Lazy Synthesis*

Bernd Finkbeiner[1] and Swen Jacobs[2]

[1] Universität des Saarlandes
finkbeiner@cs.uni-saarland.de
[2] École Polytechnique Fédérale de Lausanne
swen.jacobs@epfl.ch

Abstract. We present an automatic method for the synthesis of processes in a reactive system from specifications in linear-time temporal logic (LTL). The synthesis algorithm executes a loop consisting of three phases: Solve, Check, and Refine. In the Solve phase, a candidate solution is obtained as a model of a Boolean constraint system; in the Check phase, the candidate solution is checked for reachable error states; in the Refine phase, the constraint system is refined to eliminate any errors found in the Check phase. The algorithm terminates when an implementation without errors is found. We call our approach "lazy," because constraints on possible process implementations are only considered incrementally, as needed to rule out incorrect candidate solutions. This contrasts with the standard "eager" approach, where the full specification is considered right away. We report on experience in the arbiter synthesis for the AMBA bus protocol, where lazy synthesis leads to significantly smaller implementations than the previous eager approach.

1 Introduction

A major advantage of synthesis over verification is that manual programming is no longer required: synthesis automatically derives an implementation that is correct by construction. A major disadvantage is that synthesis requires a much more detailed specification. While the specifications used for verification typically focus on a small set of *safety-critical* properties, specifications for synthesis must describe *all* relevant properties of the process one wishes to synthesize as well as of the cooperating processes in the remainder of the system. This results in a *state explosion* problem similar to the infamous problem in verification, because the state space of the synthesized implementation is based on the product of all these properties.

An interesting example for this phenomenon is the synthesis of the AMBA bus protocol, which is currently the largest published case study carried out with automatic synthesis methods. Bloem *et al.* [1, 2] report that the automatically

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by the Swiss NSF Grant #200021_132176.

generated implementation is about 100 times larger than the manually written code and, furthermore, grows exponentially with the number of bus masters, even though the manually written code almost remains constant. The approach used by Bloem *et al.* may in fact not even show the full scale of the problem, because the simplifying assumption is made that the synthesized process has access to the full system state. Under incomplete information, i.e., when some state variables are hidden from the synthesized process, an additional subset construction is required that causes a further exponential blowup [3].

In this paper, we address the explosion of the state space during synthesis with a novel combination of synthesis and verification. Rather than running a synthesis procedure based on a full specification, we use verification to *lazily* identify and add constraints on the synthesized process that are actually needed to rule out incorrect implementations. Our starting point is a *partial design*, which includes an implementation for the already implemented part of the system, which we call the *white-box* process, and the interface to the part of the system that is to be synthesized, which we call the *black-box* process. The implementation of the white-box process is given as a labeled transition system; for the black-box process, an implementation is to be synthesized such that the composition of white-box and black-box implementation satisfies the specification, which is given as a formula of linear-time temporal logic (LTL). Nondeterminism in the white-box process is interpreted as hostile: the black-box process must ensure the satisfaction of the specification for all possible behaviors. Disjunctions in the LTL specification, by contrast, represent friendly nondeterminism, leaving design choices open to the synthesis of the black-box process.

Starting with an initial (trivial) constraint on the black-box process, we use an SMT-solver to generate a sequence of candidate implementations. Each candidate is combined with the white-box processes and checked for reachable errors. As long as such errors exist, we extract new constraints on the black-box process that exclude the error in future iterations. The algorithm terminates when an implementation without errors is found.

The new synthesis technique, which we call *lazy synthesis*, thus alternates between constraint solving, which produces new candidates, and model checking, which identifies errors in the candidates that lead to a refinement of the constraints. We refer to the individual phases of this process as Solve, Check, and Refine. The Solve-Check-Refine loop of lazy synthesis can be understood as an extension of the CEGAR (Counter-Example Guided Abstraction Refinement) loop [4] commonly used in verification, with the difference that the counterexamples that drive the refinement process are not found in abstractions of the given implementation, but rather in the continuously changing candidate solutions produced by the SMT-solver.

As described so far, lazy synthesis tends to find implementations that are significantly smaller than those found by eager methods, because we avoid the full construction of the product state space, but the implementations are not necessarily minimal. To further reduce the size of the synthesized implementations, we have integrated the *bounded synthesis* technique [5] into lazy synthesis.

Bounded synthesis searches for implementations up to a given bound on the number of states. To ensure the minimality of the synthesized implementation, we maintain a constraint that limits the number of states of the implementation. Starting with an initial low value (such as a single state), we increase the bound whenever the constraint system becomes unsatisfiable. In our experience, the number of states that is actually needed for a correct implementation is usually very small compared to the product state space constructed by eager methods.

To evaluate this observation experimentally, we have repeated the AMBA case study with lazy synthesis. It turns out that a large part of the protocol specification [6] is already deterministic. We modeled this part as the white-box process and thus focused the synthesis effort on the arbitration policy, which is left open in the protocol specification. Unlike Bloem *et al.*, we do not assume complete information, and were therefore able to minimize the number of signals the arbitration policy depends on. For typical fairness properties such as "every bus master that requests a grant, will eventually get one," expressed in LTL, lazy synthesis finds implementations with a linear number of states in the number of bus masters. This is in contrast with the exponential growth of the size of the implementations reported by Bloem *et al.* [1, 2] using eager synthesis.

The remainder of the paper is structured as follows. Section 2 introduces the AMBA protocol case study as a motivating example. In Section 3, we introduce the basic notions needed to discuss the synthesis problem; these notions allow us, in Section 4, to formalize the synthesis problem of the AMBA protocol case study. In Section 5, we describe the lazy synthesis algorithm. Section 6 gives some details on our current implementation of the approach, and in Section 7 we demonstrate how we used the approach and its implementation to synthesize arbiters for the AMBA specification. We conclude in Section 8 with a summary and some ideas for future research directions.

2 The AMBA Case Study

We will use the *Advanced Microcontroller Bus Architecture* (AMBA) specification [6] as a motivating example. This specification describes a communication bus for a number of masters and clients on a microchip. The bus controller keeps track of requests, and assigns the bus to one master at a time. Additionally, masters can ask for different kinds of *locked bursts*, i.e., sequences of transfers during which only this master is allowed to use the bus. We introduce briefly the signals that are used to realize the controller of this bus.

Requests and grants. To request the bus, master i will raise a signal HBUSREQi. The controller decides who will be granted the bus by raising signal HGRANTi. HMASTER[n:0] is an $n+1$-bit signal, where n is chosen such that the number of masters fits into $n+1$ bits. It always contains the identifier of the master which is currently active. Whenever the client raises HREADY, it is updated by letting HMASTER[n:0] = i, where HGRANTi is currently active.

Locks and bursts. A master can request a locked access by raising HLOCKi (in addition to HBUSREQi). If the locked access is granted, the master can set HBURST[1:0] to either SINGLE (single cycle access), BURST4 (four cycle burst) or INCR (unspecified length burst). For a BURST4 access, the bus will remain locked until the client has accepted 4 inputs from the master (signaled by raising HREADY 4 times). In case of an INCR access, the bus will remain locked until HBUSREQi is lowered. The arbiter raises signal HMASTLOCK if the bus is currently locked.

3 The Synthesis Problem

In this section we formalize the setting of our synthesis approach.

Partial designs. A *partial design* is a tuple $\mathcal{D} = (V, I, O, \mathcal{T}_W)$, where V is a set of boolean system variables, which also serve as the *atomic propositions*, the disjoint subsets $I, O \subseteq V$, $I \cap O = \emptyset$, are the *input* and *output* variables, respectively, of the black-box process. Input variables of the white-box are all variables from V, and outputs all variables from $V \setminus O$. $\mathcal{T}_W$ is the implementation of the white-box process, given as a labeled transition system, which is defined in the following.

Implementations. We represent implementations as labeled transition systems. For a given finite set Υ of directions and a finite set Σ of labels, a Σ-labeled Υ-*transition system* is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a finite set of states T, an initial state $t_0 \in T$, a (nondeterministic) transition function $\tau : T \times \Upsilon \to 2^T$, and a labeling function $o : T \to \Sigma$.

A *path* in a labeled transition system is a sequence $\mu : \omega \to T \times \Upsilon$ of states and directions that follows the successor relation, i.e., for all $i \in \omega$ if $\mu(i) = (t_i, e_i)$ then $\mu(i+1) = (t_{i+1}, e_{i+1})$ where $t_{i+1} \in \tau(t_i, e_{i+1})$. We call the path *initial* if it starts with the initial state and initial environment input: $\mu(0) = (t_0, e_0)$.

A process with input variables I and output variables O is implemented as a 2^O-labeled 2^I-transition system. Let $\mathcal{T}_1 = (T_1, t_{0,1}, \tau_1, o_1)$ be a 2^{O_1}-labeled 2^{I_1}-transition system, representing a process with inputs I_1 and outputs O_2, and let, likewise, $\mathcal{T}_2 = (T_2, t_{0,2}, \tau_2, o_2)$ be a 2^{O_2}-labeled 2^{I_2}-transition system, representing a second process with inputs I_2 and outputs O_2. The *parallel composition* of $\mathcal{T}_1$ and $\mathcal{T}_2$, denoted by $\mathcal{T}_1 \| \mathcal{T}_2$, is the $2^{O_1 \cup O_2}$-labeled $2^{(I_1 \cup I_2) \setminus (O_1 \cup O_2)}$-transition system $\mathcal{T} = (T, t, \tau, o)$, where the states consist of the product $T = T_1 \times T_2$, $t_0 = (t_{0,1}, t_{0,2})$, the transition function matches inputs with outputs generated in the previous step: $\tau((s_1, s_2), l) = \tau_1(s_1, (l \cup o_2(s_2)) \cap I_1) \times \tau_2(s_2, (l \cup o_1(s_1)) \cap I_2)$, and the labeling function is the union $o(s_1, s_2) = o_1(s_1) \cup o_2(s_2)$. We call the parallel composition of the white-box implementation and the black-box implementation the *system implementation*.

Specifications. We use linear-time temporal logic (LTL) [7], with the usual modalities Next $\bigcirc$, Until $\mathcal{U}$, Eventually $\Diamond$, and Globally $\Box$, as the specification

logic. If a sequence $\pi \in \omega \to 2^V$ satisfies an LTL formula φ, we say that π is a *model* of φ, denoted by $\pi \models \varphi$. A $2^{V \setminus O_{env}}$-labeled $2^{O_{env}}$-transition system (T, t_0, τ, o) *satisfies* an LTL formula φ if, for all initial paths $\mu : \omega \to T \times 2^{O_{env}}$ of the transition system, the sequence $\sigma_\mu : i \mapsto \tilde{o}(\mu(i))$ is a model of φ, where $\tilde{o}(t, e) = o(t) \cup e$.

Realizability and synthesis. An LTL specification φ is (finite-state) *realizable* in a partial design $\mathcal{D} = (V, I, O, \mathcal{T}_W)$ iff there exists an implementation $\mathcal{T}_B$ for the black-box process, such that the system implementation $\mathcal{T}_W \| \mathcal{T}_B$ satisfies φ. In this case, we say that the black-box implementation is *correct*.

Following the *bounded synthesis* approach [5], we introduce a bound $n \in \mathbb{N}$ on the size of the black-box implementation. Given an architecture $D = (V, I, O, \mathcal{T})$, a specification φ, and a bound n, we say that φ is n-realizable in D if there exists a correct implementation $\mathcal{T}_B$ of the black-box process, such that T_B has no more than n states.

The *synthesis problem* is to compute a correct black-box implementation if the given LTL specification is realizable in the given partial design.

4 The Partial Design of the AMBA Protocol

The starting point of the AMBA case study is the informal specification [6] available from the ARM website. In order to apply lazy synthesis, the informal specification needs to be formalized into a partial design and an LTL specification. In this section, we discuss these modeling decisions.

The white-box process. Upon inspection of the AMBA specification, one can easily see that at any given time, the valuations of variables HMASTER[n:0] and HMASTLOCK are completely determined by the history of the other variables of the system: whenever HREADY holds, the specification requires that in the next state HMASTER[n:0] will be equal to i, for every i such that HGRANTi holds in the current state. In addition to determining HMASTER[n:0] wrt. HREADY and the HGRANTi, this indirectly imposes a mutual exclusion property on the HGRANTi, since the property cannot be satisfied for multiple HGRANTi at the same time. In a similar fashion, HMASTLOCK is determined: whenever a master i is granted an access, variable HLOCKi determines whether it will be a locked access. If this is the case, and HBURST[1:0] is either BURST4 or INCR, then HMASTLOCK has to be set until the desired burst access is over, i.e. either until the client accepted 4 transmissions from the master (each signaled by HREADY being high), or until the master lowers HBUSREQi. Using this deterministic specification, we can easily build a white-box process that governs variables HMASTER[n:0] and HMASTLOCK and satisfies this part of the specification.

The black-box process. The remaining variables controlled by the system are the HGRANTi variables. Except for their valuation in the initial state, these

are only indirectly specified by their influence on the other variables, and the global requirements on the overall system. These variables are controlled by the black-box process.

The interface of the white-box process. To keep the interface of the black-box small, we add an auxiliary variable DECIDE to the white-box process, which is set whenever the access of a master is finished. We will see that the right definition of DECIDE allows the lazy synthesis algorithm to find a correct black box process without knowing about the valuations of any other variables.

Figure 1 gives a slice of the resulting white-box process. In all of the depicted states, HMASTER[n:0] has the same value. The overall white-box consists of such a slice for every master, and transitions to states with a different valuation of HMASTER[n:0] are only possible from state 0, or the corresponding state in the given slice. HMASTLOCK is true in all states except 0 and 1, and DECIDE is true in the states depicted as dashed circles. Transitions that do not contain any conditions are taken unconditionally, and whenever none of the outgoing transitions is possible, we remain in the state. From 0, transitions into several different states of the other slices of the system are possible.

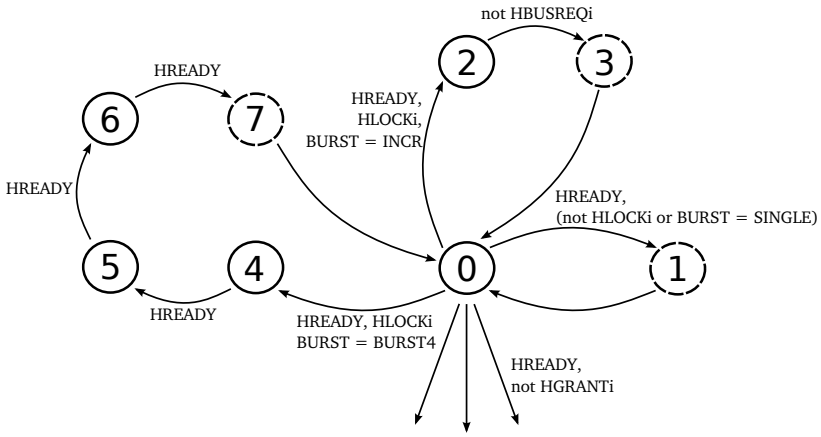

Fig. 1. Slice for one master of the AMBA white-box process

LTL specification. The LTL specification consists of the formula $A1 \wedge A2 \Rightarrow G1 \wedge G2 \wedge G3 \wedge G4$ with the assumptions and guarantees shown in Figure 2. Formula (A1) and (A2) are assumptions on the environment: neither are the clients busy forever, nor is the bus locked forever. The second formula is an indirect assumption on the environment, as the only way HMASTLOCK can be true forever in our white-box process is if HBUSREQi holds forever after master i acquires a lock on an INCR burst. ($G1$) and ($G2$) are guarantees that follow from the requirement that whenever HREADY is high, the white-box process

Assumptions :

$$\square\diamond \text{ HREADY} \qquad (A1)$$

$$\square\diamond \neg\text{HMASTLOCK} \qquad (A2)$$

Guarantees :

$$\square\ (\text{HREADY} \rightarrow \bigvee_{\text{i}} \text{HGRANTi}) \qquad (G1)$$

$$\forall \text{i} \neq \text{j} : \square\ (\text{HREADY} \rightarrow \neg(\text{HGRANTi} \wedge \text{HGRANTj})) \qquad (G2)$$

$$\forall \text{i} : \square\ (\text{HBUSREQi} \rightarrow \diamond(\neg\text{HBUSREQi} \vee \text{HMASTER} = \text{i})) \qquad (G3)$$

$$\forall i : \square\ (\neg\text{DECIDE} \rightarrow (\text{HGRANTi} \leftrightarrow \bigcirc\text{HGRANTi})) \qquad (G4)$$

Fig. 2. The LTL specification of the AMBA specification.

must update HMASTER with any i s.t. HGRANTi is true. As HMASTER can only hold exactly one value, this implies that always exactly one grant must be true. ($G3$) is the fairness guarantee of the system: a HBUSREQi that is not lowered again will eventually be answered by setting HMASTER[n:0] = i. Finally, ($G4$) is an optional constraint on the auxiliary DECIDE variable. Similar to the definition of auxiliary variables for verification, this property of DECIDE will help guide the lazy synthesis algorithm.

5 Lazy Synthesis

We now describe the lazy synthesis algorithm, which solves the synthesis problem for a given partial design and LTL specification. The first subsection gives an overview of the Solve-Check-Refine loop, the individual building blocks of the loop are described in more detail in the following subsections.

5.1 The Solve-Check-Refine loop

Figure 3 shows the main loop of the lazy synthesis algorithm. Given a partial design $\mathcal{D}$ and a specification φ, Procedure LazySynthesis($\mathcal{D}, \varphi$) computes the least bound $n \in \mathbb{N}$ such that φ is n-realizable in $\mathcal{D}$ and returns a black-box implementation with n states.

The algorithm incrementally increases the bound n on the number of states of the black-box implementation until an implementation is found. For each bound, we incrementally strengthen the constraint C, starting with *init_constraint*, until either the constraint becomes unsatisfiable and we try with higher bound n, or a correct implementation is found, at which point the algorithm terminates.

The algorithm builds on the following subroutines, which will be explained in the following subsections.

- Solve. The constraint C is a ground formula over booleans (representing inputs) and integers (representing states), with function symbols that represent transitions and outputs of the black-box component. It is used to forbid certain input/output patterns of the black-box process. Given such a constraint

```
LazySynthesis(D, φ)
 1  n ← 1
 2  correct ← false
 3  C ← init_constraint
 4  while correct = false
 5        do
 6           (model-found, T_B) = Solve(C, n)
 7           if model-found = true
 8              then
 9                   (correct, error-sequence) = Check(D, φ, model)
10                   if correct = false
11                      then C ← Refine(C, error-sequence)
12              else n ← n + 1
13                   C ← init_constraint
14  return T_B
```

Fig. 3. Algorithm for lazy synthesis. Given a partial design $\mathcal{D}$ and a specification φ, procedure LazySynthesis($\mathcal{D}, \varphi$) computes the least bound n such that φ is n-realizable in $\mathcal{D}$ and returns a black-box implementation with n states.

C and a bound n, procedure Solve(C, n) checks if there exists a black-box implementation with at most n states that satisfies the constraint C. The result is a pair (*model-found*, $\mathcal{T}_B$), where the first component *model-found* is a boolean flag indicating whether a solution has been found, and if this flag is true, then the second component is a candidate implementation for the black-box process.

- Check. Given a partial design $\mathcal{D}$, a specification φ, and a black-box implementation $\mathcal{T}_B$ constructed by Solve, Check($\mathcal{D}, \varphi, \mathcal{T}_B$) verifies whether the composition of white-box and black-box implementation satisfies φ. The procedure returns a pair (*correct*, *error-sequence*), where the first component is a boolean flag indicating whether the implementation is correct, and the second component is a representation of the error paths found if the implementation is not correct.
- Refine. Procedure Refine(C, *error-sequence*) organizes the error paths found by procedure Check into a tree representation that starts with the initial state. This error tree is then translated into a new conjunct in the constraint that forbids all error paths collected by procedure Check.

5.2 Solve

The goal of procedure Solve is to find an implementation for the black-box process that satisfies the constraints collected so far. For a black-box process with at most n states, we assume, without loss of generality, that the states are the natural numbers from 0 to $n-1$ and that the initial state is 0. We can also assume that the black-box implementation is deterministic, because any given

nondeterministic implementation, for which the system implementation satisfies the specification, can obviously safely be replaced by any of its deterministic restrictions. We represent the unknown transition function using an uninterpreted function symbol *trans* of type $\mathbb{B}^{|I|} \times \{0, \dots n-1\} \to \{0, \dots n-1\}$. The unknown labeling function is represented by an uninterpreted function symbol *label* of type $\{0, \dots, n-1\} \to 2^O$.

We denote with $\mathbb{T}(X)$ the set of terms over a set X of function symbols and constants, and with $\mathbb{C}(X)$ the set of constraints over X. We use terms in $\mathbb{T}(\{trans, label, 0\})$ to symbolically identify the states that are reached after a certain sequence of inputs and outputs, and constraints in $\mathbb{C}(\{trans, label, 0\})$ to describe conditions on such states. Any interpretation τ, o of the symbols *trans* and *label* defines an implementation of the black-box process, the 2^O-labeled 2^I-transition system $\mathcal{T}_B = (\{0, \dots n-1\}, 0, \tau, o)$. To improve readability, we will also use, for a given interpretation o of *label*, directly the variable names to denote functions from states to Boolean values. I.e., HGRANT0(1)= true iff HGRANT0 $\in$ $o(1)$. In the synthesis loop, procedure SOLVE(C, n) uses an SMT-solver to find such interpretations. To enforce the limit on the size of the implementation, we extend the constraint system that is passed to the solver with an appropriate type constraint (i.e, $\forall\ \overline{b} \in \mathbb{B}^{|I|}, t \in \{0, \dots, n-1\}.\ 0 \leq trans(\overline{b}, t) \leq n-1$).

Example 1. In the AMBA specification, the black-box process initially is only constrained by an upper bound on the size of the implementation we are currently looking for and the valuations of its output variables in the initial state 0. Suppose we want to synthesize an implementation for 2 masters, we are looking for models of size up to 3, the only input to the black-box process is DECIDE, and initially HGRANT0 should be high, and HGRANT1 low. Thus, we assert

$$\text{HGRANT0}(0) \wedge \neg\text{HGRANT1}(0) \wedge \forall\ b \in \mathbb{B},\ n \in \{0, \dots, n-1\}.\ 0 \leq trans(b, n) \leq 2$$

in the SMT solver. We may get the model

$$\begin{array}{rr@{\qquad}rr} \text{HGRANT0} : & 0 \mapsto true & trans : & (false, 0) \mapsto 1 \\ & 1 \mapsto false & & (true, 0) \mapsto 1 \\ & 2 \mapsto true & & (false, 1) \mapsto 2 \\ \text{HGRANT1} : & 0 \mapsto false & & (true, 1) \mapsto 0 \\ & 1 \mapsto false & & (false, 2) \mapsto 0 \\ & 2 \mapsto true & & (true, 2) \mapsto 0, \end{array}$$

representing a candidate implementation of the black box.

5.3 CHECK

Procedure CHECK verifies whether the composition of the candidate black-box implementation constructed by SOLVE with the white-box implementation satisfies the specification φ. If φ is violated, we extract a set of counterexamples. We are interested in *finite* counterexamples, because they can easily be eliminated in the subsequent REFINE phase. Since counterexamples to LTL specifications are

in general infinite, we first translate the LTL formula φ to a safety property, for which all counterexamples are finite. As pointed out in [5], a reduction to safety is possible whenever the size of the implementation is bounded. To construct a monitor process for an LTL specification, we adapt a reduction given in [5], Theorem 4 (there stated in terms of a translation from universal co-Büchi tree automata to deterministic safety tree automata) to our setting.

Recall that a Büchi word automaton over alphabet Σ is a tuple $\mathcal{A} = (Q, Q_0, \Delta, F)$, where Q is a finite set of states, $Q_0 \subseteq Q$ a subset of initial states, $\Delta \subseteq Q \times \Sigma \times Q$ a set of transitions, and $F \subseteq Q$ a subset of accepting states. A Büchi automaton accepts an infinite word $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega$ iff there exists a run r of $\mathcal{A}$ on w, i.e., an infinite sequence $r_0 r_1 r_2 \ldots \in Q^\omega$ of states such that $r_0 \in Q_0$ and $(r_i, w_i, r_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$, such that $r_j \in F$ for infinitely many $j \in \mathbb{N}$. The set of sequences accepted by $\mathcal{A}$ is called the *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$. Let $\mathcal{A}_{\neg\varphi} = (Q_{\neg\varphi}, Q_{0,\neg\varphi}, \Delta_{\neg\varphi}, F_{\neg\varphi})$ be a Büchi automaton that accepts all sequences in $(2^V)^\omega$ that satisfy $\neg\varphi$, and therefore violate φ.

Proposition 1. *For every LTL formula φ and every bound $m \in \mathbb{N}$ on the number of states of the system implementation, there exists a family of monitor processes $\{\mathcal{T}_{\neg\varphi,m'} \mid m' \in \mathbb{N}\}$ with error state err, such that*

1. *any system implementation $\mathcal{T}$ satisfies φ* if *err is unreachable in $\mathcal{T} \| \mathcal{T}_{\neg\varphi,m'}$, and*
2. *for $m' \geq m \cdot |Q_{\neg\varphi}| + 1$, any system implementation $\mathcal{T}$ with at most m states satisfies φ* if and only if *err is unreachable in $\mathcal{T} \| \mathcal{T}_{\neg\varphi,m'}$.*

Proof. We construct a monitoring process $\mathcal{T}_{\neg\varphi,m'} = (T, t_0, \tau, o)$ with designated error state *err*:

- $T = (Q \to \{0, \ldots m', \_\}) \cup \{err\}$;
- t_0 is the function $t_0 : Q \to \{0, \ldots, m', \_\}$ with $t_0(q) = 0$ if $q \in Q_0$ and $t_0(q) = \_$ otherwise;
- $\tau(err, \sigma) = \{err\}$,
  $\tau(f, \sigma) = \{err\}$ if there are two states $q \in F, q' \in Q$ such that $f(q) = m'$ and the transition (q, σ, q') is in Δ, and
  $\tau(f, \sigma) = \{f'\}$, otherwise, with $f'(q') = \max\{f(q) + g(q) \mid f(q) \neq \_, (q, \sigma, q') \in \Delta\}$, where $g(q) = 1$ if $q \in F$ and $g(q) = 0$ if $q \notin F$, and $\max \emptyset = \_$;
- $o(t) = \emptyset$ for all $t \in T$.

Each state of the monitoring process thus maintains, for each state q of $\mathcal{A}_{\neg\varphi}$, two pieces of information: (1) whether or not q is, in the current position, visited on some run (if not, q is assigned a blank $\_$ symbol), and (2) the maximum number of visits to accepting states on any run prefix of $\mathcal{A}_{\neg\varphi}$ ending in state q. If the number of visits to accepting states is bounded by m', the monitor does not reach *err* and the system implementation satisfies φ. For system implementations with up to m states, it suffices to use $m' = m \cdot |Q_{\neg\varphi}| + 1$. Consider the product $\mathcal{A}' = (Q \times T, Q_0 \times \{t_0\}, \{(q,t), \sigma, (q',t')) \mid (q, \sigma, q') \in \Delta, o(t') = \sigma\}, F_{\neg\varphi} \times T)$ of the system implementation and $\mathcal{A}_{\neg\varphi}$. If, on some run of the product automaton,

the accepting states of $\mathcal{A}_{\neg\varphi}$ have been visited more than $m \cdot |Q_{\neg\varphi}|$ times, some product state consisting of some state of the implementation and some accepting state of $\mathcal{A}_{\neg\varphi}$ must have been visited twice, and we can hence construct a path in the implementation and an accepting run of $\mathcal{A}_{\neg\varphi}$ by repeating the cycle infinitely often. □

Let $\mathcal{D} = (V, I, O, \mathcal{T}_W)$ be a partial design, φ an LTL specification, $m' \in \mathbb{N}$ a natural number, and T_W the states of $\mathcal{T}_W$. We call the pair $\mathcal{E} = \big((V, I, O, \mathcal{T}_W \| \mathcal{T}_{\neg\varphi,m'}), T_W \times \{err\}\big)$, consisting of a partial design and a set of error states, the *extended* partial design. The white-box process of $\mathcal{E}$ additionally keeps track of the state of the monitor process $\mathcal{T}_{\neg\varphi,m'}$.

An *error path* of a system implementation $\mathcal{T}$ of an extended partial design is a finite prefix $\mu(0)\mu(1)\dots\mu(k)$ of a path μ such that $\mu(k)$ is an error state. A *counterexample* is an initial error path. If no counterexamples have been found, the algorithm terminates and returns $\mathcal{T}_B$. Otherwise, the set of counterexamples for $m' = m \cdot |Q_{\neg\varphi}|$ is collected in the form of an *error sequence* $E_0, E_1, \dots E_k \in (2^T)^*$, such that for each $0 \leq i \leq k$, the states in E_i have a minimal error path of length i.

Procedure Check assumes a fixed bound m'. While $m' = m \cdot |Q_{\neg\varphi}| + 1$ is a safe choice, in practice it is more efficient to start with small bounds and incrementally increase m' if no implementation is found.

Example 2. The properties from Figure 2 are translated into a monitoring process. For simplicity, assume we only have a monitor for $(G2)$, with i=0 and j=1. The monitor moves from its initial state 0 into the error state *err* whenever HREADY, HGRANT0 and HGRANT1 are simultaneously true.

In the system implementation of the extended partial design, with the white-box implementation from Fig. 1 and the black-box implementation from Example 1, error states E_0 are all tuples (a, b, err), where a is any state of the black-box process and b is any state of the white-box process. We will denote this set of states as $(*, *, err)$. The backwards reachable states from $(*, *, err)$ are all states in which the black-box is in state 2, since this triggers the monitor to move into *err*. The black-box process only moves into 2 when it is in 1 and DECIDE is false, so in all pre-states the white-box needs to be in one of the states in $S_1 = \{0, 2, 3, 4, 5, 6\}$, or the corresponding states with HMASTER = 1. Denoting these states by S_1', the backwards reachable states from $E_1 = (2, *, *)$ are $E_2 = (1, S_1 \cup S_1', *)$. Finally, the black-box process can only reach state 1 from state 0, and does so without further conditions. Pre-states of $S_1 \cup S_1'$ in the white-box are $S_2 = \{0, 1, 2, 4, 5, 7\}$ and the corresponding S_2', so backwards reachable states from $(1, S_1 \cup S_1', *)$ are $E_3 = (0, S_2 \cup S_2', *)$. Since E_3 contains the initial state $(0, 0, 0)$, the sequence $E_0, \dots, E_3$ is an error sequence.

5.4 Refine

Refine uses the error sequence found by Check to refine the constraint on the black-box process. For this purpose, we first organize the error sequence into a

tree that starts with the initial state and branches according to the values of the variables visible to the black-box process. We denote a Σ-labeled finite tree over a set Υ of directions as a pair (N, l), where $N \subseteq \Upsilon^*$ is a prefix-closed set of finite words over Υ, identifying the nodes of the tree, and $l : N \to \Sigma$ is the labeling function. The root of the tree is the empty word ϵ. A node $w \in N$ is a leaf if it has no children, i.e., $\{w \cdot \upsilon \mid \upsilon \in \Upsilon\} \cap N = \emptyset$. Let $V_B = I \cup O$ be the set of variables visible to the black-box process.

A *counterexample tree* for a system implementation $\mathcal{T} = (T, t_0, \tau, o)$, an extended partial design $\mathcal{E} = \big((V, I, O, \mathcal{T}_W \| \mathcal{T}_{\neg\varphi,m'}), T_W \times \{err\}\big)$ and an error sequence $E_0, E_1, \ldots E_k \in (2^T)^*$ is a finite 2^T-labeled tree (N, l) with directions $\Upsilon = 2^I$ such that the following conditions hold:

- The root of the tree is labeled with the singleton set $\{t_0\}$ consisting of the initial state.
- For each node $w \in N$ and each direction $\upsilon \in \Upsilon$ there is a child $w \cdot \upsilon \in N$ iff (1) the label of w does not contain an error state, i.e., $l(w) \cap E_0 = \emptyset$, and (2) the set of states in $E_{k-|w|-1}$ that are υ-successors of states in the label of the parent is non-empty. In this case, the child is labeled with this set:
  $w \cdot \upsilon \in N$ iff $l(w) \cap E_0 = \emptyset$ and $\{\tau(q, \upsilon) \mid q \in l(w)\} \cap E_{k-|w|-1} \neq \emptyset$, and $l(w \cdot \upsilon) = \{\tau(q, \upsilon) \mid q \in l(w)\} \cap E_{k-|w|-1}$.

To refine the constraint on the black-box process, we translate the counterexample tree (N, l) into a constraint that ensures that, in future iterations, each counterexample is prevented by the black-box process.

Proposition 2. *Let (N, l) be a counterexample tree. There exists a constraint $C_{(N,l)}$ that eliminates exactly those black-box implementations for which the system implementation has one of the counterexamples in (N, l).*

Proof. We set $C_{(N,l)} := \mathit{constr}(\epsilon, 0)$, where the function $\mathit{constr} : (N \times \mathbb{T}(\{\mathit{trans}, \mathit{label}, 0\})) \to \mathbb{C}(\{\mathit{trans}, \mathit{label}, 0\})$ is defined inductively as follows:

- for a leaf node $w \in N, l(w) \cap E_0 \neq \emptyset$,
  $\mathit{constr}(w, t) = \mathit{false}$;
- for a non-leaf node $w \in N, l(w) \cap E_0 = \emptyset$,
$$\mathit{constr}(w, t) = \bigwedge_{w \cdot \upsilon \in N, \upsilon \in 2^I} \begin{pmatrix} \mathit{label}(t) \neq (\upsilon \cap O) \\ \vee\ \mathit{constr}(w \cdot \upsilon, \mathit{trans}(t, \upsilon)) \end{pmatrix}.$$

□

Example 3. We inspect the error sequence obtained during the CHECK phase in Example 2. Forward reachable states from $(0, 0, 0)$ are $(1, S_3 \cup S_3', 0)$, where $S_3 = \{0, 1, 2, 4\}$.

Construction of the counterexample tree can be seen as a branching model checking procedure, which first partitions S_3 into states $S_4 = \{1\}$ where DECIDE holds, and $S_5 = \{0, 2, 4\}$ where it does not hold. Then, state sets

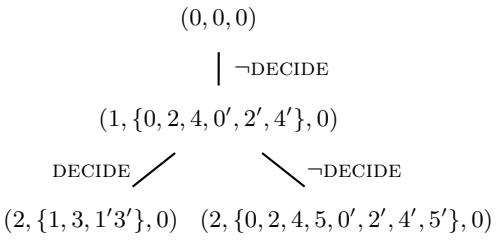

Fig. 4. Counterexample Tree

$(1, S_4 \cup S_4', 0)$ and $(1, S_5 \cup S_5', 0)$ are intersected with E_2 from Example 2, resulting in the empty set and $(1, S_5 \cup S_5', 0)$, respectively. Forward reachable from $(1, S_5 \cup S_5', 0)$ are $(2, S_6 \cup S_6', 0)$, where $S_6 = \{0, 1, 2, 3, 4, 5\}$.

Partitioning states again wrt. DECIDE gives us $(2, \{1, 3, 1'3'\}, 0)$, and $(2, \{0, 2, 4, 5, 0', 2', 4', 5'\}, 0)$. Since we have reached state 2 of the black-box process (which triggers the monitor to move into *err*), all successor states of these will be error states. Figure 4 depicts the resulting counterexample tree (leaving out the leaves labeled with *false*). The corresponding counterexample constraint is

$$\begin{aligned}
&\neg\text{HGRANT0}(0) \vee \text{HGRANT1}(0) \\
\vee\ &\text{HGRANT0}(trans(false, 0)) \vee \text{HGRANT1}(trans(false, 0)) \\
\vee\ &((\neg\text{HGRANT0}(trans(true, trans(false, 0))) \\
&\quad \vee\ \neg\text{HGRANT1}(trans(true, trans(false, 0)))) \\
&\wedge (\neg\text{HGRANT0}(trans(false, trans(false, 0))) \\
&\quad \vee\ \neg\text{HGRANT1}(trans(false, trans(false, 0))))))).
\end{aligned}$$

6 Symbolic Implementation

We have implemented the algorithm described in Section 5 in OCaml, tightly integrating the SMT solver Z3 [8] and the BDD package CUDD [9].

Initialization. The input to our tool contains the partial design $\mathcal{D}$ and specification φ of the desired system in one file. White box and monitor automata are translated into a BDD representation of their initial and error states, as well as their respective transition relations. These will not change during the main loop of the algorithm.[3]

Main Loop: Solve**,** Check**,** Refine

- Solve. The solve phase is handled by the SMT solver, which receives the current set C of constraints on the black-box process, and either returns a model or the result *unsatisfiable*. In the latter case, we increase the bound and try again.

[3] This means that monitor automata currently do not grow with the size bound, but their size m' must be chosen large enough from start.

– Check. The model obtained from the SMT solver is translated into a BDD representation, and we construct a BDD representation of the complete system, including the candidate black box. We apply backward model checking, storing BDD representations of the error sequence $E_0, \ldots, E_k$.
– Refine. To obtain the *counterexample tree*, we start another model checking run, this time going forward from the backwards reachable initial states identified in the Check phase. In every iteration, we partition the reachable states according to the valuations of input variables of the black-box process, resulting in a branching model checking process. To allow efficient partitioning, we enforce an ordering on the BDD which always keeps input variables on top. Furthermore, every element of this partition is intersected with E_{k-j}, where j is the number of steps we have taken in the forward model checking process. As a consequence, state sets that will not lead to an error in the minimal number of steps become empty, and these branches of the process are pruned.
  During the branching model checking process, we store constraints on input variables that correspond to the partitioning of the reachable states in the counterexample tree. By construction, every branch of the process will have reached the error states after k steps, and we obtain an error tree of depth k. The constraints in this tree are combined as described in Section 5, such that they exclude all minimal error paths from this model in the future candidate models produced by Solve.

7 Experiments

Table 1 gives experimental results on the AMBA case study obtained with our prototype implementation of the lazy synthesis approach. We synthesized arbiters for architectures with 2, 4 or 6 masters, using lazy synthesis with monitors with a fixed valuation of m' of 10, 14, or 18. For the interface of the black-box, we tested the cases $I = \{\text{DECIDE}\}$, $I = \{\text{DECIDE}, \text{HMASTLOCK}\}$, and $I = \{\text{DECIDE}, \text{HMASTLOCK}, \text{HREADY}\}$. DEC indicates that we used the optional constraint ($G4$) from Figure 2 to guide the search.

Times are given in seconds, on an Intel Core i7 CPU @ 2.67GHz. TO marks cases where a timeout of 5 hours has been reached, "unsat" cases where the specification is unsatisfiable for the given monitor. The size of the synthesized black-box is equal to the number of masters in the system. For 8 masters, the tool timed out for all options mentioned above.

Comparison to Bloem *et al.* The AMBA case study has been carried out with an eager synthesis method by Bloem *et al.* [1, 2]. Similar to our auxiliary variable DECIDE, they defined several auxiliary variables and constraints that help guide the synthesis process. However, in contrast to our approach, Bloem *et al.* synthesized the complete controller, including the deterministic parts we have included in our white box. The advantage of synthesizing the complete controller is that it justifies the assumption of complete information, which results

Table 1. Experimental Results

	DEC					w/o DEC				
	m'=10			m'=14	m'=18	m'=10			m'=14	m'=18
	$\|I\|=1$	$\|I\|=2$	$\|I\|=3$	$\|I\|=1$	$\|I\|=1$	$\|I\|=1$	$\|I\|=2$	$\|I\|=3$	$\|I\|=1$	$\|I\|=1$
2 Masters	0.6	0.6	0.6	0.4	0.7	0.3	0.3	0.5	0.4	0.5
4 Masters	14.6	18.4	61.0	39.1	242.9	47.6	162.6	TO	332.6	923.9
6 Masters	unsat	unsat	unsat	9952.0	TO	unsat	unsat	unsat	TO	TO

in a simpler synthesis problem. However, focusing the synthesis on the black-box process, which does not have access to the full state, allows us to obtain smaller implementations. Using lazy synthesis, we can minimize both the interface between black and white box (and thus, find the signals on which the arbitration policy depends) and minimize the number of states of the the black-box implementation. The size of the implementation synthesized using the lazy approach is linear in the number of masters, while Bloem *et al.* report exponential growth.

8 Conclusions

We have presented *lazy synthesis*, a novel combination of synthesis and verification. Lazy synthesis focuses the synthesis effort on the relevant part of the design, the *black-box* process and ensures that only constraints that are needed to rule out incorrect implementations are considered. The main practical advantage of lazy synthesis is that it produces dramatically smaller implementations than eager methods. This has three main reasons. First, unnecessary constraints are avoided. Second, the incomplete information of the black-box process is treated accurately, and, hence, irrelevant dependencies are avoided. Third, lazy synthesis integrates bounded synthesis, and thus ensures that the number of states in the implementation is minimal.

A related method, called *counter-example guided inductive synthesis* (CEGIS), has been proposed for functional synthesis of sequential and concurrent programs [10, 11]. Like lazy synthesis, the approach is based on generating candidate solutions to the synthesis problem, and refining them based on error traces, but there are several differences. One of the main differences to lazy synthesis is that program executions in CEGIS are finite, while we consider properties of reactive systems on possibly infinite traces. Furthermore, synthesis in CEGIS is restricted to specific constructs, like finding values for constants and regular expressions, or reordering program statements given in the partial implementation. Finally, in case the candidate implementation does not satisfy the specification, we produce a constraint that excludes all minimal error paths from subsequent models, while the CEGIS approach only excludes one particular error per iteration.

Future Work. There are two major issues that deserve further investigation. The first issue concerns the limitation of the presented approach to finite-state

white-box processes. This limitation could be avoided by integrating lazy synthesis with automatic abstraction refinement (cf. [12]). The second issue is the limitation to single black-box processes. In distributed systems, there are typically multiple processes that each have an incomplete view of the global state. Even though the synthesis problem for distributed architectures is, in general, undecidable, lazy synthesis should, in principle, be applicable to distributed architectures, because both the verification problem and the bounded synthesis problem are decidable.

Acknowledgments. We thank Bertrand Jeannet for help with the OCaml interface of CUDD.

References

1. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: Proc. DATE. (2007) 1188–1193
2. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. In: Proc. COCV. (2007) 3–16
3. Reif, J.H.: The complexity of two-player games of incomplete information. J. Comput. Syst. Sci. **29**(2) (1984) 274–301
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In Emerson, E.A., Sistla, A.P., eds.: CAV. Volume 1855 of Lecture Notes in Computer Science., Springer (2000) 154–169
5. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Proc. ATVA. Volume 4762 of Lecture Notes in Computer Science., Springer-Verlag (2007) 474–488
6. ARM Ltd.: AMBA specification (rev.2). Available from www.arm.com (1999)
7. Pnueli, A.: The temporal logic of programs. In: Proc. FOCS, IEEE Computer Society Press (1977) 46–57
8. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS, Springer-Verlag (2008) 337–340
9. Somenzi, F.: CUDD: CU Decision Diagram Package, Release 2.4.2. University of Colorado at Boulder (2009)
10. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. (2006) 404–415
11. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: PLDI. (2008) 136–148
12. Dimitrova, R., Finkbeiner, B.: Abstraction refinement for games with incomplete information. In Hariharan, R., Mukund, M., Vinay, V., eds.: FSTTCS. (2008)