Functional Synthesis via Input-Output Separation

Supratik Chakraborty
IIT Bombay
Mumbai, India
supratik@cse.iitb.ac.in

Dror Fried
Rice University
Houston, USA
dror.fried@rice.edu

Lucas M. Tabajara
Rice University
Houston, USA
lucasmt@rice.edu

Moshe Y. Vardi
Rice University
Houston, USA
vardi@cs.rice.edu

***Abstract*—Boolean functional synthesis is the process of constructing a Boolean function from a Boolean specification that relates input and output variables. Despite significant recent developments in synthesis algorithms, Boolean functional synthesis remains a challenging problem even when state-of-the-art methods are used for decomposing the specification. In this work we bring a fresh decomposition approach, orthogonal to existing methods, that explores the decomposition of the specification into separate input and output components. We make use of an input-output decomposition of a given specification described as a CNF formula, by alternatingly analyzing the separate input and output components. We exploit well-defined properties of these components to ultimately synthesize a solution for the entire specification. We first provide a theoretical result that, for input components with specific structures, synthesis for CNF formulas via this framework can be performed more efficiently than in the general case. We then show by experimental evaluations that our algorithm performs well also in practice on instances which are challenging for existing state-of-the-art tools, serving as a good complement to modern synthesis techniques.**

I. Introduction

Boolean functional synthesis is the problem of constructing a Boolean function from a Boolean specification that describes a relation between input and output variables [2], [12], [19], [35]. This problem has been explored in a number of settings including circuit design [20], QBF solving [27], and reactive synthesis [36], and several tools have been developed for its solution. Nevertheless, scalability of Boolean functional synthesis methods remains a concern as the number of variables and size of the formula grows. This is not surprising since Boolean functional synthesis is in fact CO-NP$^{\text{NP}}$-hard.

A standard practice for handling the problem of scalability is based on decomposing the given formula into smaller sub-specifications and synthesizing each component separately [2], [19], [35]. The most common form of such decomposition, called *factorization*, is when the formula is represented as a conjunction of constraints, in which each conjunct can be seen as a sub-specification [19], [35]. The main challenge in this approach is that most factors cannot be synthesized entirely separately due to the dependencies created by shared input and output variables. The ways to meet this challenge are usually to either merge factors that share variables [35] or perform additional computations in order to combine the functions synthesized for different factors [19]. All these result in additional work that must be performed during the synthesis.

In this work, we propose an alternative decomposition framework, which follows naturally from the fact that variables in the specification are separated into input and output variables. This idea was originally inspired by [11], which explores the notion of *sequential relational decomposition*, in which a relation is decomposed into two by introducing an intermediate domain. Differently from factorization, this form of decomposition allows the two components to be synthesized completely independently. That work, however, shows that decomposition is hard in general, and if the relation is given as a Boolean circuit, decomposition is NEXPTIME-complete. Furthermore, there is no guarantee that synthesizing the two components independently would be easier than synthesizing the original specification, since the synthesis of one component might ignore useful information given by the other component.

We instead suggest a more relaxed notion of decomposition for specifications described as CNF formulas, in which every clause is split into an input and an output clause and the independent analyses of the input/output components "cooperate" to synthesize a function for the entire specification. Based on this concept, we describe a novel synthesis algorithm for CNF formulas called the "Back-and-Forth" algorithm, where rather than synthesizing the input and output components entirely independently we share information back and forth between the two components to guide the synthesis. More specifically, our algorithm alternates between SAT calls that follow the input-component structure analysis and MaxSAT calls that follow the output-component structure analysis. Thus, this approach builds on recent progress with SAT and MaxSAT solving [21], [30]. A notable consequence of our method is that, as the number of SAT calls is dependent on the structure of the input component, for specifications with some well-defined input structure we can perform synthesis in P$^{\text{NP}}$, compared to the generally mentioned CO-NP$^{\text{NP}}$-hardness. An additional advantage of our algorithm is that it constructs the synthesized function as a *decision list* [29]. Compared to other data structures for representing Boolean functions, such as ROBDDs or AIGs, decision lists have significant benefits in term of explainability, allowing domain specialists to validate and analyze their behavior (see discussion in Section VI for more details).

Work supported in part by NSF grants CCF-1319459 and IIS-1527668, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by a grant from MHRD, Govt of India, under the IMPRINT-1 scheme, and by the Brazilian agency CNPq through the Ciência Sem Fronteiras program. We thank Assaf Marron for useful discussions, and the anonymous reviewers for their suggestions. Extended version of the paper available on arXiv.

We experimentally evaluate the "Back-and-Forth" algorithm on a suite of standard synthesis benchmarks, comparing its performance with that of state-of-the-art synthesis tools. Although these tools perform very well on many families of benchmarks, our results show that the "Back-and-Forth" algorithm is able to handle classes of benchmarks that these tools are unable to synthesize, indicating that it belongs in a portfolio of synthesis algorithms.

II. Related Work

Constructing explicit representations of implicitly specified functions is a fundamental problem of interest to both theoreticians and practitioners. In the contexts of Boolean functional synthesis and certified QBF solving, such functions are also called *Skolem functions* [8], [14], [19]. Boole [9] and Lowenheim [22] studied variants of this problem when computing most general unifiers in resolution-based proofs. Unfortunately, their algorithms, though elegant in theory, do not scale well in practice [23]. The close relation between Skolem functions and proof objects in specialized QBF proof systems has been explored in [8], [14]. One of the earliest applications of Boolean functional synthesis has been logic synthesis - see [34] for a survey. More recently, Boolean functional synthesis has found applications in diverse areas such as temporal strategy synthesis [3], [16], [36], certified QBF solving [6], [7], [26], [28], automated program synthesis [31], [33], circuit repair and debugging [18], and the like. This has resulted in a new generation of Boolean functional synthesis tools, cf. [1], [2], [12], [14], [19], [27], [28], [35], that are able to synthesize functions from significantly larger relational specifications than what was possible a decade back.

Recent tools for Boolean functional synthesis can be broadly categorized based on the techniques employed by them. Given a specification $F(\vec{x}, \vec{y})$, where $\vec{x}$ denotes inputs and $\vec{y}$ denotes outputs, the work of [14] extracts Skolem functions for $\vec{y}$ in terms of $\vec{x}$ from a proof of validity of $\forall\vec{x}.\exists\vec{y}.F(\vec{x}, \vec{y})$ expressed in a specific format. The efficiency of this technique crucially depends on the existence and size of a proof in the required format. *Incremental determinization* [27] is a highly effective synthesis technique that accepts as input a CNF representation of a specification and builds on several successful heuristics used in modern conflict-driven clause-learning (CDCL) SAT solvers [30].

In [12], the composition-based synthesis approach of [17] is adapted and new heuristics are proposed for synthesizing Skolem functions from an ROBDD representation of the specification. The technique has been further improved in [35] to work with factored specifications represented as implicitly conjoined ROBDDs. CEGAR-based techniques that use modern SAT solvers as black boxes [1], [2], [19] have recently been shown to scale well on several classes of large benchmarks. The idea behind these techniques is to start with an efficiently computable initial estimate of Skolem functions, and use a SAT solver to test if the estimates are correct. A satisfying assignment returned by the solver provides a counterexample to the correctness of the function estimates, and can be used to iteratively refine the estimates. In [1], it is shown that transforming the representation of the specification to a special negation normal form allows one to efficiently synthesize Skolem functions.

Both ROBDD and CEGAR-based approaches make use of decomposition techniques to improve performance, the most common of which is *factorization* [19], [35]. In this method, every conjunct of a conjunctive specification is considered individually. The main drawback in this approach is that the dependencies between conjuncts limit how much each of them can be analyzed independently of the others, requiring either partially combining components, as in [35], or going through a process of refinement of the results [19]. This issue motivates the search for alternative notions of decomposition for synthesis problems. Our approach is loosely inspired by the idea of *sequential relational decomposition* explored in depth in [11]. A more direct application of this idea to synthesis might still be possible, but requires further exploration. In addition to the above techniques, templates or sketches have been used to synthesize functions when information about the possible functional forms is available a priori [32], [33].

As is clear from above, several orthogonal techniques have been found to be useful for the Boolean functional synthesis problem. In fact, there remain difficult corners, where the specification is stated simply, and yet finding Skolem functions that satisfy the specification has turned out to be hard for all state-of-the-art tools. Our goal in this paper is to present a new technique and algorithm for this problem, that does not necessarily outperform existing techniques on all benchmarks, but certainly outperforms them on instances in some of these difficult corners. We envisage our technique being added to the existing repertoire of techniques in a portfolio Skolem-function synthesizer, to expand the range of problems that can be solved.

III. Preliminaries

A. Boolean Functional Synthesis

A specification for the Boolean functional synthesis problem is a (quantifier-free) Boolean formula $F(\vec{x}, \vec{y})$ over *input variables* $\vec{x} = (x_1, \ldots, x_m)$ and *output variables* $\vec{y} = (y_1, \ldots, y_n)$. Note that F can be interpreted as a relation $F \subseteq X \times Y$, where X is the set of all assignments $\hat{x}$ to $\vec{x}$ and Y is the set of all assignments $\hat{y}$ to $\vec{y}$. With that in mind, we denote by $Dom(F) = \{\hat{x} \mid \exists\hat{y}.(F(\hat{x}, \hat{y}) = 1)\}$ and $Img(F) = \{\hat{y} \mid \exists\hat{x}.(F(\hat{x}, \hat{y}) = 1)\}$ the domain and image of the relation represented by F. We also use $Img_{\hat{x}}(F) = \{\hat{y} \mid F(\hat{x}, \hat{y}) = 1\}$ to denote the image of a specific element $\hat{x} \in X$. If $Dom(F) = X$, then we say that F is *realizable*.

Two Boolean formulas $F(\vec{w})$ and $F'(\vec{w})$ are said to be *logically equivalent*, denoted by $F \equiv F'$, if they have the same solution space; that is, for every assignment $\hat{w}$ to $\vec{w}$, $F(\hat{w}) = 1$ iff $F'(\hat{w}) = 1$. Unless stated otherwise, all Boolean formulas mentioned in this work are quantifier free.

We say that a partial function $g : X \to Y$ *implements* a relation $F \subseteq X \times Y$ if for every $\hat{x} \in Dom(F)$ we have that $(\hat{x}, g(\hat{x})) \in F$. Such a g is also called a *Skolem function* of F.

Note that if F is realizable, then g is a total function. Finally, we define the *Boolean-synthesis problem* as follows:

Problem 1. *Given a specification $F(\vec{x}, \vec{y})$, construct a partial function g that implements F.*

For more information on Boolean synthesis, see [12], [19].

B. Decision lists

Our choice of representation of Skolem functions in this work is inspired by the idea that we can represent an arbitrary Boolean function f by a *decision list* [29]. A decision list is an expression of the form `if` $f_1(\vec{x})$ `then` $\hat{y}_1$ `else if` $f_2(\vec{x})$ `then` $\hat{y}_2$ `else ... else` $\hat{y}_k$, where each f_i is a formula in terms of the input variables $\vec{x}$ and each $\hat{y}_i$ is an assignment to the output variables $\vec{y}$. The length k of the list corresponds to the number of decisions. Clearly, for a specification $F(\vec{x}, \vec{y})$ with m input variables we can always synthesize as an implementation a decision list of length 2^m, where for every possible assignment of $\vec{x}$ we choose an assignment of $\vec{y}$ that satisfies the specification. Many specifications, however, can be implemented by significantly smaller decision lists, by taking advantage of the fact that multiple inputs can be mapped to the same output. Our analysis identifies and exploits these cases.

Despite being a natural representation, decision lists might not be appropriate for a physical implementation of the synthesized function as a circuit. In this case, it might make sense to collect the decisions into a more compact representation, such as an ROBDD.

C. Conjunctive Normal Form

A Boolean formula $F(\vec{w})$ is in *conjunctive normal form* (CNF) if F is a conjunction of clauses $C_1 \wedge \ldots \wedge C_k$, where every clause C_i is a disjunction of literals (a variable or its negation). A subset S of the clauses of a CNF formula F is *satisfiable* if there exists an assignment $\hat{w}$ to the variables $\vec{w}$ in F such that $C_i(\hat{w}) = 1$ for every clause $C_i \in S$. Similarly, a subset S of the clauses of F is *all-falsifiable* if there exists an assignment $\hat{w}$ such that $C_i(\hat{w}) = 0$ for every clause $C_i \in S$. A subset S of clauses is a *maximal satisfiable subset* (MSS) if S is satisfiable and every superset $S' \supset S$ is unsatisfiable. Similarly, S is a *maximal falsifiable subset* (MFS) if S is all-falsifiable and every superset $S' \supset S$ is not all-falsifiable. For more information on MSS and MFS, refer to [15].

IV. Synthesis via Input-Output Separation

In this section, we present a novel algorithm for Boolean functional synthesis from CNF specifications. Our approach is based on a separation of every clause into an input part and an output part. First, we describe how a decision list implementing the specification can be constructed by enumerating MFSs of the input clauses, or similarly by enumerating MSSs of the output clauses. Then, we show how we can benefit from alternating between the two: the MFSs can be used to avoid useless MSSs, while the MSSs can be used to cover multiple MFSs at the same time without enumerating all of them.

Given a CNF formula $F(\vec{x}, \vec{y})$, assume $F(\vec{x}, \vec{y}) = \bigwedge_{i=1}^{k} C_i$, where $C_1, \ldots, C_k$ are clauses over $\vec{x}$ and $\vec{y}$. Let $C_i|_{\vec{x}}$ denote the x-part of clause C_i, that is, the disjunction of all x literals in C_i. Similarly, let $C_i|_{\vec{y}}$ be the y-part of clause C_i, the disjunction of all y literals in C_i. We call $S_{\vec{x}} = \{C_i|_{\vec{x}} \mid C_i \text{ is a clause in } F\}$ and $S_{\vec{y}} = \{C_i|_{\vec{y}} \mid C_i \text{ is a clause in } F\}$ the set of input and output clauses of the specification, respectively.

In the following sections, we describe how to perform separate analyses of the input component $S_{\vec{x}}$ and the output component $S_{\vec{y}}$, and then how to combine these analyses into a single synthesis algorithm that alternates between the two components.

A. Analysis of the Input Component

In this subsection we assume that the specification F is realizable. First, consider a single assignment $\hat{x}$ to the input variables $\vec{x}$. Let $Fals(\hat{x}) = \{C_i|_{\vec{x}} \in S_{\vec{x}} \mid C_i|_{\vec{x}}(\hat{x}) = 0\}$ be the subset of input clauses that $\hat{x}$ falsifies. For a set $S'_{\vec{x}} \subseteq S_{\vec{x}}$ of input clauses, let $Co(S'_{\vec{x}}) = \{C_i|_{\vec{y}} \in S_{\vec{y}} \mid C_i|_{\vec{x}} \in S'_{\vec{x}}\}$ be the corresponding set of output clauses and let $MustSat(\hat{x}) = Co(Fals(\hat{x}))$. Note that $C_i \equiv (C_i|_{\vec{x}} \vee C_i|_{\vec{y}}) \equiv (\neg C_i|_{\vec{x}} \rightarrow C_i|_{\vec{y}})$ for every clause C_i. Therefore $MustSat(\hat{x})$ is the subset of output clauses that must be satisfied in order to satisfy F when $\hat{x}$ is the input assignment.

A key observation is that for two different input assignments $\hat{x}$ and $\hat{x}'$, if $Fals(\hat{x}') \subseteq Fals(\hat{x})$, then $MustSat(\hat{x}') \subseteq MustSat(\hat{x})$, and therefore every output assignment $\hat{y}$ that satisfies the specification for $\hat{x}$ also satisfies the specification for $\hat{x}'$. Hence, it is enough to consider only assignments for $\vec{x}$ that falsify a maximal number of input clauses. This leads to the following lemma:

Lemma 1. *Let $M_{\vec{x}}$ be an MFS of $S_{\vec{x}}$, and $\hat{y}$ be an assignment that satisfies $Co(M_{\vec{x}})$. Then: (1) For every assignment $\hat{x}$ such that $Fals(\hat{x}) \subseteq M_{\vec{x}}$, the assignment $(\hat{x}, \hat{y})$ satisfies $F(\vec{x}, \vec{y})$; and (2) There is no assignment $\hat{x}$ such that $Fals(\hat{x}) \supset M_{\vec{x}}$.*

Proof. (1) For every clause $C_i|_{\vec{x}} \in Fals(\hat{x})$, since $C_i|_{\vec{x}} \in M_{\vec{x}}$, we have that $C_i|_{\vec{y}}$ is in $Co(M_{\vec{x}})$ and therefore is satisfied by $\hat{y}$. Therefore, every clause C_i in $F(\vec{x}, \vec{y})$ that is not satisfied by $\hat{x}$ is satisfied by $\hat{y}$. Note that (2) follows from $M_{\vec{x}}$ being maximal. □

From Lemma 1 and our assumption that $F(\vec{x}, \vec{y})$ is realizable, we can conclude the following.

Corollary 1. *F can be implemented by a decision list of length equal to the number of MFS of $S_{\vec{x}}$, where each f_i in the decision list is of size linear in the size of the specification.*

Proof. Construct $f_i(\vec{x})$ by taking the conjunction of all input clauses $C|_{\vec{x}}$ not contained in the i-th MFS M_i. Then, $f_i(\vec{x})$ is satisfied exactly by those assignments $\hat{x}$ such that $Fals(\hat{x})$ is a subset of M_i. Then, set the corresponding output assignment $\hat{y}_i$ to an arbitrary satisfying assignment of $Co(M_i)$. □

Example 1. *Let $F(x_1, x_2, y_1, y_2) = (x_1 \vee \neg x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \neg y_1) \wedge (x_2 \vee y_1 \vee \neg y_2) \wedge (\neg x_1 \vee x_2 \vee y_2)$. We first construct*

input clauses $S_{\vec{x}} = \{(x_1 \vee \neg x_2), (x_1 \vee x_2), (x_2), (\neg x_1 \vee x_2)\}$ *and output clauses* $S_{\vec{y}} = \{(y_1), (\neg y_1), (y_1 \vee \neg y_2), (y_2)\}$. $S_{\vec{x}}$ *has three MFS:* $\{(x_1 \vee \neg x_2)\}$, $\{(x_1 \vee x_2), (x_2)\}$ *and* $\{(x_2), (\neg x_1 \vee x_2)\}$. *From these MFS we can construct a decision list implementing* F *in the way described above. Note that this decision list necessarily covers every possible input assignment:*

$$\texttt{if } (x_1 \vee x_2) \wedge (x_2) \wedge (\neg x_1 \vee x_2) \texttt{ then } (y_1 := 1; y_2 := 0)$$
$$\texttt{else if } (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \texttt{ then } (y_1 := 0; y_2 := 0)$$
$$\texttt{else if } (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \texttt{ then } (y_1 := 1; y_2 := 1)$$

Note that we require $F(\vec{x}, \vec{y})$ to be realizable because otherwise we cannot guarantee that $Co(M_{\vec{x}})$ will be satisfiable for every MFS $M_{\vec{x}}$ of the input clauses. If $Co(M_{\vec{x}})$ is unsatisfiable, however, it is not enough to simply remove the corresponding $f_i(\vec{x})$ from the decision list, because there might be a subset $M'_{\vec{x}} \subset M_{\vec{x}}$ for which $Co(M'_{\vec{x}})$ is satisfiable.

This is the first time to our knowledge that MFS are used for synthesis purposes. An advantage of enumerating MFS is that finding an MFS can be easily done, in a precise sense discussed below. One way to do this is through the *conflict graph* of the set of input clauses [13]. Given a set of clauses S, the conflict graph of S is the graph where every vertex corresponds to a clause in S, and there is an edge between two vertices iff the corresponding clauses have a complementary pair of literals between them (that is, the same variable appears in positive form in one clause and in negative form in the other). The complement of the conflict graph is called a *consensus graph* [13].

Since two clauses can be falsified at the same time iff there is no edge between them in the conflict graph, or alternatively there is an edge between them in the consensus graph, there is a one-to-one correspondence between MFS of the set of clauses, maximal independent sets (MIS) in the conflict graph, and maximal cliques in the consensus graph. Therefore, we can enumerate the MFS in a set of clauses by either enumerating MIS in the conflict graph or maximal cliques in the consensus graph. The benefit of this reduction is that maximal cliques display a so called *polynomial-time listability*, meaning that finding a maximal clique can be performed in polynomial time, and therefore enumeration takes polynomial time in the number of maximal cliques [15].

This relation between the set of MFS and maximal cliques implies that the size of the smallest decision list that implements a given specification is upper bounded by the number of maximal cliques in the consensus graph of the input clauses. Therefore we have the following result.

Theorem 1. *Synthesis can be performed in* P^{NP} *for specifications for which the consensus graph of* $S_{\vec{x}}$ *has a polynomial number of maximal cliques (such as planar or chordal graphs).*

Proof. Given a specification F, construct the consensus graph of the input component, enumerate the maximal cliques and for each one use a SAT solver to obtain a corresponding satisfying assignment for the output clauses. Since the number of maximal cliques is polynomial, only a polynomial number of SAT calls is required. □

Theorem 1 demonstrates an improvement relative to the general CO-$\mathrm{NP}^{\mathrm{NP}}$-hardness of synthesis. Moreover, constructing the consensus graph of the input component is easy, as is testing for certain graph properties, such as planarity, that ensure a small number of maximal cliques. Therefore, Theorem 1 provides an elegant method of deciding whether synthesis can be performed efficiently in practice before even beginning the synthesis process.

To summarize this section, the analysis of the input component provides two insights. First, a decision list implementing the specification can be constructed from the list of MFS of the input clauses. Second, analyzing the graph structure of the input component allows us to identify classes of specifications for which synthesis can be performed more efficiently. Note that this analysis, however, does not take into account the properties of the output component, and as such the decision list produced by ignoring the output component may be longer than necessary. With that in mind, the next section presents a complementary analysis of the output component that can help to produce a smaller decision list.

B. Analysis of the Output Component

For the analysis of the output component, consider the set $MustSat(\hat{x})$, defined in the previous subsection, of output clauses that must be satisfied when $\hat{x}$ is the input assignment. Then for every two input assignments $\hat{x}$ and $\hat{x}'$, if $MustSat(\hat{x}') \subseteq MustSat(\hat{x})$, every output assignment $\hat{y}$ that satisfies the specification for $\hat{x}$ also satisfies the specification for $\hat{x}'$. Therefore, it is enough when constructing the decision list to consider only those satisfiable subsets of $S_{\vec{y}}$ that are of maximal size. Similarly to Lemma 1 in the previous section, this insight allows us to state the following lemma:

Lemma 2. *Let* $M_{\vec{y}}$ *be an MSS of* $S_{\vec{y}}$ *and* $\hat{y}$ *be an assignment that satisfies* $M_{\vec{y}}$. *Then: (1) for every assignment* $\hat{x}$ *such that* $MustSat(\hat{x}) \subseteq M_{\vec{y}}$, *the assignment* $(\hat{x}, \hat{y})$ *satisfies* $F(\vec{x}, \vec{y})$; *and (2) for every assignment* $\hat{x}$ *such that* $MustSat(\hat{x}) \supset M_{\vec{y}}$, *there is no* $\hat{y}'$ *such that the assignment* $(\hat{x}, \hat{y}')$ *satisfies* $F(\vec{x}, \vec{y})$.

Proof. (1) Since $\hat{y}$ satisfies every clause $C_i|_{\vec{y}}$ in $M_{\vec{y}}$, it must be that $\hat{y}$ also satisfies every clause in $MustSat(\hat{x})$. Therefore, for every clause C_i in F, either $C_i|_{\vec{x}}$ is satisfied by $\hat{x}$ (and therefore $C_i|_{\vec{y}} \notin MustSat(\hat{x})$) or $C_i|_{\vec{y}}$ is satisfied by $\hat{y}$. Therefore $(\hat{x}, \hat{y})$ satisfies $F(\vec{x}, \vec{y})$. (2) Since $M_{\vec{y}}$ is maximal, then in this case $MustSat(\hat{x})$ must be unsatisfiable. Therefore there is no $\hat{y}'$ that can satisfy all clauses that $\hat{x}$ does not already satisfy. □

Therefore, similarly to the analysis of the input component, we have:

Corollary 2. F *can be implemented by a decision list of length equal to the number of MSS of* $S_{\vec{y}}$, *where each* f_i *in the decision list is of size linear in the size of the specification.*

Proof. Construct $f_i(\vec{x})$ by taking the conjunction of all input clauses $C|_{\vec{x}}$ such that $C|_{\vec{y}}$ is not contained in the i-th MSS M_i. Then, $f_i(\vec{x})$ is satisfied exactly by those assignments $\hat{x}$ such that $MustSat(\hat{x})$ is a subset of M_i. Then, set the corresponding output assignment $\hat{y}_i$ to an arbitrary satisfying assignment of M_i. □

Example 2. *Let F, $S_{\vec{x}}$ and $S_{\vec{y}}$ be the same as in Example 1. $S_{\vec{y}}$ has three MSS: $\{(y_1), (y_1 \vee \neg y_2), (y_2)\}$, $\{(\neg y_1), (y_1 \vee \neg y_2)\}$ and $\{(\neg y_1), (y_2)\}$. From these MSS we can construct a decision list implementing F in the way described above. Note that some decisions in the list might be redundant:*

$$\texttt{if } (x_1 \vee x_2) \texttt{ then } (y_1 := 1; y_2 := 1)$$
$$\texttt{else if } (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \texttt{ then } (y_1 := 0; y_2 := 0)$$
$$\texttt{else if } (x_1 \vee \neg x_2) \wedge (x_2) \texttt{ then } (y_1 := 0; y_2 := 1)$$

Unlike the input component, the output analysis does not require the specification to be realizable to produce the correct answer: for every input $\hat{x}$ for which an output $\hat{y}$ exists, $MustSat(\hat{x})$ will be contained in some MSS, and therefore will be covered by the decision list. On the other hand, we do not care about the case where an input $\hat{x}$ has no corresponding output $\hat{y}$. Note, however, that unlike the input component, we do not have here a simple graph structure that can be exploited to obtain the list of MSSs, and finding an MSS is clearly NP-hard. Therefore, it is unlikely for us to be able to efficiently identify instances where the number of MSS is polynomial.

More importantly, however, is that taking into account only the output component and ignoring the input component may also lead to a large decision list that includes many MSSs that would never be activated by an input. This fact emphasizes the drawbacks of independent synthesis of the components, and motivates the development of an algorithm that combines the input and output analyses to produce a decision list that is smaller than either of the ones produced by each analysis individually.

C. Alternating between Input and Output Components

Our next goal is to combine the input and output analyses obtained so far into a synthesis procedure that constructs a decision list of length upper-bounded by the minimum among the number of MFS of the input clauses and the number of MSS of the output clauses. Due to the restrictions of the input analysis, if the specification is unrealizable the procedure terminates without producing a decision list. Extending the synthesis to unrealizable specifications is left for future work. We first state the following lemma:

Lemma 3. *If $F(\vec{x}, \vec{y})$ is realizable, then for every MFS $M_{\vec{x}}$ of $S_{\vec{x}}$, $Co(M_{\vec{x}}) \subseteq M_{\vec{y}}$ for some MSS $M_{\vec{y}}$ of $S_{\vec{y}}$.*

Proof. For every MFS $M_{\vec{x}}$, since $M_{\vec{x}}$ is all-falsifiable, there exists an input assignment $\hat{x}$ such that $Fals(\hat{x}) = M_{\vec{x}}$. Then, since F is realizable, $MustSat(\hat{x}) = Co(M_{\vec{x}})$ is satisfiable, and therefore is contained in some MSS. □

Given an MFS $M_{\vec{x}}$ for the input clauses, we say that an MSS $M_{\vec{y}}$ for the output clauses *covers* $M_{\vec{x}}$ if $Co(M_{\vec{x}}) \subseteq M_{\vec{y}}$.

Algorithm 1 Back-and-Forth synthesis algorithm combining MFS and MSS analysis.

1: initialize a list of MSSs L to the empty list
2: **while** there are still MFS left to generate **do**
3: $\quad M_{\vec{x}} \leftarrow$ MFS of $S_{\vec{x}}$ not covered by any MSS in L
4: $\quad$ **if** MSS $M_{\vec{y}} \subseteq S_{\vec{y}}$ covering $M_{\vec{x}}$ exists **then**
5: $\quad\quad$ add $M_{\vec{y}}$ to L
6: $\quad$ **else**
7: $\quad\quad$ FAIL: specification is unrealizable
8: $\quad$ **end if**
9: **end while**
10: construct decision list from L

Lemma 3 says that for every MFS $M_{\vec{x}}$, there exists at least one MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$. Therefore, instead of producing a satisfying assignment for $Co(M_{\vec{x}})$, we can produce a satisfying assignment for $M_{\vec{y}}$. In fact, such satisfying assignment also takes care of every other MFS covered by $M_{\vec{y}}$, making it unnecessary to generate them.

The above insight gives rise to Algorithm 1, which we call the "Back-and-Forth" algorithm. In this algorithm, we maintain a list L of MSSs that is initially empty. At every iteration of the algorithm, we produce a new MFS that is not covered by the MSSs already in L. Then, we find an MSS that covers this new MFS. If no such MSS exists, it means the specification is unrealizable, and so the algorithm emits an error message and terminates. Otherwise, we add this MSS to L. After all the MFS have been covered, we construct a decision list from the obtained list L of MSS in the same way as described in Section IV-B: $f_i(\vec{x})$ is a formula that is satisfied exactly when $MustSat(\vec{x})$ is a subset of the i-th MSS, and the corresponding output assignment $\hat{y}_i$ is a satisfying assignment for that MSS.

Example 3. *Let F, $S_{\vec{x}}$ and $S_{\vec{y}}$ be the same as in Examples 1 and 2. In the first iteration, we generate the MFS $M^1_{\vec{x}} = \{(x_1 \vee \neg x_2)\}$. Then, we expand $Co(M^1_{\vec{x}}) = \{(y_1)\}$ into the MSS $M^1_{\vec{y}} = \{(y_1), (y_1 \vee \neg y_2), (y_2)\}$ and add $M^1_{\vec{y}}$ to L. Note that $M^1_{\vec{y}}$ also covers, besides $M^1_{\vec{x}}$, the MFS $\{(x_2), (\neg x_1 \vee x_2)\}$, and therefore this MFS will not need to be generated. The only remaining MFS is $M^2_{\vec{x}} = \{(x_1 \vee x_2), (x_2)\}$. $M^2_{\vec{y}} = Co(M^2_{\vec{x}}) = \{(\neg y_1), (y_1 \vee \neg y_2)\}$ is already an MSS, so we add it to L. Since all MFS have been covered, the procedure terminates. Note that we did not need to add the MSS $\{(\neg y_1), (y_2)\}$ to L, since no MFS is covered by this MSS. From L, we can now construct a decision list as described earlier:*

$$\texttt{if } (x_1 \vee x_2) \texttt{ then } (y_1 := 1; y_2 := 1)$$
$$\texttt{else if } (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \texttt{ then } (y_1 := 0; y_2 := 0)$$

a) Implementation details: The key steps of Algorithm 1 are the generation of the MFS $M_{\vec{x}}$ in line 3 and the MSS $M_{\vec{y}}$ in line 4. These steps are similar to the input and output analyses in Sections IV-A and IV-B. Since, however, we use communication between the input and output components, we have additional constraints on the MFS and MSS being

generated. At each step the generated MFS must not be covered by the previously-generated MSSs, and the generated MSS must cover the most recently generated MFS.

While generating an arbitrary MFS can be done in polynomial time, we prove that adding the restriction that the MFS must not be covered by a previous MSS makes the MFS generation an NP-complete problem (see extended version of the paper for proper theorem and proof). Therefore, we implement the MFS generation in the following way. First, we use a SAT solver as an NP oracle to find an (not-necessarily maximal) all-falsifiable subset of $S_{\vec{x}}$ not covered by the previous MSSs. Then, we extend this subset to an MFS by iterating over the remaining input clauses and at each step adding to the growing set a clause that does not conflict with the clauses already present in that set. This process of obtaining an MFS from $S_{\vec{x}}$ is easier to implement when we use the conflict graph representation of $S_{\vec{x}}$. Given k previous MSSs $M_1, \ldots, M_k$ and the conflict graph $G = (V, E)$, we use the following SAT query to generate an all-falsifiable subset:

$$\varphi \equiv \bigwedge_{i=1}^{k} \left(\bigvee_{C_j|_{\vec{y}} \in S_{\vec{y}} \setminus M_i} z_j \right) \wedge \bigwedge_{(C_i|_{\vec{x}}, C_j|_{\vec{x}}) \in E} (\neg z_i \vee \neg z_j)$$

We use variable z_i to indicate whether clause $C_i|_{\vec{x}}$ is present in the all-falsifiable subset. The first conjunction encodes that for every previous MSS, the subset must include a clause $C_j|_{\vec{x}}$ not covered by that MSS. The second conjunction expresses that if two clauses conflict with each other, they cannot both be added to the subset. Note that whenever we generate a new MFS, we only need to add extra clauses of the first form to this query, allowing us to employ incremental capabilities of SAT solvers.

After extending the subset produced by the SAT solver to an MFS $M_{\vec{x}}$, we have to generate a new MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$. For that we use a partial MaxSAT solver as an oracle. In a partial MaxSAT problem, some clauses are set as hard clauses and others are set as soft clauses [4]. The solver then returns an assignment that satisfies all hard clauses and the maximum possible number of soft clauses. We call the MaxSAT solver on the set of output clauses $S_{\vec{y}}$, where the clauses in $Co(M_{\vec{x}})$ are set as hard clauses, and all other clauses are set as soft clauses. This way, the MaxSAT solver is guaranteed to return a satisfiable set of clauses containing $Co(M_{\vec{x}})$ and of maximum size. Since a satisfiable subset of maximum size is necessarily maximal, the satisfied clauses returned by the MaxSAT solver is an MSS, as desired.

b) Analysis and Correctness: Since exactly one new MFS and one new MSS are generated at every iteration, the number of iterations in Algorithm 1 is upper bounded by $\min(\#MFS, \#MSS)$. Yet, since Algorithm 1 does not generate redundant MFS and MSS, the number of iterations, and thus the size of the decision list, can be much smaller.

We now formalize and prove the correctness of Algorithm 1.

Lemma 4. *For a realizable specification $F(\vec{x}, \vec{y})$, let $\langle (f_1, \hat{y}_1), \ldots, (f_k, \hat{y}_k) \rangle$ be the decision list produced by Algorithm 1. Then (1) For every $\hat{x}$ such that $f_i(\hat{x}) = 1$, $F(\hat{x}, \hat{y}_i) = 1$; (2) For every $\hat{x}$ there is at least one i such that $f_i(\hat{x}) = 1$.*

Proof. (1) Let $M_{\vec{y}}$ be the i-th MSS generated by the algorithm. Then, by construction, $f_i(\hat{x}) = 1$ iff $MustSat(\hat{x}) \subseteq M_{\vec{y}}$, and $\hat{y}_i$ is a satisfying assignment to $M_{\vec{y}}$. Therefore, if $f_i(\hat{x}) = 1$ then $\hat{y}_i$ satisfies $MustSat(\hat{x})$, and so $(\hat{x}, \hat{y}_i)$ satisfies F.

(2) For every $\hat{x}$, there exists an MFS $M_{\vec{x}}$ such that $Fals(\hat{x}) \subseteq M_{\vec{x}}$. If $M_{\vec{x}}$ was generated by the algorithm, then an MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$ was added to the MSS list. If $M_{\vec{x}}$ was not generated by the algorithm, it must be because there was already a previously generated MSS $M_{\vec{y}}$ that covers $M_{\vec{x}}$. Either way, since $M_{\vec{y}}$ covers $M_{\vec{x}}$ and $Fals(\hat{x}) \subseteq M_{\vec{x}}$, $M_{\vec{y}}$ covers $Fals(\hat{x})$. Therefore, the corresponding f_i in the decision list is such that $f_i(\hat{x}) = 1$. □

From Lemma 4 we obtain the following corollary.

Corollary 3. *Given a realizable specification $F(\vec{x}, \vec{y})$, the decision list produced by Algorithm 1 implements F.*

It is worth noting that if the number of MFS is small as discussed in Section IV-A, then purely enumerating MFS, as in Section IV-A can be theoretically faster than using Algorithm 1. That is because finding an MFS can be done in polynomial time, while Algorithm 1 requires calls to a SAT and MaxSAT solvers. In practice, however, we observed that the Back-and-Forth algorithm often avoids a large number of redundant MFS, which makes up for the extra complexity in generating each MFS. Still, for specifications that are known to have a small number of MFS, restriction to the analysis of the input component as in Section IV-A can be sufficient.

D. Partitioning the Specification into Distinct Output Variables

Some of the cases in the back-and-forth analysis which cause the number of MFS or MSS to be exponential can be simplified by partitioning the specification into sets of clauses that do not share output variables. As an example, consider the specification for the identity function:

$$F(\vec{x}, \vec{y}) = (x_1 \leftrightarrow y_1) \wedge \ldots \wedge (x_k \leftrightarrow y_k)$$

or in a CNF form:

$$F(\vec{x}, \vec{y}) = (\neg x_1 \vee y_1) \wedge (x_1 \vee \neg y_1) \wedge \ldots \wedge (\neg x_k \vee y_k) \wedge (x_k \vee \neg y_k)$$

It is easy to see that both the number of MFS and MSS for this formula are 2^k. Each output variable, however, does not appear in the same clause with other output variables. Therefore, we can consider each pair $(\neg x_i \vee y_i) \wedge (x_i \vee \neg y_i)$ of clauses as a separate specification and synthesize it independently as a decision list of size 2. As such, the total number of MFS and MSS grow linearly with k.

Therefore we propose the following preprocessing step.

1) Given the specification F, construct a graph with a vertex for each clause and an edge between two vertices iff the corresponding clauses share an output variable.

2) Separate the graph into connected components $\mathbb{C}_1, \ldots, \mathbb{C}_k$. Note that the $\mathbb{C}_i$ are completely disjoint in terms of output variables.
3) For every $\mathbb{C}_i$, define a sub-specification F_i by taking only the clauses in F whose corresponding vertex is in $\mathbb{C}_i$.
4) Call Algorithm 1 for each specification F_i. This gives us a decision list D_i for F_i that decides on an assignment for only the output variables in F_i.

Since the F_i have disjoint sets of output variables, every D_i decides on an assignment for a different partition of output variables. Therefore, given an input $\hat{x}$ we can produce a corresponding output $\hat{y}$ by simply evaluating each D_i independently on $\hat{x}$ and combining the results.

V. Experimental Evaluation

In order to evaluate the performance of the Back-and-Forth synthesis algorithm, we ran the algorithm on benchmarks from the 2QBF track of the QBFEVAL'16 QBF-solving competition [25]. This track is composed of QBF benchmarks of the form $\forall\vec{x}.\exists\vec{y}.F(\vec{x},\vec{y})$, where F is a CNF formula. We can see these benchmarks as synthesis problems asking if we can synthesize a Skolem function for the existential variables in terms of the universal variables such that the formula F is satisfied. For this experimental evaluation we used only those benchmarks that are realizable, since adjusting the Back-and-Forth algorithm to handle unrealizable benchmarks is future work. The benchmarks can be classified into seven families: MutexP (7 instances), Qshifter (6 instances), RankingFunctions (49 instances), ReductionFinding (34 instances), SortingNetworks (22 instances), Tree (5 instances) and FixpointDetection (93 instances). Because benchmarks in the same family tend to have similar properties, it makes sense to evaluate performance over each family, rather than over specific instances.

We compared the running time of the Back-and-Forth algorithm on these benchmarks with three state-of-the-art tools that employ different synthesis approaches: the CDCL-based CADET [27], the ROBDD-based RSynth [35], and the CEGAR-based BFSS [1]. Since the Back-and-Forth algorithm, CADET and RSynth are all sequential algorithms, to ensure fair comparison of computational effort, the version of BFSS used was compiled with the MiniSAT SAT solver [10] instead of the parallelized UniGen sampler used in [1]. We leave for future work the exploration of performance of the different tools in a parallel scenario.

Our implementation of the Back-and-Forth algorithm used the Glucose SAT solver [5], based on MiniSAT, and the Open-WBO MaxSAT solver [24]. The implementation also used the partitioning described in Section IV-D. 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. Our algorithm has not been parallelized, so the cluster was solely used to run multiple experiments simultaneously. Each instance had a timeout of 8 hours.

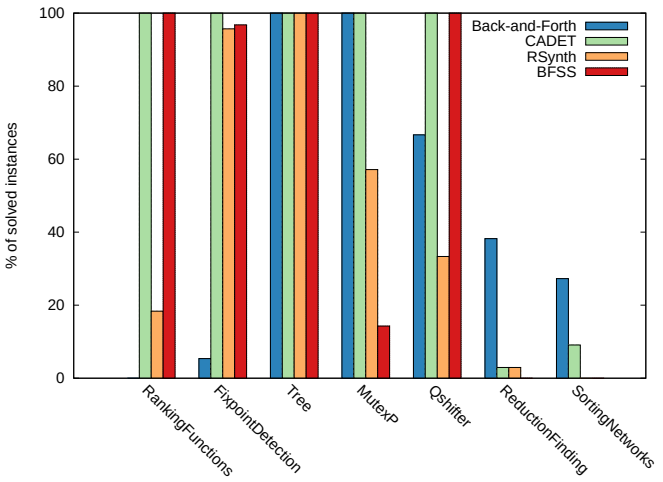

Fig. 1. Percentage of instances solved by each synthesis algorithm for each of the benchmark families.

Figure 1 shows for each family the percentage of instances each tool was able to solve in the time limit. We can divide the results into three parts:

In the RankingFunctions and FixpointDetection families the Back-and-Forth algorithm timed out on almost all instances, only being able to solve the easiest instances of FixpointDetection. CADET, on the other hand, performed very well, being able to solve all instances. RSynth and BFSS also outperformed the Back-and-Forth algorithm, although they did not perform as well as CADET.

The Tree, MutexP, and Qshifter families had almost all instances solved by the Back-and-Forth algorithm in under 45 seconds (except for the two hardest instances of Qshifter, which timed out), in many cases outperforming RSynth or BFSS. Even so, CADET still performed the best in these classes, solving all instances faster than our algorithm.

Lastly, ReductionFinding and SortingNetworks seem to be the most challenging families for existing tools, with CADET only being able to solve two instances in total, RSynth one, and BFSS none. In contrast, our Back-and-Forth algorithm solved 13 cases in ReductionFinding and 6 in SortingNetworks. Furthermore, as can be seen in Figure 2, every instance that was solved by other tools was also solved by the Back-and-Forth algorithm, which was faster by over an order of magnitude.

In summary, the Back-and-Forth algorithm performed competitively in 5 out of 7 families, and was strictly superior in 2 out of 7 families. Due to the difficulty of analyzing CNF formulas, the exact reason why the algorithm performs well in these particular families and not in others remains an open question, to be explored in future work. Still, the results suggest that the Back-and-Forth algorithm can serve as a good complement to modern synthesis tools, performing well exactly in the cases in which these tools struggle the most, and therefore it would be a good candidate for membership in

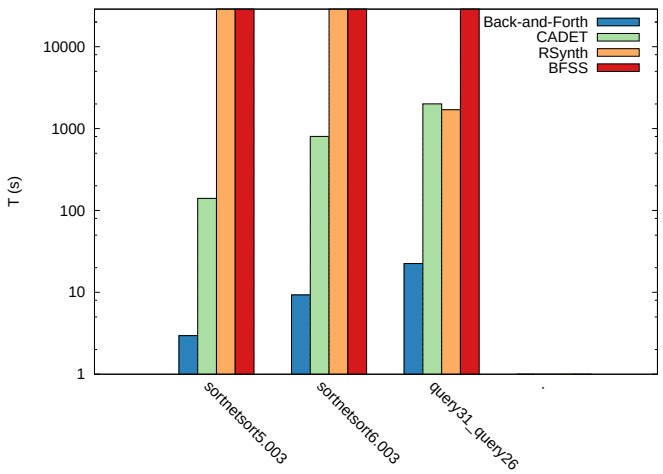

Fig. 2. Running time, in seconds, of each synthesis algorithm on instances of the REDUCTIONFINDING and SORTINGNETWORKS families that were solved by at least one algorithm besides Back-and-Forth. Bars of maximum height indicate the algorithm timed out on the benchmark.

a portfolio of synthesis algorithms.

VI. DISCUSSION

A recurrent observation in recent evaluations [1], [2], [19], [35] of Boolean functional synthesis tools has been that no single tool or algorithm dominates the others in all classes of benchmarks. To build industry-strength Boolean functional solvers, it is therefore inevitable that a portfolio approach be adopted. Since decomposition-based techniques (beyond factored specifications) have not been used in existing tools so far, our original motivation was to develop a decomposition-centric framework for Boolean functional synthesis that complements (rather than dominates) the strengths of existing tools. As our experiments with the Back-and-Forth algorithm show, we have been able to take the first few steps in this direction by successfully solving some classes of benchmarks that state-of-the-art tools choke on. While we have tried to understand features of these benchmarks that make them particularly amenable to our technique, a lot more work remains to be done to elucidate this relation clearly.

Yet another motivation for exploring a decomposition-centric synthesis approach was to be able to generate Skolem functions in a format that lends itself to easy independent validation by domain experts. Interestingly, despite the singular importance of this aspect, it has been largely ignored by existing Boolean functional synthesis tools, most of which construct a circuit representation of the function using an acyclic-graph data structure such as an ROBDD or an And-Inverter Graph. While these are known to be efficient representations of Boolean functions, they are not amenable to easy validation by a domain expert, especially when their sizes are large, often requiring a satisfiability solver to check that the generated Skolem functions indeed satisfy the specifications. Synthesizing functions as decision lists is a natural and well-studied choice for meeting this objective. Along with each decision in the decision list, we can also identify the clauses that contribute to the generation of the outputs (these are clauses whose input components are falsified by the decision), thereby providing clues about which part of the specification is responsible for the outputs generated in a particular branch of the decision-list representation. Our work shows that decomposition-based techniques lend themselves easily to such representations.

In order to be consistent with performance comparison experiments reported in the literature, all specifications used in our evaluation were prenex CNF (PCNF) formulas taken from the QBFEVAL'16 benchmark suite. While this certainly presents challenging instances of Boolean functional synthesis, PCNF is not a natural choice of representing specifications in several important application areas. For example, the industry standard (IEC 1131-3) for reactive programs for programmable logic controllers (PLC) includes a set of languages that allow the user to specify combinations of outputs based on different combinations of input conditions. The same is also true in the specification of several bus protocols like the VME Bus or AMBA Bus. Scenario-based specifications such as these are much more amenable to our decomposition-based approach, since there is a natural separation of input and output components of the specification. In addition, with such specifications, it is meaningful to analyze the structure of dependence between the input and output components, and exploit structural properties (viz. the size of the MIS in the conflict graph as explained in Section IV) in synthesis. We believe that as we look beyond PCNF representations of specifications, techniques like those presented in this paper will be even more useful in a portfolio approach to synthesis.

In our experimental evaluation, we chose CADET as a representative of the state-of-the-art on Boolean synthesis stemming from the QBF community. This is due to its focus on 2QBF (which suffices for Boolean synthesis of realizable specifications) and its performance on recent QBFEVAL competitions. Another certifying QBF solver, CAQE [28], uses techniques that are similar to the clause splitting used in our algorithm. But CAQE targets QBF instances with arbitrary quantifier alternation, requiring additional mechanisms for handling these cases, and furthermore does not perform the same analysis as here, based on MFS and MSS. Due to their similarities, it would be interesting to perform a comparison between the two algorithms in the future.

Finally, the techniques presented in this work are clearly not the only ways to achieve synthesis via decomposition, and there exists scope for significant innovation and creativity, both in the manner in which a specification is decomposed, and in the way the decomposition is exploited to arrive at an efficient synthesis algorithm. One example lies in identifying algorithms for sequential decomposition, as presented in [11], which are applicable to a synthesis context. In summary, synthesis based on input-output decomposition presents uncharted territory that deserves systematic exploration in order to complement the strengths of existing synthesis tools.

REFERENCES

[1] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah. What's Hard About Boolean Functional Synthesis? In *Computer Aided Verification - 30th International Conference, CAV 2018*, pages 251–269, 2018.

[2] 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*, pages 337–353, 2017.

[3] R. Alur, P. Madhusudan, and W. Nam. Symbolic Computational Techniques for Solving Games. *STTT*, 7(2):118–128, 2005.

[4] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference*, pages 427–440, 2009.

[5] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 399–404, 2009.

[6] V. Balabanov and J.-H. R. Jiang. Unified QBF Certification and Its Applications. *Form. Methods Syst. Des.*, 41(1):45–65, Aug. 2012.

[7] V. Balabanov and J. R. Jiang. Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In *Computer Aided Verification - 23rd International Conference, CAV 2011*, pages 149–164, 2011.

[8] V. Balabanov, M. Widl, and J. R. Jiang. QBF Resolution Systems and Their Proof Complexities. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 154–169, 2014.

[9] G. Boole. *The Mathematical Analysis of Logic*. Philosophical Library, 1847.

[10] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing - SAT 2003 - 6th International Conference*, pages 502–518, 2003.

[11] D. Fried, A. Legay, J. Ouaknine, and M. Y. Vardi. Sequential Relational Decomposition. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 432–441, 2018.

[12] D. Fried, L. M. Tabajara, and M. Y. Vardi. BDD-Based Boolean Functional Synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016*, pages 402–421, 2016.

[13] R. Ganian and S. Szeider. New Width Parameters for Model Counting. In *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference*, pages 38–52, 2017.

[14] M. Heule, M. Seidl, and A. Biere. Efficient Extraction of Skolem Functions from QRAT Proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2014*, pages 107–114, 2014.

[15] A. Ignatiev, A. Morgado, J. Planes, and J. Marques-Silva. Maximal Falsifiability - Definitions, Algorithms, and Applications. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19*, pages 439–456, 2013.

[16] S. Jacobs, R. Bloem, R. Brenguier, R. Könighofer, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker. The Second Reactive Synthesis Competition (SYNTCOMP 2015). In *Proceedings Fourth Workshop on Synthesis, SYNT 2015*, pages 27–57, 2015.

[17] J. R. Jiang. Quantifier Elimination via Functional Composition. In *Computer Aided Verification, 21st International Conference, CAV 2009*, pages 383–397, 2009.

[18] S. Jo, T. Matsumoto, and M. Fujita. SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions. In *Proceedings of the 2012 IEEE 21st Asian Test Symposium*, ATS '12, pages 19–24. IEEE Computer Society, 2012.

[19] 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*, pages 73–80, 2015.

[20] J. H. Kukula and T. R. Shiple. Building Circuits from Relations. In *Computer Aided Verification, 12th International Conference, CAV 2000*, pages 113–123, 2000.

[21] C. M. Li and F. Manyà. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, pages 613–631. 2009.

[22] L. Lowenheim. Über die Auflösung von Gleichungen in Logischen Gebietkalkul. *Math. Ann.*, 68:169–207, 1910.

[23] E. Macii, G. Odasso, and M. Poncino. Comparing Different Boolean Unification Algorithms. In *Proceedings of 32nd Asilomar Conference on Signals, Systems and Computers*, pages 17–29, 2006.

[24] R. Martins, V. M. Manquinho, and I. Lynce. Open-WBO: A Modular MaxSAT Solver. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 438–445, 2014.

[25] M. Narizzano, L. Pulina, and A. Tacchella. The QBFEVAL web portal. In *Logics in Artificial Intelligence*, pages 494–497. Springer Berlin Heidelberg, 2006.

[26] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Bierre. Resolution-Based Certificate Extraction for QBF - (Tool Presentation). In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, pages 430–435, 2012.

[27] M. N. Rabe and S. A. Seshia. Incremental Determinization. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference*, pages 375–392, 2016.

[28] M. N. Rabe and L. Tentrup. CAQE: A Certifying QBF Solver. In *Formal Methods in Computer-Aided Design, FMCAD 2015*, pages 136–143, 2015.

[29] R. L. Rivest. Learning Decision Lists. *Machine Learning*, 2(3):229–246, 1987.

[30] J. P. M. Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, pages 131–153. 2009.

[31] A. Solar-Lezama. Program Sketching. *STTT*, 15(5-6):475–495, 2013.

[32] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 281–294, 2005.

[33] S. Srivastava, S. Gulwani, and J. S. Foster. Template-Based Program Verification and Program Synthesis. *STTT*, 15(5-6):497–518, 2013.

[34] L. M. Tabajara. BDD-Based Boolean Synthesis. Master's thesis, Rice University, 2018.

[35] L. M. Tabajara and M. Y. Vardi. Factored Boolean Functional Synthesis. In *Formal Methods in Computer Aided Design, FMCAD 2017*, pages 124–131, 2017.

[36] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi. Symbolic LTLf Synthesis. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*, pages 1362–1369, 2017.