Symbolically Synthesizing Small Circuits

Rüdiger Ehlers[1], Robert Könighofer[2], and Georg Hofferek[2]

[1]Reactive Systems Group, Saarland University, Germany
[2]Institute for Applied Information Processing and Communications, Graz University of Technology, Austria

***Abstract*—*Reactive synthesis*, where a finite-state system is automatically generated from its specification, is a particularly ambitious way to engineer correct-by-construction systems. In this paper, we propose *implementation-extraction based on computational learning of Boolean functions* as a final synthesis step in order to obtain *small and fast circuits* for realizable specifications in a symbolic way. Our starting point is a restriction of the system player's choices in a synthesis game such that all remaining strategies are winning. Such games are used in most symbolic synthesis tools, and hence, our technique is not tied to one specific synthesis workflow, but rather supports a large variety of these. We present several variants of our implementation-learning approach, including one based on Bshouty's monotone theory. The key idea is the efficient use of the system player's freedom in the game. Our experimental results show a significant reduction of implementation size compared to previous methods, while maintaining reasonable computation times.**

I. Introduction

A common criticism on formal methods for the verification of reactive systems is that they only aid the system engineer with ensuring correctness *after* the system is constructed. The idea of *reactive synthesis* is to change this situation by automatically computing a correct-by-construction system after the specification is stated. While the theoretical complexity of the synthesis problem is long-established for many important specification formalisms, only recently, progress on the practical solution of this problem has gained momentum, as new results on symbolic synthesis [18], [17], smart specification decomposition techniques [16], [29], and specification formalisms explicitly targeting synthesis have emerged [7].

Early theory on the subject was mainly concerned with checking the *realizability* of a specification, i.e., testing if there exists an implementation. Procedures for obtaining an implementation in case of a positive answer were rudimentary and did not consider the *quality* of the synthesized solutions. With the rise of synthesis technology from its infancy, a growing interest in synthesizing solutions of high quality (e.g., requiring only a small on-chip area, or reacting quickly) emerged, witnessed by the introduction of synthesis methods that take quantitative criteria into account. Introducing quantitative criteria at the start of the synthesis process however typically breaks the possibility to perform the synthesis process in a symbolic way, e.g., using binary decision diagrams (BDDs).

This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), by the German Science Foundation (DFG) through the AVACS Transregional Collaborative Research Center (SFB/TR 14), and by the Austrian Science Fund (FWF) through project QUAINT (I774-N23) and the national research network RiSE (S11406-N23).

The key idea to solve this problem is to consider the quality only at a later stage in the synthesis process, when the realizability of a specification has already been determined. Most synthesis approaches reduce the realizability problem of a given specification to solving a two-player game in which one player models the environment and provides the input to a system to be synthesized, whereas the other player models the system and provides the output. If and only if the system player can always win the game, the specification is realizable. We call the characterization of a set of moves a *general strategy* if the system player wins when taking only moves from this set. Calling it *general* is justified by the fact that there are often situations in which the strategy can be *non-deterministic*, i.e., it has multiple choices for the system player to win in that situation. Any implementation that resolves this non-determinism, and is thus a *specialization* of such a general strategy, satisfies the original specification. Since a general strategy is a natural by-product of most game solving algorithms used in synthesis, we can easily take the general strategy as input to a process for finding a small implementation. This way, we have separated the problems of synthesizing *any* solution and obtaining a *good* one. While we might miss the smallest implementation this way, we do not introduce any additional computational hassle by combining the two goals in one step.

Theoretical work on computing small implementations from general strategies shows that approximating the size of a smallest finite-state machine within any polynomial quality function [14] from a general strategy is NP-hard.[1] This holds even if we do not have any input to the system. The fact that for scalable synthesis, we also have to be able to cope with symbolically represented general strategies[2], and also want a circuit rather than an explicit finite-state machine as result, does not quite make the problem easier in terms of complexity. As a consequence, current methods minimize the circuit size heuristically while only guaranteeing correctness. Experience however shows that with these techniques, the circuit sizes are often prohibitively large, calling for a better approach.

In this paper, we present a learning-based approach to com-

[1]For example, any algorithm that (1) outputs $\mathbf{false}$ if for some given general strategy and value of n, there exists no finite-state machine of size n that behaves in a way allowed by the general strategy, (2) outputs $\mathtt{true}$ if there exists such a finite-state machine of size at most n^{10}, and (3) outputs an arbitrarily result otherwise, solves an NP-hard problem.

[2]A symbolic representation of the synthesis game and the general strategy is crucial for scalability of reactive synthesis because the transformation of a specification into a game can lead to huge state spaces.

puting small circuits in symbolic reactive synthesis. In contrast to previous approaches, we do not exploit special properties of the data structure involved for symbolic reasoning (such as BDDs), but rather use computational learning of Boolean functions. This allows us to utilize the *non-determinism* of general strategies in a much more effective way. We learn a CNF (conjunctive normal form), DNF (disjunctive normal form), CDNF (conjunction of DNFs), or DCNF (disjunction of CNFs) representation of output and next-state bit valuations, which can immediately be translated into circuits. These circuits are not only typically smaller, but also more shallow than those of previous approaches, which allows running them at higher clock rates.

Our approach is not bound to one specific synthesis workflow, but supports any flow that computes a general strategy. For our experimental evaluation, we used two different BDD-based synthesis tools, namely RATSY [4], and UNBEAST [15]. RATSY provides us with general strategies stemming from the generalized reactivity(1) synthesis approach [7]. UNBEAST is a symbolic implementation [16] of a bounded synthesis variant [18]. In our experiments, we obtained circuit-size improvements of around one order of magnitude, when compared to the built-in approaches of these tools. The computation times are longer but still reasonable, thus allowing the new approach to be applied also to large problem instances.

This paper is structured as follows. In the next section, we give an overview of related work and provide experiences with previous approaches to circuit computation in synthesis. Then, we briefly discuss preliminaries and give literature pointers to the computation of general strategies in synthesis workflows. In Section IV, we describe our new learning-based approach, followed by an experimental evaluation in Section V. We conclude with a summary and ideas for future work.

II. Previous Approaches and Related Work

Computing an implementation in case of a realizable specification is the last step of every reactive synthesis approach. There are a few of these for which this last step is an easy one. In SMT-based bounded synthesis [18], the realizability of a specification by some finite-state machine with b states is encoded into a satisfiability modulo theory (SMT) formula, whose solution is an explicit implementation. Anti-chains-based bounded synthesis [17] uses anti-chains, rather than BDDs, as symbolic data structure during a game-solving process. It is then trivial to extract an implementation with as many states as there are elements in the final anti-chain.

Both approaches come at a price. SMT-based bounded synthesis is only reasonable if there exist small implementations and the number of input/output signals is not too high. Anti-chains-based bounded synthesis requires the specification to be a conjunction of relatively small sub-specifications in order to scale well. To counter these limitations, we are concerned with general circuit extraction approaches that start with symbolic general strategies. In the remainder of this section, we describe previous techniques for this task, state our experiences with them, and discuss techniques similar to our new approach.

Kukula and Shiple [23] described a simple technique to compute a circuit from a general strategy in BDD form. The main idea is to take the graph structure of the BDD and instantiate an 8-gate building block for all nodes to obtain an implementation. The resulting circuits have a very high depth (more than two times the number of state and input variables) and experience shows that they are often huge [6].

ANZU [21] uses a simple, cofactor-based approach [6] to compute a completely specified Boolean function for each output signal. The BDDs that represent these functions are then dumped into a network of multiplexers. Bloem et al. [6] also mention a simple but effective optimization. For each output, they remove unnecessary input variables by existential quantification. This method has also been implemented in RATSY [4]. To the best of our knowledge, this is the most effective circuit synthesis approach previously known, and it will be used as a baseline for comparison in Section V. We will subsequently refer to this method as the *cofactor approach*.

Baneres et al. [3] present a recursive paradigm for extracting completely specified Boolean functions from general strategies. Their approach is based on first computing the single output functions independently, without resubstitution. In a second stage they recursively resolve inconsistencies resulting from uncoordinated choices during the first stage. They also introduce a recursion-depth limit. If the limit is reached, their algorithm falls back to an arbitrary other relation-solving method. We reimplemented their approach within RATSY and applied it to its general strategies. Unfortunately, first experimental results were rather discouraging. Without any recursion limit, the approach timed out even for rather small benchmarks. However, using a recursion limit, we (almost) always hit the fall-back mechanism. The result of the fall-back mechanism is in almost all cases the same as if the recursive approach of [3] had not been used at all. Therefore, this approach does not provide any improvement concerning circuit size, but only increases computation time significantly. We believe that this is due to the fact that our general strategies are highly non-deterministic, and in particular have many vertices that [3] calls "non-don't-care extendable".

Another approach that we tried was implementing the Minato-Morreale algorithm for computing an *Irredundant Sum-of-Products* [24], [26]. It is a recursive procedure that takes a general strategy as an input and computes a Sum-of-Products form for a compatible completely specified function. The final result is *irredundant* in the sense that no single literal or cube can be deleted without changing the function. We use the recursive structure of the algorithm to build a multi-level Boolean circuit along the way. The resulting circuits are comparable in size to the ones obtained through the cofactor approach. Computation times, however, are significantly higher. To further improve these results, we also tried using a "cache". In each step, the algorithm first checks whether a function lying in the desired interval of functions has already been built as a circuit in previous steps. If so, this function (and the corresponding wire in the circuit) is reused. To keep the memory footprint of the cache small and to speed up the process of a cache look-up, we did not store the BDDs of the functions, but rather used a signature-based approach as in [25]. We only store the function's output for some random input vectors. These outputs are called a *signature*. Signatures have a very low memory footprint. When doing a look-up, we can use the signature to perform a fast pre-test. This pre-test may, however, create false positives. Thus, whenever the pre-test yields a positive result, we (recursively) reconstruct a BDD for the function in question from the structure of the circuit generated so far. We subsequently use this BDD to perform a sound comparison to check whether or not the function really lies within the desired interval. Experimental results have shown that, unfortunately, we get almost no cache hits. The hits we do get are mostly very small, almost trivial functions, consisting of only a handful of gates. Thus, the gain due to sharing is negligible. On the other hand, computation time rises significantly due to the many look-up checks that have to be performed. We also noticed that, when extended from completely specified functions [25] to intervals, the signature-based pre-test gives too many false positives to be of use.

Jiang et al. [19] presented a SAT-solver-based approach to compute functions from a general strategy. Their method is based on Craig interpolation [13], which is supported by many modern SAT solvers. Also here, preliminary experimental results suggest that this method cannot deal well with the high degree of non-determinism which is characteristic for general strategies in reactive synthesis. First tests produced circuits that were at least one order of magnitude larger than the ones obtained by the cofactor approach.

Our method for computing circuits is based on computational learning. It starts with simple candidate functions and refines them based on the counterexamples that are returned by a teacher oracle. Counterexample-guided refinements have already been used in program sketching [30] to synthesize missing program parts, and for program repair [22], [10]. Natively, these methods can only synthesize integer constants. Templates or user-provided generators containing unknown integers are used to synthesize more sophisticated program parts. In contrast, our method is able to compute circuits directly and without the help of the user.

Computational learning of Boolean function has many applications apart from implementation extraction. Becker et al. [2] use the concept to turn a quantified Boolean formula (QBF) solver into a tool for obtaining a compact representation of *all* solutions to a Boolean formula that may or may not have some quantifiers. While the representation types for the solutions are the same as in this work (CNF, DNF, and a conjunction of DNF formulas), Becker et al. focus on integrating a QBF solver into the classical learning algorithms for these representations. In this paper, on the other hand, we are not concerned with such low-level technical considerations, and simplify the details of our approach by taking BDDs as data structure for symbolic reasoning. This allows us to start right away with tackling the special properties of the implementation extraction domain, in particular how to obtain efficient circuits with multiple output signals, and how to make the best use of the non-determinism in the general strategies.

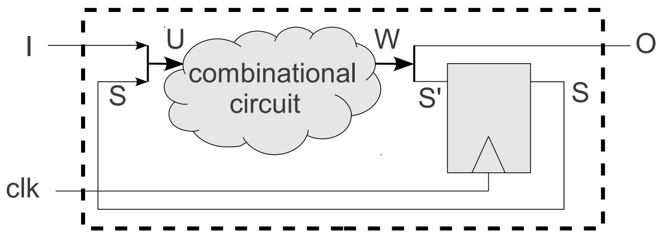

Fig. 1. Implementation of a general strategy.

The work in [11] addresses learning of Boolean functions over enlarging sets of variables, especially for loop invariant generation and assumption synthesis. The learning method is based on Bshouty's monotone theory [8], just like one of our algorithms. However, while [11] concentrates on efficiency in presence of an unbounded number of variables, we focus on utilizing non-determinism effectively to obtain small circuits.

III. Preliminaries

A. Basic Notation

Let V be a set of Boolean variables. To simplify notation, we treat subsets X of V and their characteristic functions interchangeably. Thus, a subset X of V induces a variable valuation $x : V \to \mathbb{B}$ by setting $x(v) = \mathbf{true}$ for some $v \in V$ if and only if $v \in X$, and likewise, a variable valuation $x : V \to \mathbb{B}$ induces a subset X of V by choosing $X = \{v \in V \mid x(v) = \mathbf{true}\}$. A model of a Boolean formula is a variable valuation that satisfies the Boolean formula.

B. General strategies

A *general strategy* is a tuple $\mathcal{S} = (S, I, O, s_0, \delta)$, where S is a set of *state bits*, I is a set of *input bits*, O is a set of *output bits*, $s_0 \subseteq S$ is the *initial state* and $\delta \subseteq 2^S \times 2^I \times 2^S \times 2^O$ is the *transition relation*. To separate the two occurrences of state bit sets in the transition relation, we will henceforth write S' for their second copy. In this paper, we are concerned with *extracting* a small circuit for this strategy, i.e., a net with input bits $U = S \cup I$ and output bits $W = S' \cup O$ such that for every input $(s, i) \in 2^S \times 2^I$, if s is a *reachable state* and the circuit outputs some $(s', o) \in 2^{S'} \times 2^O$ for this input, then $(s, i, s', o) \in \delta$. We call such a circuit a *specialization* of $\mathcal{S}$ as it exhibits only behavior that is allowed by the strategy, and chooses one particular output/next state combination whenever more than one is possible. A state s is considered to be reachable if there exists some sequence $s_0 \xrightarrow[i_0]{o_0} s_1 \xrightarrow[i_1]{o_1} \ldots \xrightarrow[i_n]{o_n} s_n$ with $s_n = s$ such that for all $j \in \{0, \ldots, n\}$, (s_{j+1}, o_j) is the output of the circuit for the input bit valuation (s_j, i_j). Thus, we assume that some flip-flops feed back the output state bits as input to the net in the next computation cycle. This implementation of $\mathcal{S}$ in a circuit is illustrated in Fig. 1. For the scope of this paper, the sizes and depths of combinatorial circuits are considered at the gate level.

General strategies as defined above are computed in many modern synthesis workflows as a by-product. Normally, not all possible implementations of a specification are specializations of the general strategy computed, but rather some unfavorable ones have already been filtered out. For example, if we used a mutual-exclusion protocol specification as input to a synthesis tool, we would want that the general strategy computed ensures that grants are given quickly in order not to let the requester wait unnecessarily. For the synthesis approaches considered in this paper, this *good reactivity* is ensured: in generalized reactivity(1) synthesis [7], [27], the general strategy is built such that transitions that help towards the fulfillment of liveness objectives are preferred, whereas in BDD-based bounded synthesis [16], a strict bound on delays in such a situation is imposed. Thus, in both cases, for synthesizing implementations of high quality, we can restrict our attention to circuit size and depth for the scope of this paper. For details on the computation of general strategies in these two synthesis approaches, the interested reader is referred to [16], [7], [27].

C. Binary Decision Diagrams

To handle Boolean functions and general strategies symbolically, we use *reduced ordered binary decision diagrams* (BDDs) [9], which represent characteristic functions $f : 2^V \rightarrow \mathbb{B}$ for some finite set of variables V. Let f and f' be two BDDs and $V' \subseteq V$ be a set of variables. We denote the conjunction, disjunction, negation, existential quantification, and universal quantification of BDDs as $f \wedge f'$, $f \vee f'$, $\neg f$, $\exists V' . f$ and $\forall V' . f$, respectively. To represent the transition relation of a general strategy $\mathcal{S} = (S, I, O, s_0, \delta)$, we take $V = S \uplus I \uplus S' \uplus O$ and build the characteristic function of δ by setting $f_\delta(X) = \mathbf{true}$ for some $X \subseteq V$ iff $(X \cap S, X \cap I, X \cap S', X \cap O) \in \delta$.

IV. Learning Small Circuits

In this section, we present the core contribution of this work: Given a general strategy $\mathcal{S} = (S, I, O, s_0, \delta)$, we show how to compute a combinatorial circuit with input bits $U = S \cup I$ and output bits $W = S' \cup O$ that implements the strategy as illustrated in Fig. 1. We break down this problem into obtaining $|W|$ circuits with a single output bit, each getting $|U|$ bits as input. First, we describe this decomposition process. Then, in Section IV-B, we discuss how a circuit for a single output bit can be computed using computational learning.

A. Decomposition

One reason for computing $|W|$ one-output-bit circuits, as we do in this paper, is to increase the freedom in the circuits due to unreachable states. If a circuit for one output bit has been found, we can recompute the set of states reachable by any specialization that uses this circuit. Typically, this reachable state set shrinks with every additional circuit, which allows our specialization to ignore more and more input valuations $X \subseteq U$ as the algorithm proceeds. The following algorithm describes the overall process:

```
1:  procedure OBTAINCIRCUIT($S, I, O, s_0, \delta$)
2:      $A := \delta$
3:      for $v \in W$ do
4:          $r :=$ states reachable from $s_0$ under $A$ as BDD
5:          $c := r \wedge (\neg(\exists W . (A \wedge v)) \vee \neg(\exists W . (A \wedge \neg v)))$
6:          if optimize then
7:              $(f, c) :=$ MINVARS($A, v, c$)
8:          else
9:              $f := \exists W . (A \wedge v)$
10:         end if
11:         $g :=$ LEARN($U, f, c$)
12:         Take $g$ as output circuit for $v$
13:         $A := A \wedge (\neg v \oplus g)$
14:     end for
15: end procedure
```

The algorithm iterates over all outputs $v \in W$ of the combinatorial circuit we wish to build. In every iteration, we first compute which states are reachable in any specialization with the circuits computed so far. Next, we compute the care set, i.e., the set of input variable valuations $X \subseteq U$ for which the output matters. There are two reasons why a valuation might not be in the care set: (1) the state is not reachable, and (2) both values of the output bit v are allowed. The computation of c in line 5 reflects these cases. Ignore the optional optimization in line 7 for a moment. Line 9 now computes the target function, and line 11 uses a black-box function LEARN to obtain a corresponding circuit using computational learning. We assume that LEARN returns a BDD representation of the one-output-bit circuit that resembles f on variable valuations $X \subseteq V$ for which $c(X) = \mathbf{true}$ holds. We describe two variants of a function LEARN in the next subsection.

After a circuit for one output $v \in W$ has been obtained, we need to update the general strategy to only allow output variable valuations $X \subseteq W$ that are still possible when using the circuits we already have. This happens in line 13.

The optimization in line 7 minimizes the number of input bits $v' \in U$ on which the output bit v may depend. This idea has been introduced in [6] and can be implemented as follows.

```
1:  procedure MINVARS($A, v, c$)
2:      $m_0 := \neg((\exists W . (A \wedge v)) \vee \neg c)$
3:      $m_1 := \neg((\exists W . (A \wedge \neg v)) \vee \neg c)$
4:      for $v' \in U$ do
5:          $(m'_0, m'_1) := (\exists v' . m_0, \exists v' . m_1)$
6:          if $m'_0 \wedge m'_1 =$ false then
7:              $(m_0, m_1) := (m'_0, m'_1)$
8:          end if
9:      end for
10:     return $(m_1, m_0 \vee m_1)$
11: end procedure
```

The lines 2 and 3 compute the input variable valuations for which the circuit to be learned has to output $\mathbf{false}$ and $\mathbf{true}$, respectively. If an existential quantification of an input variable v' does not make the two regions m_0 and m_1 overlap, then this means that the function can still be implemented without taking the input bit v' into account. Otherwise, v' is crucial for distinguishing input variable valuations for which the circuit has to output $\mathbf{true}$ from those where it has to output $\mathbf{false}$. In this case, v' cannot be disregarded. The check is done in line 6. After all unnecessary input bits have been discarded using existential quantification, the target function f and the care set c are recomputed and returned. We also use the above algorithm in a heuristic to find a good ordering for the output bits in line 3 of OBTAINCIRCUIT: output bits v for which f depends on fewer variables are considered simpler to handle, and are thus processed first.

B. Learning circuits with a single output bit

When computing a small one-output-bit circuit from a problem instance (V, f, c), our aim is to utilize *don't care* input bit valuations (i.e., $X \subseteq V$ with $c(X) = \mathbf{false}$) as effectively as possible, while obtaining circuits with appealing properties, such as low depth and few gates. We describe here how to apply the concept of *computational learning* to obtain small and shallow circuits. We decompose f into a Boolean formula that only needs to be correct on the *care set*, i.e., the input bit valuations that c maps to $\mathbf{true}$. The Boolean formula is built in an incremental fashion, i.e., we start with a small formula that we iteratively refine until it is correct with respect to the care set. After learning the formula, it can easily be translated to a circuit by using only AND, OR, and NOT gates.

We describe two variants of the learning process here, one for which the target Boolean formula is in CNF form, and one for which it is in CDNF form, i.e., a conjunction of disjunctive normal form Boolean formulas. Both variants are instances of Angluin-style [1] learning algorithms, in which the learning process proceeds by performing *queries* of various types to some teacher oracle. In our context, queries reduce to operations on BDDs. We can use the CNF and CDNF algorithms to also learn DNF and DCNF formulas: we simply dualize f and the output formula.

1) Learning CNFs: A formula in conjunctive normal form is a conjunction of clauses, each being a disjunction of literals. Given a one-bit output circuit problem (V, f, c), a clause C is *sound* in a CNF for (V, f, c) iff $\neg C(X)$ implies $\neg f(X) \vee \neg c(X)$ for all $X \subseteq V$. That is, a sound clause only evaluates to $\mathbf{false}$ if the variable valuation can be mapped to $\mathbf{false}$. On the other hand, if a CNF formula of sound clauses maps every $X \subseteq V$ with $c(X) \wedge \neg f(X)$ to $\mathbf{false}$, then it has enough clauses to be a valid solution. Using BDDs, we can easily check if a CNF formula has enough clauses or if a clause is sound. The following algorithm iteratively searches for sound clauses until we have enough of them:

```
1:  procedure LEARNCNF($V, f, c$)
2:    $r := \mathbf{true}$
3:    while $r \wedge (\neg f) \wedge c \neq \mathbf{false}$ do
4:      $b :=$ pick some variable valuation in $r \wedge (\neg f) \wedge c$
5:      $V' := V$
6:      $C := \bigvee_{v' \in V'} v' \oplus b(v')$
7:      for $v \in V$ do
8:        $C' := \bigvee_{v' \in (V' \setminus \{v\})} v' \oplus b(v')$
9:        if $((\neg C') \wedge c \wedge f) = \mathbf{false}$ then
10:         $(C, V') := (C', V' \setminus \{v\})$
11:       end if
12:     end for
13:     $r := r \wedge C$
14:   end while
15:   return $r$
16: end procedure
```

The variable r stores the candidate CNF formula as BDD. In practice, we store the BDD together with the corresponding CNF formula to avoid reconstructing the CNF at the end. In line 3, we check if we have found enough sound clauses already. If this is not the case, we pick some variable valuation b that witnesses this fact. We use b to derive a new sound clause in line 6. In the lines 7 to 12 we shorten it as much as possible while retaining its soundness. This way, we keep both the length and number of the clauses in the formula small.

Let $\mathcal{X} = \{X \subseteq V \mid c(X) \wedge \neg f(X) \wedge r(X)\}$ be the set of variable valuations that r must, be but does not yet, map to $\mathbf{false}$. LEARNCNF terminates because it quits on $|\mathcal{X}| = 0$, and $|\mathcal{X}|$ decreases in every loop iteration. It is correct because it adds only sound clauses to r, and enough to have $|\mathcal{X}| = 0$.

After LEARNCNF is finished, we remove all clauses from r for which removing leaves the learned function consistent with f on c. This reduces the size of the learned function.

Example 1. *We illustrate* LEARNCNF *on the learning problem* $(\{v_1, v_2\}, f, c)$*, defined in the left part of the following truth table.*

v_1	v_2	f	c	r^1	$C^1{=}\neg v_1 \vee v_2$	$C'^1{=}r^2{=}v_2$
false	false	false	false	true	true	false
false	true	false	false	true	true	true
true	false	false	true	true	false	false
true	true	true	true	true	true	true

We write a^i to denote variable a in iteration i of LEARNCNF*. The first candidate is $r^1 = \mathbf{true}$. An input valuation b^1 satisfying $r_1 \wedge (\neg f) \wedge c$ is $b^1(v_1) = \mathbf{true}$, $b^1(v_2) = \mathbf{false}$; the corresponding clause is $C^1 = \neg v_1 \vee v_2$. Next, C^1 is simplified by removing literals as long as soundness is preserved. $C'^1 = v_2$ renders $(\neg C'^1) \wedge c \wedge f$ unsatisfiable, so C'^1 is still sound. Since the empty clause is not sound, r is refined to $r^2 = C'^1 = v_2$. Now, $r_2 \wedge (\neg f) \wedge c$ is unsatisfiable, i.e, there is no more input valuation for which the circuit must, but does not yet, output* $\mathbf{false}$*. Hence, the circuit outputting $r_2 = v_2$ is a solution to the learning problem $(\{v_1, v_2\}, f, c)$.*

2) Learning CDNFs: The CNF learning approach above computes two-level combinatorial circuits. While these are shallow, there are many functions for which we need more levels in order to obtain a circuit with few gates. Here, we use Bshouty's learning algorithm, based on his *monotone theory* [8] as a basis for learning a CDNF representation of the target function, which leads to three levels in the computed circuits. This is still a low number, and thus allows to drive the resulting circuit with high clock rates, but offers a better chance for minimizing the number of gates. We start with an explanation of the necessary theory, and then describe how it can be applied when f and c are given in BDD form.

Let V be some set of variables and X, Y, and Z be subsets of V. We write $X \subseteq_Z Y$ if and only if $(X \cap (V \setminus Z)) \subseteq$

$(Y \cap (V \setminus Z))$ and $(X \cap Z) \supseteq (Y \cap Z)$. A Boolean function $f : 2^V \rightarrow \mathbb{B}$ is Z-monotone if for all $X, Y \subseteq V$, if $X \subseteq_Z Y$ and $f(X) = \mathbf{true}$, then $f(Y) = \mathbf{true}$.

Bshouty's learning algorithm represents the function-to-learn as a conjunction of Boolean formulas in disjunctive normal form. Each of this DNF formulas is Z-monotone for some $Z \subseteq V$. From the computational learning theory perspective [1], Bshouty's CDNF learning algorithm employs two types of *queries*: membership queries and equivalence queries. In this paper, we extend its idea by modifying the algorithm in order to make use of don't care variable valuations. Performing a membership query in this context then means that for a variable valuation X, we check if $f(X) \vee \neg c(X) = \mathbf{true}$. Performing an equivalence query means checking if the variable valuations that c maps to $\mathbf{true}$ are models of the candidate CDNF formula if and only if they are mapped to $\mathbf{true}$ by f. The following code describes the learning process:

```
1: procedure LEARNCDNF($V, f, c$)
2:   $P = \emptyset$
3:   while $\mathbf{true}$ do
4:     $g := \bigwedge_{(h,d)\in P} h$
5:     if $(g \wedge (\neg f) \wedge c) \neq \mathbf{false}$ then
6:       $b :=$ pick some variable valuation in $g \wedge (\neg f) \wedge c$
7:       $P := P \cup \{(\mathbf{false}, b)\}$
8:     else if $((\neg g) \wedge f \wedge c) \neq \mathbf{false}$ then
9:       $b :=$ pick some variable valuation in $(\neg g) \wedge f \wedge c$
10:      for $\{(h,d) \in P \mid b \not\models h\}$ do
11:        $b' := b$
12:        for $v \in \{v' \in V \mid b'(v') \neq d(v')\}$ do
13:          $b'' := b'$
14:          $b''(v) := \neg b''(v)$
15:          if $f(b'') \vee \neg c(b'')$ then
16:            $b' := b''$
17:          end if
18:        end for
19:        $h' := h \vee \bigwedge_{v\in V, b'(v)\neq d(v)} \begin{cases} v & \text{if } b'(v) = \mathbf{true} \\ \neg v & \text{else} \end{cases}$
20:        $P := P \setminus \{(h,d)\} \cup \{(h',d)\}$
21:      end for
22:    else
23:      return $g$
24:    end if
25:  end while
26: end procedure
```

The algorithm maintains a candidate CDNF formula in P. Every DNF formula is stored together with its monotonicity base. Line 5 checks for *false-positives*, and line 8 for *false-negative* variable valuations. False-positives are valuations $X \subseteq V$ with $c(X) = \mathbf{true}$ that are models of the candidate formula, but for which $f(X) = \mathbf{false}$. Likewise, false-negatives are valuations X for which $c(X) = f(X) = \mathbf{true}$, but X is not a model of the candidate formula. Both witness the misclassification of a variable valuation. Whenever we find a false-positive X, we add a DNF that is kept X-monotone during the run of the algorithm. For a false-negative X, we update all DNFs with a cube (a conjunction of literals) that ensures that X becomes a model of the DNF. For this, we first make the false-negative as similar to the monotonicity base as possible without changing the fact that the circuit is allowed to output $\mathbf{true}$ for this valuation. Then we add a cube that contains only literals that point away from the monotonicity base. This way, the DNF formula remains d-monotonous with respect to its base d, but stays small at the same time. For more details on Bshouty's CDNF learning algorithm, the reader is referred to [28].

The algorithm terminates because in every iteration, either a false-positive or a false-negative is resolved, and the maximum number of potential misclassifications is finite. Note that resolving a false-positive will typically add new false-negatives because the newly added DNF is initially empty, i.e. **false**. However, resolving a false-positive X eliminates it once and for all. The reason is that the new DNF that is added is kept X-monotone. It is extended with cubes containing literals that point away from the monotonicity base only. Hence, X can never become a model of that DNF and $g(X)$ will always remain **false**. The algorithm is correct because upon termination, there are no more misclassifications.

In a post-processing step, we simplify the formula produced by LEARNCDNF. We remove all DNFs, cubes and literals for which removing leaves the CDNF consistent with f on c.

Example 2. *We apply* LEARNCDNF *to the learning problem* $(\{v_1, v_2\}, f, c)$ *from Example 1, defined by the following table.*

v_1	v_2	f	c	g^1	g^2	g^3
false	**false**	**false**	**false**	**true**	**false**	**false**
false	**true**	**false**	**false**	**true**	**false**	**true**
true	**false**	**false**	**true**	**true**	**false**	**false**
true	**true**	**true**	**true**	**true**	**false**	**true**

We have that $P^1 = \emptyset$, *so* $g^1 = \mathbf{true}$. *Since* $g^1 \wedge (\neg f) \wedge c$ *is satisfiable, there exists a false-positive* b^1 *with* $b^1(v_1) = \mathbf{true}$ *and* $b^1(v_2) = \mathbf{false}$. *To resolve it, a new DNF formula, initialized to* **false**, *is added to* P *together with its monotonicity base* b^1. *In the next iteration,* g^2 *is* **false**. *Consequently,* $g^2 \wedge (\neg f) \wedge c$ *is unsatisfiable, so there exists no false-positive. However, there is a false-negative* b^2, *defined as* $b^2(v_1) = b^2(v_2) = \mathbf{true}$. *To resolve it, the DNF formula* $h^2 = \mathbf{false}$ *is weakened with an additional cube. To get a small cube,* b^2 *is modified to match the monotonicity base* b^1 *as well as possible.* b^1 *and* b^2 *differ only in* v_2, *so this value is flipped to obtain* b''^2 *as* $b''^2(v_1) = \mathbf{true}$, $b''^2(v_2) = \mathbf{false}$. *However, for* b''^2 *it is not allowed to output* $\mathbf{true}$, *because* $f(b''^2) \vee \neg c(b''^2)$ *is* **false**, *so the flip is retracted. Since* b^1 *and* b^2 *differ only in* v_2 *and* $b'^2(v_2) = \mathbf{true}$, *the empty DNF in* h^2 *is extended to* v_2. *Since* P^3 *now contains only the DNF* v_2, g^3 *is* v_2 *in the next iteration. Using* g^3, *there is neither a false-positive nor a false-negative, so* $g^3 = v_2$ *is reported as solution.*

V. EXPERIMENTAL RESULTS

In this section, we first briefly describe our implementation and experimental setup. Then we present our experimental results with the synthesis tools RATSY and UNBEAST.

A. Implementation and Experimental Setup

We implemented the learning algorithms in a circuit extraction tool that can be run in 9 different modes. For every mode, the following table summarizes the learning method, whether the basic method is complemented, whether variables are minimized using MINVARS, and the output format. We say that a method is complemented if we negate the function to learn before applying the learning algorithm. By duality, we can then use the CNF learning algorithm for obtaining a DNF result, and the CDNF learner to get a DCNF (disjunction of DNFs) output. Mode 8 is special: for every output, it applies the learning methods of modes 1, 3, 5, and 7, and picks the smallest implementation.

Mode	Learning Method	Compl.	MINVARS	Outcome
0	Bshouty	no	no	CDNF
1	Bshouty	no	yes	CDNF
2	Bshouty	yes	no	DCNF
3	Bshouty	yes	yes	DCNF
4	CNF Refinement	no	no	CNF
5	CNF Refinement	no	yes	CNF
6	CNF Refinement	yes	no	DNF
7	CNF Refinement	yes	yes	DNF
8	both	both	yes	all

As input, our tool takes a file containing a general strategy $\mathcal{S}$. As output, it produces a circuit in SMV or BLIF format. All symbolic computations are done using BDDs. We use CUDD [31] as BDD library with dynamic variable reordering enabled. Our tool is written in C++. The implementation as well as the input files and all scripts to reproduce the experimental results are available for download[3].

In our experiments, we run RATSY and UNBEAST to synthesize circuits for several specifications. We also export the general strategies and synthesize circuits with our learning-based extractor. ABC[4] 70930 is used to map circuits to standard cells. The gates in our standard cell library have a fan-in of at most 4 (which can increase the depth of the circuit). All circuits produced by our extractor have been successfully model-checked against their original specifications, using NUSMV 2.5.4 [12] with bounded model checking.

All experiments were performed on an Intel Xeon E5430 CPU with 4 cores running at 2.66 GHz, 64 GB of RAM, and a 64 bit Linux. All programs run single-threaded, so only one core was actually used. The maximum memory consumption of our new circuit extractor was 3.3 GB in our experiments.

B. Experiments with RATSY

RATSY's built-in circuit extractor uses the cofactor approach sketched in Section II. Table I compares this technique with our new approach. Due to space constraints, the comparison includes mode 1 and 7 (see Section V-A) only. These modes were chosen because they achieved good results. Results for the other modes can be found in Table III in the appendix. We evaluate the methods on three parametrized specifications. The first one defines an arbiter for ARM's AMBA AHB bus [6]. It is parametrized with the number of masters it can handle. These specifications are denoted as Ai, where i is the number of masters. The second specification, denoted A'i, is a less optimized variant of the former. It is described in [5]. The third specification is denoted by Gi and defines a generalized buffer [6] connecting i senders to two receivers. The bit numbers of the general strategies range from $|U| = 24$ and $|W| = 12$ (for G2) to $|U| = 129$ and $|W| = 63$ (for A15). Table III contains the exact numbers for every benchmark.

Column 1 in Table I lists the time needed by RATSY to turn the general strategy into a circuit. The size of the resulting circuit in terms of the total number of standard cells (gates plus flip-flops, but the flip-flops are typically negligible) is given in column 2. Column 3 contains the corresponding depth of the combinational circuit. The columns 4 to 7 show the results for circuit extraction with our extractor in mode 1. Column 4 gives the circuit extraction time. The columns 5 and 6 list the size and depth of the resulting circuits. Column 7 contains the circuit size improvement factor due to our method. The columns 8 to 11 show the same information for mode 7. Computation time entries preceded by a ">" indicate time-outs. A "-" stands for missing data due to a time-out. The suffix k stands for a multiplication of the respective number by 1 000. The table does not contain entries for A'i with $i > 5$ because for these specifications, RATSY did not finish within 100 000 seconds. The sums and averages in the last two lines only take into account benchmarks for which all methods terminate.

C. Experiments with UNBEAST

UNBEAST is a bounded synthesis tool that applies the Kukula/Shiple method (see Section II) for circuit extraction. The comparison with our new approach is summarized in Table II for the modes 1 and 7 of our new circuit extraction tool. Results for the other modes can be found in the appendix. For the comparison, we use the (realizable) LILY benchmarks [20], which are denoted as Li. For some of these benchmarks we created several variants. They are named Li-j, where j is a size parameter. We also use a specification for a load balancer [16] (the final version), which is parameterized by the number i of clients. These specifications are referred to as Bi. Table II is organized just like Table I. A circuit depth of 0 means that the combinatorial circuit could be implemented without gates, i.e., all outputs are either equal to an input or to the constants **true** or **false**. The bit numbers of the general strategies range from $|U| = 9$ and $|W| = 9$ (on L13) to $|U| = 93$ and $|W| = 92$ (on B5). The individual bit numbers can be found in Table IV in the appendix.

D. Discussion

Fig. 2 shows a scatter plot comparing the size of the circuits produced by our new extractor in mode 1 against those produced by RATSY or UNBEAST. On most benchmarks run through RATSY, an improvement of around one order of

[3]http://www.iaik.tugraz.at/content/research/design_verification/others/
[4]http://www.eecs.berkeley.edu/~alanmi/abc/

TABLE I
COMPARISON WITH RATSY.

Col.	1	2	3	4	5	6	7	8	9	10	11
	RATSY			Mode 1				Mode 7			
	time: code generation	circuit size	circuit depth	time	circuit size	circuit depth	shrinking factor	time	circuit size	circuit depth	shrinking factor
	[sec]	[#cells]	[-]	[sec]	[#cells]	[-]	[-]	[sec]	[#cells]	[-]	[-]
`A2`	1.2	733	18	1.4	269	5	2.7	1.3	260	5	2.8
`A3`	19	5.6 k	25	19	355	5	16	21	565	5	9.9
`A4`	67	10 k	33	461	1.8 k	7	5.5	480	3.7 k	8	2.7
`A5`	135	5.7 k	25	221	663	5	8.5	239	1.2 k	6	4.9
`A6`	204	8.2 k	26	233	819	6	10	251	1.3 k	6	6.4
`A7`	840	16 k	36	452	1.0 k	6	16	488	1.6 k	6	10
`A8`	6.6 k	135 k	46	>100 k	–	–	–	>100 k	–	–	–
`A9`	1.8 k	22 k	41	4.2 k	1.3 k	6	16	6.7 k	2.4 k	6	9.3
`A10`	3.0 k	19 k	44	8.9 k	1.5 k	6	12	7.0 k	2.4 k	7	7.6
`A11`	4.0 k	39 k	46	7.4 k	1.8 k	7	21	7.5 k	3.1 k	6	13
`A12`	10 k	38 k	50	16 k	2.0 k	6	19	37 k	3.1 k	7	12
`A13`	15 k	65 k	54	45 k	2.4 k	6	28	31 k	3.5 k	7	19
`A14`	15 k	47 k	42	36 k	2.6 k	7	18	83 k	3.6 k	7	13
`A15`	19 k	70 k	59	75 k	3.0 k	7	24	99 k	4.0 k	7	18
`A'2`	1.7	1.0 k	16	1.9	224	5	4.6	2.4	306	5	3.4
`A'3`	169	17 k	26	77	465	5	38	103	781	6	22
`A'4`	914	28 k	32	984	677	5	41	5.1 k	5.2 k	8	5.3
`A'5`	9.7 k	101 k	37	18 k	893	5	113	16	1.9 k	6	54
`G2`	0.1	249	11	0.1	53	3	4.7	0.1	65	3	3.8
`G3`	0.2	394	12	0.3	123	4	3.2	0.3	174	4	2.3
`G4`	0.5	721	18	0.5	119	3	6.1	0.5	262	5	2.8
`G5`	1.2	1.8 k	18	2.3	444	6	4.2	1.9	674	6	2.7
`G6`	7.7	6.2 k	22	6.8	1.1 k	6	5.8	2.1	828	6	7.5
`G7`	3.3	3.5 k	23	11	1.7 k	7	2.0	15	3.4 k	7	1.0
`G8`	8.1	5.8 k	26	2.6	278	4	21	7.4	5.5 k	8	1.1
`G9`	5.9	3.4 k	25	430	5.8 k	9	0.6	74	10 k	8	0.3
`G10`	14	6.5 k	29	8.0 k	22 k	9	0.3	57	13 k	9	0.5
`G11`	18	9.8 k	33	71 k	47 k	10	0.2	157	28 k	9	0.4
`G12`	35	14 k	34	>100 k	–	–	–	711	62 k	10	0.2
sum	80 k	531 k	827	292 k	100 k	160	5.3	294 k	101 k	173	5.3
avg.	3.0 k	20 k	31	11 k	3.7 k	5.9	16	11 k	3.7 k	6.4	8.7

TABLE II
COMPARISON WITH UNBEAST.

Col.	1	2	3	4	5	6	7	8	9	10	11
	UNBEAST			Mode 1				Mode 7			
	time: code generation	circuit size	circuit depth	time	circuit size	circuit depth	shrinking factor	time	circuit size	circuit depth	shrinking factor
	[sec]	[#cells]	[-]	[sec]	[#cells]	[-]	[-]	[sec]	[#cells]	[-]	[-]
`L3`	0.1	845	24	0.1	19	1	44	0.1	19	1	44
`L3-6`	0.3	6.3 k	86	15	2.6 k	6	2.4	0.9	79	3	80
`L5`	0.1	908	23	0.1	24	1	38	0.1	27	2	34
`L6`	0.1	2.5 k	40	0.1	38	3	65	0.1	42	2	59
`L7`	0.1	551	21	0.1	22	1	25	0.1	25	2	22
`L8`	0.1	113	11	0.1	10	0	11	0.1	10	0	11
`L9`	0.1	450	15	0.1	16	1	28	0.1	16	1	28
`L10`	0.1	1.4 k	35	0.1	22	0	64	0.1	22	0	64
`L12`	0.1	524	21	0.1	15	0	35	0.1	15	0	35
`L13`	0.1	28	7	0.1	9	1	3.1	0.1	9	1	3.1
`L14`	0.1	3.4 k	31	0.1	17	1	203	0.1	17	1	203
`L15`	0.1	277	15	0.1	16	2	17	0.1	16	2	17
`L16`	0.1	830	21	0.1	29	3	29	0.1	37	3	22
`L17`	0.1	1.5 k	37	0.1	21	1	71	0.1	25	2	60
`L18`	0.3	23 k	73	0.2	79	4	286	0.2	73	4	309
`L19`	0.1	531	26	0.1	21	1	25	0.1	21	1	25
`L20`	0.1	4.7 k	52	0.5	120	4	39	0.3	84	4	56
`L21`	0.4	13 k	70	2.6	114	5	112	3.6	205	5	62
`L22`	0.1	1.1 k	43	0.1	107	4	11	0.1	43	3	26
`L22-5`	0.1	950	35	0.1	104	4	9.1	0.1	40	3	24
`L22-6`	0.1	995	40	0.1	107	4	9.3	0.1	43	3	23
`L22-7`	0.1	1.2 k	43	0.1	107	4	11	0.1	42	3	28
`L22-8`	0.1	1.0 k	35	0.1	106	4	9.5	0.1	41	3	24
`L22-9`	0.1	910	35	0.1	105	4	8.7	0.1	41	3	22
`L23`	0.1	354	18	0.1	16	1	22	0.1	16	1	22
`B2`	0.1	3.5 k	51	0.1	35	2	99	0.1	43	2	81
`B3`	0.7	27 k	79	1.9	437	6	62	0.8	131	5	208
`B4`	5.5	171 k	151	1.1 k	6.5 k	9	26	3.5 k	23 k	9	7.4
`B5`	650	816 k	189	99 k	17 k	9	49	>100 k	–	–	–
sum	7.5	268 k	1.1 k	1.1 k	11 k	77	25	3.5 k	24 k	69	11
avg.	0.3	9.6 k	41	39	387	2.8	49	125	872	2.5	57

magnitude can be observed, with a tendency to greater improvements for larger circuits. For UNBEAST the improvement reaches almost two orders of magnitude on many benchmarks. Mode 8 (only included in the appendix) produces even smaller circuits, but at the costs of higher running times. In contrast to the many methods we have already tried (cf. Section II), these results are very promising. Table I and II also show a significant improvement of the circuit depths. For RATSY, the average is reduced from 31 to 6 after mapping to standard cells. For UNBEAST there is an average reduction from 41 to less than 3 in our experiments.

The downside of our new method is that computation times grow. For the RATSY benchmarks `A`i and `A'`i, the slow-down factor is mostly below 4. Only for the benchmarks `G`i, the circuit extraction times grow much faster with increasing i than with the cofactor approach implemented in RATSY. Also compared to the Kukula/Shiple method of UNBEAST, a considerable slow-down can be observed on the `B`i benchmarks. On the `G`i benchmarks, mode 7 appears to scale better than mode 1, but at the costs of producing larger circuits.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new approach for extracting circuits from general strategies that improves the circuit size by roughly one to two orders of magnitude, compared to previous techniques. Moreover, it reduces the depths of the resulting circuits, allowing them to be run at higher clock rates. General strategies are typical intermediate results of reactive synthesis workflows, and thus our contribution significantly increases the quality of the circuits computed in reactive synthesis.

During our quest for effective and efficient circuit extraction techniques that go beyond the cofactor approach of Bloem at al. [6], we tried a large number of older techniques from literature. Our experience shows that exploiting the large degree of non-determinism that we have in general strategies is not a simple task and techniques not geared towards such cases do not perform well. Our approach on the other hand is built around the idea of computational learning of Boolean

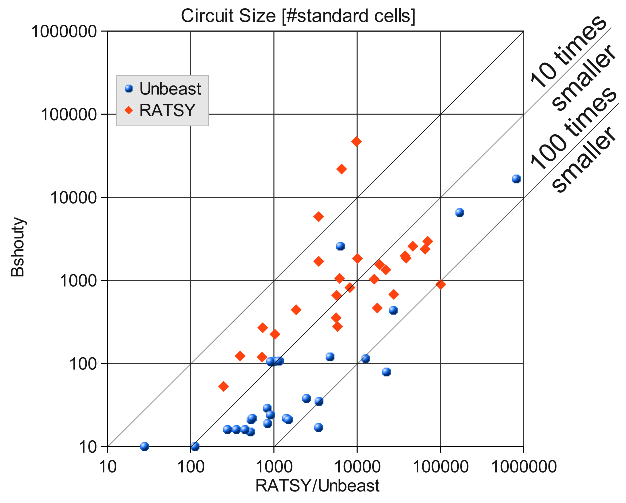

Fig. 2. Circuit size improvement.

functions. It allows exploiting non-determinism in a natural way, which is the reason for the effectiveness of our approach.

In the future, we plan to implement more learning algorithms, refine them with heuristics for selecting better false-positives, false-negatives, and variable orderings, and to implement the algorithms also with SAT-solvers instead of BDDs. Furthermore, we want to compare the produced circuits with manual implementations to see how much potential for optimizations still exists, and to get new inspirations.

References

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[2] M. Lewis B. Becker, R. Ehlers and P. Marin. ALLQBF solving by computational learning. In *Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 370–384. Springer, 2012.

[3] D. Bañeres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve Boolean relations. In *Design Automation Conference (DAC'04)*, pages 416–421. ACM, 2004.

[4] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY - A new requirements analysis tool with synthesis. In *Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

[5] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *Design, Automation and Test in Europe (DATE'07)*, pages 1188–1193, 2007.

[6] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007.

[7] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[8] N. H. Bshouty. Exact learning Boolean functions via the monotone theory. *Electronic Colloquium on Computational Complexity (ECCC)*, 2(8), 1995.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[10] K.-H. Chang, I. L. Markov, and V. Bertacco. Fixing design errors with counterexamples and resynthesis. In *Asia and South Pacific Design Automation Conference (ASP-DAC'07)*, pages 944–949. IEEE, 2007.

[11] Y.-F. Chen and B.-Y. Wang. Learning Boolean functions incrementally. In *Computer Aided Verification (CAV'12)*, volume 7358 of *LNCS*, pages 55–70. Springer, 2012.

[12] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000.

[13] W. Craig. Three uses of the Herbrand-Gentzen Theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[14] R. Ehlers. Short witnesses and accepting lassos in ω-automata. In *Language and Automata Theory and Applications (LATA'10)*, volume 6031 of *LNCS*, pages 261–272. Springer, 2010.

[15] R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.

[16] R. Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012.

[17] E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011.

[18] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 2012. 10.1007/s10009-012-0228-z.

[19] J. R. Jiang, H. Lin, and W. Hung. Interpolating functions from large Boolean relations. In *International Conference on Computer-Aided Design (ICCAD'09)*, pages 779–784. IEEE, 2009.

[20] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design (FMCAD'06)*, pages 117–124. IEEE, 2006.

[21] B. Jobstmann, S. J. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 258–262, 2007.

[22] R. Koenighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer Aided Design (FMCAD'11)*, pages 91–100. IEEE, 2011.

[23] J. H. Kukula and T. R. Shiple. Building circuits from relations. In *Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 113–123. Springer, 2000.

[24] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Synthesis and Simulation Meeting and International Interchange (SASIMI'92)*, pages 64–73, 1992.

[25] A. Mishchenko, S. Chatterjee, and R. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.

[26] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Transactions on Computers*, 100(6):504–509, 1970.

[27] M. Schlaipfer, G. Hofferek, and R. Bloem. Generalized reactivity(1) synthesis without a monolithic strategy. In *Haifa Verification Conference (HVC'11)*, 2011. To appear.

[28] R. H. Sloan, B. Szörényi, and G. Turán. Learning Boolean functions with queries. In *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Cambridge University Press, 2010.

[29] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for LTL games. In *Formal Methods in Computer-Aided Design (FMCAD'09)*, pages 77–84. IEEE, 2009.

[30] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pages 404–415. ACM, 2006.

[31] F. Somenzi. CUDD: CU decision diagram package, release 2.4.2, 2009.

APPENDIX

TABLE III
EXTENSIVE PERFORMANCE RESULTS USING RATSY. "T" INDICATES A TIME-OUT AFTER 100 000 SECONDS.

	RATSY				Mode 0		Mode 1		Mode 2		Mode 3		Mode 4		Mode 5		Mode 6		Mode 7		Mode 8	
	$\|U\|$	$\|W\|$	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size
	[-]	[-]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]
A2	32	18	1.2	733	4.2	339	1.4	269	4.4	358	1.5	259	3.4	358	1.3	217	7.6	551	1.3	260	2.0	249
A3	41	23	19	5.6k	48	406	19	355	162	732	23	540	29	420	20	423	94	732	21	565	21	303
A4	48	26	67	10k	5.1k	2.5k	461	1.8k	65k	11k	1.8k	5.3k	4.4k	3.4k	123	1.2k	T	–	480	3.7k	2.7k	2.1k
A5	56	30	135	5.7k	921	703	221	663	2.5k	1.2k	235	824	717	850	233	756	1.4k	1.6k	239	1.2k	265	581
A6	63	33	204	8.2k	1.1k	835	233	819	5.7k	1.9k	260	1.3k	714	1.4k	292	960	3.3k	2.1k	251	1.3k	299	691
A7	71	37	840	16k	2.4k	1.1k	452	1.0k	8.7k	2.2k	498	1.6k	1.8k	1.8k	964	1.2k	9.8k	2.6k	488	1.6k	664	912
A8	78	40	6.6k	135k	T	–	T	–	T	–	T	–	T	–	T	–	T	–	T	–	T	–
A9	86	44	1.8k	22k	24k	1.3k	4.2k	1.3k	26k	3.4k	4.4k	2.1k	8.6k	2.7k	6.5k	1.6k	37k	3.4k	6.7k	2.4k	4.9k	1.2k
A10	93	47	3.0k	19k	T	–	8.9k	1.5k	T	–	7.0k	2.5k	24k	3.2k	9.3k	1.9k	T	–	7.0k	2.4k	13k	1.4k
A11	100	50	4.0k	39k	77k	1.9k	7.4k	1.8k	T	–	7.4k	2.9k	16k	3.9k	12k	2.2k	T	–	7.5k	3.1k	11k	1.6k
A12	107	53	10k	38k	T	–	16k	2.0k	T	–	20k	3.1k	T	–	48k	2.4k	T	–	37k	3.1k	36k	1.7k
A13	114	56	15k	65k	T	–	45k	2.4k	T	–	28k	3.6k	45k	5.3k	87k	2.7k	T	–	31k	3.5k	T	–
A14	121	59	15k	47k	T	–	36k	2.6k	T	–	38k	3.9k	58k	5.9k	T	–	T	–	83k	3.6k	92k	2.3k
A15	129	63	19k	70k	T	–	75k	3.0k	T	–	T	–	T	–	T	–	T	–	99k	4.0k	T	–
A′2	35	21	1.7	1.0k	4.7	298	1.9	224	5.2	400	2.2	265	3.6	365	1.6	188	8.4	540	2.4	306	2.4	183
A′3	43	25	169	17k	142	307	77	465	114	450	99	561	120	310	103	555	159	555	103	781	83	444
A′4	52	30	914	28k	T	–	984	677	T	–	4.1k	4.8k	61k	4.8k	1.7k	1.2k	T	–	5.1k	5.2k	1.1k	656
A′5	60	34	9.7k	101k	7.6k	466	18k	893	8.6k	842	12k	1.1k	6.7k	475	15k	900	11k	1.4k	16k	1.9k	9.1k	834
G2	24	12	0.1	249	0.3	79	0.1	53	0.4	81	0.1	61	0.2	80	0.1	57	0.2	87	0.1	65	0.1	56
G3	28	14	0.2	394	0.8	153	0.3	123	0.9	165	0.3	181	0.7	173	0.3	147	0.7	214	0.3	174	0.4	136
G4	32	16	0.5	721	1.7	140	0.5	119	2.0	169	0.5	190	1.7	149	0.5	111	1.7	267	0.5	262	0.5	119
G5	36	18	1.2	1.8k	11	261	2.3	444	9.2	356	2.5	673	11	332	1.7	387	20	565	1.9	674	3.3	268
G6	39	19	7.7	6.2k	20	502	6.8	1.1k	15	371	2.4	601	8.7	463	4.3	1.0k	14	617	2.1	828	5.8	212
G7	42	20	3.3	3.5k	41	483	11	1.7k	121	518	27	3.5k	179	668	5.3	1.4k	927	1.2k	15	3.4k	47	1.4k
G8	46	22	8.1	5.8k	22	417	2.6	278	28	483	18	3.9k	17	414	2.7	261	178	626	7.4	5.5k	21	278
G9	50	24	5.9	3.4k	249	872	430	5.8k	551	790	180	9.9k	22k	1.1k	91	5.2k	34k	1.4k	74	10k	644	1.4k
G10	53	25	14	6.5k	871	1.9k	8.0k	22k	418	790	95	9.8k	14k	1.3k	1.3k	19k	8.4k	1.3k	57	13k	3.6k	410
G11	56	26	18	9.8k	7.4k	3.4k	71k	47k	300	969	263	20k	1.4k	1.5k	3.7k	35k	T	–	157	28k	25k	470
G12	59	27	35	14k	740	2.2k	T	–	1.4k	1.2k	1.1k	44k	37k	1.7k	14k	75k	95k	1.6k	711	62k	T	–
avg.	62	31	3.0k	23k	5.8k	930	11k	3.7k	5.7k	1.3k	4.6k	4.7k	12k	1.7k	7.7k	6.0k	11k	1.1k	11k	5.8k	8.0k	794

TABLE IV
EXTENSIVE PERFORMANCE RESULTS USING UNBEAST. "T" INDICATES A TIME-OUT AFTER 100 000 SECONDS.

	UNBEAST				Mode 0		Mode 1		Mode 2		Mode 3		Mode 4		Mode 5		Mode 6		Mode 7		Mode 8	
	$\|U\|$	$\|W\|$	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size
	[-]	[-]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]
L3	21	19	0.1	845	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19
L3-6	59	57	0.3	6.3k	1.2	115	15	2.6k	34	3.7k	0.9	90	1.2	118	2.5	2.7k	13	3.1k	0.9	79	18	89
L5	26	24	0.1	908	0.1	24	0.1	24	0.1	27	0.1	27	0.1	24	0.1	24	0.1	27	0.1	27	0.1	24
L6	34	32	0.1	2.5k	0.1	40	0.1	38	0.1	49	0.1	41	0.1	41	0.1	43	0.1	54	0.1	42	0.1	37
L7	24	22	0.1	551	0.1	22	0.1	22	0.1	25	0.1	25	0.1	22	0.1	22	0.1	25	0.1	25	0.1	22
L8	10	10	0.1	113	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10
L9	16	16	0.1	450	0.1	20	0.1	16	0.1	18	0.1	16	0.1	20	0.1	16	0.1	18	0.1	16	0.1	16
L10	22	22	0.1	1.4k	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22
L12	15	15	0.1	524	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15
L13	9	9	0.1	28	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9
L14	17	17	0.1	3.4k	0.1	17	0.1	17	0.1	21	0.1	17	0.1	17	0.1	17	0.1	21	0.1	17	0.1	17
L15	14	14	0.1	277	0.1	15	0.1	16	0.1	21	0.1	16	0.1	15	0.1	16	0.1	21	0.1	16	0.1	16
L16	18	18	0.1	830	0.1	39	0.1	29	0.1	47	0.1	41	0.1	40	0.1	28	0.1	48	0.1	37	0.1	29
L17	20	21	0.1	1.5k	0.1	33	0.1	21	0.1	35	0.1	25	0.1	36	0.1	21	0.1	37	0.1	25	0.1	21
L18	34	35	0.3	23k	1.3	249	0.2	79	3.0	465	0.2	85	1.0	224	0.2	70	1.9	451	0.2	73	0.3	83
L19	21	21	0.1	531	0.1	32	0.1	21	0.1	33	0.1	21	0.1	31	0.1	21	0.1	28	0.1	21	0.1	21
L20	28	29	0.1	4.7k	0.4	76	0.5	120	0.4	126	0.3	102	0.3	77	0.3	99	0.4	92	0.3	84	0.6	86
L21	32	32	0.4	13k	2.2	123	2.6	114	16	468	12	419	3.3	248	4.1	261	4.3	218	3.6	205	4.6	101
L22	28	26	0.1	1.1k	0.1	61	0.1	107	0.4	215	0.1	40	0.1	63	0.1	109	0.2	169	0.1	43	0.2	38
L22-5	24	22	0.1	950	0.1	53	0.1	104	0.3	205	0.1	38	0.1	59	0.1	105	0.1	175	0.1	40	0.1	34
L22-6	28	26	0.1	995	0.1	61	0.1	107	0.4	215	0.1	40	0.1	63	0.1	109	0.1	169	0.1	43	0.2	38
L22-7	27	25	0.1	1.2k	0.1	56	0.1	107	0.4	209	0.1	39	0.1	62	0.1	108	0.2	178	0.1	42	0.1	37
L22-8	26	24	0.1	1.0k	0.1	55	0.1	106	0.4	207	0.1	38	0.1	61	0.1	107	0.1	177	0.1	41	0.1	36
L22-9	25	23	0.1	910	0.1	54	0.1	105	0.4	206	0.1	39	0.1	60	0.1	105	0.1	176	0.1	41	0.1	35
L23	16	16	0.1	354	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16
B2	33	32	0.1	3.5k	0.2	45	0.1	35	0.5	113	0.1	44	0.2	54	0.1	33	0.2	66	0.1	43	0.2	36
B3	44	43	0.7	27k	4.2	453	1.9	437	7.9	280	1.6	165	2.5	467	1.1	447	1.1	169	0.8	131	3.3	160
B4	77	76	5.5	171k	T	–	1.1k	6.5k	T	–	T	–	19k	7.9k	264	8.1k	8.1k	22k	3.5k	23k	T	–
B5	93	92	650	816k	T	–	99k	17k	T	–	T	–	T	–	T	–	T	–	T	–	T	–
avg.	27	26	23	37k	0.4	64	3.4k	947	2.4	250	0.6	54	663	349	9.7	451	291	993	125	872	1.0	40