Noname manuscript No.
(will be inserted by the editor)

A Symbolic Algorithm for Lazy Synthesis of Eager Strategies

Swen Jacobs · Mouhammad Sakr

the date of receipt and acceptance should be inserted later

Abstract We present an algorithm for solving two-player safety games that combines a mixed forward/backward search strategy with a symbolic representation of the state space. By combining forward and backward exploration, our algorithm can synthesize strategies that are eager in the sense that they try to prevent progress towards the error states as soon as possible, whereas standard backwards algorithms often produce permissive solutions that only react when absolutely necessary. We provide experimental results for two classes of crafted benchmarks, the benchmark set of the Reactive Synthesis Competition (SYNTCOMP) 2017, as well as a set of randomly generated benchmarks. The results show that our algorithm in many cases produces more eager strategies than a standard backwards algorithm, and solves a number of benchmarks that are intractable for existing tools. Finally, we observe a connection between our algorithm and a recently proposed algorithm for the synthesis of controllers that are robust against disturbances, pointing to possible future applications.

Keywords Reactive Systems · Synthesis · Safety Games

1 Introduction

Automatic synthesis of digital circuits from logical specifications is one of the most ambitious and challenging problems in circuit design. The problem was first identified by Church [6]: given a requirement ϕ on the input-output behavior of a boolean circuit, compute a circuit C that satisfies ϕ. Since then, several

S. Jacobs
CISPA Helmholtz Center for Information Security, Germany
Tel.: +49-681-3205666
E-mail: jacobs@cispa.saarland

M. Sakr
CISPA Helmholtz Center for Information Security, Germany
Tel.: +49-681-3205662
E-mail: sakr@react.uni-saarland.de

approaches have been proposed to solve the problem [4,24], which is usually viewed as a game between two players: the system player tries to satisfy the specification and the environment player tries to violate it. If the system player has a winning strategy for the game, then this strategy represents a circuit that is guaranteed to satisfy the specification. Recently, there has been much interest in approaches that leverage efficient data structures and automated reasoning methods to solve the synthesis problem in practice [12,10,26,14,2, 20].

In this paper, we restrict our attention to safety specifications. In this setting, most of the successful implementations *symbolically* manipulate sets of states via their characteristic functions, represented as Binary Decision Diagrams (BDDs) [17]. The "standard" algorithm works backwards from the unsafe states and computes the set of all states from which the environment can force the system into these states. The negation of this set is the winning region of the system player, and it defines the most permissive winning strategy: the strategy that allows any move, except those that leave the winning region. Computing the winning region is a very general and in some cases desirable approach, since all winning strategies must operate within the winning region, which can therefore be used to find a more specific strategy with desirable properties in a second computation step. However, this approach may be suboptimal if the generality of the most permissive strategy is not necessary—either because any solution would fit, or because we know how to compute a strategy with the desired properties directly and more efficiently.

We aim at the generation of *eager* strategies that avoid progress towards the error whenever possible, in stark contrast to the most permissive strategy. Such strategies are desirable in many applications, e.g., if the system should be tolerant to hardware faults or perturbations in the environment [9]. When used to find eager strategies, the standard algorithm may first spend a lot of time on the exploration of states that could easily be avoided by the system player, and only in the second step will find that they are not necessary for the eager solution. To avoid this and keep the explored state space small, some kind of forward search from the initial states is necessary. However, for pure forward search no efficient symbolic algorithm is known, and most existing approaches that integrate forward search into backwards algorithms do so in a rather limited fashion [17]. Notably, Brenguier et al. [3] have integrated forward search into an abstraction-based synthesis algorithm, however their experimental evaluation showed only few benchmarks where the approach was faster than the standard backwards approach.

1.1 Contributions

In this work, we introduce a lazy synthesis algorithm that combines a forward search for candidate solutions with backward model checking of these candidates. All operations are such that they can be efficiently implemented with a fully symbolic representation of the state space and the space of candidate so-

lutions. The combined forward/backward exploration allows us to detect small subsets of the winning region that are sufficient to define a winning strategy. As a result, it produces less permissive solutions than the standard approach and can solve certain classes of problems more efficiently.

We evaluate a prototype implementation of our algorithm on three sets of benchmarks, including the benchmark set of the Reactive Synthesis Competition (SYNTCOMP) 2017 [16]. We show that on many benchmarks our algorithm detects remarkably small subsets of the winning region that are sufficient to solve the synthesis problem: on the benchmark set from SYNTCOMP 2017, the biggest measured difference is by a factor of 10^{68}. Moreover, it solves a number of instances that have not been solved by any participant in SYNTCOMP 2017.

Finally, we observe a relation between our algorithm and the approach of Dallal et al. [9] for systems with perturbations, and provide the first implementation of their algorithm as a variant of our algorithm. On the SYNTCOMP benchmark set, we show that whenever a given benchmark admits controllers that give stability guarantees under perturbations, then our lazy algorithm will terminate after exploring a small subset of the winning region and can provide quantitative safety guarantees similar to those of Dallal et al. without any additional cost.

A preliminary version of this paper was presented at the International Symposium on Automated Technology for Verification and Analysis (ATVA), 2018 [18].

1.2 Overview

We introduce the synthesis problem in Section 2 and recapitulate a number of existing approaches to solve it in Section 3. In Section 4 we introduce our lazy synthesis algorithm, followed by a number of optimizations in Section 5. The experimental evaluation of our algorithms is presented in Section 6, and we discuss further experiences with implementing forward exploration in Section 7.1. In Section 8 we discuss connections of our approach to approaches for the synthesis of controllers that are resilient against certain faults, before we conclude in Section 9.

2 Preliminaries

Given a specification ϕ, the reactive synthesis problem consists in finding a system that satisfies ϕ in an adversarial environment. The problem can be viewed as a game between two players, Player 0 (the system) and Player 1 (the environment), where Player 0 chooses controllable inputs and Player 1 chooses uncontrollable inputs to a given transition function. In this paper we consider synthesis problems for safety specifications: given a transition system that may raise a BAD flag when entering certain states, we check the existence

of a function that reads the current state and the values of uncontrollable inputs, and provides valuations of the controllable inputs such that the BAD flag is not raised on any possible execution. We consider systems where the state space is defined by a set L of boolean state variables, also called *latches*. We write $\mathbb{B}$ for the set $\{0,1\}$. A state of the system is a valuation $q \in \mathbb{B}^L$ of the latches. We will represent sets of states by their characteristic functions of type $\mathbb{B}^L \to \mathbb{B}$, and similarly for sets of transitions etc.

Definition 1 A **controllable transition system** (or short: controllable system) TS is a 6-tuple $(L, X_u, X_c, \mathcal{R}, BAD, q_0)$, where:

- L is a set of state variables for the latches
- X_u is a set of uncontrollable input variables
- X_c is a set of controllable input variables
- $\mathcal{R} : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \times \mathbb{B}^{L'} \to \mathbb{B}$ is the transition relation, where $L' = \{l' \mid l \in L\}$ stands for the state variables after the transition
- $BAD : \mathbb{B}^L \to \mathbb{B}$ is the set of unsafe states
- q_0 is the initial state where all latches are initialized to 0.

We assume that the transition relation $\mathcal{R}$ of a controllable system is *deterministic* and *total* in its first three arguments, i.e., for every state $q \in \mathbb{B}^L$, uncontrollable input $u \in \mathbb{B}^{X_u}$ and controllable input $c \in \mathbb{B}^{X_c}$ there exists exactly one state $q' \in \mathbb{B}^{L'}$ such that $(q, u, c, q') \in \mathcal{R}$.

In our setting, characteristic functions are usually applied to a fixed vector of variables. Therefore, if $C : \mathbb{B}^L \to \mathbb{B}$ is a characteristic function, we write C as a short-hand for $C(L)$. Characteristic functions of sets of states can also be applied to next-state variables L', in that case we write C' for $C(L')$.

Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables, and $Y \subseteq X \setminus \{x_i\}$ for some x_i. For boolean functions $F : \mathbb{B}^X \to \mathbb{B}$ and $f_{x_i} : \mathbb{B}^Y \to \mathbb{B}$, we denote by $F[x_i \leftarrow f_{x_i}]$ the boolean function that substitutes x_i by f_{x_i} in F.

Definition 2 Given a controllable system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$, the *synthesis problem* consists in finding for every $x \in X_c$ a solution function $f_x : \mathbb{B}^L \times \mathbb{B}^{X_u} \to \mathbb{B}$ such that if we replace $\mathcal{R}$ by $\mathcal{R}[x \leftarrow f_x]_{x \in X_c}$, we obtain a safe system, i.e., no state in BAD is reachable.

If such a solution does not exist, we say the system is *unrealizable*.

A set of solution functions for all $x \in X_c$ is also called a *strategy* (for Player 0). We call the states that are reachable under a given strategy the *care-set* of the strategy. Note that the behavior of the system does not change if the strategy is modified on states outside of the care-set. If BAD is unreachable under a given strategy, we call it a *winning strategy*.

To determine the possible behaviors of a controllable system, two forms of image computation can be used: i) the *image* of a set of states C is the set of states that are reachable from C in one step, and the *preimage* are those states from which C is reachable in one step—in both cases ignoring who controls the input variables; ii) the *uncontrollable preimage* of C is the set of states from which the environment can force the next transition to go into C, regardless of the choice of controllable variables. Formally, we define:

Definition 3 Given a controllable system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$ and a set of states C, we have:

- $image(C) = \{q' \in \mathbb{B}^{L'} \mid \exists (q, u, c) \in \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} : C(q) \wedge \mathcal{R}(q, u, c, q')\}$. We also write this set as $\exists L\ \exists X_u\ \exists X_c\ (C \wedge \mathcal{R})$.
- $preimage(C) = \{q \in \mathbb{B}^L \mid \exists (u, c, q') \in \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \times \mathbb{B}^{L'} : C(q') \wedge \mathcal{R}(q, u, c, q')\}$. We also write this set as $\exists X_u\ \exists X_c\ \exists L'\ (C' \wedge \mathcal{R})$.
- $UPRE(C) = \{q \in \mathbb{B}^L \mid \exists u \in \mathbb{B}^{X_u}\ \forall c \in \mathbb{B}^{X_c}\ \exists q' \in \mathbb{B}^L : C(q') \wedge \mathcal{R}(q, u, c, q')\}$. We also write this set as $\exists X_u\ \forall X_c\ \exists L'\ (C' \wedge \mathcal{R})$.

A direct correspondence of the uncontrollable preimage $UPRE$ for forward computation does not exist: if the environment can force the next transition out of a given set of states, in general the states that we reach are not uniquely determined and depend on the choice of Player 0.

2.1 Efficient symbolic computation

BDDs are a suitable data structure for the efficient representation and manipulation of boolean functions, including all operations needed for the computation of *image*, *preimage*, and $UPRE$. Between these three, *preimage* can be computed most efficiently, while *image* and $UPRE$ are more expensive: there exist a number of optimizations for the computation of *preimage* that cannot be used when computing *image* (see Section 5); and $UPRE$ contains a quantifier alternation, which makes it much more expensive than the other two operations.

3 Existing Approaches

As mentioned before, the safety synthesis problem is usually seen as a game between Player 1, who chooses the uncontrollable inputs, and Player 0, who chooses the controllable inputs. The goal of Player 0 is to choose the inputs in a way that he never visits an unsafe state. The classical approach to solve such a game is to compute the so-called winning regions of the two players, where the winning region of Player 1 is the set of states from which he can force Player 0 into an unsafe state and the winning region for Player 0 is any state that is not winning for Player 1.

Before we introduce our new approach, we recapitulate three existing approaches and point out their benefits and drawbacks.

3.1 Backward fixed-point algorithm

Given a controllable transition system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$ with $BAD \neq 0$, the standard backward BDD-based algorithm (see e.g. [17]) computes the winning region of Player 1, i.e., the set of states from which the

environment can force the system into unsafe states, in a fixed-point computation that starts with the unsafe states. The winning region of Player 1 is the least fixed-point of $UPRE$ on $BAD : \mu C.\ UPRE(C') \cup BAD \cup C$.

Since safety games are determined, the complement of the computed set is the winning region for Player 0, i.e., the set of all states from which the system can win the game. Thus, this set also represents the most permissive winning strategy for Player 0. We note two things regarding this approach:

1. To obtain the winning region, it computes the set of all states that cannot avoid moving into an error state, using the rather expensive $UPRE$ operation.
2. The most permissive winning strategy will not avoid progress towards the error states unless we reach the border of the winning region.

3.2 A forward algorithm [21,5]

A forward algorithm is presented by Cassez et al. [5] for the dual problem of solving reachability games, based on the work of Liu and Smolka [21]. The algorithm starts from the initial state and explores all states that are reachable in a forward manner. Whenever a state is visited, the algorithm checks whether it is losing; if it is, the algorithm revisits all reachable states that have a transition to this state and checks if they can avoid moving to a losing state. Although the algorithm is optimal in that it has linear time complexity in the state space, two issues should be taken into account:

1. The algorithm explicitly enumerates states and transitions, which is impractical even for moderate-size systems.
2. A fully symbolic implementation of the algorithm does not exist, and it would have to rely heavily on the expensive forward *image* computation.

We will discuss the difficulties of implementing a symbolic forward algorithm in more detail in Section 7.1.

3.3 Lazy Synthesis [13]

Lazy Synthesis interleaves a backwards model checking algorithm that identifies possible error paths with the synthesis of candidate solutions. To this end, the error paths are encoded into a set of constraints, and an SMT solver produces a candidate solution that avoids all known errors. If new error paths are discovered, more constraints are added that exclude them. The procedure terminates once a correct candidate is found (see Figure 1). The approach works in a more general setting than ours, for systems with multiple

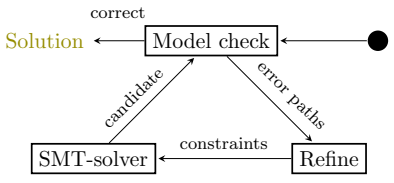

Fig. 1 High-level description of the lazy synthesis algorithm

components and partial information.
When applied to our setting and challenging benchmark problems, the following issues arise:

1. Even though the error paths are encoded as constraints, the representation is such that it explicitly branches over valuations of all input variables, for each step of the error paths. This is clearly impractical for systems that have more than a dozen input variables (which is frequently the case in the classes of problems we target).
2. In each iteration of the main loop a single deterministic candidate is checked. Therefore, many iterations may be needed to discover all error paths.

4 Symbolic Lazy Synthesis Algorithms

In the following, we present symbolic algorithms that are inspired by the lazy synthesis approach and overcome some of its weaknesses to make it suitable for challenging benchmark problems like those from the SYNTCOMP library. We show that in our setting, we can avoid the explicit enumeration of error paths. Furthermore, we can use non-deterministic candidate models that are restricted such that they avoid the known error paths. When choosing these restrictions, we prioritize the removal of transitions that are close to the initial state, which can help us avoid error paths that are not known yet. The high-level control flow of the algorithm is depicted in Figure 2.

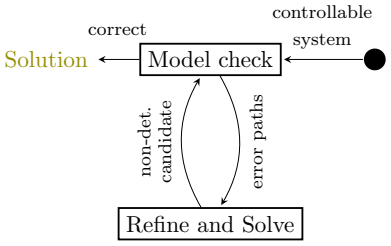

Fig. 2 High-level description of the symbolic lazy synthesis algorithm

4.1 The basic algorithm

To explain the algorithm, we need some additional definitions. Fix a controllable system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$.

An *error level* E_i is a set of states that are on a path from q_0 to BAD, and all states in E_i are reachable from q_0 in i steps. Formally, E_i is a subset of

$$\left\{ q_i \in \mathbb{B}^L \;\middle|\; \begin{array}{l} \exists q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n \in \mathbb{B}^L : \\ q_n \in BAD \text{ and } \exists (q_j, u, c, q_{j+1}) \in \mathcal{R} \text{ for } 0 \leq j < n \end{array} \right\}.$$

We call $(E_0, ..., E_n)$ a *sequence of error levels* if i) each E_i is an error level, ii) each state in each E_i has a transition to a state in E_{i+1}, and iii) $E_n \subseteq BAD$. Note that the same state can appear in multiple error levels of a sequence, and E_0 contains only q_0.

Given a sequence of error levels $(E_0, ..., E_n)$, an *escape* for a transition (q, u, c, q') with $q \in E_i$ and $q' \in E_{i+1}$ is a transition (q, u, c', q'') such that

$\forall m > i : q'' \notin E_m$. We say the transition (q, u, c, q') *matches* the escape (q, u, c', q'').

Given two error levels E_i and E_{i+1}, we denote by RT_i the following set of tuples, representing the "removable" transitions, i.e., all transitions from E_i to E_{i+1} that match an escape:

$$RT_i = \{(q, u, q') \mid q \in E_i, q' \in E_{i+1} \text{ and } \exists (q, u, c, q') \in \mathcal{R} \text{ that has an escape}\}.$$

4.1.1 Overview

Figure 3 sketches the control flow of the algorithm where all operations are performed symbolically on set of states. It starts by model checking the controllable system, without any restriction on the transition relation wrt. the controllable inputs. If unsafe states are reachable, the model checker returns a sequence of error levels. Iterating over all levels, we identify the transitions from the current level for which there exists an escape, and temporarily remove them from the transition relation. Based on the new restrictions on the transition relation, the algorithm then prunes the current error level by removing states that do not have transitions to the next level anymore. Whenever we prune at least one state, we move to the previous level to propagate back this information. If this eventually allows us to prune the first level, i.e., remove the initial state, then this error sequence has been invalidated and the new transition system (with deleted transitions) is sent to the model checker. Otherwise the system is unrealizable. In any following iteration, we accumulate information by merging the new error sequence with the ones we found before, and reset the transition relation before we analyze the error sequence for escapes.

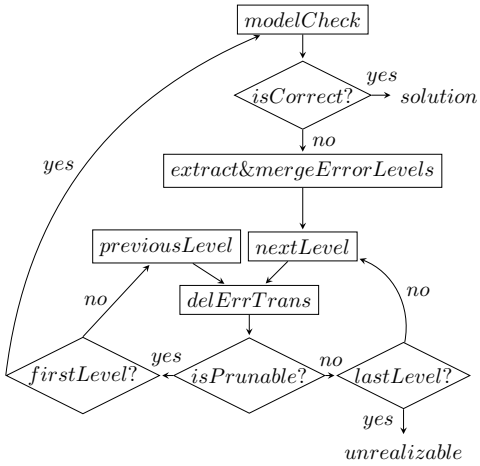

Fig. 3 Control flow of the algorithm

4.1.2 Detailed Description

In more detail, Algorithm 1 describes a symbolic lazy synthesis algorithm. The method takes as input a controllable system and checks if its transition relation can be fixed in a way that error states are avoided. Upon termination, the algorithm returns either *unrealizable*, i.e., the system can not be fixed, or a restricted transition relation that is safe and total. From such a transition relation, a (deterministic) solution for the synthesis problem can be extracted in

Algorithm 1 Lazy Synthesis

```
 1: procedure LazySynthesis(ControllableSystem sys)
 2:     TR ← sys.R,   E ← ()
 3:     while true do
 4:         isCorrect, mcLvls ← ModelCheck(TR)
 5:         if isCorrect then
 6:             return TR
 7:         E ← mergeLevels(E, mcLvls)
 8:         isUnrealizable, TR ← PruneLevels(sys.R, E)
 9:         if isUnrealizable then
10:             return Unrealizable
 1: procedure PruneLevels(TransitionRelation TR, ErrorSequence E)
 2:     i ← 0
 3:     while i < length(E) − 1 do
 4:         isPrunable, TR, E ← ResolveLevel(E, i, TR)
 5:         if isPrunable then
 6:             if i == 0 then   // we have removed the initial state from E[0]
 7:                 return false, TR
 8:             i ← i − 1
 9:         else
10:             i ← i + 1
11:     while i ≥ 1 do   // i == length(E) − 1 when we enter the loop
12:         i ← i − 1
13:         isPrunable, TR, E ← ResolveLevel(E, i, TR)
14:     if isPrunable then   // we have removed the initial state from E[0]
15:         return false, TR
16:     else  // we could not remove the initial state from E[0]
17:         return true, ∅
 1: procedure ResolveLevel(ErrorSequence E, Int i, TransitionRelation TR)
 2:     RT ← (∃L′ (( ∃X_c TR ) ∧ ¬E[i + 1 : n]′ )) ∧ E[i] ∧ E[i + 1]′
 3:     TR ← TR ∧ ¬RT
 4:     AVSet ← ∀X_u (E[i] ∧ ∃L′( ∃X_c TR ∧ ¬E[i + 1 : n]′ ) )
 5:     E[i] ← E[i] ∧ ¬AVSet
 6:     return AVSet ≠ ∅, TR, E
```

the same way as for existing algorithms. Therefore, we restrict the description of our algorithm to the computation of the safe transition relation.

LazySynthesis: In Line 2, we initialize TR to the unrestricted transition relation $\mathcal{R}$ of the input system and E to the empty sequence, before we enter the main loop. Line 4 uses a model checker to check if the current TR is correct, and returns a sequence of error levels $mcLvls$ if it is not. In more detail, procedure ModelCheck(TR) starts from the set of error states and uses the *preimage* function (see Definition 3) to iteratively compute a sequence of error levels.[1] It terminates if a level contains the initial state or if it reaches a fixed point. If the initial state was reached, the model checker uses the *image* function to remove from the error levels any state that is not reachable from

[1] This part is the light-weight backward search: unlike $UPRE$ in the standard backward algorithm, *preimage* does not contain any quantifier alternation.

the initial state.[2] Otherwise, in Line 6 we return the safe transition relation. If TR is not safe yet, Line 7 merges the new error levels with the error levels obtained in previous iterations by letting $E[i] \leftarrow E[i] \vee mcLvls[i]$ for every i. In Line 8 we call PruneLevels($sys.\mathcal{R}, E$), which searches for a transition relation that avoids all error paths represented in E, as explained below. If pruning is not successful, in Lines 9-10 we return "*Unrealizable*".

PruneLevels: In the first loop, ResolveLevel(E, i, TR) is called for increasing values of i (Line 4). Resolving a level is explained in detail below; roughly it means that we remove transitions that match an escape, and then remove states from this level that are not on an error path anymore. If ResolveLevel has removed states from the current level, indicated by the return value of *isPrunable*, we check whether we are at the topmost level — if this is the case, we have removed the initial state from the level, which means that we have shown that every path from the initial state along the error sequence can be avoided. If we are not at the topmost level, we decrement i before returning to the start of the loop, in order to propagate the information about removed states to the previous level(s). If *isPrunable* is false, we instead increment i and continue on the next level of the error sequence.

The first loop terminates either in Line 7, or if we reach the last level. In the latter case, we were not able to remove the initial state from $E[0]$ with the local propagation of information during the main loop (that stops if we reach a level that cannot be pruned). To make sure that all information is completely propagated, afterwards we start another loop were we resolve all levels bottom-up, propagating the information about removed states all the way to the top. If we arrive at $E[0]$ and still cannot remove the initial state, we conclude that the system is unrealizable. This last propagation is needed because, unlike previous propagations, it propagates all information up lo level $E[0]$ even if some error level is not prunable. To see why this is necessary, consider an error sequence obtained after merging error sequences from different iterations, where a state q can be in more than one error level at the same time, say in levels i and j with $i < j$. Now if some error level between i and j is not prunable, then level i will not be resolved again, and escapes for transitions from q will not be used to prune level i, even if they are used to prune level j.

ResolveLevel: Line 2 computes the set of transitions that have an escape: $\exists L'$ (($\exists X_c$ TR) $\wedge \neg E[i+1:n]'$) is the set of all (q, u) for which there exists an escape (q, u, c, q'), and by conjoining this set with $E[i] \wedge E[i+1]'$ we compute all tuples (q, u, q') that represent transitions from $E[i]$ to $E[i+1]$ matching an escape. Line 3 removes the corresponding transitions from the transition relation TR. Line 4 computes *AvSet* which represents the set of all states such that all their transitions within the error levels match an escape. $\forall X_u$ ($E[i] \wedge \exists L'($ $\exists X_c$ $TR \wedge \neg E[i+1:n]'$)) returns the set of states that have an

[2] This is the only place where our algorithm uses *image*, and it is only included to keep the definitions and correctness argument simple - the algorithm also works if the model checker omits this last *image* computation step, see Section 5.

escape for every uncontrollable input. After removing $AVSet$ from the current level, we return.

4.1.3 Illustration of the Algorithm

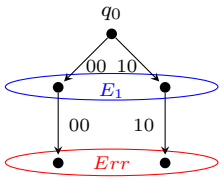

Fig. 4 Error levels from iteration 1

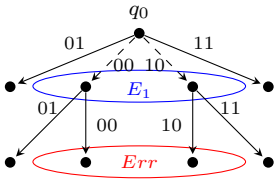

Fig. 5 solution for iteration 1

As an example, Figure 4 shows error levels that may be obtained from the model checker in a first iteration. The transitions are labeled with vectors of input bits, where the left bit is uncontrollable and the right bit controllable. The last level is a subset of BAD. After the first iteration of the algorithm, the transitions that are dashed in Figure 5 will be deleted. Note that another solution exists where instead we delete the two outgoing transitions from level E_1 to the error level Err. This solution can be obtained by a backward algorithm. However, our solution makes all states in E_1 unreachable and thus has a care-set that is much smaller than the winning region.

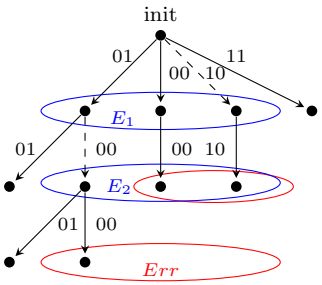

Fig. 6 Error levels from iteration 2

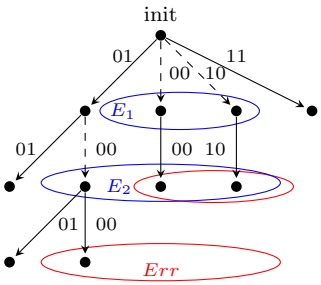

Fig. 7 Solution for iteration 2

Figure 6 depicts merged error levels obtained from iteration 1 and 2 where you can see that the initial state *init* cannot avoid the error level E_1 on uncontrollable input 0. Figure 7 shows that a state can be pruned from level E_1 as it state can avoid level E_2. Pruning E_1 allows *init* to find an escape for uncontrollable input 0 and as a consequence it can avoid E_1 completely.

4.1.4 Comparison

Having defined our symbolic lazy synthesis algorithm formally, let us again compare it to the existing lazy synthesis algorithm, as well as to the standard backwards algorithm.

Lazy Synthesis: The approach depicted in Figure 1 uses model checking to obtain information on paths to the error states, just like our new approach. However, in contrast to our approach the error paths are encoded into SMT constraints, and based on these constraints the SMT solver chooses a deterministic strategy that avoids all known error paths. Thus, the two essential differences are:

1. The SMT encoding explicitly branches over all possible decisions in the error paths, making it impractical to encode long error paths due to the exponential growth of the encoding.
2. The candidate generated by the SMT solver is deterministic, in contrast to the *non-deterministic* strategy generated by the symbolic lazy algorithm, where the strategy is only determinized after being found correct by the model checker.

To evaluate the impact of the second point, we have implemented a version of the algorithm where the strategy is determinized before being sent to the model checker. As expected, it can only solve a few small instances from the challenging SYNTCOMP benchmark set, and the approach with non-deterministic strategies performs much better.

Standard Backwards Algorithm: Compared to the standard backward fixed-point approach (see Section 3.1), an important difference is that we explore the error paths in a forward analysis starting from the initial state, and avoid progress towards the error states as soon as possible. As a consequence, our algorithm can find strategies with a care-set that is much smaller than the winning region, and may solve the problem faster than the standard approach. We give a detailed comparison of the performance of our algorithm against the standard algorithm in Section 6.

4.2 Correctness of Algorithm 1

Theorem 1 (Soundness) *Every transition relation returned by Algorithm 1 is safe, and total in the first two arguments.*

Proof The model checker guarantees that the transition relation is safe, i.e., unsafe states are not reachable. To see that the returned transition relation is total in the first two arguments, i.e., $\forall q \in \mathbb{B}^L \ \forall u \in \mathbb{B}_u^X \ \exists c \in \mathbb{B}_c^X \ \exists q' \in \mathbb{B}^{L'} : (q, u, c, q') \in TR$, observe that this property holds for the initial TR, and is preserved by *ResolveLevels*: Lines 2 and 3 of the procedure ensure that a

transition $(q, u, c, q') \in TR$ can only be deleted if $\exists c' \in \mathbb{B}_c^X \ \exists q'' \neq q' \in \mathbb{B}^{L'} : (q, u, c', q'') \in TR$, i.e., if there exists another transition with the same state q and uncontrollable input u.

To prove completeness of the algorithm, we define formally what it means for an error level to be resolved.

Definition 4 (Resolved) Given a sequence of error levels $E = (E_0, ..., E_n)$ and a transition relation TR, an error level E_i with $i < n$ is *resolved* with respect to TR if the following conditions hold:

- $RT_i = \emptyset$
- $\forall q_i \in E_i \setminus BAD : \ \exists u \in \mathbb{B}^{X_u} \ \exists c \in \mathbb{B}^{X_c} \ \exists q_{i+1} \in E_{i+1} : \ (q_i, u, c, q_{i+1}) \in TR$

E_i is unresolved otherwise, and E_n is always resolved.

Informally, E_i is resolved if every state in E_i, on some uncontrollable input u, cannot avoid reaching lower levels (i.e. each controllable input of u leads to some E_j where $i < j \leq n$). We can conclude the following lemma.

Lemma 1 *A controllable system is unrealizable iff there exists an error sequence $E_0, E_1, ..., E_n$ where $E_0 = \{q_0\}$, and for all $i \leq n$, E_i is resolved and non-empty.*

Proof Suppose the system is unrealizable, i.e., Player 1 has a strategy to always reach BAD. Then for some $n \in \mathbb{N}$ there exists a sequence of (non-empty) sets of states $E_0, E_1, \ldots, E_n$ such that $E_0 = \{q_0\}$, $E_n \subseteq BAD$, and for every E_i and every $q \in E_i$, Player 1 can force the game into E_{i+1} in one step, i.e., $\forall q \in E_i \ \forall c \in \mathbb{B}^{X_c} \ \exists u \in \mathbb{B}^{X_u} : (q, u, c, q') \in TR$ with $q' \in E_{i+1}$. In particular, $E_0, E_1, \ldots, E_n$ is an error sequence. To see that it is resolved, assume that it was not: then from some E_i, RT_i would have to be non-empty, i.e., for some $q \in E_i$and $u \in \mathbb{B}^{X_u}$ there would have to be a transition $(q, u, c, q') \in TR$ with $q' \notin E_{i+1}$, contradicting the properties of our error sequence.

In the other direction, suppose there exists an error sequence $E_0, E_1, ..., E_n)$ with $E_0 = \{q_0\}$ and $\forall i \leq n$, E_i is resolved and non-empty. Then we can construct a strategy for Player 1 to win the game: in each E_i, there must exist a state q and inputs u, c such that there is $(q, u, c, q') \in TR$ with $q' \in E_{i+1}$, for which there is no escape. A winning strategy for Player 1 is to always choose such an uncontrollable input u.

Theorem 2 (Completeness) *If Algorithm 1 returns "Unrealizable", then the controllable system is unrealizable.*

Proof Observe that the algorithm returns unrealizable only when there exists an error sequence $E_0, E_1, ..., E_n$ where $E_0 = \{q_0\}$ and all levels are resolved and non-empty. Lines 2 and 3 of ResolveLevel guarantee that all transitions from E_i to E_{i+1} that match an escape will be deleted, so the only remaining transitions between E_i and E_{i+1} are those that have no escapes. Line 4 computes all states in E_i that no longer have transitions to lower levels

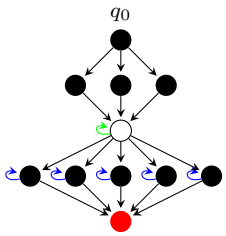

Fig. 8 Example with small solution

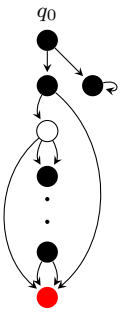

Fig. 9 Example that is solved fast

(levels with greater index) and Line 5 removes these states. Thus, after calling ResolveLevel, the current level will be resolved.

However, since ResolveLevel may remove states from E_i, the levels E_j with $j < i$ could become unresolved. To see that this is not an issue note that before we output *Unrealizable*, we go through the second loop that resolves all levels from n to 0. After execution of this second loop all levels are resolved, and if E_0 still contains q_0, then from our sequence of error levels we can extract a subsequence[3] of resolved and non-empty error levels, which by Lemma 1 implies unrealizability.

Theorem 3 (Termination) *Algorithm 1 always terminates.*

Proof Every call of the procedure PruneLevels returns a transition relation that is guaranteed to avoid all error paths returned by the model checker in all previous iterations (see Line 7 of procedure LazySynthesis). This is accomplished by making at least one state on every path from the initial state to an error state unreachable (see Lines 6-7,14-15 of PruneLevels). In particular, any transition relation returned by PruneLevels is different from all previous transition relations. Since for a fixed controllable system there is only a finite number of possible transition relations, the procedure will eventually terminate.

4.3 Example Problems

We want to highlight the potential benefit of our algorithm on two families of examples.

First, consider a controllable system where all paths from the initial state to the error states have to go through a bottleneck, e.g., a single state, as depicted in Figure 8, and assume that Player 0 can force the system not to go beyond this bottleneck. In this case, the care-set of our solution only includes the states between the initial state and the bottleneck, whereas the winning

[3] It may be a subsequence due to the merging of error levels from different iterations of the main loop.

region detected by the standard algorithm may be much bigger (in the example including all the states in the fourth row). Moreover, the strategy produced by our algorithm will be very simple: if we reach the bottleneck, we force the system to stay there. In contrast, the strategy produced by the standard algorithm will in general be much more complicated, as it has to define the behavior for a much larger number of states.

Second, consider a controllable system where the shortest path between error and initial state is short, but Player 1 can only *force* the system to move towards the error on a long path. Moreover, assume that Player 0 can avoid entering this long path, for example by entering a separate part of the state space like depicted in Figure 9. In this case, our algorithm will quickly find a simple solution: move to that separate part and stay there. In contrast, the standard algorithm will have to go through many iterations of the backwards fixed-point computation, until finally finding the point where moving into the losing region can be avoided.

5 Optimization

As presented, Algorithm 1 requires the construction of a data structure that represents the full transition relation $\mathcal{R}$, which causes a significant memory consumption. In practice, the size of a BDD that represents the full transition relation can be prohibitive even for moderate-size models.

Since the transition relation is deterministic, it can alternatively be represented by a vector of functions, each of which updates one of the state variables. Such a partitioning of the transition relation is an additional computational effort, but it results in a more efficient representation that is necessary to handle large systems. In the following we describe optimizations based on such a representation.

Definition 5 A **functional controllable system** is a 6-tuple $TS_f = (L, X_u, X_c, \mathbf{F}, BAD, q_0)$, where

- L is a set of state variables for the latches
- X_u is a set of uncontrollable input variables
- X_c is a set of controllable input variables
- $\mathbf{F} = (f_1, ..., f_{|L|})$ is a vector of update functions $f_i : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \to \mathbb{B}$ for $i \in \{1, \ldots, |L|\}$
- $BAD : \mathbb{B}^L \to \mathbb{B}$ is the set of unsafe states
- q_0 is the initial state where all latches are initialized to 0.

In a functional system with current state q and inputs u and c, the next-state value of the ith state variable l_i is computed as $f_i(q, u, c)$. Thus, we can compute image and preimage of a set of states C in the following way:

- $image_f(C) = \exists L\ \exists X_u\ \exists X_c\ (\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C)$
- $preimage_f(C) = \exists L'\ \exists X_u\ \exists X_c\ (\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C')$

However, computing $\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C'$ is still very expensive and might be as hard as computing the whole transition relation. To optimize the preimage computation, we instead directly substitute the state variables in the boolean function that represents C by the function that computes their new value:

$$preimage_s(C) = \exists X_u\ \exists X_c\ C[l_i \leftarrow f_i]_{l_i \in L}$$

Since substitution cannot be used to compute $image(C)$, forward exploration of the state space is in practice much more expensive than backwards exploration. This even holds for alternative, more efficient ways to compute *image*, such as using the *range* function [19]. We consider a forward algorithm based on this alternative in Section 7.1.

5.1 The Optimized Algorithm

The optimized algorithm takes as input a functional controllable system, and uses the following modified procedures:

OptimizedLazySynthesis: This procedure replaces LazySynthesis, to which it is different in two aspects concerning the model checker:

1. the preimage is computed using $preimage_s$, and
2. unreachable states are not removed, in order to avoid image computation. Thus, the error levels are over-approximated.

OptimizedResolveLevel: This procedure replaces ResolveLevel and computes *RT* and *AvSet* more efficiently. Note that for a given set of states C, the set $pretrans(C) = \{(q,u,c) \in \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \mid \mathbf{F}(q,u,c) \in C\}$ can efficiently be computed as $C[l_i \leftarrow f_i]_{l_i \in L}$. Based on this, we get the following:

- *RT*: we compute the transitions that can be avoided as the conjunction of the transitions from E_i to E_{i+1}, given as $pretrans(E[i+1]') \wedge E[i]$, with those transitions that have an escape, $\exists X_c\ pretrans(\neg E[i+1:n]') \wedge E[i]$.

- *AvSet*: The states that can avoid all transitions to the lower levels can now be computed as $\forall X_u\ [\ \exists X_c\ pretrans(\neg E[i+1:n]') \wedge E[i]\ ]$.

5.1.1 Generalized Deletion of Transitions

In addition, we consider a variant of our algorithm that uses the following heuristic to speed up computation: whenever we find an escape (q,u,c,q') with $q \in E_i$, then we not only remove all matching transitions that start in E_i, but matching transitions that start anywhere, and lead to a state $q'' \in E_j$ with $j > i$. Thus, we delete more transitions per iteration of the algorithm, all of which are known to lead to an error.

6 Experimental Evaluation

We implemented our algorithm in Python, using the BDD package CUDD [27]. We evaluate our prototype on a family of parameterized benchmarks based on the examples in Section 4.3, on the benchmark set of SYNTCOMP 2017 [16], and on a set of random benchmarks.

We evaluate two versions of our algorithm: a version without generalized deletion (see Section 5.1.1), in the following called *Lazy*, and a version with generalized deletion, in the following called $Lazy_{GD}$. We compare them against our own implementation of the standard backward approach, in order to have a fair comparison between algorithms that use the same BDD library and programming language. For the SYNTCOMP benchmarks, we additionally compare against the results of the participants in SYNTCOMP 2017. Our implementations of all algorithms include the most important general optimizations for this kind of algorithms, including a functional transition relation and automatic reordering of BDDs (see Jacobs et al. [17]).

6.1 Parameterized Benchmarks

On the parameterized versions of the examples from Section 4.3, we observe the expected behaviour:

- for the first example, the care-set of the strategy found by our algorithm is always only about half as big as the winning region found by the standard algorithm. Even more notable is the size of the synthesized controller circuit: for example, our solution for an instance with 2^{18} states and 10 input variables has a size of just 9 AND-gates, whereas the solution obtained from the standard algorithm has 800 AND-gates.
- for the second example, we observe that for systems with 15 to 25 state variables, our algorithm solves the problem in constant time of 0.1s, whereas the solving time increases sharply for the standard algorithm: it uses 1.7s for a system with 15 latches, 92s for 20 latches, and 4194s for 25 latches.

6.2 SYNTCOMP Benchmarks

We compared our algorithms against the standard algorithm on the benchmark set that was used in the safety track of SYNTCOMP 2017, with a timeout of 5000s on an Intel Xeon processor (E3-1271 v3, 3.6 GHz) and 32 GB RAM.

First, we observe that our algorithms often detect care-sets that are much smaller than the full winning region: out of the 76 realizable benchmarks that the *Lazy* algorithm solved, we found a strictly smaller care-set in 28 cases. In 14 cases, the care-set is smaller by a factor of 10^3 or more, in 8 cases by a factor of 10^{20} or more, and in 4 cases by a factor of 10^{30} or more. The biggest difference in size is by a factor of 10^{68}. For the $Lazy_{GD}$ algorithm, the care-sets are somewhat bigger, but the tendency is the same. Table 1 gives detailed information for a selection of such examples.

Table 1 Comparison of care-set and winning region size for selected benchmarks (number of states)

Instance	Standard	Lazy	Lazy_{GD}	Difference factor
load_2c_comp_comp5	$1.08 * 10^{40}$	$\mathbf{5.67 * 10^{13}}$	$3.79 * 10^{22}$	$> 10^{26}$
load_3c_comp_comp4	$2.39 * 10^{52}$	$\mathbf{1.21 * 10^{18}}$	$8.5 * 10^{37}$	$> 10^{44}$
load_3c_comp_comp7	$4.97 * 10^{86}$	$\mathbf{1.21 * 10^{18}}$	$6.28 * 10^{57}$	$> 10^{68}$
load_4c_comp_comp4	$4.03 * 10^{63}$	*TO*	$\mathbf{4.79 * 10^{52}}$	$> 10^{10}$
load_4c_comp_comp6	$9.03 * 10^{92}$	*TO*	$\mathbf{2.2 * 10^{71}}$	$> 10^{21}$
load_full_2_comp5	$2.52 * 10^{80}$	*TO*	$\mathbf{4.21 * 10^{65}}$	$> 10^{15}$
load_full_2_comp7	$4.99 * 10^{108}$	*TO*	$\mathbf{3.11 * 10^{85}}$	$> 10^{23}$
ltl2dba_C2-6_comp3	$2.46 * 10^{35}$	$\mathbf{4.55 * 10^{25}}$	$\mathbf{4.55 * 10^{25}}$	$> 10^{9}$
ltl2dba_E4_comp3	$2.96 * 10^{79}$	$\mathbf{3.74 * 10^{50}}$	$1.05 * 10^{65}$	$> 10^{28}$
demo-v10_5	$1.93 * 10^{25}$	$\mathbf{1.31} * 10^{5}$	$2.25 * 10^{15}$	$> 10^{20}$
demo-v12_5	$2.81 * 10^{14}$	$\mathbf{1.64} * 10^{4}$	$6.98 * 10^{10}$	$> 10^{10}$
demo-v14_5	$1.23 * 10^{14}$	**356**	$2.69 * 10^{8}$	$> 10^{11}$
demo-v16_5	$9.03 * 10^{7}$	$\mathbf{1.36 * 10^{4}}$	$3.09 * 10^{5}$	$> 10^{3}$
demo-v18_5	$3.67 * 10^{27}$	*TO*	$\mathbf{6.97 * 10^{16}}$	$> 10^{10}$
demo-v19_5	$1.27 * 10^{11}$	**305**	$2.68 * 10^{8}$	$> 10^{8}$
demo-v20_5	$2.31 * 10^{41}$	$\mathbf{3.44 * 10^{10}}$	$1.22 * 10^{24}$	$> 10^{30}$
demo-v22_5	$3.4 * 10^{38}$	$\mathbf{1.71 * 10^{15}}$	$4.76 * 10^{21}$	$> 10^{23}$
demo-v23_5	$1.37 * 10^{12}$	$\mathbf{9.22 * 10^{3}}$	$1.09 * 10^{9}$	$> 10^{8}$
demo-v24_5	$3.27 * 10^{63}$	$\mathbf{1.17 * 10^{31}}$	$4.23 * 10^{28}$	$> 10^{32}$

However, note that our smaller sets do not necessarily correspond to smaller symbolic representations of these sets. Table 2 compares the sizes of BDDs instead of explicit number of states, showing that in some cases the BDD is smaller, but more often the symbolic representation for the smaller set of states is actually more complex. The results are also mixed when regarding the size of the synthesized circuits: in 11 cases the *Lazy* algorithm produces a smaller solution than the standard algorithm, in 21 cases it is the other way around. The $Lazy_{GD}$ algorithm produced smaller circuits in 15 cases. Table 3 contains a sample of these results, including also the size of the symbolic representation of the winning strategy. It is also important to note that the *Lazy* algorithm, for 10 out of the 11 benchmarks with smaller synthesized circuits, has produced smaller care-sets. Furthermore $Lazy_{GD}$ has produced smaller care-sets for 11 out of the 15 benchmarks with smaller synthesized circuits.

Out of the 234 benchmarks, the *Lazy* algorithm solved 99 before the timeout, and the $Lazy_{GD}$ algorithm solved 116. While the standard algorithm solves a higher number of instances overall (163), for a number of examples our algorithms are faster. In particular, both versions each solve 7 benchmarks that are not solved by the standard algorithm, as shown in Table 4. Moreover, we compare against the participants of SYNTCOMP 2017: with a timeout of 3600s, the best single-threaded solver in SYNTCOMP 2017 solved 155 problems, and the virtual best solver (VBS; i.e., a theoretical solver that on each benchmark performs as good as the best participating solver) would have solved 186 instances. If we include our two algorithms with a timeout of 3600s, the VBS can additionally solve 7 out of the 48 instances that could not be solved by any of the participants of SYNTCOMP before. As our algorithms

Table 2 Comparison of care-set and winning region size for selected benchmarks (number of BDD nodes in symbolic representation)

Instance	Standard	Lazy	Lazy_{GD}
load_2c_comp_comp5	**299**	986	518
load_3c_comp_comp4	**345**	9299	669
load_3c_comp_comp7	**442**	11253	1507
load_4c_comp_comp4	**2075**	*TO*	3263
load_4c_comp_comp6	**4413**	*TO*	7814
load_full_2_comp5	**1308**	*TO*	2182
load_full_2_comp7	**2068**	*TO*	5071
ltl2dba_C2-6_comp3	**199**	484	501
ltl2dba_E4_comp3	7361	**454**	508
demo-v10_5	**16**	83	49
demo-v12_5	**10**	44	34
demo-v14_5	**53**	83	87
demo-v16_5	262	233	**132**
demo-v18_5	125535	*TO*	**52687**
demo-v19_5	156	83	**76**
demo-v20_5	**87**	135	600
demo-v22_5	**226**	1373	1768
demo-v23_5	**46**	139	92
demo-v24_5	**195**	4075	561

Table 3 Comparison of the size of solutions for selected benchmarks (number of AND-gates in synthesized circuit / number of BDD nodes in winning strategy)

Instance	Standard	Lazy	Lazy_{GD}
amba10c6n	28137 / 18612	28621 / 18711	**25816** / **17466**
driver_d10y	226581 / 140789	**156776** / **105244**	*TO*
factory_assembly_7x5_2_0errors	31469 / 22841	**19853** / 18541	21453 / **17713**
genbuf12c30n	3914 / 4808	**2153** / 3278	2278 / **3172**
genbuf24c30y	23974 / 15528	18495 / 12365	**9789** / **6529**
genbuf56c40n	135025 / 59629	*TO*	**65284** / **32154**
genbuf8c30n	5767 / 5536	5753 / 5463	**5189** / **4890**
genbuf16c4y	**7137** / 10605	49373 / 55322	7375 / **10469**
demo-v18_5_REAL	**50620** / **88189**	*TO*	140172 / 105279
driver_d8n	**171348** / **108099**	*TO*	180917 / 116347
factory_assembly_5x5_2_0errors	**11187** / **8578**	21078 / 13728	22586 / 16680
factory_assembly_5x5_2_1errors	**21300** / **17118**	46963 / 33821	52730 / 33430

also solve some instances much faster than the existing algorithms, they would be worthwhile additions to a portfolio solver for SYNTCOMP.

6.3 Random Benchmarks

The SYNTCOMP benchmark library consists of crafted benchmarks that were submitted by the participants. An inspection of these benchmarks shows that in many cases these benchmarks are such that progress towards the error states can *only* be avoided when we reach the border of the winning region. Obviously, benchmarks with such a structure benefit the standard backward approach and do not allow the lazy synthesis approach to show its strengths.

To obtain additional benchmarks that avoid the potential bias of hand-crafted examples, we developed a scheme for generating random benchmarks. Our prototype implementation takes as input the number of controllable vari-

Table 4 Comparison of solving time for benchmarks solved by a lazy algorithm, but not by the standard algorithm (seconds)

Instance	Standard	Lazy	Lazy_{GD}	SYNTCOMP 2017 Participants
gb_s2_r3_comp1	*TO*	38	*TO*	solved by 1
genbuf48c6y	*TO*	*TO*	3839	solved by 4
ltl2dba_E6_comp4	*TO*	2435	*TO*	not solved
ltl2dba_Q4_comp5	*TO*	125	304	solved by 1
ltl2dba_U1-6_Comp3	*TO*	*TO*	4590	not solved
ltl2dpa_alpha5_Comp2	*TO*	*TO*	1880	not solved
ltl2dpa_alpha5_Comp3	*TO*	*TO*	2651	not solved
ltl2dpa_E4_comp2	*TO*	1081	*TO*	not solved
ltl2dpa_E4_comp4	*TO*	2122	*TO*	not solved
ltl2dpa_U14_comp2	*TO*	4019	615	not solved
ltl2dpa_U14_comp35	*TO*	2605	1681	not solved

ables c, the number of uncontrollable variables u and the number of latches l, and generates an AIGER benchmark based on a uniformly random distribution over all controllable systems with these parameters. The implementation uses ROBDD as an intermediate representation of the benchmark to be generated and therefore the uniformity is guaranteed by the canonicity of ROBDDs. For a fixed u, c, and l there are $2^{2^{u+c+l}}$ different boolean functions, and we need one such function to update each latch, and in addition we need a boolean function only over l that determines the BAD output. Thus, overall we have a space of $(2^{2^{u+c+l}})^l \cdot (2^{2^l})$ possible benchmarks.

Optionally, we can restrict the number o of latches that are used to define BAD. Using fewer latches for BAD decreases the expected size of the error states and increases the chance of obtaining a realizable benchmark.

We have generated thousands of random benchmarks from different classes, where a class is defined by the number of controllable variables, uncontrollable variables, latches, and output function variables. A primary observation is that whenever a benchmark is easy to solve because there are many winning strategies (e.g., if parameters o and u are much smaller than l and c, respectively), then the standard algorithm is usually able to find a solution faster. However, when it is hard to find a winning strategy, then the results change. For instance, we compared the lazy algorithm with and without general deletion against the standard algorithm on 100 random benchmarks with $c = 3, u = 1, l = 13$ (i.e., 17 variables overall), and $o = 12$. For 66 benchmarks, both of our algorithms synthesized circuits that were smaller or equal to the solutions of the standard algorithm (out of these 66, 30 where strictly smaller). Moreover, the $Lazy$ algorithm solved 26 faster than the standard algorithm, and the $Lazy_{GD}$ algorithm was faster on 33 benchmarks.

Figures 10, 11, 12 compare solving time between the $Lazy$ and the standard algorithm, for benchmarks with 17,18, and 19 variables respectively. For the benchmarks with 19 variables the $Lazy$ algorithm solved 6 instances out of 100 that the standard algorithm could not solve, visible on the line on the right-hand side, marked with TO. The remaining benchmarks that we generated

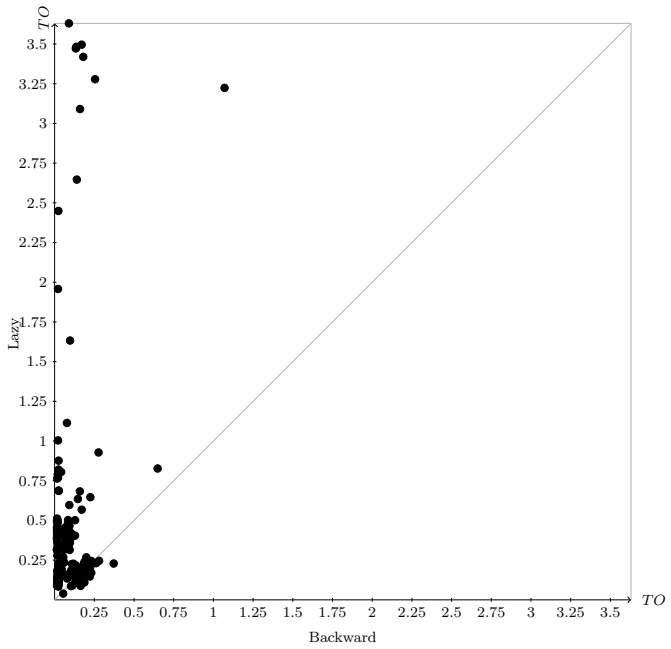

Fig. 10 Time comparison between backward and lazy approaches for benchmarks with 17 variables. 100 random benchmarks with $c = 3, u = 1, l = 13, o = 12$, and 100 random benchmarks with $c = 3, u = 1, l = 13, o = 11$

had 16 or fewer variables, and every single benchmark could be solved by all three algorithms, usually in a few seconds.

For the benchmarks we generated, we chose the parameters of our random generator in order to obtain interesting benchmarks, i.e.,

1. not too easy to solve (benchmarks with ≤ 16 variables can almost always be solved very quickly)
2. not too hard to solve (for 19 variables, both tools already run into a timeout on many examples), and
3. preferably realizable (by having more controllable than uncontrollable inputs).

We prefer realizable benchmarks since we also want to compare properties of the solutions, such as care-set/winning region or the size of the synthesized circuit.

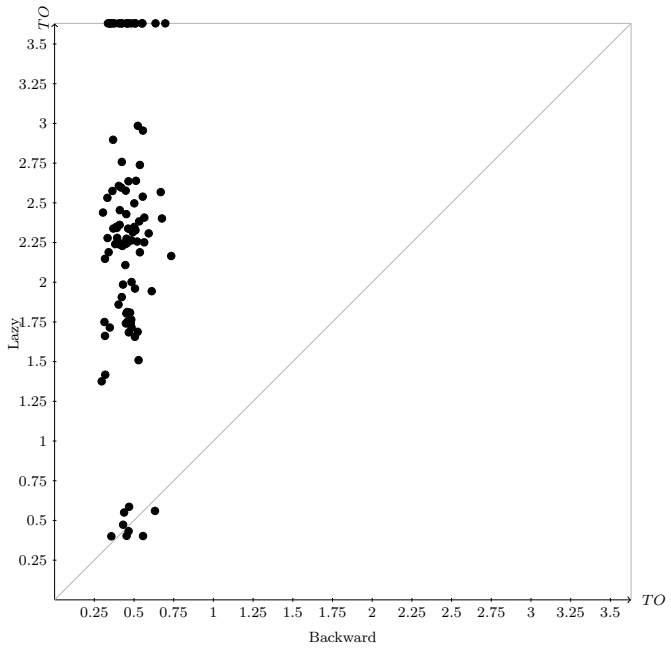

Fig. 11 Time comparison between backward and lazy approachses for benchmarks with 18 latches. 100 random benchmarks with $c = 3, u = 1, l = 14, o = 12$

7 Why Not a Purely Forward Exploration?

7.1 A Forward Algorithm

In Section 3.2 we mentioned a completely forward algorithm presented by Cassez et al. [5]. The algorithm starts from the initial state and explores all states that are reachable in a forward manner and checks if they can avoid moving to a losing state. The algorithm is not symbolic and it explicitly enumerates states and transitions. In this section, we propose a symbolic implementation and report on our experiences with integrating this form of forward exploration into our algorithms.

For a symbolic implementation, given a set of states, we need to compute all states that are reachable from this set in one transition. This can be accomplished by computing the image as defined in Section 5. However, image computation is very expensive in terms of memory and may be as hard as computing the whole transition relation. We explain below two methods that aim to reduce the overhead of image computation.

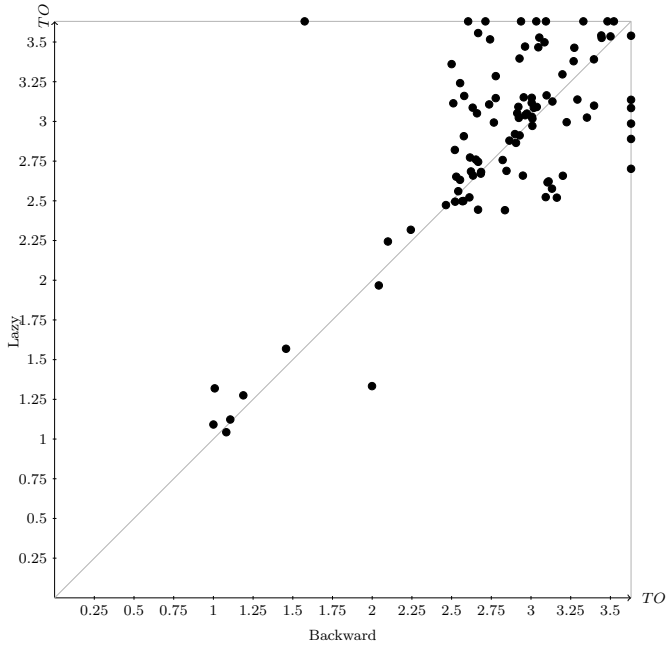

Fig. 12 Time comparison between backward and lazy approachses for benchmarks with 19 latches. 100 random benchmarks with $c = 4, u = 2, l = 13, o = 12$

7.1.1 Early quantification [7]

When computing a BDD respresentation of an expression that contains existential quantification, it is desirable to evaluate terms under existential quantifiers as early as possible: existentially quantified variables can be completely eliminated from the ROBDD, which often results in a considerable reduction in the size of the BDD. Unfortunately, existential quantification is not distributive over conjunction and therefore we often have to first compute the result of the conjunction before we can remove the quantified variables.

However, if a term contains variables that are used only in this term, then existential quantification of these variables can always be performed locally. For synthesis algorithms, this would be useful for the update functions f_i of latches l_i. To take advantage of this property, one can use heuristics to search for a convenient ordering of update functions, represented as a permutation $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ such that as many existentially quantified subexpressions can be evaluated as early as possible. Let $E_i \leftarrow \mathit{support}(f_{\pi(i)}) \setminus \bigcup_{k=1}^{i-1} \mathit{support}(f_{\pi(k)})$ then the image can be computed as follows:

1. $S_1 \leftarrow C$
2. $S_{i+1} \leftarrow \exists x \in E_i \, (S_i \wedge (l'_{\pi(i)} \equiv f_{\pi(i)}))$
3. $\mathit{image}_f(C) = S_{n+1}$

We did not implement this optimization since an inspection of the benchmark set from SYNTCOMP has shown that the support of most of the update functions and the ROBDDs that represent them contain all the variables, which makes this optimization unlikely to be very useful. It remains open if this could be a worthwhile addition to synthesis tools, at least on certain kinds of benchmarks.

7.1.2 The range function

Definition 6 [19] Given a function $f : \mathcal{A} \to \mathcal{B}$ and $C \subseteq \mathcal{A}$, define:

- $range(f) = \{y \in \mathcal{B} \mid \exists x \in \mathcal{A} \text{ with } y = f(x)\}$.
- $image(f, C) = \{y \in \mathcal{B} \mid \exists x \in C \text{ with } y = f(x)\}$.

Given $f : \mathcal{A} \to \mathcal{B}$ one can easily see that *range* can be used to compute the image of f on $\mathcal{A}$, since $range(f) = image(f, \mathcal{A})$. However what we need for forward exploration of the state space is a way to compute $image(f, C)$ for arbitrary $C \subseteq \mathcal{A}$.

Another way to view this is that instead of $range(f)$, where f covers the whole set $\mathcal{A}$, we are interested in $range(f')$ for some restriction f' of f that is only defined on C. To this end, Coudert et al. [8] introduced the *constraint operator*, a variant of the generalized cofactor [28].

Definition 7 Let $f_i : \mathbb{B}^m \to \mathbb{B}$, $\mathbf{F} : \mathbb{B}^m \to \mathbb{B}^n$ with $\mathbf{F} = (f_1, \dots f_n)$ and $C : \mathbb{B}^m \to \mathbb{B}$. The *generalized cofactor* $\mathbf{F}|_C = (f_1|_C, \dots f_n|_C)$ is defined as

$$\mathbf{F}|_C(x) = \begin{cases} \mathbf{X} & \text{if } C(x) = False \\ \mathbf{F}(x) & \text{if } C(x) = True \end{cases}$$

In the above definition $\mathbf{X}$ denotes a vector of "don't care" values, and these can be chosen in a way that makes the BDD that represents $\mathbf{F}|_C$ smaller than the BDD of $\mathbf{F}$.

Now, we can use the following theorem to implement the desired image computation:

Theorem 4 *[8, 28]* $range(\mathbf{F}|_C) = image(\mathbf{F}, C)$

Although this method alleviates the memory requirements of $\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C'$, the algorithm to compute $range(\mathbf{F}|_C)$ is a recursive algorithm [19] that requires a large number of recursive calls.

We have tried to integrate this form of forward search into our algorithms in order to detect unreachable states and prune the corresponding error paths. In our experiments, we experienced unacceptably large computation times, even on rather small examples, and with optimizations such as storing intermediate computation results. Therefore, we do not expect an algorithm based on *image* or *range* computation to be competitive on the class of benchmarks that we considered.

8 Synthesis of Resilient Controllers

As mentioned in Section 1, our algorithm produces strategies that avoid progress towards the error states as early as possible, which can be useful for generating controllers that allow for a margin of error, e.g. in the presence of sporadic faults or perturbations. In this section we review an algorithm that generates strategies with resilience to perturbations, compare it to the lazy synthesis algorithm and observe commonalities of their behavior on certain benchmarks.

8.1 Controllable Systems with Perturbations

Dallal et al. [9] have modeled systems with *perturbations*, which are defined as extraordinary transitions where values for the controllable inputs, or a subset thereof, are chosen non-deterministically. Thus, in a perturbation step Player 0 has only limited control over the behavior of the system, or none at all.

Formally, we modify controllable transition systems by fixing a subset $X_P \subseteq X_C$ and considering a transition relation of the form $\mathcal{R} : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_P} \times \mathbb{B}^{X_c \setminus X_P} \times \mathbb{B}^{L'} \to \mathbb{B}$. Given a set of solution functions for X_c and a bound k on the number of perturbations, the semantics of the composed system is the same as before, except that in a given run of the system, up to k times the values for variables in X_P are not chosen according to the solution function, but can be arbitrary.

Then, we are interested in an upper bound on the number of perturbations such that the synthesis problem can still be solved, and in strategies for the system with this number of perturbations, called *maximally resilient strategies*.

8.2 An Algorithm for Synthesis of Resilient Controllers

Dallal et al. [9] introduced an algorithm that produces maximally resilient strategies. It can be summarized as follows:

1. use the standard fixed-point algorithm to compute the winning region without perturbations,
2. use a mixed forward/backward analysis to find a strategy that makes as little progress towards the losing region as possible.

The second part can be seen as a variant of our lazy synthesis algorithm, except that it only has to handle a restricted setting: instead of the error states, the winning region can be used as a basis for the backwards analysis, and the forward analysis is simplified by the fact that from all states inside the winning region there is a winning strategy, so no backtracking is necessary to remove states from which winning is impossible.

8.3 Implementation, Experiments and Comparison to Lazy Synthesis

We have implemented this algorithm as a combination of the backward fixed-point algorithm and symbolic lazy synthesis, providing to our knowledge its first implementation. An evaluation on the SYNTCOMP 2017 benchmarks provides interesting insights: only on 6 out of the 234 benchmarks the algorithm can give a guarantee of resilience against one or more perturbations, as shown in Table 5.

Table 5 Benchmarks with Resilience Guarantees.
For Lazy Synthesis, we give the distance to error states, regardless of controllability after a perturbation. For Dallal et al., we give the distance to the losing region, taking controllability into account, i.e., the number of perturbations that the controller is resilient against.

Instance	**Lazy Synthesis**	**Dallal et al. [9]**
beembrdg2f1_c0to1	32	32
demo-v10_5	6	6
demo-v12_5	7	7
demo-v20_5	6	5
ltl2dba_C2-6_comp3	0	3
ltl2dba_E4_comp3	4	4

When inspecting the behavior of our lazy algorithms on these benchmarks, we find that for 5 out of 6 benchmarks, our algorithms can give an additional *quantitative safety guarantee* by measuring the distance between the error states and any states that can be visited under the given strategy. Note that this information can be extracted without additional cost, simply by inspection of the final sequence of error levels. However, also note that this distance is not the same as the resilience property of Dallal et al., since (i) we compute the distance to the unsafe states, not to the losing region, and (ii) we do not take into account whether after a single perturbation there is still a winning strategy for Player 0. Thus, a distance of k to the unsafe states does not imply that the strategy is resilient to k perturbations—in fact, such a strategy does not always exist, as the results for benchmark demo-v20_5 in Table 5 show.

Furthermore, we observe that on all of these examples our algorithms detect a care-set that is much smaller than the full winning region. The results in Table 1 include 5 of the 6 benchmarks, and show that the care-sets provided by lazy synthesis are smaller by a factor of 10^9 or more. This leads us to the conjecture that lazy synthesis performs particularly well on synthesis problems that allow resilient controllers, together with the observation that not many of these appear in the SYNTCOMP 2017 benchmark set that we have tested against.

9 Conclusion

We have introduced lazy synthesis algorithms with a novel combination of forward and backward exploration. Our experimental results show that in many

cases our algorithms detect solutions with care-sets that are much smaller than the full winning region. Moreover, they can solve a number of problems that are intractable for existing synthesis algorithms, both in crafted and random benchmarks. Finally, our algorithm produces eager solutions, and in some cases we can give quantitative safety guarantees, i.e., we can determine the minimum distance to any error state that a system running on our generated strategy will keep during execution.

In the future, we want to further investigate which classes of benchmarks are difficult for the standard backwards algorithm, and for which classes of benchmarks the lazy algorithms are preferable. Based on this, we further want to explore how lazy synthesis can be integrated into portfolio solvers and hybrid algorithms. Finally, we also want to explore the applications of eager strategies in the synthesis of resilient controllers [9,23,11,15] and connections to symbolic synthesis algorithms for infinite-state systems [25,1,22].

Acknowledgements. We thank Bernd Finkbeiner and Martin Zimmermann for fruitful discussions. This work was supported by the German Research Foundation (DFG) under the project ASDPS (JA 2357/2-1).

References

1. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: S. Jagannathan, P. Sewell (eds.) POPL, pp. 221–234. ACM (2014). DOI 10.1145/2535838.2535860. URL `https://doi.org/10.1145/2535838.2535860`
2. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. In: VMCAI, *LNCS*, vol. 8318, pp. 1–20. Springer (2014)
3. Brenguier, R., Pérez, G.A., Raskin, J., Sankur, O.: AbsSynthe: abstract synthesis from succinct safety specifications. In: SYNT, *EPTCS*, vol. 157, pp. 100–116 (2014). DOI 10.4204/EPTCS.157.11
4. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. Amer. Math. Soc. **138**, 295–311 (1969). DOI 10.2307/1994916
5. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: CONCUR, *LNCS*, vol. 3653, pp. 66–80. Springer (2005)
6. Church, A.: Applications of recursive arithmetic to the problem of circuit synthesis. Summaries of the Summer Institute of Symbolic Logic **I**, 3–50 (1957)
7. Clarke, J.B.E., Long, D.: Representing circuits more efficiently in symbolic model checking. In: 28th ACM/IEEE Design Automation Conference, pp. 403–407 (1991)
8. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Automatic Verification Methods for Finite State Systems, *Lecture Notes in Computer Science*, vol. 407, pp. 365–373. Springer (1989). DOI 10.1007/3-540-52148-8_30. URL `https://doi.org/10.1007/3-540-52148-8_30`
9. Dallal, E., Neider, D., Tabuada, P.: Synthesis of safety controllers robust to unmodeled intermittent disturbances. In: CDC, pp. 7425–7430. IEEE (2016). DOI 10.1109/CDC.2016.7799416
10. Ehlers, R.: Symbolic bounded synthesis. Formal Methods in System Design **40**(2), 232–262 (2012). DOI 10.1007/s10703-011-0137-x
11. Ehlers, R., Topcu, U.: Resilience to intermittent assumption violations in reactive synthesis. In: HSCC, pp. 203–212. ACM (2014). DOI 10.1145/2562059.2562128. URL `http://doi.acm.org/10.1145/2562059.2562128`
12. Filiot, E., Jin, N., Raskin, J.F.: Antichains and compositional algorithms for LTL synthesis. Formal Methods in System Design **39**(3), 261–296 (2011). DOI 10.1007/s10703-011-0115-3

13. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: VMCAI, *LNCS*, vol. 7148, pp. 219–234. Springer (2012). DOI 10.1007/978-3-642-27940-9_15
14. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5-6), 519–539 (2013). DOI 10.1007/s10009-012-0228-z
15. Huang, C., Peled, D.A., Schewe, S., Wang, F.: A game-theoretic foundation for the maximum software resilience against dense errors. IEEE Trans. Software Eng. **42**(7), 605–622 (2016). DOI 10.1109/TSE.2015.2510001. URL `https://doi.org/10.1109/TSE.2015.2510001`
16. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In: SYNT@CAV, *EPTCS*, vol. 260, pp. 116–143 (2017). DOI 10.4204/EPTCS.260.10
17. Jacobs, S., Bloem, R., Brenguier, R., Ehlers, R., Hell, T., Könighofer, R., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The first reactive synthesis competition (SYNTCOMP 2014). STTT **19**(3), 367–390 (2017). DOI 10.1007/s10009-016-0416-3
18. Jacobs, S., Sakr, M.: A symbolic algorithm for lazy synthesis of eager strategies. In: ATVA, *Lecture Notes in Computer Science*, vol. 11138, pp. 211–227. Springer (2018). DOI 10.1007/978-3-030-01090-4_13. URL `https://doi.org/10.1007/978-3-030-01090-4_13`
19. Kropf, T.: Introduction to formal hardware verification. Springer Science & Business Media (2013)
20. Legg, A., Narodytska, N., Ryzhyk, L.: A SAT-based counterexample guided method for unbounded synthesis. In: CAV (2), *LNCS*, vol. 9780, pp. 364–382. Springer (2016). DOI 10.1007/978-3-319-41540-6_20
21. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points (extended abstract). In: ICALP, *LNCS*, vol. 1443, pp. 53–66. Springer (1998). DOI 10.1007/BFb0055040
22. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: M. Chechik, J. Raskin (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 9636, pp. 204–221. Springer (2016). DOI 10.1007/978-3-662-49674-9_12. URL `https://doi.org/10.1007/978-3-662-49674-9_12`
23. Neider, D., Weinert, A., Zimmermann, M.: Synthesizing optimally resilient controllers. In: CSL, *LIPIcs*, vol. 119, pp. 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018). DOI 10.4230/LIPIcs.CSL.2018.34. URL `https://doi.org/10.4230/LIPIcs.CSL.2018.34`
24. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989). DOI 10.1145/75277.75293
25. Raman, V., Donzé, A., Sadigh, D., Murray, R.M., Seshia, S.A.: Reactive synthesis from signal temporal logic specifications. In: HSCC, pp. 239–248. ACM (2015). DOI 10.1145/2728606.2728628. URL `http://doi.acm.org/10.1145/2728606.2728628`
26. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for the synthesis of reactive systems. STTT **15**(5-6), 433–454 (2013). DOI 10.1007/s10009-012-0224-3
27. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.0. University of Colorado at Boulder (2009)
28. Touati, H.J., Savoj, H., Lin, B., Brayton, R.K., Sangiovanni-Vincentelli, A.: Implicit state enumeration of finite state machines using bdd's. In: Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on, pp. 130–133. IEEE (1990)