Factored Boolean Functional Synthesis

Lucas M. Tabajara
Department of Computer Science, Rice University
Houston, USA
lucasmt@rice.edu

Moshe Y. Vardi
Department of Computer Science, Rice University
Houston, USA
vardi@cs.rice.edu

***Abstract*—Boolean functional synthesis allows the automated construction of Boolean functions from declarative specifications. BDD-based techniques for this problem can be very efficient when the specification can be compactly represented by a BDD, but this is not always possible. In model checking, a way around this problem has been found by using factored representations, where formulas are represented as a conjunction of subformulas, each encoded individually as a BDD. We show how techniques and heuristics for quantifier elimination on factored formulas can also be lifted to perform synthesis, and show that this approach allows the synthesis of many problem instances that are intractable when represented by a single BDD. We compare our approach to other tools for Boolean synthesis that are not BDD-based. Our empirical evaluation shows that, while no approach dominates across the board, our tool outperforms other tools on several problem instances.**

***Index Terms*—Binary Decision Diagrams, Boolean synthesis, factored representation**

I. Introduction

The problem of synthesizing Boolean functions from relations is central to many areas of formal methods. Boolean functions can represent both logical circuits and programs over finite data types, and Boolean synthesis is also an essential component of synthesis from temporal specifications [1].

In the Boolean synthesis problem, we are given as specification a Boolean formula $f(\vec{x}, \vec{y})$, where $\vec{x}$ is a vector of input variables $x_1, \ldots, x_m$ and $\vec{y}$ is a vector of output variables $y_1, \ldots, y_n$. Our goal is twofold: first, to characterize the set of valid inputs by a formula $p(\vec{x})$ that is satisfied exactly by those inputs for which there is an output that satisfies f; second, to construct a Boolean function $g : \mathbb{B}^m \rightarrow \mathbb{B}^n$ such that $p(\vec{x}) \Rightarrow f(\vec{x}, g(\vec{x}))$. In other words, for every input $\vec{x}$ that satisfies $p(\vec{x})$, setting $\vec{y} = g(\vec{x})$ satisfies $f(\vec{x}, \vec{y})$.

Early approaches to this problem have used techniques based on Binary Decision Diagrams (BDDs) [2], [3]. The efficiency of BDDs, however, is highly dependent on finding a good variable ordering, which is a hard optimization problem. Furthermore, it is well-known that there are interesting Boolean formulas that cannot be represented by a polynomial-sized BDD. Because of this, more recent works have avoided BDDs in favor of approaches using SAT solvers [4].

Nevertheless, BDD-based techniques have been shown to be very competitive when the specification can be efficiently represented by a BDD and a good variable ordering is known [5]. This raises the question of whether it is possible, instead of discarding BDDs, to find a way to employ BDD-based techniques even in cases when a BDD for the specification cannot be constructed. In other applications where the use of BDDs is common, the solution to this problem came in the form of *factored representations* of formulas, which allow a much wider range of instances to be effectively computed [6].

Factored representations are based on the fact that it is common for Boolean formulas of practical importance to be represented by conjunctions of constraints. In other words, a Boolean formula $f(\vec{x}, \vec{y})$ might be written in the format $f_1(\vec{x}, \vec{y}) \wedge \ldots \wedge f_k(\vec{x}, \vec{y})$. In this case, rather than constructing a single *monolithic* BDD B for f, we can instead represent the formula as a collection of BDDs $B_1, \ldots, B_k$ for each of the *factors* $f_1, \ldots, f_k$, implicitly interpreted as a conjunction. Since conjoining multiple BDDs can lead to a combinatorial explosion, the factored representation is usually significantly more compact.

In symbolic model checking, where the idea of factored BDD representations originated, this approach was able to reduce the size of representations of transition relations by an order of magnitude [6]. Since then, different techniques have been developed for further improving performance, including heuristics for clustering and reordering factors [7]. Similar techniques have been used for processing factored formulas in the context of symbolic satisfiability [8]. In this approach to the satisfiability problem, a CNF formula is encoded by partitioning the set of clauses and representing each partition as a BDD. Then, symbolic quantifier elimination is used to find if there is a satisfying assignment to the formula. In this paper we show how techniques and heuristics used in these applications can be adapted to perform synthesis from factored specifications.

Other approaches have been developed for synthesis of factored formulas that do not employ BDDs. A recent work [9] uses And-Inverter Graphs (AIGs) for representing Boolean formulas and a counterexample-guided abstraction refinement (CEGAR) loop for synthesizing the function. A downside to this approach is that the CEGAR loop requires repeated calls to a SAT solver, which can have a high cost in running time. Furthermore, BDDs can be very compact for small formulas, which poses the question of whether they can produce smaller functions than AIGs when using factored representation.

A different synthesis approach is based on the close relation between Boolean synthesis and QBF solving. The CNF formulas given as input to QBF solvers are special cases of factored formulas, and a number of modern solvers are capable of computing Skolem functions for the existential variables

in terms of the universal variables [10] [11]. Therefore, by writing a specification $f(\vec{x}, \vec{y})$ in CNF, the synthesis problem can be encoded as a QBF $\forall\vec{x}.\exists\vec{y}.f(\vec{x}, \vec{y})$.

Although QBF solvers can be very efficient in solving these formulas, they are able to synthesize Skolem functions only when the QBF $\forall\vec{x}.\exists\vec{y}.f(\vec{x}, \vec{y})$ evaluates to *true*. This corresponds to the case when the specification $f(\vec{x}, \vec{y})$ is *realizable*, that is, when every input $\vec{x}$ has an output $\vec{y}$ that satisfies f, and consequently $p(\vec{x}) \equiv 1$. In many applications of Boolean synthesis we are interested, however, in unrealizable specifications as well. One such a case is LTL_f synthesis using DFA games [1], in which a winning strategy might not exist for every state of the automaton, but we would like to synthesize this strategy for all states for which it exists.

In our experimental evaluation, we first compare our implementation using factored representation with the monolithic approach, allowing us to confirm that indeed factoring the specification allows us to synthesize a number of instances that would be otherwise intractable. We then compare our implementation using BDDs to two other tools: CEGARSKOLEM [9] [12], which uses the CEGAR-based approach, and the QBF solver CADET [11]. The results show that no approach is universally better, and every tool outperforms the others in some subset of the benchmarks. Although the QBF approach has a clear advantage for realizable specifications, being unable to handle unrealizable instances limits its applicability in a number of practical cases.

Beyond performance, an advantage of using BDDs is that this makes the approach easier to integrate in temporal synthesis applications, such as [1]. This is because such applications usually employ some kind of fixpoint computation, for which BDDs are particularly suited due to the ease of checking if two BDDs are equivalent. Using other representations for Boolean formulas, such as AIGs or CNF, it becomes harder to perform such computations.

II. PRELIMINARIES

A. Boolean Formulas and Functions

We denote by $\mathbb{B} = \{0, 1\}$ the set of Boolean values. We use the notation $\vec{x}$ to represent a Boolean vector $(x_1, \ldots, x_m) \in \mathbb{B}^m$, for some m. We identify a Boolean formula f over Boolean variables $x_1, \ldots, x_m$ with the Boolean function $f : \mathbb{B}^m \to \mathbb{B}$ such that $f(\vec{x}) = 1$ if and only if $\vec{x}$ is a satisfying assignment of formula f.

We use $\neg$, $\wedge$, $\vee$ to denote the usual Boolean operators of negation, conjunction and disjunction, and $\equiv$ to denote logical equivalence of two Boolean formulas. Given two formulas $f(x_1, \ldots, x_m)$ and $f'(y_1, \ldots, y_n)$, we use $f[x_i \mapsto f']$ to denote the formula $f(x_1, \ldots, x_{i-1}, f'(y_1, \ldots, y_n), x_{i+1}, \ldots, x_m)$. We say that a variable x_i is in the *support* of a formula f if x_i determines the value of f, that is, $f[x_i \mapsto 0] \not\equiv f[x_i \mapsto 1]$.

We also use $\forall$ and $\exists$ to denote universal and existential quantification over Boolean variables. Given a quantified Boolean formula, we can use the following lemma to obtain a logically equivalent quantifier-free formula:

Lemma 1 (Self-Substitution [5])**.** *Let $f(\vec{x}, y)$ be a Boolean formula over variables $\vec{x} = (x_1, \ldots, x_m)$ and y. Then*

- $\forall y.f(\vec{x}, y) \equiv f(\vec{x}, f(\vec{x}, 0))$
- $\exists y.f(\vec{x}, y) \equiv f(\vec{x}, f(\vec{x}, 1))$

Given a formula $f(\vec{x}, y)$ with $\vec{x} = (x_1, \ldots, x_m)$, and a function $g : \mathbb{B}^m \to \mathbb{B}$, if $f(\vec{x}, g(\vec{x})) \equiv \exists y.f(\vec{x}, y)$, we say that g is a *witness* for y in f. It is clear from Lemma 1 that, for every formula $f(\vec{x}, y)$, $f(\vec{x}, 1)$ is a witness for y.

B. Boolean Synthesis

We use the following formulation of the Boolean synthesis problem:

Problem 1 (Boolean Synthesis)**.** *Given a Boolean formula $f(\vec{x}, \vec{y})$ where $\vec{x} = (x_1, \ldots, x_m)$ and $\vec{y} = (y_1, \ldots, y_n)$, called the* specification*, compute a Boolean formula $p(\vec{x})$, called the* precondition*, and a Boolean function $g(\vec{x}) : \mathbb{B}^m \to \mathbb{B}^n$, called the* implementation*, such that $\exists\vec{y}.f(\vec{x}, \vec{y}) \equiv p(\vec{x}) \equiv f(\vec{x}, g(\vec{x}))$.*

In this context, we call $\vec{x}$ the *input variables* and $\vec{y}$ the *output variables*. Intuitively, f specifies a relation between inputs and outputs of the desired Boolean function g, and p identifies valid inputs for g. If there is an output $\vec{y}$ that satisfies f for input $\vec{x}$, then a) $p(\vec{x})$ is true, and b) $\vec{y} = g(\vec{x})$ satisfies f. The implementation g can be represented by a sequence of functions $\langle g_1, \ldots, g_n \rangle$, $g_i : \mathbb{B}^m \to \mathbb{B}$. In this work we focus on specifications of the form $f(\vec{x}, \vec{y}) = f_1(\vec{x}, \vec{y}) \wedge \ldots \wedge f_k(\vec{x}, \vec{y})$.

In the context of Boolean synthesis, we say that a specification $f(\vec{x}, \vec{y})$ with input variables $\vec{x}$ and output variables $\vec{y}$ is *realizable* if for every assignment to $\vec{x}$ there exists an assignment $\vec{y}$ such that $f(\vec{x}, \vec{y})$ evaluates to *true*. In other words, if a specification f is realizable then $\exists\vec{y}.f(\vec{x}, \vec{y}) \equiv p(\vec{x}) \equiv f(\vec{x}, g(\vec{x})) \equiv 1$. In this case, every input $\vec{x}$ is a valid input for $g(\vec{x})$.

C. Binary Decision Diagrams

A *[Reduced Ordered] Binary Decision Diagram*, or BDD, is a data structure that represents a Boolean function as a directed acyclic graph [13]. BDDs can be seen as a reduced representation of a binary decision tree of a Boolean function. We require that variables are ordered the same way along every path of the BDD ("ordered") and that the BDD is minimized to eliminate duplication ("reduced"). For a given variable order, the reduced BDD is *canonical*. The variable order used can have a major impact on BDD size, and two BDDs representing the same function but with different orders can have an exponential difference in size. Since BDDs represent Boolean functions, they can be manipulated using standard Boolean operations. We overload the notation of the operators $\neg$, $\wedge$, $\vee$ and functional composition (e.g. $B[x_i \mapsto B']$) with equivalent semantics to their counterparts for Boolean formulas.

III. SYNTHESIS FROM FACTORED FORMULAS

In this section, we start by formally defining the notion of factored representations and present some of their properties.

We then describe a method for performing synthesis over factored representations.

A. Factored Representation of Boolean Formulas

Let the specification f for an instance of the Boolean synthesis problem be of the form $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge f_2(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$. Each formula f_i is called a *factor* of f. The sequence of BDDs $\langle B_1, B_2, \ldots, B_k\rangle$, where B_i is the BDD encoding of f_i, is called the *factored representation* of f. In contrast, the representation of f as a single BDD B is called the *monolithic representation*.

Note that it is possible for a formula to have an exponential monolithic representation and a polynomial factored representation. In particular, the factored representation of a formula in CNF can always be linear, since the BDD of a single clause is linear in size.

Although factored representations can be exponentially more compact than monolithic representations, they introduce complications into the synthesis procedure. To understand why, first note from the definition of Boolean synthesis that there is a close connection between Boolean synthesis and quantifier elimination. In fact, substituting the implementation $g(\vec{x})$ in the specification $f(\vec{x},\vec{y})$ is equivalent to existentially quantifying $\vec{y}$, and the precondition $p(\vec{x})$ is exactly the result of this quantification. Then, recall that existential quantifiers do not distribute over conjunction. That is, in general $\exists y_1,\ldots,y_n.\bigwedge_{i=1}^{k} f_i(\vec{x},\vec{y}) \not\equiv \bigwedge_{i=1}^{k} \exists y_1,\ldots,y_n.f_i(\vec{x},\vec{y})$.

More precisely, as pointed out in [9], the right-hand side is an over-approximation of the left-hand side, meaning that every assignment of $\vec{x}$ that satisfies the left-hand side satisfies the right-hand side, but not vice-versa.

As a consequence, if we are given a factored representation of a Boolean formula, it is not clear how to perform existential quantifier elimination, and consequently synthesis, without conjoining the factors. However, the insight first employed in [6] is that it is possible to move conjuncts outside an existential quantifier if the quantified variable does not appear in the support of the conjunct. Formally, let $F_j \subseteq \{1,\ldots,k\}$ be the set of indices i such that y_j is in the support of f_i. Then,

$$\begin{aligned}&\exists y_1,\ldots,y_n.\bigwedge_{i=1}^{k} f_i(\vec{x},\vec{y})\\ &\equiv \exists y_1,\ldots,y_{n-1}.\left(\exists y_n.\bigwedge_{i\in F_n} f_i(\vec{x},\vec{y})\right)\wedge\bigwedge_{i\notin F_n} f_i(\vec{x},\vec{y})\end{aligned}$$

Using the relation between synthesis and existential quantification, we obtain the following result:

Lemma 2. *Let $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y})\wedge f_2(\vec{x},\vec{y})\wedge\ldots\wedge f_k(\vec{x},\vec{y})$ be a specification and $g_j(\vec{x})$ be a witness to y_j in $\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})$. Then, $g_j(\vec{x})$ is a witness to y_j in $f(\vec{x},\vec{y})$.*

Proof. Since $g_j(\vec{x})$ is a witness to y_j in $\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})$, then by definition $\left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\right)[y_j \mapsto g_j(\vec{x})] \equiv \exists y_j.\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})$.

To prove that $g_j(\vec{x})$ is also a witness of $f(\vec{x},\vec{y})$, it needs to be shown that $f(\vec{x},\vec{y})[y_j \mapsto g_j(\vec{x})] \equiv \exists y_j.f(\vec{x},\vec{y})$. But

$$\begin{aligned}&f(\vec{x},\vec{y})[y_j \mapsto g_j(\vec{x})]\\ &\equiv \left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\wedge\bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\right)[y_j \mapsto g_j(\vec{x})]\\ &\equiv \left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\right)[y_j \mapsto g_j(\vec{x})]\wedge\bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\\ &\equiv \left(\exists y_j.\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\right)\wedge\bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\\ &\equiv \exists y_j.\left(\bigwedge_{i\in F_j} f_i(\vec{x},\vec{y})\wedge\bigwedge_{i\notin F_j} f_i(\vec{x},\vec{y})\right)\\ &\equiv \exists y_j.f(\vec{x},\vec{y})\end{aligned}$$

Therefore, $g_j(\vec{x})$ is a witness of $f(\vec{x},\vec{y})$. □

From Lemma 2 we have that a witness for a variable in a factored formula can be constructed from only the factors in which that variable appears. Since in practice each variable will only be in the support of a small subset of the factors, this insight means that it is possible to perform synthesis without converting entirely from the factored to the monolithic representation. Instead, we can design a strategy for synthesis directly over factored formulas.

B. Synthesis from Factored Specifications

Algorithm 1 presents a synthesis framework that takes advantage of the factored representation of the specification, using the insight from Lemma 2 to avoid conjoining all factors at once. Instead, we conjoin the factors one-by-one, and after each conjunction synthesize and eliminate the variables that do not appear in the support of any of the remaining factors. This strategy is similar to the ones followed in model checking [6] and symbolic satisfiability [8] from factored representations.

We assume the existence of a monolithic Boolean synthesis procedure, denoted by $synth(B, X, Y)$, which receives a BDD B, a set of input variables X and a set of output variables Y, and returns a BDD P representing the precondition and a sequence of BDDs $(W_j)_{y_j\in Y}$ representing the implementation.

We start, in line 2, by partitioning the output variables into sets $Y_1,\ldots,Y_k$ such that $y_j \in Y_i$ if and only if B_i is the last factor where y_j appears. In other words, $y_j \in Y_i$ if and only if $\max F_j = i$. We maintain a BDD B which accumulates the factors. In line 3, B is initialized to the empty conjunction, which is equivalent to the constant 1. We then iterate over the factors, conjoining the next factor to B at every iteration in line 5. Once B_i is conjoined, none of the output variables in Y_i appear in any of the remaining factors. The monolithic synthesis procedure is then called in line 6 to synthesize witnesses for every variable in Y_i, in terms of the input variables $x_1,\ldots,x_m$ and the output variables in $Y_{i+1},\ldots,Y_k$. Then, in line 7, B is updated to the precondition

Fig. 1. Synthesis from Factored Specifications

Input: Factored specification $\langle B_1, \ldots, B_k \rangle$.
Output: Precondition BDD P, witness BDDs $\langle W_1, \ldots, W_n \rangle$.

1: $X \leftarrow \{x_1, \ldots, x_m\}$
2: $Y_i \leftarrow \{y_j \mid B_i$ is the last factor where y_j appears$\}$
3: $B \leftarrow 1$
4: **for** $i \leftarrow 1 \ldots k$ **do**
5: $\quad B \leftarrow B \wedge B_i$
6: $\quad P_i, (W_j)_{y_j \in Y_i} \leftarrow synth(B, X \cup Y_{i+1} \cup \ldots \cup Y_k, Y_i)$
7: $\quad B \leftarrow P_i$
8: **end for**
9: **for** $i \leftarrow 1 \ldots k$, $i' \leftarrow (i+1) \ldots k$ **do**
10: $\quad W_\ell \leftarrow W_\ell[y_j \mapsto W_j]$, for all $y_\ell \in Y_i$, $y_j \in Y_{i'}$
11: **end for**
12: $P \leftarrow B$
13: **return** $P, W_1, \ldots, W_n$

P_i, which corresponds to the conjunction of the first i factors with the output variables in $Y_1 \cup \ldots \cup Y_i$ existentially quantified.

After the end of the loop, every witness W_j for $y_j \in Y_i$ has the variables from $Y_{i+1}, \ldots, Y_k$ in its support set. In the last step, performed by the loop in lines 9-11, these extra variables are eliminated by substituting their respective witnesses, making every W_j dependent only in the input variables $x_1, \ldots, x_m$.

The following theorem states the correctness of Algorithm 1, which follows from the correctness of the monolithic synthesis procedure and Lemma 2.

Theorem 1. *If $P, W_1, \ldots, W_n$ are computed according to Algorithm 1, then $\exists y_1, \ldots, y_n.(B_1 \wedge \ldots \wedge B_k) \equiv P \equiv (B_1 \wedge \ldots \wedge B_k)[y_1 \mapsto W_1, \ldots, y_n \mapsto W_n]$.*

Proof. We assume the correctness of the monolithic synthesis procedure $synth$, meaning that $synth(B, X, Y)$ returns a precondition P, and a witness W_j for each variable $y_j \in Y$ in terms of the variables in X, such that $\exists Y.B \equiv P \equiv B[y_j \mapsto W_j]_{y_j \in Y}$.

Consider the loop in lines 4-8. We will first prove that if at the start of the i-th iteration $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})$, then at the end of the i-th iteration $B \equiv \exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i)$. We will use this to prove that $P \equiv \exists y_1, \ldots, y_n.(B_1 \wedge \ldots \wedge B_k)$

Assume that at the start of the i-th iteration $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})$. Then, after line 5, $B \equiv (\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})) \wedge B_i$. Since $Y_1, \ldots, Y_{i-1}$ do not appear in B_i, the quantifier can be moved outside the conjunction, so $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$.

Then, in line 6 the monolithic synthesis procedure is called on B, with input variables $X \cup Y_{i+1} \cup \ldots \cup Y_k$ and output variables Y_i. By the correctness of the monolithic procedure, $P_i \equiv \exists Y_i.B \equiv \exists Y_1, \ldots, Y_{i-1}, Y_i.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$. Then, after line 7, when B is updated to P_i, $B \equiv \exists Y_1, \ldots, Y_{i-1}, Y_i.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$.

Therefore, if $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})$ at the start of the i-th iteration, $B \equiv \exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i)$ at the end of the i-th iteration. Taking $i = 1$, this means that if $B \equiv 1$ (the empty conjunction) before the loop then at the end of the first iteration $B \equiv \exists Y_1.B_1$. Since the invariant $B \equiv \exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i)$ is maintained, at the end of the last iteration $B \equiv \exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$. Therefore, after line 12, $P \equiv \exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$, as desired.

We now prove that $(B_1 \wedge \ldots \wedge B_k)[y_1 \mapsto W_1, \ldots, y_n \mapsto W_n] \equiv \exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$. In iteration i, we construct W_j for every $y_j \in Y_i$. Since at this time $B \equiv \exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$, by the correctness of the $synth$ procedure, $(\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i))[y_j \mapsto W_j]_{y_j \in Y_i} \equiv \exists Y_1, \ldots, Y_{i-1}, Y_i.(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_i)$. Then, since no variables in $Y_1, \ldots, Y_{i-1}$ appear in B_i,

$$\begin{aligned}
&\exists Y_1, \ldots, Y_i.(B_1 \wedge \ldots \wedge B_i) \\
&\equiv (\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_i))[y_j \mapsto W_j]_{y_j \in Y_i} \\
&\equiv ((\exists Y_1, \ldots, Y_{i-1}.(B_1 \wedge \ldots \wedge B_{i-1})) \wedge B_i)[y_j \mapsto W_j]_{y_j \in Y_i}
\end{aligned}$$

Applying this transformation recursively to $\exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k)$ results in $(\ldots (B_1[y_j \mapsto W_j]_{y_j \in Y_1} \wedge B_2)[y_j \mapsto W_j]_{y_j \in Y_2} \wedge \ldots \wedge B_k)[y_j \mapsto W_j]_{y_j \in Y_k}$. Applying Lemma 2, we can move the composition operators outside the conjunction, giving

$$\begin{aligned}
&\exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k) \\
&\equiv (B_1 \wedge \ldots \wedge B_k)[y_j \mapsto W_j]_{y_j \in Y_1} \ldots [y_j \mapsto W_j]_{y_j \in Y_k}
\end{aligned}$$

Recall that each W_j for $y_j \in Y_i$ might contain variables from $Y_{i+1}, \ldots, Y_k$ in its support set. Because of this, we cannot change the order of the composition operators. However, the loop in lines 9-11 performs the composition of each witness with the ones that succeed it, making every W_j dependent only on $x_1, \ldots, x_m$. This allows the compositions to be performed in any order, so that $\exists Y_1, \ldots, Y_k.(B_1 \wedge \ldots \wedge B_k) \equiv (B_1 \wedge \ldots \wedge B_k)[y_1 \mapsto W_1, \ldots, y_n \mapsto W_n]$. □

A problem with Algorithm 1 is that performance will be very dependent on the order of the factors. Consider for example a specification in which for every i, the output support of f_i is $\{y_1, \ldots, y_i\}$. Then, $Y_1 = Y_2 = \ldots = Y_{k-1} = \{\}$ and $Y_k = \{y_1, \ldots, y_n\}$. Processing the factors in order will result in all factors being conjoined before any witness can be synthesized, thus degenerating into the monolithic synthesis procedure. On the other hand, processing the factors in the reverse order would allow one variable to be synthesized immediately after each conjunction. Therefore, it is clear that the algorithm can benefit from reordering the factors before starting the synthesis. Finding the optimal order is a combinatorially hard problem, but a number of heuristics can be used instead. Another possible improvement in the algorithm is clustering, a technique that has been employed in other applications which use factored representations of formulas [8], [14], [15]. In clustering, the set of factors is first partitioned, and the factors in each partition are conjoined into monolithic clusters. The algorithm is then applied over the clusters rather than the individual factors. The next section explores different heuristics for clustering and reordering.

C. Clustering and Reordering

As noted in [14], if the individual BDDs for each factor are small, it is often better to combine different factors into monolithic clusters. If the clusters are constructed so that they remain of reasonable size, clustering reduces the number of iterations while not excessively increasing the cost in space.

Formally, given a factored formula $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge f_2(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$ a *clustering heuristic* partitions the set of factors $\{f_1, f_2, \ldots, f_k\}$ into κ disjoint non-empty subsets $C_1, \ldots, C_\kappa$, called the *clusters*. In practice, each cluster C_ι is represented by a BDD $\mathcal{B}_\iota$ encoding the formula $\bigwedge_{f_i \in C_\iota} f_i(\vec{x},\vec{y})$. Since conjunction is associative and commutative, $\langle \mathcal{B}_1, \ldots, \mathcal{B}_\kappa \rangle$ is itself a factored representation of the original formula f. Therefore, Algorithm 1 can be applied normally to this representation.

The goal of clustering is to create a balance between the number of factors and size of the factors. An example of clustering strategy is *rank-based clustering*, employed in [8]. In this strategy, for every variable y_j, cluster $C_j = \{f_i \mid rank(f_i) = j\}$, where $rank(f_i)$ is the highest index among the variables in the support of f_i.

Rank-based clustering naturally gives rise to some reordering heuristics, in which clusters are ordered either by increasing or decreasing rank. Two more options for reordering factors appear in the context of model checking in [7]. In that work, factored formulas are used to represent transition relations, and different reordering heuristics are used in the forward and backward simulation steps. The following are the four heuristics used in this work:

a) Bouquet's method: [8] Order by increasing rank.

b) Bucket elimination: [8] Order by decreasing rank.

c) Forward: [7] Greedily order factors by number of variables that can be eliminated once the factor is conjoined. In other words, at every step choose the factor that has the greatest number of output variables that do not appear in any of the remaining factors.

d) Backward: [7] Order factors such that at every step the next factor will be the one that has the fewest new variables, that is, variables that have not appeared in any of the previous factors. This heuristic tries to avoid as much as possible increasing the size of the conjoined BDD.

All of the above heuristics for clustering and reordering can be applied to synthesis from factored representations, but it is unclear which would give better results. Section IV describes an experimental evaluation of the different techniques.

D. BDD Variable Ordering

The size of BDDs is strongly influenced by the ordering of the variables. Part of the goal of using factored representations is to be able to represent specifications for which a good variable ordering is not known beforehand. Rather than using an arbitrary variable ordering for these cases, it would be good to be able to compute one by analyzing the structure of the formula. Similarly to clustering, finding the optimal variable ordering is a hard combinatorial problem, but numerous heuristics have been developed to find good enough approximations.

One such heuristic is the inverse *maximum cardinality search* (MCS) ordering [16]. This variable ordering is constructed based on the *Gaifman graph* of the formula $f(\vec{x},\vec{y}) = f_1(\vec{x},\vec{y}) \wedge \ldots \wedge f_k(\vec{x},\vec{y})$, defined as $G = (V, E)$, where $V = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ and $E = \{(v_1, v_2) \mid$ there exists an i such that v_1 and v_2 are in the support of $f_i\}$. In other words, the Gaifman graph of a factored formula has one vertex for each variable and has an edge between every pair of variables that share a factor.

The inverse MCS order can be computed from the Gaifman graph by the following procedure: 1) initialize an empty list L; 2) at each step, select the vertex $v \in V$ not in L with the largest number of neighbors in L, and add v to L; 3) after all vertices have been added, reverse L, so that vertices added later come first in the ordering.

Other heuristics for variable ordering were studied in [8], but among them the inverse MCS heuristic had the best results in that work. Therefore, this heuristic was chosen for the experiments in this paper.

IV. Experimental Evaluation

We performed the experiments using QBF benchmarks taken from the QBFLIB collection [17]. All benchmarks selected were of the form $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$, where $f(\vec{x},\vec{y})$ is a CNF formula. In this case, synthesis corresponds to finding a Skolem function to the existential variables. Every clause in $f(\vec{x},\vec{y})$ can be considered one factor.

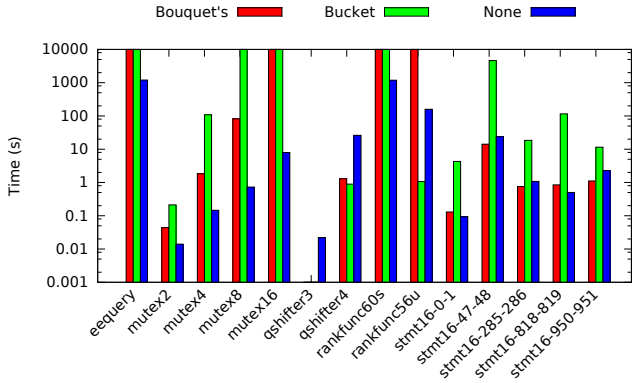
We implemented the factored algorithm from Section III and the various heuristics for clustering and reordering factors in our tool RSynth, in C++11 and using the CUDD [18] package for manipulating BDDs. As of version 3.0.0, CUDD includes a monolithic Boolean synthesis procedure `SolveEqn`, which we used in our implementation as the $synth$ subroutine.

All experiments were executed in the DAVinCI cluster at Rice University, consisting of 192 Westmere nodes of 12 processor cores each, running at 2.83 GHz with 4 GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2 GHz with 8 GB of RAM per core. The algorithm has not been parallelized, so the cluster was solely used to run multiple experiments simultaneously.

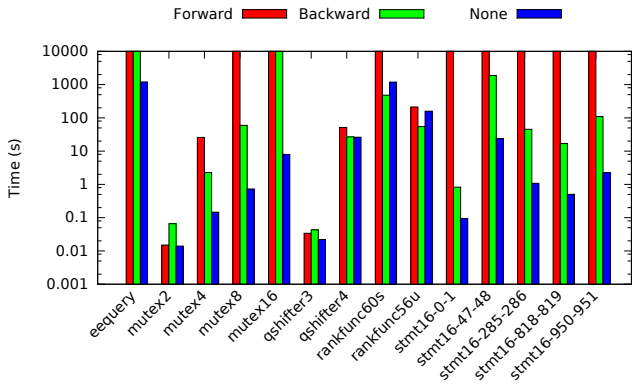
Besides comparing the monolithic and factored algorithms and evaluating different reordering heuristics, we also compare our tool RSynth with two existing tools for Boolean synthesis. The first is the CegarSkolem tool from [9], which uses a SAT-based CEGAR loop and AIGs to perform synthesis from factored formulas. The second is the 2QBF solver CADET [11].

All plots[1] in this section are shown in log scale. Each benchmark was given a time limit of two hours. Only a subset of the total set of benchmarks is included in the plots. Benchmarks for which the results were similar to already-included benchmarks were omitted, as well as benchmarks for which all or almost all of the methods timed out.

[1]Plots are best viewed online for ease of reading.



(a)



(b)

Fig. 2. Performance of the factored algorithm using different reordering heuristics, in log scale. The values include both the time spent reordering the factors and time running the algorithm. Bars of maximum height indicate instances that timed out. Bars not displayed mean that the instance took less that 1ms.

A. Heuristics for Factor Reordering

We first measure the performance of the factored algorithm using different reordering heuristics. The bar plots on Figure 2 show the running time of each heuristic on different benchmarks. Figure 2(a) shows the results for *Bouquet's Method* and *Bucket Elimination*, and Figure 2(b) shows the results for the *Forward* and *Backward* heuristics. The bars labeled *None* show the running time when no heuristic is used and the factors are simply processed in the order they are given in the input file.

Surprisingly, the results show that using no reordering is often preferable. In most of the instances, the best running time was achieved with no reordering. In fact, some benchmarks were able to be synthesized in the time limit only when no reordering was used. What this result suggests is that the process by which CNF formulas are generated already produces clauses in a good order. This makes sense because, when constructing a CNF formula, clauses with the same variables will generally be close to each other.

To confirm this point, we also ran experiments where the clauses were reordered randomly. In this case, regardless of the benchmark, the synthesis almost always timed out. We conclude that we can generally assume that the input is given in a good order, but, if this is not the case, using a moderately good heuristic, such as *Bouquet's Method*, already improves significantly over an arbitrary ordering. The performance of the other heuristics varied depending on the type of benchmark. Every heuristic outperformed the others on at least one case. Overall, the *Forward* heuristic seems to have the worst scalability, timing out for most of the instances. This is likely due to it being a greedy heuristic which tries to synthesize as many variables as possible at each step, causing the size of the BDDs to quickly increase.

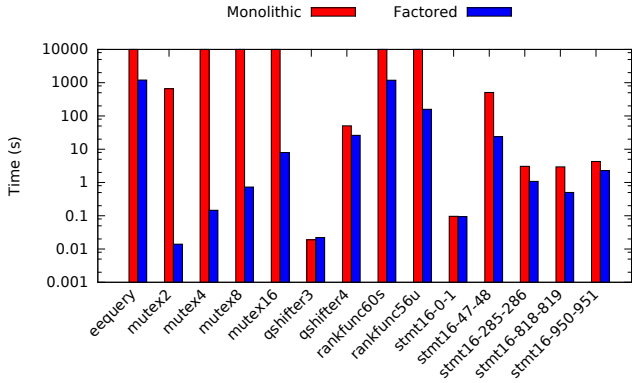


Fig. 3. Performance of the monolithic and factored synthesis algorithms, in log scale. Bars of maximum height indicate instances that timed out.

B. Factored vs. Monolithic

Next, we compare the running time of the factored algorithm with synthesis using the monolithic procedure. In the latter, the running time includes the time necessary to conjoin all the factors to create the monolithic representation. Given the previous results, no reordering was used for the factored approach. Results are shown in the bar plot on Figure 3.

It is immediately noticeable that the monolithic approach in most cases displays a much poorer performance compared with the factored one. In the few cases where the monolithic algorithm outperforms the factored algorithm, it is only by a small margin. On the other hand, there are several cases where the factored algorithm outperforms the monolithic one by an order of magnitude or more. There are additionally a number of cases synthesized by the factored algorithm which the monolithic algorithm is not able to solve in the time limit. This indicates that it is worthwhile to take advantage of factored representation for synthesis, and that it allows a number of instances to become feasible compared to a monolithic representation.

C. Comparison with CEGARSKOLEM and CADET

We compare the performance of RSYNTH with the CEGAR-based tool CEGARSKOLEM and the QBF solver CADET. Given the results of previous experiments, we select the factored algorithm with no reordering for the comparison.

Figure 4 shows a comparison of running time between RSYNTH, CEGARSKOLEM and CADET on the same benchmarks used in the previous experiments. All of the benchmarks

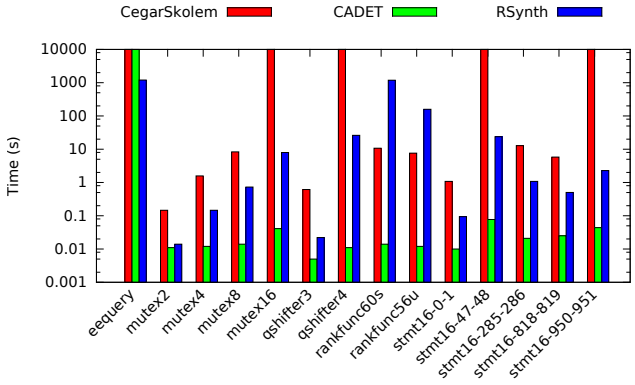


Fig. 4. Comparison of running time between RSYNTH, CEGARSKOLEM and CADET, in log scale. Bars of maximum height indicate instances that timed out.

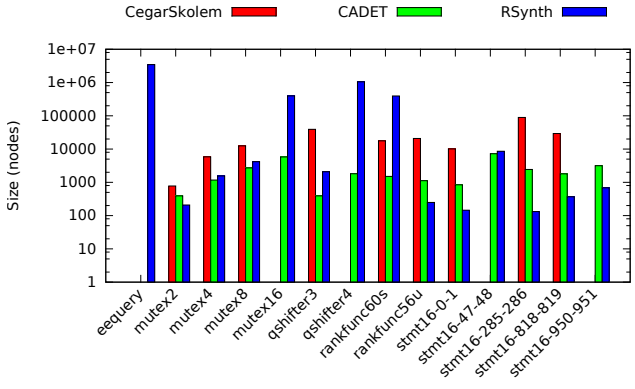


Fig. 5. Comparison of function size, in number of nodes, between RSYNTH, CEGARSKOLEM and CADET, in log scale. The size is not displayed for those instances in which the tool timed out.

are realizable, allowing CADET to be used for them. Out of 161 total benchmarks, RSYNTH was able to synthesize 87 and CEGARSKOLEM 52. There were only 6 benchmarks in which CEGARSKOLEM outperformed RSYNTH, all from the *rankfunc* class. However, CADET had by far the best performance in almost all instances, usually by orders of magnitude, and was able to synthesize all but one of the 161 benchmarks. This leads to the conclusion that the QBF approach is preferable when the specification is realizable.

Figure 5 shows a comparison of the size of the synthesized functions between the three tools. RSYNTH produces functions in the form of BDDs, while CEGARSKOLEM and CADET produce functions in the form of AIGs, therefore the comparison is in number of nodes of these data structures. Missing bars mean that the tool timed out for that particular instance. RSYNTH produced smaller functions for about half of the benchmarks, while CADET had smaller functions for the other half. This demonstrates that in many cases BDDs are indeed able to produce a more compact representation than the one obtained by AIGs.

The main conclusion that we can draw from this comparison is that, for realizable specifications, synthesis approaches

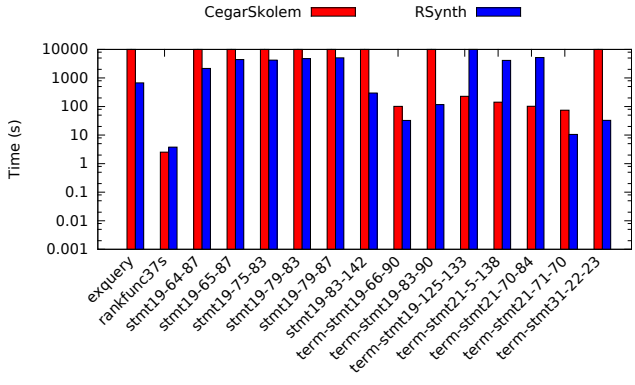


Fig. 6. Comparison of running time between RSYNTH and CEGARSKOLEM tools, in log scale, over unrealizable benchmarks. Bars of maximum height indicate instances that timed out.

based on QBF will likely dominate in terms of running time. In general, however, QBF solvers do not support the generation of Skolem functions when the formula $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$ evaluates to *false*, i.e., $f(\vec{x},\vec{y})$ is unrealizable. Therefore, for unrealizable specifications it becomes necessary to turn to other synthesis approaches. This brings up the question of how RSYNTH and CEGARSKOLEM perform in synthesizing unrealizable instances. The next section presents an evaluation dedicated to answering this question.

D. Unrealizable Specifications

For this comparison, we also used QBF benchmarks of the form $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$ from QBFLIB. This time, however, the quantified formulas evaluate to *false*, meaning that $f(\vec{x},\vec{y})$ is unrealizable. Since CADET is unable to handle such cases, we only perform a comparison between RSYNTH and CEGARSKOLEM for these formulas.

Figure 6 shows the running time of each tool in a set of unrealizable benchmarks. Comparing RSYNTH and CEGARSKOLEM, we see that the results vary depending on the instance, with either tool outperforming the other on a subset of the benchmarks. There are also many cases which one of the tools is able to synthesize while the other times out. In total, 227 benchmarks were solved by at least one of the tools, with CEGARSKOLEM performing best in 118 cases, and RSYNTH performing best in the remaining 109. This result suggests that no approach is strictly better than the other, and the best choice will likely depend on the specific instance of the problem.

The performance of QBF solvers when the specification is realizable invites the question of whether we can find a way to exploit them in synthesizing unrealizable formulas as well. It turns out that it is possible to transform an unrealizable formula into a realizable one with the same witnesses by adding an additional quantifier alternation. This idea is well known in the context of arithmetic realizability [19]. In our case, given a quantified formula $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$, we can construct a formula $\forall\vec{x}.\exists p.(p \leftrightarrow \exists\vec{y}.f(\vec{x},\vec{y}))$, which is always true. By a few simple transformations, we obtain $\forall\vec{x}.\exists p.(\neg p \vee \exists\vec{y}.f(\vec{x},\vec{y})) \wedge (p \vee \forall\vec{y}.\neg f(\vec{x},\vec{y}))$, and by renaming

variables and moving the quantifiers to the front, the resulting formula is $\forall\vec{x}.\exists p.\exists\vec{y}.\forall\vec{z}.((\neg p \vee f(\vec{x},\vec{y})) \wedge (p \vee \neg f(\vec{x},\vec{z}))$. Because of the additional quantifier, the formula is no longer in 2QBF, and therefore can no longer be handled by CADET, but other certifying QBF solvers might be able to synthesize it. Note additionally that the Skolem function for the additional existentially quantified variable p now corresponds exactly to the realizability precondition. This might be a promising approach, but a number of factors will have to be taken into consideration. Besides having to deal with the additional quantifiers, if f is originally in CNF, its negation in the second conjunct is now in DNF. Dealing with this may impose another computational challenge. We leave to future work to explore the possibilities of this transformation and the resulting synthesis approach.

V. Discussion

In this paper, we adapted techniques for processing factored representations of Boolean formulas using BDDs to the problem of Boolean functional synthesis. We show that these techniques allow synthesis from a number of specifications which cannot be handled when using a monolithic representation.

We performed an experimental comparison of our tool RSynth with other tools for Boolean synthesis, namely the CEGAR-based tool CegarSkolem [9] and the QBF solver CADET [11]. Our experiments show the QBF approach to be very efficient when the specification is realizable, significantly outperforming the others. However, QBF solvers are not generally able to synthesize functions for unrealizable specifications, which motivates the use of alternative approaches such as the one presented in this paper. For unrealizable specifications, the results of the comparison between RSynth and CegarSkolem vary, with the best tool depending on the specific instance. Therefore, we conclude that there is no single approach that dominates over all cases, rather every tool is able to handle some specifications that the others cannot.

An advantage of BDD-based techniques lies on their ease of applicability to synthesis from temporal specifications, in which Boolean synthesis is a subproblem. The use of partitioned transition relations is a common technique in these problems, and BDDs are a popular representation due to being canonical. Furthermore, these applications usually require synthesis from unrealizable formulas, where these formulas represent the subset of winning states in a game. This suggests that a synthesis approach based on a factored representation using BDDs might be a good choice for this problem. Therefore, we are also interested in pursuing forms of integrating the techniques presented here in frameworks for temporal synthesis.

Acknowledgments

Work supported in part by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering" and by the Brazilian agency CNPq through the Ciência Sem Fronteiras program. We thank Supratik Chakraborty's group, from IIT Bombay, and Markus Rabe, from UC Berkeley, for their help with the tools used in the experiments and for valuable discussions.

References

[1] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi, "Symbolic LTL$_f$ Synthesis," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017 (to appear).

[2] J. Kukula and T. Shiple, "Building Circuits from Relations," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, E. Emerson and A. Sistla, Eds., vol. 1855. Springer, 2000, pp. 113–123.

[3] E. Tronci, "Automatic Synthesis of Controllers from Formal Specifications," in *ICFEM*, 1998, pp. 134–143.

[4] J. Jiang, H. Lin, and W. Hung, "Interpolating Functions From Large Boolean Relations," in *2009 International Conference on Computer-Aided Design (ICCAD'09), November 2-5, 2009, San Jose, CA, USA*. IEEE, 2009, pp. 779–784.

[5] D. Fried, L. M. Tabajara, and M. Y. Vardi, "BDD-Based Boolean Functional Synthesis," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, S. Chaudhuri and A. Farzan, Eds. Springer, 2016.

[6] J. Burch, E. Clarke, and D. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," in *Proc. 28th ACM/IEEE Design Automation Conference*. ACM, 1991, pp. 403–407.

[7] D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," in *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, 1994, pp. 299–310.

[8] G. Pan and M. Vardi, "Symbolic Techniques in Satisfiability Solving," *J. Autom. Reasoning*, vol. 35, no. 1-3, pp. 25–50, 2005.

[9] A. K. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay, "Skolem Functions for Factored Formulas," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, 2015, pp. 73–80.

[10] M. Heule, M. Seidl, and A. Biere, "Efficient Extraction of Skolem Functions from QRAT Proofs," in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, 2014, pp. 107–114.

[11] M. N. Rabe and S. A. Seshia, "Incremental Determinization," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, 2016, pp. 375–392.

[12] S. Akshay, S. Chakraborty, A. K. John, and S. Shah, "Towards Parallel Boolean Functional Synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, 2017, pp. 337–353.

[13] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[14] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, "Efficient BDD Algorithms for FSM Synthesis and Verification," in *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV*, 1995.

[15] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, 2002, pp. 359–364.

[16] R. E. Tarjan and M. Yannakakis, "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs," *SIAM J. Comput.*, vol. 13, no. 3, pp. 566–579, 1984.

[17] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella, "Quantified Boolean Formulas satisfiability library (QBFLIB)," 2005, www.qbflib.org.

[18] F. Somenzi, *CUDD: CU Decision Diagram Package Release 3.0.0*, University of Colorado at Boulder, 2015.

[19] U. Kohlenbach, *Applied Proof Theory - Proof Interpretations and their Use in Mathematics*, ser. Springer Monographs in Mathematics. Springer, 2008.