# Just-In-Time Reactive Synthesis

Shahar Maoz
Tel Aviv University
Israel

Ilia Shevrin
Tel Aviv University
Israel

## ABSTRACT

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. GR(1) is an expressive assume-guarantee fragment of LTL that enables efficient synthesis and has been recently used in different contexts and application domains.

In this work we present just-in-time synthesis (JITS) for GR(1), a novel means to execute synthesized reactive systems. Rather than constructing a controller at synthesis time, we compute next states during system execution, and only when they are required. We prove that JITS does not compromise the correctness of the synthesized system execution. We further show that the basic algorithm can be extended to enable several variants.

We have implemented JITS in the Spectra synthesizer. Our evaluation, comparing JITS to existing tools over known benchmark specifications, shows that JITS reduces (1) total synthesis time, (2) the size of the synthesis output, and (3) the loading time for system execution, all while having little to no effect on system execution performance.

## CCS CONCEPTS

• **Software and its engineering** → *Formal methods.*

## KEYWORDS

reactive synthesis, GR(1)

**ACM Reference Format:**
Shahar Maoz and Ilia Shevrin. 2020. Just-In-Time Reactive Synthesis. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416557

## 1 INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [33]. Rather than manually constructing an implementation and using model checking to verify it against a specification, synthesis offers an approach where a correct implementation of the system is automatically obtained for a given specification, if such an implementation exists.

GR(1) is a fragment of linear temporal logic [32] (LTL) that has an efficient symbolic synthesis algorithm [5, 31] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [12, 23]. GR(1) specifications include assumptions and guarantees about what needs to hold on all initial states, on all states and transitions (safety), and infinitely often on every run (justice). GR(1) synthesis has been used and extended in different contexts and for different application domains, including robotics [19, 22, 24], scenario-based specifications [28], aspect languages [27], event-based behavior models [11], hybrid systems [14], and device drivers [37], to name a few.

The synthesis algorithm in [5], as implemented (with some variations) in existing GR(1) synthesizers RATSY [4], Slugs [13], and Spectra [25], consists of two phases. In the first phase, the algorithm checks for realizability. If the specification is found to be realizable in the first phase, synthesis continues to the second phase, where it constructs a controller using memory collected in the first phase. According to experience reported on in [5] and supported by evidence we report in our evaluation, in many cases the first phase of realizability checking takes only a small fraction of the total synthesis time. Most of the time is spent in the second phase of controller construction.

**In this work we introduce just-in-time synthesis (JITS), a novel means to execute synthesized reactive systems, which skips the controller construction phase. Instead, it efficiently stores the memory collected in the first phase of realizability checking, and uses it during system execution to compute next states, only when they are required. The controller is never constructed. However, importantly, as the next state computation is performed using the memory that was collected during realizability checking, JITS keeps system execution correct-by-construction.**

As JITS skips construction, it is expected to dramatically reduce the overall synthesis time. Moreover, JITS symbolic representation of the memory that is required for correct execution, consists of chains of very similar BDDs, and makes storing it at the end of synthesis and loading it just before system execution very efficient. During system execution, at each step, JITS manipulates only a small number of BDDs, each of relatively small size, and thus the effect on performance of the just-in-time next states computation is kept to a minimum.

Beyond performance, from architectural and methodological perspectives, JITS decouples synthesis (realizability checking) from system execution. This decoupling is significant, as indeed, synthesis takes place at development time and is executed on development machines, while system execution, i.e., the actual running of the synthesized system, will typically run on very different machines, e.g., on a robot. The decoupling, which is unique to JITS, enables flexibility in system execution, and supports, for example, extensions and variants that aim to improve qualitative aspects of the execution, such as eagerness, recovery, and configurability. See Sect. 4.

Finally, JITS is defined for GR(1) and thus seamlessly supports any specification that is reduced to GR(1). In particular, the use of JITS is independent of the use of advanced constructs such as past LTL operators, patterns, counters, etc. as supported, e.g., in Spectra [25]. Our evaluation of JITS (see below) indeed includes such rich specifications.

We have implemented JITS on top of the Spectra synthesizer [2] (using its implementation of realizability checking, ignoring its implementation of controller construction). To evaluate JITS, we compared it against existing tools over specifications from publicly available benchmarks. The results show that JITS outperforms existing tools in all aspects of synthesis, including time and memory, and that its outputs of the synthesis phase are smaller and typically faster to load. In practical terms, it makes the execution of systems with rather large specifications feasible even on environments with limited resources. See Sect. 5.

The remainder of the paper is structured as follows. In Sect. 2 we provide necessary background on GR(1) realizability and controller construction. In Sect. 3 we present the JITS approach and prove its correctness. In Sect. 4 we discuss several optional extensions to JITS. In Sect. 5 we present our evaluations for JITS synthesis and execution as compared to other tools. Finally, in Sect. 6 and Sect. 7 we discuss related work and conclude.

## 2 PRELIMINARIES

We recall the notions of realizability and synthesis, GR(1) realizability problem, and the controller construction algorithm from [5].

### 2.1 LTL Realizability and Synthesis

The interaction of a *reactive system* [16] with its environment is best seen as a turn-based infinite duration game where in each of its rounds the environment first chooses an assignment to its input variables, and the system responds by choosing an assignment to its output variables. Then, an LTL formula over these Boolean variables can be used as a specification for the behavior of a desired reactive system.

We consider reactive systems that can be implemented by *controllers* of finite size, i.e., by finite Mealy machines that compute their outputs. A controller *realizes* an LTL specification $\psi$ if for all environment inputs, it prescribes outputs that result in computations which satisfy $\psi$. We say that $\psi$ is *realizable* if such a controller exists. Otherwise, $\psi$ is *unrealizable.* It is well-known that *finite* controllers are sufficient for realizability of LTL specifications [33]. The goal of LTL *synthesis* is, given an LTL specification, to construct a controller that realizes it, if it is realizable.

### 2.2 GR(1) Realizability Problem

*GR(1) synthesis* [5, 31] handles a fragment of LTL where specifications have a predefined syntactic structure. Specifically, they contain initial assumptions and guarantees over initial states (denoted $\theta^e$ and $\theta^s$ resp.), safety assumptions and guarantees relating the current and next state (denoted $\rho^e$ and $\rho^s$ resp.), and justice assumptions and guarantees requiring that an assertion holds infinitely many times during a computation (denoted $J^e$ and $J^s$ resp.).

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

$$(\theta^e \wedge \mathbf{G}\rho^e \wedge \bigwedge_{i\in 1..m} \mathbf{GF}J_i^e) \rightarrow (\theta^s \wedge \mathbf{G}\rho^s \wedge \bigwedge_{j\in 1..n} \mathbf{GF}J_j^s)$$

The GR(1) realizability problem is formulated as a two-player game between the system and the environment, and reduces to deciding the winner in such game. Intuitively, the system wins if it has a strategy that allows it to satisfy its $n$ justice guarantees infinitely often while maintaining initial and safety guarantees, or force the environment to violate any of its $m$ justice assumptions, or its initial or safety assumptions.

We denote the states from which the system can force the environment to visit a state in $X$ by $\circledcirc(X)$, also called the controlled predecessors, defined as:

$$\circledcirc(X) = \{c \in 2^{\mathcal{X}\cup\mathcal{Y}} \mid \forall x \in 2^{\mathcal{X}} : \neg\rho^e(c,x) \vee \exists y \in 2^{\mathcal{Y}} : (\rho^s(c,\langle x,y\rangle) \wedge \langle x,y\rangle \in X)\} \quad (1)$$

The winning states are characterized using the following three-level nested $\mu$-calculus formula from [5]:

$$W_{sys} = \nu Z.\bigcap_{j=1}^{n}\mu Y.\bigcup_{i=1}^{m}\nu X.(J_j^s \cap \circledcirc(Z)) \cup \circledcirc(Y) \cup (\neg J_i^e \cap \circledcirc(X)) \quad (2)$$

The output of formula 2 is the set of winning states for the system player.

### 2.3 GR(1) Original Controller Construction

The realizability checking algorithm outputs additional intermediate results of the nested fixed-point computations from Eq. 2, stored in `mX[][][]` and `mY[][]`. These intermediate results, along with the winning set $W_{sys}$, are presented as follows. We use 1-based indexes for the justice assumptions and guarantees to conform with the notation in [5].

*2.3.1* . mX[$j$][$r$][$i$] represents the set of states in which the system is at most $r$ steps from the satisfaction of $J_j^s$ (the system is at rank $r$), and from any state in this set, the system can either keep forcing the environment to violate $J_i^e$, or alternatively move one step closer towards satisfying $J_j^s$. It is the result of the safety game innermost greatest fixed-point computation from formula 2.

Specifically, when $r = 1$, mX[$j$][1][$i$] represents the set of states in which the system either satisfies $J_j^s$, or forces the environment to violate $J_i^e$.

*2.3.2* . mY[$j$][$r$] represents the set of states in which the system is at most $r$ steps from the satisfaction of $J_j^s$. From any state in this set, the system can either force the environment to violate at least one $J_i^e$ for some $i$, or move one step closer towards satisfying $J_j^s$. mY[$j$][$r$] is defined as follows:

$$\mathtt{mY}[j][r] = \bigcup_{i\in[1..m]} \mathtt{mX}[j][r][i] \quad (3)$$

[1]There exists a strict, more involved variant of realizability [5]. However, the difference is not relevant for the paper.

It is an intermediate result of the inner reachability game least fixed-point computation from formula 2. We denote by $r_j$ the number of iterations during this computation. It holds that:

$$\forall j, r < r_j.\Big(\mathtt{mY}[j][r] \subseteq \mathtt{mY}[j][r+1]\Big) \tag{4}$$

Specifically, $\mathtt{mY}[j][r_j]$ is the result of the reachability game inner least fixed-point computation. From this set, the system can eventually satisfy $J^s_j$, or force the environment to violate some justice assumption.

*2.3.3 .* Finally, $W_{sys}$, denoted $\mathsf{Z}$ from now on, represents the set of states in which the system can act towards satisfying *all* its justice guarantees in a round-robin fashion, or can force the environment to violate any of its assumptions. It is the winning set of the GR(1) game and the result of the safety game outermost greatest fixed-point computation. During the iterations of this computation, the inner reachability games are adjusted each time to stop within the boundaries of $\mathsf{Z}$, hence finally:

$$\forall j.\mathsf{Z} = \mathtt{mY}[j][r_j] \tag{5}$$

From these intermediate results, Bloem et al. [5] show how to construct a controller implementing a winning strategy as described in Alg. 1. We briefly present the algorithm.

Let $\mathcal{Z}_n$ denote the variable representing the index of the current justice goal. Primed BDDs, e.g., $X'$, denote the set of states $X$, but where variables refer to their next value. Denote $\rho$ as $\rho^e \wedge \rho^s$. $\rho$, as the BDD responsible for the transitions safety, contains both primed and unprimed variables. Denote $\oplus$ as the addition operation modulo $n$.

The construction is divided into three parts. Each part conjuncts valid transitions to the controller, i.e., relations between current and next states:

- Given $j \in [1 \dots n]$, a $\rho_1$ transition (lines 2-6) is taken from $\mathsf{Z}$ states, where current state satisfies justice guarantee $J^s_j$. In this case, $\mathcal{Z}_n$ is updated with $j \oplus 1$ and next states are in $\mathsf{Z}$.
- Given $j \in [1 \dots n]$ and $r \in [2 \dots r_j]$, a $\rho_2$ transition (lines 7-15) is taken from $\mathtt{mY}[j][r]$ states, where rank $r$ is minimal, namely, current state is in $\mathtt{mY}[j][r]$ but not in $\bigcup_{r'<r} \mathtt{mY}[j][r']$. In this case, controller proceeds to $\mathtt{mY}[j][r-1]$.
- Given $j \in [1 \dots n]$, $r \in [1 \dots r_j]$, and $i \in [1 \dots m]$, a $\rho_3$ transition (lines 16-26) is taken from $\mathtt{mX}[j][r][i]$ states, where current state violates $J^e_i$, and $r$ and $i$ are minimal, namely, current state is in $\mathtt{mX}[j][r][i]$ but not in $\bigcup_{r',i'} \mathtt{mX}[j][r'][i']$ for $r' < r$ or $r' = r$ and $i' < i$. In this case, the controller remains in $\mathtt{mX}[j][r][i]$.

The output of Alg. 1 is a single BDD over variables $\mathcal{X} \cup \mathcal{Y}$ and $\mathcal{Z}_n$ that describes a reactive controller, i.e., a finite Mealy machine. Since this representation uses a pre-computed BDD, we refer to it here as the *static* approach, to differentiate from our just-in-time approach.

Bloem et al. [5] show that this construction is sound and complete, i.e., that all next states returned are winning for the system and that if there is a winning next state, at least one is returned.

**Algorithm 1** Controller construction for the static approach

```
    input: $\mathsf{Z}, \mathtt{mY}[][], \mathtt{mX}[][][]$
1:  $trans \leftarrow TRUE$
2:  for all $j \in [1 \dots n]$ do                                  // $\rho_1$ transitions
3:      $trans_j \leftarrow (\mathcal{Z}_n = j) \wedge \mathsf{Z} \wedge J^s_j \wedge \rho$
4:      $trans_j \leftarrow trans_j \wedge \mathsf{Z}' \wedge (\mathcal{Z}'_n = j \oplus 1)$
5:      $trans \leftarrow trans \vee trans_j$
6:  end for
7:  for all $j \in [1 \dots n]$ do                                  // $\rho_2$ transitions
8:      $low \leftarrow \mathtt{mY}[j][0]$
9:      for all $r \in [2 \dots r_j]$ do
10:         $trans_{jr} \leftarrow (\mathcal{Z}_n = j) \wedge \mathtt{mY}[j][r] \wedge \neg low \wedge \rho$
11:         $trans_{jr} \leftarrow trans_{jr} \wedge low' \wedge (\mathcal{Z}'_n = j)$
12:         $trans \leftarrow trans \vee trans_{jr}$
13:         $low \leftarrow low \vee \mathtt{mY}[j][r]$
14:     end for
15: end for
16: for all $j \in [1 \dots n]$ do                                  // $\rho_3$ transitions
17:     $low \leftarrow \mathtt{mY}[j][0]$
18:     for all $r \in [1 \dots r_j]$ do
19:         for all $i \in [1 \dots m]$ do
20:             $trans_{jri} \leftarrow (\mathcal{Z}_n = j) \wedge \mathtt{mX}[j][r][i] \wedge \neg low \wedge \neg J^e_i \wedge \rho$
21:             $trans_{jri} \leftarrow trans_{jri} \wedge \mathtt{mX}'[j][r][i] \wedge (\mathcal{Z}'_n = j)$
22:             $trans \leftarrow trans \vee trans_{jri}$
23:             $low \leftarrow low \vee \mathtt{mX}[j][r][i]$
24:         end for
25:     end for
26: end for
27: return $trans$
```

## 2.4 GR(1) Original Controller Execution

Before system execution, this BDD, denoted *trans*, is loaded, and is used during execution to compute next possible assignments to all system variables, as we show in Alg. 2. One step of the execution consists of the environment choosing values for the next input, after which the system must choose values for the next state (output) from the result set of Alg. 2.

In Alg. 2, $c \in 2^{\mathcal{X} \cup \mathcal{Y}}$ is a BDD describing the current state of the variables. $x \in 2^{\mathcal{X}}$ is a BDD describing environment inputs. $j$ is the index of the current justice goal. The execution itself is straightforward. It conjuncts *trans* with current state $c$, inputs $x$ primed, and the assignment $\mathcal{Z}_n = j$. Observe that the resulting BDD $T$ represents a subset of transitions encoded in *trans* that conform to the given inputs. The controller extracts the next states from $T$ by quantifying out unprimed variables and unpriming the result (intuitively, it means moving to the next step in the game).

From the output of Alg. 2, the controller picks one assignment to variables over $\mathcal{Y} \cup \{\mathcal{Z}_n\}$, which is regarded as the next state. This assignment can be chosen at random, or by applying some deterministic rule.

We assume that environment inputs are checked for correctness prior to the call to Alg. 2, namely, they do not violate the safety assumptions. In case of an invalid input the controller may raise an error message.

**Algorithm 2** Controller next states for the static approach

```
    input: $c \in 2^{\mathcal{X}\cup\mathcal{Y}}$, $x \in 2^{\mathcal{X}}$, $j \in [1 \dots n]$
1:  $T \leftarrow (\mathcal{Z}_n = j) \wedge c \wedge x' \wedge trans$   // where $trans$ is from Alg. 1
2:  $T \leftarrow QuantifyOutUnprimedVars(T)$
3:  $T \leftarrow Unprime(T)$
4:  return $T$
```

## 3 JUST-IN-TIME SYNTHESIS

We are now ready to present the main contribution. The JITS solution is divided into two separate phases: storing and execution. We use set theoretic and logic notation interchangeably when describing operations on BDDs (as representations of sets and as Boolean functions, respectively).

### 3.1 Storing

Given the intermediate results computed by the realizability checking algorithm, we make the following key observations.

*3.1.1* . mY[][] and Z are derived from the array of mX[][][] fixed-points, as seen in Equ. 3 and Equ. 5. Thus, even though they are crucial for correct execution, storing them for later use is redundant.

*3.1.2* . The mX[][][] array holds many BDDs, each of which describes a set of states. While there are $\sum_{j \in [1 \dots n]} m \cdot r_j$ such BDDs in total, we expect them to be similar to one another because they represent monotonic sequences of sets, i.e., ordered by subset relation. Formally:

$$\forall j, r < r_j, i. \Big(\mathtt{mX}[j][r][i] \subseteq \mathtt{mX}[j][r+1][i]\Big) \tag{6}$$

Equation 6 holds following Equ. 3 and Equ. 4. One may hope to take advantage of this similarity when storing them.

*3.1.3* . When encoding multiple BDDs to a file on disk, the memory optimization achieved by sharing nodes in the internal BDD engine between these BDDs is lost if they are encoded separately.

Based on these three observations, JITS encodes the required data in a single BDD over three new variables $J$, $R$, $I$, and the variables $\mathcal{X} \cup \mathcal{Y}$ as follows:

$$Xs = \bigwedge_{j,r,i} \Big((J = j \wedge R = r \wedge I = i) \rightarrow \mathtt{mX}[j][r][i]\Big) \tag{7}$$

At the end of the construction phase, JITS stores the BDD $Xs$ along with the BDDs representing the justice assumptions $J^e_i$, justice guarantees $J^s_j$, initial and safety formulas $\theta^s$, $\theta^e$, $\rho^s$ and $\rho^e$ from the original specification. As our evaluation shows, see Sect. 5, this representation is highly efficient.

Together, as we explain next, the stored BDDs provide all the necessary information for correct execution.

### 3.2 Execution

*3.2.1 Loading.* System execution starts by loading the BDDs stored at the end of the storing phase. As part of loading, JITS extracts every mX[$j$][$r$][$i$] to an array by restricting the $Xs$ BDD to every unique assignment of the $J$, $R$, and $I$ variables, and computes mY[][] array according to definition from Equ. 3. JITS also omits Z calculation since it holds that Z = mY[1][$r_1$] from Equ. 5.

JITS loads the initial and safety BDDs, and sets $\rho = \rho^s \wedge \rho^e$. $\rho$ as well as all primed versions of mX[][][] and mY[][] BDDs are computed at load time and cached for later use. Note that the variable $\mathcal{Z}_n$ is absent from the BDD state space of JITS. As we show in our evaluation, JITS loading time is typically shorter than that of the static approach (and the longer the loading time with the static approach, the shorter JITS loading time becomes in comparison, reaching order of magnitudes relative improvement).

*3.2.2 Step-wise execution.* At the end of loading, JITS is ready for step-wise execution as we show in Alg. 3. This step-wise execution algorithm acts as an eager, just-in-time implementation of Alg. 1 static construction and Alg. 2 execution. As in the static approach, we assume correct environment inputs, and in case of an invalid input (which violates safety assumptions), can raise an error message.

During runtime, JITS maintains, on the algorithm level, a helper index $r_{min}$ that keeps the current rank (distance from $J^s_j$). JITS maintains this index, such that after initialization and at the end of each step it points to the lowest rank possible.

The algorithm first checks if the current state $c$ satisfies $J^s_j$. According to the $\rho_1$ transitions from the static construction, next states should be the whole Z set. JITS instead, performs an eager look-ahead to find the lowest rank for next justice goal given environment inputs $x$, and updates $r_{min}$. The BDD $N$, which represents the next states, is assigned with mY[$j \oplus 1$][$r_{min}$]. (lines 3-4).

Otherwise, JITS tries to decrease rank. According to the $\rho_2$ transitions from the static construction, next states is the mY[$j$][$r_{min} - 1$] set. As before, JITS performs an eager look-ahead to find the next lowest rank possible for the current goal. It updates $r_{min}$ and assigns $N$ to mY[$j$][$r_{min}$] (lines 9-10).

Note that it is not always possible to decrease rank from $c$ with inputs $x$, since for every $j$ and $r$, mY[$j$][$r$] contains not only the controlled predecessors of mY[$j$][$r-1$], but also the results of $m$ safety games for this rank that force justice assumption violation. Another possibility is that $r_{min}$ is currently 1, yet justice goal is not satisfied. In these cases, JITS resorts to force the environment to violate some $J^e_i$. It computes the minimal $i_{min}$ for that purpose and assigns $N$ to mX[$j$][$r_{min}$][$i_{min}$] (lines 12-13).

The final part is similar to Alg. 2. JITS conjuncts $N$ in its primed state with $c \wedge x' \wedge \rho$. Then it processes the result by quantifying out unprimed variables and unpriming, and finally returns a BDD that represents a set of possible next assignments over $\mathcal{Y}$. Observe that at this stage, the next justice goal $j$ is already decided. As in the static approach, a single assignment can be chosen at random or deterministically.

Remark 1 (Eagerness). *A property of JITS that one might consider eager, is that it first and foremost considers transitions that make progress towards satisfying the current justice guarantee, and resorts to justice assumption violation only if conditions are not met. The original construction from [5], on the other hand, always considers $\rho_3$ transitions from any step, since they were already added to trans BDD regardless.*

**Algorithm 3** Controller next states for JITS

```
    input: $c \in 2^{\mathcal{X}\cup\mathcal{Y}}$, $x \in 2^{\mathcal{X}}$, $j \in [1 \dots n]$
 1: if $c \in J^s_j$ then                       // corresponds to $\rho_1$ from Alg. 1
 2:     $j' \leftarrow j \oplus 1$
 3:     $r_{min} \leftarrow \min\{r \in [1 \dots r_{j'}] \mid c \wedge x' \wedge \rho \wedge \mathtt{mY}'[j'][r] \neq \emptyset\}$
 4:     $N \leftarrow \mathtt{mY}[j'][r_{min}]$
 5: else
 6:     $j' \leftarrow j$
 7:     $r_{new} \leftarrow \min\{r \in [1 \dots r_{min}] \mid c \wedge x' \wedge \rho \wedge \mathtt{mY}'[j'][r] \neq \emptyset\}$
 8:     if $r_{new} < r_{min}$ then             // corresponds to $\rho_2$ from Alg. 1
 9:         $r_{min} \leftarrow r_{new}$
10:         $N \leftarrow \mathtt{mY}[j'][r_{min}]$
11:     else                                    // corresponds to $\rho_3$ from Alg. 1
12:         $i_{min} \leftarrow \min\{i \in [1 \dots m] \mid c \in \mathtt{mX}[j'][r_{min}][i]\}$
13:         $N \leftarrow \mathtt{mX}[j'][r_{min}][i_{min}]$
14:     end if
15: end if
16: $T \leftarrow c \wedge x' \wedge \rho \wedge N'$
17: $T \leftarrow QuantifyOutUnprimedVars(T)$
18: $T \leftarrow Unprime(T)$
19: return $T$
```

*Specifically, this creates a redundancy that the static construction suffers from, and of which JITS is devoid. When the current state satisfies $J^s_j$, Alg. 1 outputs as next states both* Z *(with $\mathcal{Z}_n = j \oplus 1$) and* $\mathtt{mX}[j][1][i]$ *for some $i \in [1 \dots m]$ (with $\mathcal{Z}_n = j$), since both $\rho_1$ and $\rho_3$ transitions are available. This may result in pairs of states that agree on all variable values in $\mathcal{X} \cup \mathcal{Y}$ and disagree on the value of $\mathcal{Z}_n$. Bloem et al. [5] attempt to tackle this case and minimize the transition system, but at the cost of a much larger BDD according to their evaluation. In JITS, due to the inherent eagerness, this problem does not exist.*

## 3.3 Soundness and Deadlock-Freedom

JITS next states algorithm (Alg. 3) is sound w.r.t. the original static approach, i.e., it does not return a state that the original static approach would not have produced. Moreover, JITS is deadlock-free w.r.t. the static approach, in the following sense: it always returns a correct set of next states, if at least one such state exists. The remainder of this subsection formalizes and proves these claims.

*3.3.1 Formalization.* We bridge the gap between the output format of Alg. 2, where (multiple) possible assignments of $\mathcal{Z}_n$ are part of the result BDD $T$, and Alg. 3, where the next justice goal is deterministic and maintained outside the BDD state space.

We use the tuple $(s, k)$, where $s \in 2^{\mathcal{Y}}$ and $k \in [1 \dots n]$, to describe an assignment $s$ to system variables over $\mathcal{Y}$ along with a justice goal index $k$. We describe the BDD outputs of the static approach and JITS next state functions as a set of such tuples.

Given $c \in 2^{\mathcal{X}\cup\mathcal{Y}}$, $x \in 2^{\mathcal{X}}$, and $j \in [1 \dots n]$, we denote $NS_{static}$ as the output of Alg. 2 and $NS_{JITS}$ as the output of Alg. 3, in the form of a set of tuples $(s, k)$. Let $j'$ be the value of the justice goal index by the end of Alg. 3 for inputs $c$, $x$, and $j$. Observe that for every tuple $(s, k) \in NS_{JITS}$, $k$ has the same value, which is $j'$, while for every tuple $(s, k) \in NS_{static}$, $k$ can be either $j$ or $j \oplus 1$ (according to $\mathcal{Z}_n$).

We proceed to state the theorems. We assume correct environment behavior, i.e., environment inputs that do not violate the environment assumptions. Soundness and deadlock-freedom for the case of an environment assumption violation are trivial.

Theorem 1 (JITS is sound). *For every $c \in 2^{\mathcal{X}\cup\mathcal{Y}}$, $x \in 2^{\mathcal{X}}$, and $j \in [1 \dots n]$:*

$$NS_{JITS} \subseteq NS_{static}$$

Note the use of $\subseteq$ in Thm. 1. Equality happens when a $\rho_3$ transition is the *only* transition possible according to Alg. 1. In this case, both JITS and the static approach keep the justice guarantee goal index intact and return next states in $\mathtt{mX}[j][r][i]$ for same values of $r$ and $i$. In all other cases, the relation between $NS_{JITS}$ and $NS_{static}$ is a strict inclusion. In these cases, $\rho_2$ or $\rho_1$ transitions are possible as well. While JITS will ignore any $\rho_3$ transitions then, the static approach will consider all possible transitions.

Theorem 2 (JITS is deadlock-free). *For every $c \in 2^{\mathcal{X}\cup\mathcal{Y}}$, $x \in 2^{\mathcal{X}}$, and $j \in [1 \dots n]$:*

$$NS_{static} \neq \emptyset \implies NS_{JITS} \neq \emptyset$$

*3.3.2 Proofs.* Denote the set of the intermediate results of the fixed-point computations BDDs as $FP$:

$$FP = \{\mathtt{Z}\} \cup \bigcup_{j,r} \{\mathtt{mY}[j][r]\} \cup \bigcup_{j,r,i} \{\mathtt{mX}[j][r][i]\}$$

Observe that *trans* BDD is a disjunction of transitions, each of which is a BDD of the following form:

$$ZN \wedge A \wedge \rho \wedge B' \wedge ZN'$$

$A \in FP$ represents a current set of states, $B' \in FP'$ represents a primed next set of states, $ZN$ represents an assignment to $\mathcal{Z}_n$, and $ZN'$ represents a next assignment to $\mathcal{Z}_n$. Therefore, line 1 in Alg. 2 can be rewritten as:

$$T \leftarrow (\mathcal{Z}_n = j) \wedge c \wedge x' \wedge \Big( \bigvee_{all\ transitions} ZN \wedge A \wedge \rho \wedge B' \wedge ZN' \Big)$$

Alg. 2 extracts the next states from $T$ by targeting only primed variables. Hence, we define $NS_{static}$ as follows:

$$NS_{static} = \{(s|_{\mathcal{Y}}, s|_{\mathcal{Z}_n}) \mid s \in x \wedge \rho \wedge \Big( \bigvee_{all\ transitions} B \wedge ZN \Big)\}$$

Observe that $ZN$'s support is the single variable $\mathcal{Z}_n$. We may restrict $\mathcal{Z}_n$ value to some $k$, and define $NS_{static}(k)$:

$$NS_{static}(k) = \{(s|_{\mathcal{Y}}, k) \mid s \in x \wedge \rho \wedge \Big( \bigvee_{\mathcal{Z}_n = k} B \Big)\}$$

Naturally, $NS_{static}(k) \subseteq NS_{static}$.

Similarly, JITS extracts the next states from $T$, which was computed in line 16 in Alg. 3. We define $NS_{JITS}$ as follows:

$$NS_{JITS} = \{(s, j') \mid s \in x \wedge \rho \wedge N\}$$

$N \in FP/\{\mathtt{Z}\}$ is the BDD of next states. $j'$ is the justice goal index at the end of Alg. 3 for the given inputs.

Proof of Thm. 1: JITS is sound. It is enough to show that $NS_{JITS} \subseteq NS_{static}(j')$. Considering that $x$ and $\rho$ BDDs appear on both sides, it is enough to show that $N \subseteq \bigcup_{\mathcal{Z}_n = j'} B$.

We consider three cases, based on the assignment to $N$ in Alg. 3:

- $N$ is the result of current state $c$ satisfying $J^s_j$ (lines 3-4). In this case, $j' = j \oplus 1$ and $N = \mathtt{mY}[j \oplus 1][r_{min}]$.
  There is a corresponding $\rho_1$ transition with $B = \mathtt{Z}$ that also updates $\mathcal{Z}_n$ to $j \oplus 1$. From Equ. 4 and Equ. 5 we know that $\mathtt{mY}[j \oplus 1][r_{min}] \subseteq \mathtt{mY}[j \oplus 1][r_j] = \mathtt{Z}$.
- $N$ is the result of the system progressing at least one step towards satisfying its justice goal (lines 9-10). In this case, $j' = j$ and $N = \mathtt{mY}[j][r_{min}]$ for the lowest computed $r_{min}$.
  Since previous $r$ was minimal such that $c \in \mathtt{mY}[j][r]$, there is a corresponding $\rho_2$ transition with $B = \mathtt{mY}[j][r-1]$ that retains $\mathcal{Z}_n$ value. From the fact that $r_{min} \leq r - 1$ and from Equ. 4 we know that $\mathtt{mY}[j][r_{min}] \subseteq \mathtt{mY}[j][r-1]$.
- $N$ is the result of the system staying at the same distance from the justice goal and instead forcing the environment to violate $J^e_i$ for the minimal $i$ (lines 12-13). In this case, $j' = j$ and $N = \mathtt{mX}[j][r_{min}][i_{min}]$.
  Since $i$ is minimal such that $c \in \mathtt{mX}[j][r_{min}][i]$ there is a corresponding $\rho_3$ transition with $B = \mathtt{mX}[j][r_{min}][i_{min}]$ that retains $\mathcal{Z}_n$ value.

□

Proof of Thm. 2: JITS is deadlock-free. First we show that $NS_{static} \neq \emptyset \implies NS_{static}(j') \neq \emptyset$. Consider two cases:

- There is a state $(s, j \oplus 1) \in NS_{static}$ that is a result of a $\rho_1$ transition, which also updates $\mathcal{Z}_n$ to $j \oplus 1$. Hence, current state satisfies $J^s_j$ and so $j' = j \oplus 1$ in JITS. Therefore this state is also in $NS_{static}(j')$.
- All states $(s, j) \in NS_{static}$ are the result of $\rho_2$ or $\rho_3$ transitions, which keep $\mathcal{Z}_n$ intact. Hence $j' = j$ in JITS as well, and same states are in $NS_{static}(j')$.

We complete the proof by showing that $NS_{static}(j') \neq \emptyset \implies NS_{JITS} \neq \emptyset$. Similarly to Thm. 1, we cancel out $x$ and $\rho$ and show that $\bigcup_{\mathcal{Z}_n = j'} B \neq \emptyset$ implies $N \neq \emptyset$.

We consider two cases:

- There is a $B \in FP$ in $\bigcup_{\mathcal{Z}_n = j'} B$ that is a result of a $\rho_1$ or $\rho_2$ transition. In Alg. 3, either current state satisfies the current justice goal (lines 3-4) or it is possible to decrease $r$ at least by one (lines 9-10). In any case, $N$ is assigned with one of the mY[][] BDDs.
- All $B \in FP$ in $\bigcup_{\mathcal{Z}_n = j'} B$ are the result of $\rho_3$ transitions. In this case, the controller cannot progress towards satisfying current justice goal. Alg. 3 doesn't change $j$ and $r$ indexes, and assigns $N$ with one of the mX[][][] BDDs (lines 12-13).

□

## 4 EXTENSIONS

Thanks to the decoupling of synthesis from system execution, JITS opens the way for synthesis independent extensions that can change system behavior at runtime, without the need to re-synthesize the specification. These extensions can be implemented as variants of the controller loading procedures and of Alg. 3. The extensions ought to be independent of each other, and ideally one should be able to combine multiple extensions during execution. We present three such example extensions.

### 4.1 Justice Guarantees Bookkeeping

Schlaipfer et al. [34] suggested "bookkeeping" as a method to satisfy each justice guarantee as quickly as possible. This is achieved via a Boolean array that keeps track of the guarantees satisfied in the current round and a master bit. On each step, the bookkeeping code checks whether any guarantee other than the current goal is satisfied, and marks the relevant index in the Boolean array according to the master bit. When the current justice goal is satisfied and the system chooses its next goal, it uses the array to retrieve the smallest index not yet marked. Once all indexes in the array are marked, it flips the master bit and starts a new round, with the "bookkeeping" array and the master bit having inverted values.

We have added optional bookkeeping feature as an extension to the basic JITS, with a slightly improved algorithm. Our method tracks unsatisfied guarantees both backwards and forwards, i.e., it does not wait until a full round of justice guarantees is completed before it checks on a specific guarantee, but does so on every step.

The bookkeeping extension changes the basic JITS in two locations. First, we add "bookkeeping" array management code before the call to JITS next states computation (Alg. 4). This code iterates over all justice guarantees: for each $J^s_k$ that is satisfied by the current state, the code marks the corresponding index $k$ in the *bk* array according to the *master* bit. Guarantees whose index is smaller than JITS' current goal are marked with the opposite value of *master*. This way, our method begins to track a new round even before the system has satisfied all the remaining guarantees on the list.

Second, we replaced line 2 in Alg. 3 with the code in Alg. 5, which returns the next justice goal according to *bk*, *master*, and the current goal $j$. The method iterates over the justice guarantees starting from the $(j+1)$-th place and finds the smallest index whose justice guarantee has not yet been satisfied. Once it reaches the end of the array, it flips the *master* to indicate that a round of justice guarantees has been completed, and then continues to iterate from 0 to $j$.

Note that in case all indexes in the *bk* array are marked, the algorithm simply returns to $j$. Specifically, this means that while working its way to $J^s_j$, the system has already satisfied all its other justice guarantees.

### 4.2 Recovery

Wong et al. [38] suggested controller implementations that attempt to recover in case of environment safety assumption violation. This is achieved by altering the original, static controller construction phase: removing the check whether a transition with a given environment input satisfies $\rho_e$ and is therefore allowed, and instead considering *all* next input possibilities. Then, the construction still attempts to find transitions that ensure that the next state can keep satisfying the winning conditions, although such a next state may not exist.

We have added optional recovery feature as an extension to the basic JITS. Recall that we assumed correct environment behavior in the previous sections. In this extension, we revisit this assumption

**Algorithm 4** Controller next states for JITS with bookkeeping

```
    input: $c \in 2^{\mathcal{X}\cup\mathcal{Y}}, x \in 2^{\mathcal{X}}, j \in 1 \dots n$
1:  for all $k \in [j \dots n]$ do
2:      if $bk[k] == master$ and $c \in J^s_k$ then
3:          $bk[k] \leftarrow \neg master$
4:      end if
5:  end for
6:  for all $k \in [1 \dots j-1]$ do
7:      if $bk[k] == \neg master$ and $c \in J^s_k$ then
8:          $bk[k] \leftarrow master$
9:      end if
10: end for
11: return NEXT STATES$(c, x, j)$                    // Call Alg. 3
```

**Algorithm 5** Next justice goal for JITS with bookkeeping

```
    input: $j \in [1 \dots n]$
1:  for all $k \in [j+1 \dots n]$ do
2:      if $bk[k] == master$ then
3:          return $k$
4:      end if
5:  end for
6:  $master \leftarrow \neg master$
7:  for all $k \in [1 \dots j-1]$ do
8:      if $bk[k] == master$ then
9:          return $k$
10:     end if
11: end for
12: return $j$
```

and do expect inputs that may violate environment safeties $\rho_e$ during runtime; instead of returning an error, we proceed with the JITS next states algorithm and attempt to find a valid next state. Recall that JITS always returns next states that are contained in Z, which is the winning states set. Still, as in the static controller recovery in [38], it may be the case that due to the environment safety violation, no such possible next state exists, and thus no such next state will be returned. Only in this case, the JITS recovery extension returns an error, same as in the basic JITS variant.

### 4.3 Configurable Set of Justice Guarantees

Configurability is a known extra functional requirement. Given a family of closely related products, rather than developing a separate software for each, one is encouraged to design and develop a single product that can be customized to a specific subset of the complete requirements at deployment time. Configurability is related to the idea and methodology of software product lines. To apply configurability to our domain of synthesized reactive systems, a configurable controller may be one that supports a set of justice guarantees that is chosen from a larger superset of justice guarantees.

In the static approach, in order to achieve such configurability, a unique controller must be synthesized for each relevant subset of the justice guarantees, or alternatively, a unique flag variable must be maintained inside the specification to represent the relevance of each justice guarantee, adding to the overall state space and complexity of the specification.

In JITS however, this kind of configurability can be achieved at execution time, via an extension that receives at load time a set of required justice guarantee indexes as an argument. During the extraction phase, JITS skips those indexes that are not in the set, thus loading only the BDDs relevant to the remaining indexes. This solution does not affect synthesis time. It only requires that the complete specification, containing *all* the guarantees, is realizable. Naturally, when the complete specification is realizable, any variant with only a subset of the justice guarantees is realizable as well. Moreover, any variant will be available at deployment time, since it depends only on the required justice guarantees indexes argument.

We have implemented this functionality of configurable set of justice guarantees as an optional extension to the basic JITS. The extension receives a set of required justice guarantee indexes as an argument and *loads only the BDDs relevant to these indexes*. Note that an execution based on the original basic JITS is still correct, as the required set of justice guarantees is a subset of the complete superset. However, one may expect that given a strict subset of justice guarantees indexes, the extension typically loads faster and takes less memory than the original basic JITS.

## 5 IMPLEMENTATION AND EVALUATION

We have implemented JITS in the open source Spectra [2, 25], which already includes a GR(1) synthesizer and implementations of several additional analyses. Our implementation in Java uses BDDs [6] via the CUDD 3.0 [36] package. Realizability checking includes heuristics described in [15].

All specifications used in our evaluation, the raw data we collected, and the code to reproduce our experiments, are available in supporting materials [1].

We consider the following research questions.

RQ0 How does static controller construction time compare with realizability checking time?

RQ1 How does JITS compare with existing approaches during *synthesis*, w.r.t. time, memory usage, and size of its output representation?

RQ2 How does JITS compare with existing approaches during *system execution*, w.r.t. load time, memory usage, and step time?

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

### 5.1 Corpus of Specifications

We evaluate JITS using two sets of benchmark specifications from the literature.

First, ARM's AMBA AHB arbiter (AMBA) and IBM's generalized buffer (GenBuf). These two specifications have been extensively used in the GR(1) literature for evaluation purposes, e.g., in [8, 15, 18, 20, 29, 34]. They are valuable as they are parametric and therefore suitable for examining scalability.

Second, specifications from the SYNTECH benchmarks [15], available from the Spectra website. These specifications were written by 3rd year CS undergrads in class projects taught by the authors of [15]. The benchmarks have been used in some works [9, 26, 30].

They include several versions of each specification submitted by the students. We use here the *largest* specifications, i.e., all specifications where the total number of variables is greater than 40. For each, we use the final specification the students submitted. In total, we use 5 specifications from the SYNTECH benchmarks.

## 5.2 Validation

We have systematically and automatically validated the correctness of our implementation by creating a test that runs the static approach controller and the JITS step by step in parallel, against the same environment behaviors, and in every step, checks that the choice of next state by the latter is included in the possible next states suggested by the former. We executed this test over all the specifications in our corpus, against a random environment, for runs of 10,000 steps. This validation increases our confidence in the correctness of our ideas and their implementation.

## 5.3 Experiment Setups

We used two separate experiment setups, one to examine synthesis (RQ0, RQ1), the other to examine system execution (RQ2).

*5.3.1 Synthesis setup.* For the synthesis setup, we compared JITS against two existing tools: Slugs [13] and Spectra [25]. We used the latest versions of Slugs and Spectra from GitHub and only added minimal code to measure time and memory. Both tools use (variants of) the original approach from [5]. Spectra realizability checking includes all heuristics described in [15]. We chose to compare Spectra against Slugs, and not against other GR(1) synthesizers, e.g., RATSY [4], as according to results in [15], the larger the specification, the faster Slugs realizability checking time compared to that of RATSY. Moreover, Slugs is a more recent tool; the version of Slugs that we used is the latest available, from June 2018.

We measured synthesis time, memory used, and size of output as follows. Synthesis time is the time of realizability checking combined with controller construction or storing time. Memory used is the number of active BDD nodes as reported by CUDD. Note that memory usage is as an important metric, as indeed some specifications could not be synthesized/executed successfully by the static approach due to an out-of-memory error. The size of the output is the size of the output on disk.

We used a fixed timeout of two hours. We mark timeouts by –. In some cases Slugs resulted in an out-of-memory error during synthesis. We mark these cases by `xx`. When synthesis failed due to any of these cases, we mark the other measures as `n/a`. All values we report are median values of 10 runs per specification per tool. Times we report are measured by Java in milliseconds. Even though the algorithms we deal with are deterministic, we repeated each experiment 10 times since JVM garbage collection and CUDD garbage collection add variance to running times.

For the specifications available for Slugs, AMBA and GenBuf, we took the equivalent Spectra specifications available from [15], and used them in the comparison between the three tools. For the specifications from the SYNTECH benchmarks, which are richer (e.g., include patterns) and therefore not available for Slugs, we compared only between Spectra and JITS.

In this setup, we run all experiments on an ordinary PC, Intel Xeon W-2133 CPU 3.6GHz, 32GB RAM with Windows 10 64-bit OS, Java 8 64Bit, and CUDD 3 compiled for 64Bit, using only a single core of the CPU.

**Table 1: Comparison of construction vs. realizability running times for Slugs and Spectra**

| Specification | Synthesis Time (sec) | | | |
|---|---|---|---|---|
| | Slugs | | Spectra | |
| | Real. | Constr. | Real. | Constr. |
| AMBA1 | 0.20 | 0.02 | 0.20 | 0.17 |
| AMBA2 | 3.32 | 0.25 | 1.52 | 1.63 |
| AMBA3 | 44.18 | 2.94 | 21.18 | 130.33 |
| AMBA4 | 2086.30 | 30.17 | 132.50 | 1030.89 |
| AMBA5 | – | n/a | 149.19 | – |
| AMBA10 | – | n/a | 2753.65 | – |
| GenBuf5 | 1.50 | 0.65 | 0.18 | 0.27 |
| GenBuf10 | 0.58 | 15.18 | 0.30 | 0.75 |
| GenBuf20 | 1.46 | xx | 1.53 | 6.42 |
| GenBuf30 | 3.88 | xx | 3.99 | 25.90 |
| GenBuf40 | 16.09 | xx | 8.84 | 73.18 |
| GenBuf90 | 331.15 | xx | 91.23 | 1486.84 |
| AirportShuttle | | | 48.04 | 7.79 |
| Junction2 | | | 0.83 | 13.20 |
| Junction3 | | | 9.21 | 1157.32 |
| RoboticArm | | | 3.97 | 12.33 |
| SimpleVehicle | | | 2.90 | 8.39 |

*5.3.2 System execution setup.* For the system execution setup, we compared JITS with and without the bookkeeping extension against Spectra. We could not compare againt Slugs because, to the best of our knowledge, it does not provide a direct controller execution API. We measured load time, memory used, and step time. We load the controller and execute it for 10,000 steps against a random environment.

Load time includes loading the BDDs, caching as described in Sect. 3.2, and the initial step. Memory used is the number of active BDD nodes as reported by CUDD library at the end of the execution. Step time is the time to compute the next step. We use average step time over the 10,000 steps. Times we report are median values of 10 runs (of 10,000 steps each), per specification per tool, measured by Java in milliseconds. Again, we repeated each experiment 10 times to accommodate for the variance in running times.

In some cases Spectra resulted in an out-of-memory error during load. We mark these cases by `xx`, and consequently we mark the other measures for the same specification by `n/a`.

In this setup, we run all experiments on a Raspberry Pi 2 Model B with 900MHz quad-core ARM Cortex-A7 CPU, 1GB of RAM with Raspbian 8 32-bit OS, Java 8 32Bit, and CUDD 3 compiled for 32Bit, using only a single core of the CPU. We chose this rather weak computer (different and much weaker than the computer on which we run synthesis), as an example of a target platform for a synthesized controller, e.g., a robot.

## 5.4 Results: Realizability vs. Construction

Table 1 presents a comparison between realizability and controller construction times, in seconds, for Spectra and Slugs.

The results show that in both tools, for most of the specifications, construction takes considerably more time than realizability (except for the AMBA specifications for Slugs, and AirportShuttle). For AMBA5 and up, Slugs could not complete realizability checking and Spectra could not complete construction within the 2 hours timeout. For GenBuf20 and up, Slugs construction resulted in an out-of-memory error. In the extreme case of Junction3, more than 99% of the total synthesis time was spent on controller construction.

To answer RQ0: We have evidence showing that in both tools and for most of the specifications, construction takes significantly more time than realizability checking. This evidence is consistent with data reported on in [5]. These results strengthen the motivation for JITS, which skips controller construction.

## 5.5 Results: Synthesis

Table 2 presents the results for synthesis. For each of the three tools, we show construction / storing time in seconds, number of active nodes in thousands, and space on disk in Mb. Note that JITS uses Spectra realizability checking but replaces Spectra construction with storing.

**AMBA and GenBuf**. In terms of construction time, the results show that while Slugs and Spectra grow very fast with the size of the specification (number of masters, number of senders), JITS storing time grows, but much slower. In absolute terms, in all but one specification (the smallest, AMBA1), JITS performs better than both other tools, sometime more than an order of magnitude better. In general, the larger the specification, the better the performance of JITS relative to the two other tools.

Similarily, in terms of active nodes and space on disk, the results show that while Slugs and Spectra grow very fast with the size of the specification, JITS growth is much slower.

**SYNTECH**. The table further presents the results on specifications from the SYNTECH benchmarks. This part of the table does not include data for Slugs as these specifications are rich; they include patterns, monitors, etc., which are not supported by Slugs.

In terms of construction time, Spectra is in all cases higher than JITS storing time. As expected, storing is much faster than construction.

In terms of active nodes and space on disk, the results show that JITS requires less memory and its output's size on disk is smaller, in most cases, by a factor of 3 or more.

To answer RQ1: JITS outperforms existing approaches w.r.t. construction time, memory usage, and output size, sometimes by orders of magnitude. Moreover, JITS scales better than the other two tools. The larger the specification, the better JITS synthesis time, memory usage, and size on disk compared to the other two tools.

## 5.6 Results: System Execution

Table 3 presents the results for system execution. For each of the two tools, we show load time in seconds, number of active nodes by the end of the execution in thousands, and step time in milliseconds. For the number of active nodes and step time we show results also for JITS with bookkeeping. We do not report load time for JITS with bookkeeping since this extension is independent of this part.

**AMBA and GenBuf**. In terms of load time, the results show that while Spectra load time grows very fast with the size of the specification (number of masters, number of senders), JITS load time grows very slowly. In absolute terms, JITS load time is always faster than that of Spectra.

In terms of active nodes and single step time, the results show obvious advantage to JITS (with or without bookkeeping) as the specifications grow larger.

**SYNTECH**. The table further presents the results on specifications from the SYNTECH benchmarks. In terms of load time, active nodes, and single step time, the results do not show a clear advantage to one of the approaches over the other. In particular, in terms of single step time, we do not observe that one approach is consistently better than the other.

We consider these results, for all specifications, to be a good point for JITS, as, apriori, one may expect that given the computations it has to do in every step, we would observe much slower step times for JITS than for Spectra. However, recall that the static *trans* BDD, which holds all the transitions, can be very large. Loading and applying logical operations on large BDDs can be costly, and vary greatly based on variable ordering. In contrast, while typically JITS has a higher number of operations per step, the BDDs it manipulates are much smaller.

To answer RQ2: JITS does not compromise system execution performance, and in some cases achieves even better results than existing approaches w.r.t. load time, memory usage, and step time. We also see that in many cases, JITS scales better than Spectra. The larger the specification, the better JITS load time, memory usage, and step time compared to Spectra.

## 5.7 Threats to Validity

We briefly discuss threats to the validity of our results.

First, the symbolic computations are not trivial and our implementation may have bugs. To mitigate this, we performed a thorough validation using all specifications available to us, see Sect. 5.2.

Second, even though the algorithms we deal with are deterministic, garbage collection of the JVM and of CUDD add variance to running times. To mitigate this, we repeated each experiment 10 times and we report median values. CUDD dynamic reordering may result in additional variance. During realizability checking, we used CUDD's default dynamic reordering for all tools. This is common practice in the related literature, e.g., [5, 26]. During construction, we used the default for each tool[2]. During system execution, we turned off dynamic reordering for Spectra and JITS because, for both, it creates unpredictable occasional extremely slow steps.

Third, the specifications we used in the evaluation may not be representative of real-world specifications. To alleviate this threat,

[2]Slugs turns it off, Spectra and JITS keep it on.

Table 2: **Comparison of Slugs (static), Spectra (static), and JITS: during synthesis**

| Specification | Construction Time (sec) Slugs Constr. | Spectra Constr. | JITS Stor. | Memory (# AN in thousands) Slugs | Spectra | JITS | Size on Disk (Mb) Slugs | Spectra | JITS |
|---|---|---|---|---|---|---|---|---|---|
| AMBA1 | **0.02** | 0.17 | 0.06 | 8.74 | 9.06 | **4.08** | 0.09 | 0.35 | **0.05** |
| AMBA2 | 0.25 | 1.63 | **0.16** | 57.07 | 69.77 | **13.07** | 0.69 | 2.79 | **0.24** |
| AMBA3 | 2.94 | 130.33 | **1.08** | 562.94 | 904.68 | **65.29** | 9.48 | 33.07 | **0.94** |
| AMBA4 | 30.17 | 1030.89 | **4.34** | 2649.62 | 2614.18 | **139.69** | 62.63 | 107.64 | **1.95** |
| AMBA5 | n/a | – | **10.83** | n/a | n/a | **277.75** | n/a | n/a | **6.24** |
| AMBA10 | n/a | – | **140.81** | n/a | n/a | **1641.08** | n/a | n/a | **46.00** |
| GenBuf5 | 0.65 | 0.27 | **0.09** | 40.96 | 15.13 | **6.49** | 6.52 | 0.47 | **0.09** |
| GenBuf10 | 15.18 | 0.75 | **0.17** | 13.53 | 30.07 | **6.64** | 210.50 | 1.01 | **0.11** |
| GenBuf20 | xx | 6.42 | **0.33** | n/a | 130.16 | **26.09** | n/a | 4.28 | **0.56** |
| GenBuf30 | xx | 25.90 | **1.10** | n/a | 289.60 | **42.28** | n/a | 10.76 | **0.65** |
| GenBuf40 | xx | 73.18 | **2.28** | n/a | 475.05 | **50.42** | n/a | 16.92 | **0.78** |
| GenBuf90 | xx | 1486.84 | **33.68** | n/a | 2428.81 | **199.09** | n/a | 90.80 | **4.23** |
| AirportShuttle |  | 7.79 | **3.12** |  | 145.79 | **112.40** |  | 2.00 | **1.71** |
| Junction2 |  | 13.20 | **0.27** |  | 294.16 | **13.96** |  | 10.43 | **0.29** |
| Junction3 |  | 1157.32 | **5.54** |  | 5089.20 | **139.77** |  | 189.78 | **4.87** |
| RoboticArm |  | 12.33 | **5.74** |  | 282.34 | **67.91** |  | 9.69 | **1.96** |
| SimpleVehicle |  | 8.39 | **0.79** |  | 197.01 | **47.63** |  | 6.44 | **1.23** |

Table 3: **Comparison of Spectra (static) and JITS: during system execution**

| Specification | Load Time (sec) Spectra | JITS | Memory (# AN in thousands) Spectra | JITS | JITS w/ Bk. | Single Step Time (ms) Spectra | JITS | JITS w/ Bk. |
|---|---|---|---|---|---|---|---|---|
| AMBA1 | 1.35 | **1.17** | 31.78 | **27.44** | 30.24 | 0.86 | 0.81 | **0.80** |
| AMBA2 | 5.14 | **1.72** | 126.47 | **104.89** | 110.98 | **1.78** | 2.19 | 2.14 |
| AMBA3 | 108.58 | **5.62** | 624.18 | **231.46** | 236.48 | 2.91 | 2.34 | **2.33** |
| AMBA4 | xx | **22.93** | n/a | **355.30** | 420.65 | n/a | **3.22** | 3.26 |
| AMBA5 | xx | **119.90** | n/a | **611.46** | 711.47 | n/a | 4.22 | **4.24** |
| AMBA10 | xx | **2539.67** | n/a | **2488.72** | 2724.19 | n/a | **11.58** | 11.95 |
| GenBuf5 | 1.64 | **1.43** | 98.30 | **49.77** | 66.55 | 1.95 | **1.52** | 1.67 |
| GenBuf10 | 2.59 | **1.57** | 149.22 | **98.18** | 135.81 | 2.76 | 2.50 | **2.47** |
| GenBuf20 | 13.84 | **3.17** | 349.35 | 165.27 | **161.71** | 6.03 | 5.20 | **4.97** |
| GenBuf30 | 58.46 | **6.30** | 550.22 | 238.35 | **223.16** | 9.77 | 9.37 | **9.14** |
| GenBuf40 | 170.52 | **12.61** | 980.38 | 422.54 | **394.45** | 16.12 | 14.52 | **13.48** |
| GenBuf90 | xx | **114.39** | n/a | 1646.85 | **1580.73** | n/a | 56.38 | **53.11** |
| AirportShuttle | **5.33** | 25.55 | **567.82** | 823.21 | 955.25 | **4.06** | 4.45 | 4.56 |
| Junction2 | 15.20 | **2.19** | **1513.84** | 1607.80 | 1834.58 | **4.80** | 5.20 | 5.43 |
| Junction3 | xx | **54.46** | n/a | **5321.92** | 6031.35 | n/a | 60.55 | **59.62** |
| RoboticArm | **20.63** | 24.23 | **806.99** | 1214.83 | 1258.72 | **7.25** | 8.22 | 8.17 |
| SimpleVehicle | 132.38 | **10.55** | 2249.34 | **1520.19** | 1631.03 | 8.82 | 8.18 | **8.07** |

we used several existing benchmarks from the literature. For the AMBA and GenBuf we used different number of masters, as is common practice in related literature. For the SYNTECH specifications, we chose all specifications where number of variables is greater than 40, see Sect. 5.1.

Finally, our system execution setup runs the controller for 10,000 steps against a random (yet correct) environment, see Sect. 5.3. In practice, environment behavior is not expected to be random. Real environment behavior is not available for the specifications at hand. That said, we are not aware of reasons to believe that next step performance against a real environment would be different than what we observed, neither for Spectra nor for JITS.

## 6 RELATED WORK

GR(1) synthesis was introduced in [31]. It has since been used and investigated by many, including, e.g., Kress-Gazit et al. [19, 39], who used GR(1) in robotics; Maoz and Ringert [23], who showed GR(1) synthesis for specification patterns; Cavezza and Alrajeh [8], who investigated assumptions refinement in unrealizable GR(1) specifications, to list a few. Several tools support GR(1) synthesis, e.g., RATSY [4], Slugs [13], and Spectra [25]. None of these works and tools considered just-in-time synthesis.

Heuristics to improve the running time performance of GR(1) synthesis have been suggested in [10, 15, 34]. Specifically, Firman et al. [15] present and evaluate heuristics at the level of the controlled predecessor computation and BDDs, as well as heuristics for early detection of fixed-points and early detection of unrealizability. The work, however, is *limited to accelerating the realizability checking phase*. It has no effect on the controller construction phase and no effect on system execution performance. The suggested heuristics are implemented in Spectra. As noted earlier, our comparison against Spectra includes all these heuristics.

Dathathri and Murray [10] suggest efficient GR(1) synthesis for specifications with singleton liveness guarantees, i.e, guarantees that are limited to a single state. Our work, however, is independent of any such restrictions on the GR(1) specifications.

Schlaipfer et al. [34] present an approach to GR(1) synthesis without, what they refer to as a monolithic strategy, applied to hardware synthesis. They synthesize several separate strategies, one for each justice guarantee, and manage them via auxiliary circuits. This reduces strategy construction time and memory usage at the expense of a major increase in circuit size. Evaluation is limited to the AMBA specifications.

Heuristics to improve realizability and strategy construction times have also been suggested in the series of SYNTCOMP competitions, e.g., [17]. The competition measures realizability checking and strategy construction times where the targets are hardware circuits, not software systems. Evidently, JITS is applicable to synthesized software systems, not hardware circuits. Moreover, the specifications used in the competition are either LTL or safety-only specifications. Our work focuses on GR(1) specifications, which on the one hand are not as expressive as LTL but on the other hand include not only safety but also justice assumptions and guarantees. Thus, we cannot evaluate our work against the competition's tools and specifications.

In particular, Strix [21] is an LTL reactive synthesis tool, which has recently gained attention due to promising results in the SYNTCOMP competition. Strix decomposes the LTL formula into simpler formulas, translates these on-the-fly into deterministic parity automata, solves the intermediate parity games using strategy iteration, and finally translates the winning strategy, if it exists, into a Mealy machine or an AIGER circuit with optional minimization using external tools. Only during the last step, the authors discuss encoding of the Mealy machine as a BDD. Hence, one may consider applying the concept of JITS (perhaps redesigned to adapt to the domain of deterministic parity automata instead of GR(1)) to revise this encoding and open the way for efficient execution of controllers synthesized with Strix.

Finally, one may consider JITS to be similar to on-the-fly algorithms in model checking and games analysis, e.g., [7, 35]. However, these algorithms are meant to improve the performance of verification or realizability checking. JITS, in contrast, is a novel means for the *execution* of the synthesized system. It uses the same realizability checking as the static approaches to GR(1) synthesis.

## 7 CONCLUSION

We introduced JITS, just-in-time synthesis for GR(1). JITS provides a novel, fast, and flexible means to execute synthesized controllers. It does not compromise the correct-by-construction promise, while, from architectural and methodological perspectives, it opens the way for the decoupling of realizability checking and system execution.

We have implemented JITS on top of the Spectra synthesizer. We showed that compared to existing, static approaches, JITS greatly improves overall synthesis time, memory usage, and output size, and scales better with larger specifications. Moreover, we showed that JITS remains competitive compared to the static approach in system execution performance w.r.t. time and space.

As future work we consider the following research directions. First, one may investigate how to adapt JITS to specific system execution platforms, and how to control the tradeoff between performance and memory consumption of JITS execution, potentially, for example, using BDD reordering or different, possibly adaptive, caching mechanisms.

Second, to further enhance the configurability of synthesized controllers at load time, one may suggest different storing mechanisms.

Finally, most recently, Amram et al. have presented GR(1)*, an extension of GR(1) with existential guarantees [3]. One may consider to define and implement JITS for GR(1)*.

## ACKNOWLEDGEMENTS

We thank Or Pistiner and Yaniv Sa'ar for helpful advice. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

## REFERENCES

[1] [n.d.]. JITS Supporting Materials Website. http://smlab.cs.tau.ac.il/syntech/jits/.

[2] [n.d.]. Spectra Website. http://smlab.cs.tau.ac.il/syntech/spectra/.

[3] Gal Amram, Shahar Maoz, and Or Pistiner. 2019. GR(1)*: GR(1) Specifications Extended with Existential Guarantees. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.), Vol. 11800. Springer, 83–100. https://doi.org/10.1007/978-3-030-30942-8_7

[4] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. 2010. RATSY - A New Requirements Analysis Tool with Synthesis. In *CAV (LNCS)*, Vol. 6174. Springer, 425–429. http://dx.doi.org/10.1007/978-3-642-14295-6_37

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

[6] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[7] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. 2005. Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings (Lecture Notes in Computer Science)*, Martín Abadi and Luca de Alfaro (Eds.), Vol. 3653. Springer, 66–80. https://doi.org/10.1007/11539452_9

[8] Davide G. Cavezza and Dalal Alrajeh. 2017. Interpolation-Based GR(1) Assumptions Refinement. In *TACAS (LNCS)*, Vol. 10205. 281–297. https://doi.org/10.1007/978-3-662-54577-5_16

[9] Davide G. Cavezza, Dalal Alrajeh, and András György. 2019. Minimal Assumptions Refinement for GR(1) Specifications. *CoRR* abs/1910.05558 (2019). arXiv:1910.05558 http://arxiv.org/abs/1910.05558

[10] Sumanth Dathathri and Richard M. Murray. 2017. Decomposing GR(1) games with singleton liveness guarantees for efficient synthesis. In *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017.* IEEE, 911–917. https://doi.org/10.1109/CDC.2017.8263775

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

[12] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *ICSE*. ACM, 411–420.

[13] Rüdiger Ehlers and Vasumathi Raman. 2016. Slugs: Extensible GR(1) Synthesis. In *CAV (LNCS)*, Vol. 9780. Springer, 333–339. https://doi.org/10.1007/978-3-319-41540-6_18

[14] Ioannis Filippidis, Sumanth Dathathri, Scott C. Livingston, Necmiye Ozay, and Richard M. Murray. 2016. Control design for hybrid systems with TuLiP: The Temporal Logic Planning toolbox. In *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016.* IEEE, 1030–1041. https://doi.org/10.1109/CCA.2016.7587949

[15] Elizabeth Firman, Shahar Maoz, and Jan Oliver Ringert. 2020. Performance heuristics for GR(1) synthesis and related algorithms. *Acta Inf.* 57, 1-2 (2020), 37–79. https://doi.org/10.1007/s00236-019-00351-9

[16] D. Harel and A. Pnueli. 1985. *On the Development of Reactive Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 477–498. https://doi.org/10.1007/978-3-642-82453-1_17

[17] Swen Jacobs, Roderick Bloem, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Philipp J. Meyer, Thibaud Michaud, Mouhammad Sakr, Salomon Sickert, Leander Tentrup, and Adam Walker. 2019. The 5th Reactive Synthesis Competition (SYNTCOMP 2018): Benchmarks, Participants & Results. *CoRR* abs/1904.07736 (2019). arXiv:1904.07736 http://arxiv.org/abs/1904.07736

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

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

[20] Aviv Kuvent, Shahar Maoz, and Jan Oliver Ringert. 2017. A symbolic justice violations transition system for unrealizable GR(1) specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 362–372. https://doi.org/10.1145/3106237.3106240

[21] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. 2020. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica* 57, 1-2 (2020), 3–36. https://doi.org/10.1007/s00236-019-00349-3

[22] Spyros Maniatopoulos, Philipp Schillinger, Vitchyr Pong, David C. Conner, and Hadas Kress-Gazit. 2016. Reactive high-level behavior synthesis for an Atlas humanoid robot. In *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, Danica Kragic, Antonio Bicchi, and Alessandro De Luca (Eds.). IEEE, 4192–4199. https://doi.org/10.1109/ICRA.2016.7487613

[23] Shahar Maoz and Jan Oliver Ringert. 2015. GR(1) synthesis for LTL specification patterns. In *ESEC/FSE*. ACM, 96–106. http://doi.acm.org/10.1145/2786805.2786824

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

[25] Shahar Maoz and Jan Oliver Ringert. 2019. Spectra: A Specification Language for Reactive Systems. *CoRR* abs/1904.06668 (2019). arXiv:1904.06668 http://arxiv.org/abs/1904.06668

[26] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. 2019. Symbolic repairs for GR(1) specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan (Eds.). IEEE / ACM, 1016–1026. https://dl.acm.org/citation.cfm?id=3339632

[27] Shahar Maoz and Yaniv Sa'ar. 2011. AspectLTL: an aspect language for LTL specifications. In *AOSD*, Paulo Borba and Shigeru Chiba (Eds.). ACM, 19–30. http://doi.acm.org/10.1145/1960275.1960280

[28] Shahar Maoz and Yaniv Sa'ar. 2012. Assume-Guarantee Scenarios: Semantics and Synthesis. In *MODELS (LNCS)*, Vol. 7590. Springer, 335–351. http://dx.doi.org/10.1007/978-3-642-33666-9_22

[29] Shahar Maoz and Rafi Shalom. 2020. Inherent Vacuity for GR(1) specifications. In *ESEC/FSE*. To appear.

[30] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. 2019. Specification Patterns for Robotic Missions. *CoRR* abs/1901.02077 (2019). arXiv:1901.02077 http://arxiv.org/abs/1901.02077

[31] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of Reactive(1) Designs. In *VMCAI (LNCS)*, Vol. 3855. Springer, 364–380. http://dx.doi.org/10.1007/11609773_24

[32] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[33] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *POPL*. ACM Press, 179–190.

[34] Matthias Schlaipfer, Georg Hofferek, and Roderick Bloem. 2011. Generalized Reactivity(1) Synthesis without a Monolithic Strategy. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers.* 20–34.

[35] Stefan Schwoon and Javier Esparza. 2005. A Note on On-the-Fly Verification Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.), Vol. 3440. Springer, 174–190. https://doi.org/10.1007/978-3-540-31980-1_12

[36] Fabio Somenzi. 2015. CUDD: CU Decision Diagram Package Release 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf.

[37] Adam Walker and Leonid Ryzhyk. 2014. Predicate abstraction for reactive synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014.* IEEE, 219–226. http://dx.doi.org/10.1109/FMCAD.2014.6987617

[38] Kai Weng Wong, Rüdiger Ehlers, and Hadas Kress-Gazit. 2014. Correct High-level Robot Behavior in Environments with Unexpected Events. In *Robotics: Science and Systems X, University of California, Berkeley, USA, July 12-16, 2014*, Dieter Fox, Lydia E. Kavraki, and Hanna Kurniawati (Eds.). https://doi.org/10.15607/RSS.2014.X.012

[39] Kai Weng Wong, Rüdiger Ehlers, and Hadas Kress-Gazit. 2018. Resilient, Provably-Correct, and High-Level Robot Behaviors. *IEEE Trans. Robotics* 34, 4 (2018), 936–952. https://doi.org/10.1109/TRO.2018.2830353