our new approach after the computation of the winning region finishes. Instead of building a monolithic strategy, we construct the circuits as described in Section 3. We have two slightly different implementations, which are described in the following sections.

4.1 Onion Rings without BDD Reordering

Our first approach is to create one sub-circuit after the other, immediately freeing any BDDs that are no longer required for subsequent computations. We keep a hash table of all BDD nodes for which we already constructed multiplexers. Since BDDs within the same manager may share internal nodes, we can reuse the corresponding multiplexers whenever necessary.

This advantage, however, comes at a price. We have to disable dynamic BDD reordering when we create the first circuit, because dynamic reordering may remove and/or reassign internal BDD nodes. Note, however, that reordering can of course be used before we create the first circuit. Thus, we enable dynamic reordering of BDDs during the computation of the winning region.

4.2 Onion Rings with BDD Reordering

As an alternative method, we first compute BDDs corresponding to all circuits without actually writing them out already. Thus, we can keep dynamic reordering enabled during all the computations.

Once we have computed all BDDs to be dumped, we perform a final (forced) reordering to reduce the size of the resulting multiplexer circuit, and then dump all BDDs at once, again taking advantage of node sharing.

5 Experimental Results

We used the specifications of the AMBA bus arbiter [5] for our experiments. We compared runtime, memory usage, and circuit size. We already had a working

Table 1: Experimental results for each test case and method.

Circuit	Runtime [s]			Memory usage [GB]			Circuit Size [relative to Ref]	
	Ref	Onion_{RO}	$\text{Onion}_{\text{noRO}}$	Ref	Onion_{RO}	$\text{Onion}_{\text{noRO}}$	Onion_{RO}	$\text{Onion}_{\text{noRO}}$
amba02	3	6	11	0.55	0.57	0.57	18.3	31.9
amba03	53	27	44	0.67	0.62	0.66	4.5	8.0
amba04	176	517	846	0.99	1.55	1.19	40.5	66.6
amba05	492	885	846	1.41	1.13	1.49	23.6	62.7
amba06	1,059	723	1,370	1.55	1.45	1.18	16.9	42.9
amba07	1,960	1,532	1,592	1.63	1.56	1.50	12.3	22.5
amba08	13,390	19,800	36,433	7.45	16.62	16.89	X	X
amba09	5,394	4,011	4,578	2.45	2.63	2.41	16.9	28.4
amba10	12,673	5,413	8,941	7.20	3.11	2.67	25.4	46.7
amba11	10,685	7,609	11,277	4.24	4.41	2.68	13.2	23.9
amba12	55,997	7,831	11,585	9.18	4.79	2.79	19.2	28.5
amba13	40,229	14,787	15,825	13.81	5.05	4.39	10.8	25.7
amba14	41,538	17,077	14,287	8.92	5.77	2.98	21.1	30.3
amba15	43,173	17,721	19,646	14.98	8.10	4.24	16.7	24.7

synthesis implementation, based on cofactors [6] and used it as a reference point for our new technique. All experiments were conducted on a 64-bit Linux machine powered by a 2.66GHz Intel Xeon CPU with 64GB RAM. The 3 methods we have compared are as follows:

1. **Reference:** The cofactor-based approach described in [6]. This method serves as a reference point.
2. **Onion Rings without reordering:** The method described in Section 4.1.
3. **Onion Rings with reordering:** The method described in Section 4.2.

We will use the abbreviations Ref, $\text{Onion}_{\text{noRO}}$ and Onion_{RO} to denote the methods. The results are presented in Table 1 and in Figures 5 and 6. We do not know why amba08 has a significantly higher time and memory consumption than amba09 in all three methods. A similar discrepancy has been observed before [6].

5.1 Runtime

We can see that for larger examples, method Onion_{RO} performs much better than the reference method. For smaller examples, the 3 methods perform similarly. We also see that finding a better BDD order pays off by improving the runtime. I.e., Onion_{RO} is typically faster than $\text{Onion}_{\text{noRO}}$.

5.2 Memory Usage

The memory requirements are taken from the `Cudd_PrintInfo` function of the CUDD [21] library and reflect the memory requirements of the BDD manager. Note that almost all memory used by RATSY is used by the BDD manager.

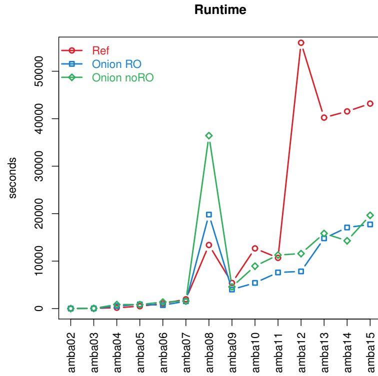


Fig. 5: Runtime for each approach and test case.

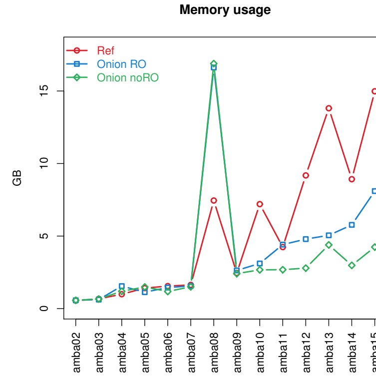


Fig. 6: Memory usage for each approach and test case.

We can see that the memory usage of our methods is better than the reference method. Again, smaller examples perform similarly, but for larger examples we gain an advantage. The method $\text{Onion}_{\text{noRO}}$ performs best, as it only needs to have the BDDs for creating a specific onion ring in memory. In contrast, Ref has to have the whole strategy BDD in memory, and Onion_{RO} has to have the BDDs for all onion rings in memory to find a consistent BDD order before building the circuits.

5.3 Circuit Size

Circuit size was measured with `abc`[3]. Table 1 shows the relative circuit sizes with respect to the Reference method. For example, when synthesizing amba02 with method Onion_{RO}, the resulting circuit is 18.3 times the size of the circuit obtained when synthesizing amba02 with the Reference method. Note that factors have been rounded to one decimal. For amba08, `abc` runs into a timeout (marked with "X" in Table 1).

6 Conclusion and Future Work

We have presented a novel approach to GR(1) synthesis. Our technique builds upon [16], but circumvents the generation of a large, monolithic strategy relation. We have shown in our experiments that, using our technique, we can, in general, reduce the runtime and memory usage significantly. For larger examples, our method is able to synthesize results, where previous methods might have run out of memory, or run into timeouts. These results, however, come at the cost of larger circuits.

[3] `http://www.eecs.berkeley.edu/~alanmi/abc/abc.htm`

Apart from that, circuits built with our new method are *eager*, meaning that they fulfill the guarantees more quickly, whenever possible. First, we get as close as we can to the next guarantee state in every time step. Second, we check which guarantees are already fulfilled in parallel, instead of sequentially. This leads to more responsive, robust systems.

For future work, we will investigate different relation determinization techniques (cf. Section 2.4), which might improve circuit sizes. Also, we plan to investigate extensive "don't-care propagation". Whenever we have detected that we are in onion ring r, the output functions corresponding to all rings $s > r$ can actually be set to arbitrary values. Such optimizations might improve circuit size at the expense of additional CPU time.

References

1. Baneres, D., Cortadella, J., Kishinevsky, M.: A recursive paradigm to solve Boolean relations. Design Automation Conference pp. 416–421 (2004)
2. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T., Jobstmann, B.: Robustness in the presence of liveness. In: Proc. Computer Aided Verification. pp. 410–424. Springer (2010)
3. Bloem, R., Chatterjee, K., Henzinger, T., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Int. Conf. Computer Aided Verification (CAV). pp. 140–156 (2009)
4. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Koenighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY — a new requirements analysis tool with synthesis. In: Proc. Computer Aided Verification. pp. 425–429 (2010), LNCS 6174
5. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: Proceedings of the Design, Automation and Test in Europe. pp. 1188–1193 (2007)
6. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware form PSL. In: 6th International Workshop on Compiler Optimization Meets Compiler Verification (2007)
7. Church, A.: Logic, arithmetic and automata. In: Proceedings International Mathematical Congress (1962)
8. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: Proc. Computer Aided Verification. pp. 263–277 (2009)
9. Jiang, J.H.R., Lin, H.P., Hung, W.L.: Interpolating functions from large Boolean relations. In: Proceedings of the 2009 International Conference on Computer-Aided Design. pp. 779–784. ICCAD '09, ACM, New York, NY, USA (2009)
10. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: 6th Conference on Formal Methods in Computer Aided Design (FMCAD'06). pp. 117–124 (2006)
11. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Computer Aided Verification. pp. 258–262 (2007)
12. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. Journal of Computer and System Sciences 78(2), 441 – 460 (2012)
13. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
14. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010 (2010)

15. Morgenstern, A., Schneider, K.: Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. In: Montanari, A., Napoli, M., Parente, M. (eds.) Games, Automata, Logics, and Formal Verification (GandALF). Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 25, pp. 89–102. Minori, Italy (2010)
16. Piterman, N., Pnueli, A., Sa´ar, Y.: Synthesis of reactive(1) designs. In: 7th International Conference on Verification, Model Checking and Abstract Interpretation. pp. 364–380. Springer (2006), LNCS 3855
17. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. Symposium on Principles of Programming Languages (POPL '89). pp. 179–190 (1989)
18. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Automated Technology for Verification and Analysis (ATVA'07). pp. 474–488 (2007)
19. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: 9th Int. Conf. on Formal Methods in Computer Aided Design. pp. 77–84 (2009)
20. Solar-Lezama, A.: The sketching approach to program synthesis. In: Asian Symp. Programming Languages and Systems. pp. 4–13. Springer (2009), LNCS 5904
21. Somenzi, F.: CUDD: CU Decision Diagram Package. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/
22. Staber, S., Jobstmann, B., Bloem, R.: Finding and fixing faults. In: Borrione, D., Paul, W. (eds.) 13th Conference on Correct Hardware Design and Verification Methods (CHARME '05). pp. 35–49. Springer-Verlag (2005), LNCS 3725
23. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: Proc. Principles of programming languages. pp. 327–338. ACM (2010)
24. Watanabe, Y., Brayton, R.: Heuristic minimization of multiple-valued relations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12(10), 1458–1472 (1993)