Functional synthesis via input–output separation

Supratik Chakraborty1 · Dror Fried2 · Lucas M. Tabajara3 · Moshe Y. Vardi3

Received: 25 June 2020 / Accepted: 25 January 2023 / Published online: 9 March 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

Boolean functional synthesis is the process of constructing a Boolean function from a Boolean specification that relates input and output variables. Despite recent developments in synthesis algorithms, Boolean functional synthesis remains a challenging problem even when state-of-the-art techniques are used for decomposing the specification. In this work, we present a new decomposition approach that decomposes the specification into separate input and output components. To begin with, we adapt the notion of “sequential decomposition” and present a framework that allows the input and output components to be independently synthesized and then re-composed to yield an implementation of the overall specification. Although theoretically appealing, this approach suffers from some practical drawbacks, as evidenced by our experiments. This motivates us to propose a relaxed approach to synthesis by decomposition. In the relaxed approach, we start with a specification given as a conjunctive normal form (CNF) formula, and obtain a decomposition of the specification by alternatingly analyzing the input and output components repeatedly. We also exploit specific properties of these components to ultimately implement the overall specification. We prove that if the input component of the CNF specification has specific structural properties, our approach can achieve synthesis in polynomial time. We also show by experimental evaluations that our algorithm performs well in practice on instances that are challenging for existing state-of-the-art tools. Thus, our decomposition-based synthesis approach serves as a good complement to other state-of-the-art techniques in a portfolio approach to Boolean functional synthesis.

Extended version of a paper published in FMCAD 2018 [13]. The datasets generated during and/or analysed during the current study are available at https://github.com/lucasmt/BackAndForthSynthesis.

Lucas M. Tabajara
lucasmt@rice.edu

Supratik Chakraborty
supratik@cse.iitb.ac.in

Dror Fried
dfried@openu.ac.il

Moshe Y. Vardi
vardi@cs.rice.edu

1 IIT Bombay, Mumbai, India
2 The Open University of Israel, Ra’anana, Israel
3 Rice University, Houston, USA

Springer
1 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 [3, 16, 24, 43]. This problem has been explored in a number of settings including circuit design [26], Quantified Boolean Formula (QBF) solving [34], and reactive synthesis [44], and several tools have been developed for solving the problem. Nevertheless, scalability of Boolean functional synthesis remains a concern as the number of variables and size of the formula grows. This is not surprising since even deciding if a Boolean specification is realizable is equivalent to solving a QBF of the form $\forall \exists$, and therefore coNPNP-complete ($\Pi_2^p$-complete) [2, 41].

A standard practice for addressing the problem of scalability is to decompose the given specification into smaller and simpler sub-specifications, and synthesize each component separately [3, 24, 43]. The most common form of such decomposition, called factorization, represents the formula as a conjunction of constraints, in which each conjunct can be seen as a sub-specification [24, 43]. The main challenge in this approach is that most factors cannot be synthesized independently due to the dependencies created by shared input and output variables. In order to overcome this challenge, one can either merge factors that share variables [43] or perform additional computations to repair the functions synthesized independently from different sub-specifications [24].

In this work, we propose an alternative decomposition approach that builds on the observation that variables in any specification are naturally separated into input and output variables. Our inspiration comes from the notion of sequential relational decomposition [15], in which a relation is decomposed into two components by introducing an intermediate domain. In contrast to decomposition by factorization, sequential relational decomposition allows the two components to be synthesized completely independently, and then re-composed to yield an overall solution. In order to adapt this idea to synthesis, we first define the notion of “good decomposition” of a specification and give a specific example of such a decomposition, called “CNF decomposition”, for specifications presented as Conjunctive Normal Form, or CNF, formulas. Although theoretically sound and appealing, this approach to synthesis turns out to be too naive in practice. In particular, the synthesis of one component often ignores information from the other component that can be exploited for significant simplifications, rendering an implementation of this idea unviable in practice for large specifications.

To make the approach more practical, we propose a more relaxed notion of decomposition for specifications given as CNF formulas. In this approach, every clause of the CNF specification is split into an input and an output clause, and the analyses of the input/output components “cooperate” in an iterative manner to synthesize a Boolean function for the overall specification. This contrasts with the independent synthesis of input and output components in the idealized “good decomposition” approach. We describe a novel synthesis algorithm for CNF formulas, called the “Back-and-Forth” algorithm, that shares information back and forth between the two components to guide the synthesis. Synthesis using input-components requires finding maximal falsifiable subsets of clauses in a suitably defined CNF formula, while synthesis using output-components requires finding maximal satisfiable subsets of clauses. These problems are naturally solved using propositional...
satisfiability (SAT) solvers and maximum satisfiability [7] (MaxSAT) solvers respectively. Indeed, our approach leverages the tremendous progress achieved in both SAT and MaxSAT solving [28, 37] over the past two decades. A notable consequence of our method is that the number of invocations of a SAT solver during the input-component based synthesis steps is dependent on well-defined structural properties of the input component. Hence, for specifications with specific input structures, synthesis can be achieved in polynomial time relative to an \( \mathbb{NP} \)-oracle. An additional advantage of our algorithm is that it constructs the synthesized function as a decision list [36]. Compared to other data structures for representing Boolean functions, such as Reduced Ordered Binary Decision Diagrams (ROBDDs) or And-Inverter-Graphs (AIGs), decision lists have significant benefits in terms of understandability by humans, allowing domain specialists to validate and analyze their behavior (see discussion in Sect. 9 for more details).

We experimentally evaluate the “Back-and-Forth” algorithm on a suite of standard Boolean functional synthesis benchmarks, comparing its performance with that of state-of-the-art synthesis tools. Our experiments show that although these other tools perform well on many families of benchmarks (often out-performing the “Back-and-Forth” algorithm), our algorithm has complementary strengths and is able to handle several classes of benchmarks that state-of-the-art tools have difficulty synthesizing. Thus, our work adds a new algorithm with complementary strengths to the portfolio of existing Boolean functional synthesis algorithms.

2 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 [10, 19, 24]. Boole [11] and Lowenheim [29] 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 [30]. The close relation between Skolem functions and proof objects in specialized QBF proof systems has been explored in [10, 19]. One of the earliest applications of Boolean functional synthesis has been logic synthesis – see [42] for a survey. More recently, Boolean functional synthesis has found applications in diverse areas such as temporal strategy synthesis [4, 21, 44], certified QBF solving [8, 9, 33, 35], automated program synthesis [38, 40], circuit repair and debugging [23], and the like. This has resulted in a new generation of Boolean functional synthesis tools, cf. [2, 3, 16, 19, 24, 34, 35, 43], 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 [19] 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 [34] 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 [37].

In [16], the composition-based synthesis approach of [22] 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 [43] to work with factored specifications represented as implicitly conjoined ROBDDs. Counterexample-guided abstraction refinement (CEGAR)-based techniques that use modern SAT solvers as black boxes [2, 3, 24] have recently been shown to scale well on several classes of large benchmarks. The idea behind these techniques is to start with initial candidate Skolem functions that are either obtained by careful analysis of the specification representation or by machine learning techniques. Subsequently, a SAT solver is used to test if the candidate functions are indeed correct Skolem functions. A satisfying assignment returned by the solver provides a counterexample to the correctness of the candidate Skolem functions, and can be used to iteratively refine the candidates. In [1], it is shown that representing the specification in a special form called Synthesis Negation Normal Form (SynNNF) allows one to synthesize Skolem functions in time polynomial in the size of the SynNNF representation. Recently, machine learning, constrained sampling and automated reasoning have been used in a synergistic manner to solve Boolean functional synthesis [18]. The resulting algorithm leverages recent advances in each of these three well-studied domains to achieve significant performance improvements in synthesizing a large number of benchmarks. 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 [39, 40].

Both ROBDD and CEGAR-based approaches have been used along with factorization (or conjunctive decomposition) to improve the performance of synthesis algorithms in earlier work [24, 43]. Effectively, these approaches try to analyze every conjunct of a conjunctive specification separately to obtain a part of the overall solution. The main drawback, however, is that dependencies between conjuncts arising from shared variables limit how much each conjunct can be analyzed truly independently of the others. To address this problem, we must either partially combine conjuncts, as in [43], or go through a process of refinement of candidate Skolem functions obtained from individual conjuncts, as in [24]. Unfortunately, both these solutions can cause the overall performance of synthesis to degrade even when the individual conjuncts are relatively small and easy to synthesize independently. This motivates our search for alternative notions of decomposition for synthesis problems as we show in the paper.

As is clear from the above discussion, 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.

3 Preliminaries

3.1 Boolean formulas

A Boolean formula $F(\vec{w})$ is in conjunctive normal form (CNF) if $F$ is a conjunction of clauses $C_1 \land \ldots \land C_k$, where every clause $C_i$ is a disjunction of literals (a Boolean variable or its negation). A Boolean CNF formula is in 3CNF if every clause has at most 3 literals.
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 [20]. The problem of satisfiability or SAT, is whether a given Boolean formula has a satisfying assignment. MaxSAT is the problem of finding an assignment that satisfies the most clauses of a given CNF formula. A **quantified Boolean formula** (QBF) is a Boolean formula in which some or all of its variables are quantified. We assume without loss of generality that a QBF formula appears in **Negated Normal Form** (NNF) in which all the quantifiers precede the formula itself.

### 3.2 Boolean functional synthesis

A specification for the Boolean functional synthesis problem is a 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 $\text{Dom}(F) = \{ \hat{x} \mid \exists \hat{y}. (F(\hat{x}, \hat{y}) = 1) \}$ and $\text{Img}(F) = \{ \hat{y} \mid \exists \hat{x}. (F(\hat{x}, \hat{y}) = 1) \}$ the domain and image of the relation represented by $F$. We also use $\text{Img}_x(F) = \{ \hat{y} \mid F(\hat{x}, \hat{y}) = 1 \}$ to denote the image of a specific element $\hat{x} \in X$. If $\text{Dom}(F) = X$, then we say that $F$ is **realizable**; otherwise $F$ is **unrealizable**.

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

We say that a partial function $g : X \rightarrow Y$ implements a relation $F \subseteq X \times Y$ if for every $\hat{x} \in \text{Dom}(F)$ we have $(\hat{x}, g(\hat{x})) \in F$. 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 functional 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 [16, 24].

### 3.3 Decision lists

We choose to use **decision lists** to represent Skolem functions. Our choice is motivated by the fact that every Boolean function can be represented by a decision list [36], and that decision lists often serve as explainable/interpretable models of knowledge (see [27], for example). 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 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,
4 On synthesis via sequential decomposition

As a first attempt to synthesize by decomposition, we explore the use of sequential decomposition, originally described in [15]. In sequential decomposition, the specification $F$ is split into an input part and an output part by adding fresh intermediate variables $\vec{z} = (z_1, \ldots, z_k)$ that define a domain $Z$ for communicating between the input domain $X$ and the output domain $Y$. The intermediate domain $Z$ must be introduced in such a way that it preserves all and only those input-output pairs that satisfy $F$. In addition, as described in [15], we would like each of the input and output parts to be synthesized independently, such that these can later be re-composed into an implementation for the entire specification. With this in mind, we define the following.

**Definition 1** Let $F(x, y), F_1(x, z), F_2(z, y)$ be Boolean formulas. Then $(F_1, F_2)$ is called a good decomposition of $F$ if

1. $F(x, y) \Leftrightarrow \exists \vec{z}(F_1(x, z) \land F_2(z, y))$, and
2. for every input $\hat{x} \in \text{Dom}(F)$, $\text{Img}_{\hat{x}}(F_1) \subseteq \text{Dom}(F_2)$.

Condition (1) in Definition 1 guarantees that for every input assignment $\hat{x}$ and output assignment $\hat{y}$, $(\hat{x}, \hat{y})$ satisfies $F$ if and only if there exists an intermediate assignment $\vec{z}$ such that $(\hat{x}, \vec{z})$ satisfies $F_1$ and $(\vec{z}, \hat{y})$ satisfies $F_2$. Condition (2) guarantees that for all implementations $g_1$ of $F_1$ and $g_2$ of $F_2$, their composition $g_2 \circ g_1$ is well-defined and is an implementation of $F$. Such a decomposition attains a complete separation of the inputs and outputs of $F$, in the sense that no direct knowledge of the output variables is necessary to synthesize $F_1$, nor of the input variables to synthesize $F_2$. Note that since Condition (2) is not a logical consequence of Condition (1), both conditions are necessary in the definition of a good decomposition.

**Theorem 1** Let $F(x, y)$ be a specification with the input variables $\vec{x}$ and the output variables $\vec{y}$. If $F_1(x, z)$ and $F_2(z, y)$ form a good decomposition of $F$, then for every implementation $g_1$ of $F_1$ and $g_2$ of $F_2$, $g_2 \circ g_1$ implements $F$.

**Proof** Since $F(x, y) \Leftrightarrow \exists \vec{z}(F_1(x, z) \land F_2(z, y))$, we have that if $(x, z)$ satisfies $F_1$ and $(z, y)$ satisfies $F_2$ then $(\hat{x}, \hat{y})$ satisfies $F$. Let $g_1 : X \rightarrow Z$ and $g_2 : Z \rightarrow Y$ be implementations...
of $F_1$ and $F_2$, respectively. Let $\hat{x} \in \text{Dom}(F)$. Since $(F_1, F_2)$ is a good decomposition of $F$, $\hat{x} \in \text{Dom}(F_1)$. Since $g_1$ is an implementation of $F_1$, $(\hat{x}, g_1(\hat{x}))$ satisfies $F_1$. Furthermore, since $\hat{x} \in \text{Dom}(F)$, $\text{Img}(g_1) \subseteq \text{Dom}(F_2)$. Then, since $g_1(\hat{x}) \in \text{Img}(g_1)$, $g_2(g_1(\hat{x})) \in \text{Dom}(F_2)$. Therefore, $(\hat{x}, g_1(\hat{x}))$ satisfies $F_2$. Since $(\hat{x}, g_1(\hat{x}))$ satisfies $F_1$ and $(\hat{x}, g_2(g_1(\hat{x})))$ satisfies $F_2$, then $(\hat{x}, g_2(g_1(\hat{x})))$ satisfies $F$. Therefore, $g_2 \circ g_1$ is an implementation of $F$. \hfill \Box

Theorem 1 suggests a decomposition framework that allows each component to be independently synthesized in a divide-and-conquer manner: Given a specification $F$, first find a good decomposition of $F$ into $(F_1, F_2)$. Then synthesize the sub-components as $g_1$ and $g_2$ respectively, and compose then to achieve an overall synthesis solution $g$ for $F$. Note that there can be many such good decompositions for a given specification, especially when the size of the intermediate domain is not constrained. Below, we describe a specific method that can be applied to all specifications in CNF, and always yields a good decomposition.

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 $C_i^x$ denote the $x$-part of clause $C_i$, i.e. the disjunction of all $x$ literals in $C_i$. Similarly, let $C_i^y$ be the $y$-part of clause $C_i$, i.e. the disjunction of all $y$ literals in $C_i$. Note that $C_i \equiv (C_i^x \cup C_i^y) \equiv (\neg C_i^x \rightarrow C_i^y)$.

The CNF decomposition of $F = C_1 \wedge \ldots \wedge C_k$ is a pair $(F_1, F_2)$ where

\begin{align}
F_1(\vec{x}, \vec{z}) &\equiv \bigwedge_{i=1}^{k} (\neg C_i^x \leftrightarrow z_i) \tag{1} \\
F_2(\vec{z}, \vec{y}) &\equiv \bigwedge_{i=1}^{k} (\neg z_i \vee C_i^y) \equiv \bigwedge_{i=1}^{k} (z_i \rightarrow C_i^y) \tag{2}
\end{align}

We call $F_1$ the input component and $F_2$ the output component arising out of the decomposition. The idea behind the CNF decomposition is to “mark” clauses that are rendered true/false by an assignment of input variables. The clauses “marked” false must then be rendered true by the assignment of output variables. This leads to a natural and simple decomposition. Furthermore, the form of $F_1$ already suggests a unique function from $\vec{x}$ to $\vec{z}$, and hence the synthesis of $g_1$ from $F_1$ is trivial – simply assign every $z_i$ to $\neg C_i^x$. Intuitively, this decomposition works by grouping assignments of input variables $\vec{x}$ into individual $z_i$ variables. Specifically, note that $z_i$ is only assigned true when $C_i^x$ is rendered false by the assignment of input variables, and we must assign values to the output variables such that $C_i^y$ is satisfied. As such, we abstract away all the assignments that make the same $z_i$ variable true. Therefore, we only need to concern ourselves with synthesizing a Skolem function $g_2$ from $F_2$.

We now prove that the CNF decomposition meets the criteria for a good decomposition, as given in Theorem 1.

**Theorem 2** If $F_1$ and $F_2$ are given by the CNF decomposition of a CNF formula $F$, then $(F_1, F_2)$ is a good decomposition of $F$.

**Proof** We first prove that $F(\vec{x}, \vec{y}) \iff \exists \vec{z}. F_1(\vec{x}, \vec{z}) \wedge F_2(\vec{z}, \vec{y})$. 

\begin{center}
\textcopyright Springer
\end{center}
\[ \exists \vec{z}. F_1(\vec{x}, \vec{z}) \land F_2(\vec{z}, \vec{y}) \equiv \exists \vec{z}. \bigwedge_{i=1}^{k} (\neg C_i |_{\vec{x}} \leftrightarrow z_i) \land \bigwedge_{i=1}^{k} (z_i \rightarrow C_i |_{\vec{y}}) \]

\[ \equiv \bigwedge_{i=1}^{k} (\exists z_i. (\neg C_i |_{\vec{x}} \leftrightarrow z_i) \land (z_i \rightarrow C_i |_{\vec{y}})) \]

\[ \equiv \bigwedge_{i=1}^{k} (\neg C_i |_{\vec{x}} \rightarrow C_i |_{\vec{y}}) \]

\[ \equiv \bigwedge_{i=1}^{k} (C_i |_{\vec{x}} \lor C_i |_{\vec{y}}) \]

\[ \equiv \bigwedge_{i=1}^{k} C_i \equiv F(\vec{x}, \vec{y}) \]

Next, we prove that \( \text{Img}_F(F_1) \subseteq \text{Dom}(F_2) \) for every input \( \vec{x} \in \text{Dom}(F) \). Assume \( \vec{x} \in \text{Dom}(F) \), that is, there exists \( \vec{y} \) such that \( F(\vec{x}, \vec{y}) = 1 \). Since \( F_1(\vec{x}, \vec{z}) = (\neg C_1 |_{\vec{x}} \leftrightarrow z_1) \land \ldots \land (\neg C_k |_{\vec{x}} \leftrightarrow z_k) \), we have that each \( z_i \) is true if and only if \( \neg C_i |_{\vec{x}} \) is true. Hence, there is a unique \( \vec{z} \) that satisfies \( F_1 \) for \( \vec{x} \). This \( \vec{z} \) is the only element of \( \text{Img}_F(F_1) \), therefore we only need to prove that \( \vec{z} \in \text{Dom}(F_2) \). But by construction of \( F_2 \), the same \( \vec{y} \) that satisfies \( F \) for \( \vec{x} \) also satisfies \( F_2 \) for \( \vec{z} \). Therefore, \( \vec{z} \in \text{Dom}(F_2) \). \( \square \)

In order to evaluate the practicality of using CNF decomposition for synthesis, we compared the performance of a representative synthesis tool using \( F_2 \) and \( F \), respectively, as the relational specification. Since the CNF decomposition and synthesis of \( F_1 \) are straightforward, and can be performed in linear time, we focused on comparing synthesis from \( F_2 \) and from \( F \).

An important detail to keep in mind is that even if \( F \) is realizable, the component \( F_2 \) produced by CNF decomposition may not be realizable. This is because the different \( C_i |_{\vec{x}} \)'s are not independent in general, and hence the set of simultaneous valuations of \( C_i |_{\vec{x}} \)'s may be constrained. In contrast, the \( z_i \)'s are completely independent inputs in \( F_2 \), and can assume arbitrary combinations of values. This may render \( F_2 \) not realizable even when \( F \) is realizable. In view of this, when synthesizing Skolem functions from \( F_2 \), we must use a synthesis tool, viz. RSynth [16], that can handle unrealizable specifications. In this way, even if \( F_2 \) is unrealizable, the tool returns a partial function \( g_2 \). If \( F \) is realizable, however, this partial function becomes a total function implementing \( F \) when composed with the function \( g_1 \) defined by \( F_1 \).

Our experiments with RSynth were run on the same benchmarks and on the same testbed as used in all our subsequent experiments. We defer a detailed description of the benchmarks and of the experimental setup to Sect. 8. For now, we simply present a comparison of the percentage of benchmarks that RSynth was able to synthesize from \( F \) and from \( F_2 \), respectively, for different interesting sub-classes of benchmarks within a timeout of 8 h. In Fig. 1, the different sub-classes of benchmarks are shown along the horizontal axis. As can be seen, in most cases the performance of RSynth worsened when trying to synthesize from \( F_2 \) after using CNF decomposition. Specifically, RSynth succeeded for a much smaller subset of benchmarks in most sub-classes when synthesizing from \( F_2 \) vis-a-vis when RSynth was used directly on \( F \). Furthermore, instances that could be solved took significantly longer in general, sometimes taking up to 10,000 times more time. The only exception was the Qshifter sub-class of benchmarks, for which CNF decomposition based
synthesis scaled significantly better than directly solving from $F$, and allowed solving all the instances.

Our conclusion from these experiments is that although theoretically appealing, CNF decomposition is not a practically viable synthesis approach, and in many cases can worsen synthesis performance considerably. We conjecture that this is largely because the synthesis of $F_2$ has no access to information about $F_1$, possibly performing superfluous work to produce correct values for assignments of $\vec{z}$ that can never be produced by the input component. Such reasoning on the practical use of the sequential decomposition approach may be generalized not only to the specific CNF decomposition approach outlined above, but to more general divide-and-conquer based decomposition frameworks. This motivates us to propose a new approach in which we build upon the CNF decomposition idea by allowing information to be shared between the synthesis processes for the input and output components.

5 Synthesis via input–output separation

The drawbacks of CNF-decomposition-based synthesis motivate us to present a more relaxed approach that is based on the same core idea of separating clauses into input and output components, but unlike in CNF decomposition, we now allow information to be shared between the two components when synthesizing them. Although such information sharing goes counter to independent synthesis of the input and output components,
our experiments show that this can be significantly beneficial in practice. We start by separating every clause of a given CNF specification into an input and an output clause, as in CNF decomposition. We then show how a decision-list based implementation of the specification can be obtained by enumerating maximally falsifiable subsets (MFS) of the input clauses. In a similar manner, a decision-list based implementation can also be obtained by enumerating maximally satisfiable subsets (MSS) of the output clauses. Interestingly, these two approaches to synthesis can benefit from each other, suggesting an algorithm that alternates between the two. Specifically, the MFS of input clauses can be used to filter out MSS of output clauses, while the MSS of output clauses can in turn be used to cover multiple MFS of input clauses, without the need to enumerate them.

Given a CNF formula $F(\vec{x}, \vec{y})$, we again assume $F(\vec{x}, \vec{y}) = \bigwedge_{i=1}^{k} C_i$, where $C_1, \ldots, C_k$ are clauses over $\vec{x}$ and $\vec{y}$, and $C_i|_x$ and $C_i|_y$ are as defined before. We call $S_x = \{C_i|_x \text{ s.t. } C_i \text{ is a clause in } F\}$ the set of input clauses of the specification. Similarly, $S_y = \{C_i|_y \text{ s.t. } C_i \text{ is a clause in } F\}$ is called the set of output clauses of $F$. In the following sections, we first describe how to synthesize Skolem functions as decision-lists into a single synthesis algorithm that alternates between the two approaches.

### 5.1 Analysis of the input component

For purposes of this subsection, we assume that the specification $F$ is realizable. Consider an assignment $\hat{x}$ to the input variables $\vec{x}$. Let $\text{Fals}(\hat{x}) = \{C_i|_x \in S_x \text{ s.t. } C_i|_x(\hat{x}) = 0\}$ be the subset of input clauses that $\hat{x}$ falsifies. For every subset $S'_x \subseteq S_x$ of input clauses, let $\text{Co}(S'_x) = \{C_i|_y \in S_y \text{ s.t. } C_i|_x \in S'_x\}$ be the corresponding set of output clauses. We define $\text{MustSat}(\hat{x})$ to be the set $\text{Co}(\text{Fals}(\hat{x}))$. Since $C_i \equiv (\neg C_i|_x \rightarrow C_i|_y)$ for every $i$, every output clause in $\text{MustSat}(\hat{x})$ must be satisfied by the assignment to output variables $\vec{y}$ if we are to have $F$ evaluate to $\text{True}$ with the input variables $\vec{x}$ set to $\hat{x}$.

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

**Lemma 1** Let $M_\hat{x}$ be an MFS of $S_x$, and let $\hat{y}$ be an assignment that satisfies $\text{Co}(M_\hat{x})$. Then:

1. For every assignment $\hat{x}$ such that $\text{Fals}(\hat{x}) \subseteq M_\hat{x}$, the assignment $(\hat{x}, \hat{y})$ satisfies $F(\vec{x}, \vec{y})$;
2. There is no assignment $\hat{x}$ such that $\text{Fals}(\hat{x}) \supset M_\hat{x}$.

**Proof**

(1) For every clause $C_i|_x \in \text{Fals}(\hat{x})$, since $C_i|_x \in M_\hat{x}$, we have that $C_i|_x$ is in $\text{Co}(M_\hat{x})$ 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 $M_\hat{x}$ 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 count of MFS of $S_x$, where each $f_i$ in the decision list is of size linear in the size of the specification.
Proof We construct a decision list of the form if \( f_1(\vec{x}) \) then \( \vec{y}_1 \) else if \( f_2(\vec{x}) \) then \( \vec{y}_2 \) else ... else \( \vec{y}_s \), where \( s \) is the number of MFS of \( S_\vec{x} \). In this list, we define \( f_j(\vec{x}) \) to be the conjunction of all input clauses \( C_j|_{\vec{x}} \) not contained in the \( i \)-th MFS \( M_i^\vec{\ell} \), that is,

\[
f_j(\vec{x}) = \bigwedge_{C_j|_{\vec{x}} \in S_\vec{x} \setminus M_i^\vec{\ell}} C_j|_{\vec{x}}.
\]

Furthermore, we define \( \vec{y} \) to be a satisfying assignment of \( \text{Co}(M_i^\vec{\ell}) \). Note that \( f_j \) is satisfied exactly by those input assignments \( \vec{x} \) that satisfy every clause \( C_i \) such that \( C_i|_{\vec{x}} \notin M_i^\vec{\ell} \). This means that \( \text{Fals}(\vec{x}) \subseteq M_i^\vec{\ell} \). Meanwhile, \( \vec{y} \) satisfies every clause \( C_j \) such that \( C_j|_{\vec{x}} \in M_i^\vec{\ell} \). As a consequence, if \( f_j(\vec{x}) = 1 \) then \( (\vec{x}, \vec{y}) \) satisfies every clause in the CNF, and therefore satisfies \( F \).

Since for every \( \vec{x} \) there is an MFS \( M_i^\vec{\ell} \) of \( S_\vec{x} \) such that \( \text{Fals}(\vec{x}) \subseteq M_i^\vec{\ell} \), therefore every \( \vec{x} \) satisfies some \( f_i \). It follows that the decision list described above implements \( F \). \( \square \)

Example 1 Let \( F(x_1,x_2,y_1,y_2) = (x_1 \lor \neg x_2 \lor y_1) \land (x_1 \lor x_2 \lor \neg y_1) \land (x_2 \lor y_1 \lor \neg y_2) \land (\neg x_1 \lor x_2 \lor y_2) \). We first construct input clauses \( S_\vec{x} = \{(x_1 \lor \neg x_2), (x_1 \lor x_2), (\neg x_1 \lor x_2)\} \) and output clauses \( S_\vec{y} = \{(y_1), (\neg y_1), (y_1 \lor \neg y_2), (y_2)\} \). \( S_\vec{\ell} \) has three MFS: \( \{x_1 \lor \neg x_2\} \), \( \{x_1 \lor x_2, (x_2)\} \) and \( \{x_2\}, (\neg x_1 \lor x_2) \). 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:

\[
\begin{align*}
\text{if}(x_1 \lor x_2) \land (x_2) \land (\neg x_1 \lor x_2) \text{then}(y_1 := 1; y_2 := 0) \\
\text{elseif}(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \text{then}(y_1 := 0; y_2 := 0) \\
\text{elseif}(x_1 \lor \neg x_2) \land (x_1 \lor x_2) \text{then}(y_1 := 1; y_2 := 1)
\end{align*}
\]

Recall that we had assumed at the beginning of this subsection that \( F(\vec{x}, \vec{y}) \) is realizable. The need for this assumption can now be appreciated by noting that if \( F(\vec{x}, \vec{y}) \) is not realizable, we cannot guarantee that \( \text{Co}(M_\vec{x}) \) will be satisfiable for every MFS \( M_\vec{x} \) of the input clauses. Indeed, if \( \text{Co}(M_\vec{x}) \) is unsatisfiable, it is not enough to simply remove the corresponding \( f_j(\vec{x}) \) from the decision list, because there might be a subset \( M_i^\vec{\ell} \subseteq M_\vec{x} \) for which \( \text{Co}(M_i^\vec{\ell}) \) is satisfiable.

This is the first time to our knowledge that MFS have been used for synthesis purposes. An advantage of enumerating MFS is that finding an MFS is computationally easy. One way to do this is by constructing the conflict graph of the set of input clauses [17]. Given a set of clauses \( S \), the conflict graph of \( S \) is an undirected 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 [17].

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 enumerating maximal cliques in the consensus graph. The benefit of this reduction is that maximal cliques enjoy the polynomial-time listability.
property, meaning that a maximal clique can be found in polynomial time, and therefore enumeration takes time polynomial in the number of maximal cliques [20].

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 count of maximal cliques in the consensus graph of the input clauses. This gives us the following result.

**Theorem 3** Boolean functional synthesis is in $\text{P}^\text{NP}$ for CNF specifications for which the consensus graph of $S_x$ 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 maximal clique thus found, 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. $\square$

Theorem 3 demonstrates an improvement relative to the hardness of the general Boolean functional synthesis problem [2]. 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 3 provides an elegant method of deciding whether synthesis can be performed efficiently in practice even before beginning the synthesis process.

To summarize this subsection, 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 relatively efficiently. Note that this analysis does not take into account the properties of the output component at all, and as such the decision list produced by ignoring the output component may be longer than necessary. With this in mind, the next section presents a complementary analysis of the output component that can help to produce a smaller decision list.

### 5.2 Analysis of the output component

For the analysis of the output component, consider the set $\text{MustSat}(\bar{x})$, defined in Sect. 5.1. This is the set of output clauses that must be satisfied when $\bar{x}$ is the input assignment, if we are to satisfy the specification $F$. For every two input assignments $\bar{x}$ and $\bar{x}'$, if $\text{MustSat}(\bar{x}') \subseteq \text{MustSat}(\bar{x})$, every output assignment $\bar{y}$ that satisfies the specification for $\bar{x}$ also satisfies the specification for $\bar{x}'$. Therefore, when constructing a decision list that implements $F$, it is sufficient to consider only those satisfiable subsets of $S_y$ that are of maximal size. Similarly to Lemma 1 in the previous subsection, this insight leads to the following lemma:

**Lemma 2** Let $M_y$ be an MSS of $S_y$ and let $\bar{y}$ be an assignment that satisfies $M_y$. Then: (1) for every assignment $\bar{x}$ such that $\text{MustSat}(\bar{x}) \subseteq M_y$, the assignment $(\bar{x}, \bar{y})$ satisfies $F(\bar{x}, \bar{y})$; and (2) for every assignment $\bar{x}$ such that $\text{MustSat}(\bar{x}) \supseteq M_y$, there is no $\bar{y}'$ such that the assignment $(\bar{x}, \bar{y}')$ satisfies $F(\bar{x}, \bar{y})$. 
Proof (1) Since \( \hat{y} \) satisfies every clause \( C_j|_{\hat{y}} \) in \( M_{\hat{y}} \), it must be that \( \hat{y} \) also satisfies every clause in \( \text{MustSat}(\hat{x}) \). Therefore, for every clause \( C_j \) in \( F \), either \( C_j|_{\hat{x}} \) is satisfied by \( \hat{x} \) (and therefore \( C_j|_{\hat{y}} \notin \text{MustSat}(\hat{x}) \)) or \( C_j|_{\hat{y}} \) is satisfied by \( \hat{y} \). Therefore \((\hat{x}, \hat{y})\) satisfies \( F(\hat{x}, \hat{y}) \). (2) Since \( M_{\hat{y}} \) is maximal, then in this case \( \text{MustSat}(\hat{x}) \) must be unsatisfiable. Therefore there is no \( \hat{y}' \) that can satisfy all clauses that \( \hat{x} \) does not already satisfy.

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

Proof As before, we construct a decision list of the form if \( f_1(\hat{x}) \) then \( \hat{y}_1 \) else if \( f_2(\hat{x}) \) then \( \hat{y}_2 \) else ... else \( \hat{y}_t \), where \( t \) is the number of MSS of \( S_{\hat{y}} \). We define \( f_i(\hat{x}) \) to be the conjunction of all input clauses \( C_j|_{i\hat{x}} \) such that \( C_j|_{i\hat{x}} \) is not contained in the \( i \)-th MSS \( M^i_{\hat{y}} \). In other words,

\[
F(\hat{x}) = \bigwedge_{C_j \in S_{\hat{y}} \setminus M_{\hat{y}}} C_j|_{i\hat{x}}.
\]

We also define \( \hat{y}_i \) to be a satisfying assignment of \( M^i_{\hat{y}} \). Note that \( f_i \) is satisfied exactly by those input assignments \( \hat{x} \) that satisfy every clause \( C_j \) such that \( C_j|_{i\hat{x}} \notin M^i_{\hat{y}} \), which means \( \text{MustSat}(\hat{x}) \subseteq M^i_{\hat{y}} \). Meanwhile, \( \hat{y}_i \) satisfies every clause \( C_j \) such that \( C_j|_{i\hat{x}} \in M^i_{\hat{y}} \). As a consequence, if \( f_i(\hat{x}) = 1 \) then \((\hat{x}, \hat{y}_i)\) satisfies every clause in the CNF, and therefore satisfies \( F \).

Since for every \( \hat{x} \in \text{Dom}(F) \), there is an MSS \( M^i_{\hat{y}} \) of \( S_{\hat{y}} \) such that \( \text{MustSat}(\hat{x}) \subseteq M^i_{\hat{y}} \), it follows that every such \( \hat{x} \) satisfies some \( f_i(\hat{x}) \). Therefore, the decision list constructed above implements \( F \). \( \square \)

Example 2 Recall Example 1, where we had

\[
F(x_1, x_2, y_1, y_2) = (x_1 \lor \neg x_2 \lor y_1) \land (x_1 \lor x_2 \lor \neg y_1) \land (x_2 \lor y_1 \lor \neg y_2) \land (\neg x_1 \lor x_2 \lor y_2)
\]

\( S_{\hat{y}} = \{(x_1 \lor \neg x_2), (x_1 \lor x_2), (x_2, (\neg x_1 \lor x_2)) \} \) and \( S_{\hat{y}} = \{(y_1), (\neg y_1), (y_1 \lor \neg y_2), (y_2)\} \). In this case, \( S_{\hat{y}} \) has three MSS: \( \{(y_1), (y_1 \lor \neg y_2), (y_2)\}, \{(\neg y_1), (y_1 \lor \neg y_2)\}\) 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:

\[
\begin{align*}
\text{if } (x_1 \lor x_2) & \text{ then } y_1 := 1; y_2 := 1 \\
\text{elseif } (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) & \text{ then } y_1 := 0; y_2 := 0 \\
\text{elseif } (x_1 \lor \neg x_2) \land (x_2) & \text{ then } y_1 := 0; y_2 := 1
\end{align*}
\]

Unlike the input component analysis, the output component 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, \( \text{MustSat}(\hat{x}) \) must be contained in some MSS of \( S_{\hat{y}} \), and therefore will be covered by the decision list. Notice that we do not care about the cases where an input \( \hat{x} \) has no corresponding output \( \hat{y} \) that satisfies the specification \( F \). Unlike in the input component analysis, however, we no longer have a simple graph structure that can be exploited to obtain the list of MSSs. In fact, finding an MSS is NP-hard. Therefore, it is unlikely that we can efficiently identify instances where there are only polynomially many MSS.

More importantly, taking into account only the output component and ignoring the input component may lead to a large decision list that includes many MSS that would never have been required to be considered for any input assignment. This emphasizes the drawbacks of independent synthesis of the input and output components. This also

 Springer
motivates the development of an algorithm that combines the strengths of the input and output analyses to produce a decision list that is smaller than what each analysis can individually produce.

## 6 The back-and-forth algorithm

Our next goal is to combine the input and output analyses described in Sect. 5, into a synthesis procedure that constructs a decision list of length upper-bounded by the smaller of the MFS count of input clauses and the MSS count of output clauses. Due to the restrictions of the input analysis, if the specification is unrealizable the procedure terminates without producing a decision list. Therefore, we assume for the remainder of the discussion that the specification is realizable. Extending our approach to unrealizable specifications is left for future work.

We start with a simple lemma:

**Lemma 3** If $F(\vec{x}, \vec{y})$ is realizable, then for every MFS $M_\vec{x}$ of $S_\vec{x}$, $\text{Co}(M_\vec{x}) \subseteq M_\vec{y}$ 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 $\text{Fals}(\hat{x}) = M_\vec{x}$. Then, since $F$ is realizable, $\text{MustSat}(\hat{x}) = \text{Co}(M_\vec{x})$ is satisfiable, and therefore is contained in some MSS of $S_\vec{y}$.

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 $\text{Co}(M_\vec{x}) \subseteq M_\vec{y}$. 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 $\text{Co}(M_\vec{x})$, we can produce a satisfying assignment for $M_\vec{y}$. In fact, such a 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 MSS of $S_\vec{y}$ that is initially empty. At every iteration of the algorithm, we produce a new MFS of $S_\vec{x}$ that is not covered by the collection of MSS already in $L$. Then, we find an MSS of $S_\vec{y}$ that covers this new MFS. If no such MSS exists, it means the specification is unrealizable, and the algorithm emits an error message and terminates. Otherwise, we add this MSS to $L$. After all the MFS of $S_\vec{x}$ have been covered, we construct a decision list from the obtained list $L$ of MSS in the same way as described in Sect. 5.2: $f_i(\vec{x})$ is a formula that is satisfied exactly when $\text{MustSat}(\vec{x})$ is a subset of the $i$-th MSS, and the corresponding output assignment $\hat{y}_i$ is a satisfying assignment for that MSS.
**Algorithm 1** Back-and-Forth synthesis algorithm combining MFS and MSS analysis.

1: Initialize a list L of MSS to the empty list.
2: while there are still MFS left to generate do
3: \( M_x \leftarrow \text{MFS of} \, S_x \text{ not covered by any MSS of} \, S_y \text{ in} \, L \)
4: if MSS \( M_x \subseteq S_y \text{ covering} \, M_x \exists \) then
5: \( \text{add} \, M_x \text{ to} \, L \)
6: else
7: \( \text{FAIL: specification is unrealizable} \)
8: end if
9: end while
10: Construct decision list from L.

**Example 3** Let \( F, S_x \) and \( S_y \) be the same as in Examples 1 and 2, in which 
\[ F(x_1, x_2, y_1, y_2) = (x_1 \lor \neg x_2 \lor y_1) \land (x_1 \lor x_2 \lor \neg y_1) \land (x_2 \lor y_1 \lor \neg y_2) \land (\neg x_1 \lor x_2 \lor y_2) \], 
\[ S_x = \{ (x_1 \lor \neg x_2), (x_1 \lor x_2), (x_2), (\neg x_1 \lor x_2) \} \] and 
\[ S_y = \{ (y_1), (\neg y_1), (y_1 \lor \neg y_2), (y_2) \} \].

In the first iteration of Algorithm 1, we generate the MFS \( M_x^1 = \{ (x_1 \lor \neg x_2) \} \).
Then, we expand \( Co(M_x^1) = \{ (y_1) \} \) into the MSS \( M_y^1 = \{ (y_1), (y_1 \lor \neg y_2), (y_2) \} \) and add \( M_y^1 \) to \( L \). Note that \( M_y^1 \) also covers, besides \( M_x^1 \), the MFS \( \{ (x_2), (\neg x_1 \lor x_2) \} \), and therefore this MFS will not need to be generated. The only remaining MFS is \( M_x^2 = \{ (x_1 \lor x_2), (x_2) \} \). Since \( M_x^2 = Co(M_x^2) = \{ (\neg y_1), (y_1 \lor \neg y_2) \} \) is already an MSS, we add it to \( L \). Now that all MFS have been covered, the algorithm terminates with \( L = \{ (y_1), (y_1 \lor \neg y_2), (y_2) \}, \{ (\neg y_1), (y_1 \lor \neg y_2) \} \). 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:

\[
\begin{align*}
\text{if}(x_1 \lor x_2) & \text{then} (y_1 := 1; y_2 := 1) \\
\text{else if}(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) & \text{then} (y_1 := 0; y_2 := 0)
\end{align*}
\]

Observe that this decision list is smaller than those obtained in Examples (1) and (2).

**6.1 Implementation details**

The key steps of Algorithm 1 are the generation of the MFS \( M_x^1 \) in line 3 and the MSS \( M_y^1 \) in line 4. These steps are similar to the input and output analyses in Sects. 5.1 and 5.2. However, we have additional constraints on the MFS and MSS being generated in Algorithm 1. Specifically, at each step, the generated MFS must not be covered by any of the previously generated MSS, and the generated MSS must cover the most recently generated MFS.

While generating an arbitrary MFS can be done in polynomial time, we show in Sect. 7.2 that adding the restriction that the MFS must not be covered by a previous MSS makes the MFS generation an NP-complete problem. 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 \( S_x \) not covered by any of the previously generated MSS. 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
We use variable \( z_i \) to indicate whether clause \( C_{i,j} \) is present in the all-falsifiable subset. The first conjunction encodes that for every previously generated MSS, the subset must include a clause \( C_{i,j} \) 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 \( M \), we have to generate a new MSS \( M' \) that covers \( M \). 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 [5]. 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 \( S \), where the clauses in \( Co(M) \) 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 \( Co(M) \) and of maximum size. Since a satisfiable subset of maximum size is necessarily maximal, the set of satisfied clauses returned by the MaxSAT solver is an MSS, as desired.

6.2 Analysis and correctness

Since exactly one new MFS and one new MSS are generated in 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 \( F(x, y) \), let \( (f_1, \hat{y}_1), \ldots, (f_k, \hat{y}_k) \) be the decision list produced by Algorithm 1. Then (1) For every \( \hat{x} \) there is at least one \( i \) such that \( f_i(\hat{x}) = 1 \); (2) For every \( \hat{x} \) such that \( f_i(\hat{x}) = 1 \), \( F(\hat{x}, \hat{y}_i) = 1 \).

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

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

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 Sect. 5.1, then purely enumerating MFS, as in Sect. 5.1 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 a MaxSAT solver. Therefore, for specifications that are known to have a small number of MFS, restriction to the analysis of the input component as in Sect. 5.1 can be sufficient. 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. In fact, we can prove that the Back-and-Forth algorithm is in a sense optimal: given the right choice of the order in which MFS and MSS are generated, the algorithm produces a decision list of minimum size. We now prove this theorem.

Theorem 4 Let $F(\vec{x}, \vec{y})$ be a realizable specification. Let $k$ be the size of the smallest decision list that implements $F$. Then, there is an execution of the Back-and-Forth algorithm that produces a decision list of size $k$.

Proof Let $\ell$ be a decision list of minimum size, i.e. $k$, that implements $F$. We will prove that there exists a decision list $\ell'$ of the same size that can be generated by the Back-and-Forth algorithm.

Assume without loss of generality that the $f_i(\vec{x})$ formulas in $\ell$ partition the space of assignments of $\vec{x}$. If this is not the case, we can modify $\ell$ by replacing $f_i(\vec{x})$ with $f_i(\vec{x}) \land \bigwedge_{j=1}^{i-1} \neg f_j(\vec{x})$. Note that every assignment $\vec{y}_i$ output by the decision list satisfies a set of output clauses, which must be contained in an MSS.

First, we prove that no two $\vec{y}_i$, $\vec{y}_j$ in $\ell$ share an MSS. Assume for the sake of contradiction that there are distinct $i$ and $j$ such that $\vec{y}_i$ and $\vec{y}_j$ share an MSS $M_{\vec{y}_i}$. Then, we can replace both $\vec{y}_i$ and $\vec{y}_j$ by a satisfying assignment $\vec{y}$ of $M_{\vec{y}_i}$. Since every clause that is satisfied by $\vec{y}_i$ or $\vec{y}_j$ is also satisfied by $\vec{y}$, this decision list remains an implementation of $F$. But now, since the $i$-th and $j$-th decisions produce the same output, we can merge them into a single decision with $f(\vec{x}) = f_i(\vec{x}) \lor f_j(\vec{x})$, and output $\vec{y}$. The resulting decision list would have size $k - 1$ and would be a correct implementation of $F$, contradicting the minimality of $\ell$. Therefore, no distinct $\vec{y}_i$ and $\vec{y}_j$ can share an MSS.

Now, choose for each $\vec{y}_i$ in $\ell$ an MSS $M_{\vec{y}_i}^i$ containing the clauses satisfied by $\vec{y}_i$. By the previous paragraph, these MSS must be all distinct. Let $\ell'$ be the decision list constructed from the list of MSS $M_{\vec{y}_1}^1, \ldots, M_{\vec{y}_k}^k$, as described earlier in this subsection. Since there are $k$ MSS, $\ell'$ has length $k$. We will prove that $\ell'$ is also an implementation of $F$, and that it can be produced by an execution of the Back-and-Forth algorithm, thus concluding the proof.

To prove that $\ell'$ is also an implementation of $F$, we will first prove that every input $\vec{x}$ that satisfies the original $f_i(\vec{x})$ in $\ell$ also satisfies the corresponding $f_i'(\vec{x})$ in $\ell'$ obtained from $M_{\vec{y}_i}^i$. Since $\ell$ is an implementation of $F$, if $\vec{x}$ satisfies $f_i(\vec{x})$ this means that MustSat($\vec{x}$) is a subset of the output clauses satisfied by $\vec{y}_i$. Therefore, MustSat($\vec{x}$) is also a subset of the MSS $M_{\vec{y}_i}^i$. Since by construction $f_i'(\vec{x})$ is satisfied if and only if MustSat($\vec{x}$) is a subset of $M_{\vec{y}_i}^i$, $\vec{x}$ satisfies $f_i'(\vec{x})$. Then, since the $\vec{y}_i'$ obtained from $M_{\vec{y}_i}^i$ necessarily satisfies a superset of MustSat($\vec{x}$), the decision list always produces a correct output. Thus, $\ell'$ is a correct implementation of $F$. 

 Springer
Lastly, we prove that $\ell'$ can be produced by an execution of the Back-and-Forth algorithm. We need to show that for every MSS $M^i_y$ there is an MFS $M^i_x$ such that:

1. $M^i_x$ is covered by $M^i_y$;
2. $M^i_x$ is not covered by $M^j_y$, for all $j < i$.

If these requirements are satisfied, then the list of MFS $M^1_x, \ldots, M^k_x$ can be produced by the Back-and-Forth algorithm. For the purpose of reaching a contradiction, assume that there is no such list of MFS. This means that there is an MSS $M^i_y$ such that no MFS $M^i_x$ satisfies the two requirements above. In other words, all MFS that are covered by $M^i_x$ have already been covered by some previous MSS $M^j_y$, for $j < i$. But this means that every input covered by $M^i_x$ is already covered by a previous MSS. Therefore, $M^i_x$ is redundant, and therefore can be removed from the list of MSS, which contradicts the minimality of $\ell'$.

Therefore, the list of MSS, and consequently $\ell'$, can be generated by the Back-and-Forth algorithm. Since we have already proved that $\ell'$ is an implementation of $F$ of minimum size, this concludes the proof of the theorem.

\[ \square \]

7 Observations of the back-and-forth algorithm

In this section we bring several properties and observations that follow our Back-and-Forth Algorithm description and analysis.

7.1 Partitioning the specification into distinct output variables

There are pathological cases in which the Back-and-Forth algorithm may end up considering an exponential number of MFS or MSS. Some of these cases can be handled better by partitioning the specification into sets of clauses that do not share output variables. As an example, consider the specification for the identity function:

$$F(\vec{x}, \vec{y}) = (x_1 \leftrightarrow y_1) \land \ldots \land (x_t \leftrightarrow y_t),$$

which can be written in CNF as:

$$F(\vec{x}, \vec{y}) = (\neg x_1 \lor y_1) \land (x_1 \lor \neg y_1) \land \ldots \land (\neg x_t \lor y_t) \land (x_t \lor \neg y_t)$$

Note that both the number of MFS and MSS for this formula are $2^t$. Each output variable, however, does not appear in the same clause with other output variables. Therefore, we can consider the sub-formula arising from each pair of clauses, viz. $(\neg x_i \lor y_i) \land (x_i \lor \neg y_i)$, as a separate sub-specification and synthesize it independently as a decision list of size 2. In this approach, the total number of MFS and MSS grow linearly with $k$.

Taking cue from the above example, 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 C₁, . . . , Cᵣ. Note that the Cᵢs are completely disjoint in terms of output variables.
3. For every Cᵢ, define a sub-specification Fᵢ by conjoining only those clauses in F that have a corresponding vertex in Cᵢ.
4. Call Algorithm 1 for each sub-specification Fᵢ. This gives us a decision list Dᵢ for Fᵢ that decides on an assignment for only those output variables that are in Fᵢ.

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

### 7.2 NP-completeness of constrained MFS generation

An essential component of the Back-and-Forth algorithm in Sect. 6 is the generation of an MFS that is not covered by any previously generated MSS. We can formulate this problem as the following decision problem, called Hitting-MFS-Among-MSS due to its resemblance to the Hitting-set problem. We say that an MFS M is covered by a list L of MSS if there exists an MSS M′ ∈ L, such that M is covered (in the sense of Sect. 6) by M′.

**Problem 2** (Hitting-MFS-Among-MSS) Let F(̄x, ̄y) be a CNF formula where S̄x is the set of input clauses and S̄y is the set of output clauses. Let L = (M̄y₁, . . . , M̄yₜ), where each M̄yᵢ ⊆ S̄y, be a collection of MSS of the output clauses S̄y. Does there exist an MFS M̄x ⊆ S̄x of the input clauses such that M̄x is not covered by L?

We next show that Hitting-MFS-Among-MSS is NP-complete. Thus our choice of using a SAT solver in the Back-and-Forth algorithm to find the MFS is justified.

**Theorem 5** Hitting-MFS-Among-MSS is NP-complete.

**Proof** First note that Hitting-MFS-Among-MSS is clearly in NP, since given M̄x ⊆ S̄x, we can verify in polynomial time whether M̄x is an MFS and whether M̄x is covered by one of the M̄yᵢ. We next prove that Hitting-MFS-Among-MSS is NP-hard by a reduction from 3SAT.

Let ϕ(̄x) = D₁ ∧ . . . ∧ D₉ be a 3CNF formula where every Dᵢ is a clause. We construct a CNF formula F(̄x, ̄y) and a list of MSS L = (M̄y₁, . . . , M̄yₚ) such that there exists an MFS that is not covered by L iff ϕ is satisfiable. The input variables ̄x of this formula are the same as the variables in ϕ, while the output variables ̄y are q fresh variables, one for each clause Dᵢ.

We first construct the set of input clauses S̄x in the following way. For every clause Dᵢ, let SatAssign(Dᵢ) be the set of all assignments to the variables of Dᵢ that satisfy Dᵢ. Since Dᵢ has at most three literals, |SatAssign(Dᵢ)| is at most 7. Next we construct a fresh input clause from every such assignment. Note that for every assignment ̄τ we can construct a clause Cᵣ over the assigned variables that is falsified exactly by ̄τ. For example if ̄τ = (x₁ = 1, x₂ = 0, x₃ = 1) then the clause Cᵣ = (¬x₁ ∨ x₂ ∨ ¬x₃) is falsified by ̄τ. Let S̄xᵢ = {Cᵣ | ̄τ ∈ SatAssign(Dᵢ)} be the set of input clauses falsified by the satisfying assignments of clause Dᵢ. Notice that by construction, no two clauses of S̄xᵢ are simultaneously falsifiable. Let S̄x = ∪ᵢ S̄xᵢ.
We next construct the set of output clauses \( S_\tilde{x} \) in the following way. We first introduce one \( y_i \) variable for every clause \( D_i \). We then construct for every \( D_i \) two clauses: \( A_i = (y_i) \) and \( B_i = (\bigvee_{j \neq i} \neg y_j) \). We define \( S_\tilde{x} = \bigcup_{i=1}^{m} \{A_i, B_i\} \).

We finally construct \( F(\tilde{x}, \tilde{y}) \) by concatenating the input and output clauses in the following way. For every clause \( D_i \) and assignment \( \tau \in \text{SatAssign}(D_i) \), create two clauses \((C_\tau \lor A_i)\) and \((C_\tau \lor B_i)\). Since the number of \( C_\tau, A_i \) and \( B_i \) is linear in the number of clauses of \( \phi \), and the size of the clauses is either constant (for \( C_\tau \) and \( A_i \)) or linear on the number of clauses (for \( B_i \)), the construction of \( F \) takes polynomial-time.

Lastly, we construct the list of MSS \( L = (M_1, ..., M_q) \) by defining \( M_r = \bigcup_{i \neq r} \{A_i, B_i\} \) for every \( r \leq q \). Clearly, \( L \) is polynomial in the size of \( \phi \). It is not hard to prove that the elements of \( L \) are indeed MSS. First, note that the only way to satisfy \( M_r \) is to set \( y_r \) to false and all other output variables to true. Then, note that the only two output clauses that are missing from \( M_r \) are \( A_i = y_i \) and \( B_i = \bigvee_{j \neq i} \neg y_j \). Since neither of these clauses can be satisfied by the unique assignment that satisfies \( M_r \), adding another clause would make the set unsatisfiable. Therefore, every \( M_r \) is both satisfiable and maximal.

We finally show that \( \phi \) is satisfiable iff there exists an MFS \( M_\tilde{x} \subseteq S_\tilde{x} \) of the input clauses that is not covered by \( L \).

\((\rightarrow)\) Assume \( \phi \) is satisfiable. Let \( \hat{x} \) be a satisfying assignment and let \( \tau \) be the projection of \( \hat{x} \) over only the variables in \( D_i \). We prove that \( M_\tilde{x} = \bigcup_{i=1}^{m} \{C_\tau\} \) is an MFS that is not covered by \( L \). To see why \( M_\tilde{x} \) is an MFS, notice the following: (1) \( M_\tilde{x} \) is all-falsifiable since every clause \( C_\tau \) is falsified by \( \hat{x} \); (2) \( M_\tilde{x} \) is maximal because it already includes one \( C_\tau \in S_\tilde{x} \) for every \( i \), and we have already noted above that no two clauses in \( S_\tilde{x} \) are simultaneously falsifiable. Now, recall that by the construction of \( F \), there must be a clause \((C_\tau \lor A_i)\) in \( F \) for every 3CNF clause \( D_i \). Therefore, \( A_i \in \text{Co}(M_\tilde{x}) \) for every \( i \in \{1, ..., q\} \). Since \( A_i \notin M_r \), \( M_r \) is not covered by \( M_\tilde{x} \). Therefore, \( M_\tilde{x} \) is an MFS that is not covered by \( L \).

\((\leftarrow)\) Assume that there exists an MFS \( M_\tilde{x} \subseteq S_\tilde{x} \) of the input clauses that is not covered by \( L \). Since \( M_\tilde{x} \) is an MFS, there must be an assignment \( \hat{x} \) that falsifies every clause \( C_\tau \in M_\tilde{x} \). Every such \( \tau \) must be consistent with \( \hat{x} \). For every \( i \), since \( M_\tilde{x} \) is not covered by \( M_r \), \( M_r \) must include an input clause \( C_\tau \) originating from clause \( D_i \). Since \( \tau \) is consistent with \( \hat{x} \) and a satisfying assignment to \( D_i \), \( \hat{x} \) must satisfy \( D_i \). Therefore, \( \hat{x} \) satisfies every clause in \( \phi \), and hence \( \phi \) is satisfiable.

\(\square\)

**Example 4** Let \( \phi(x_1, x_2, x_3) = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3) \) be a 3CNF formula. We show below a part of the construction of the formula \( F(x_1, x_2, x_3, y_1, y_2, y_3) \) with the corresponding list of MSS.

The set of input clauses constructed from the first clause of \( \phi \) is \( S_\tilde{x}^1 = \{(\neg x_1 \lor x_2 \lor x_3), (\neg x_1 \lor x_2 \lor \neg x_3), (\neg x_1 \lor \neg x_2 \lor x_3), (x_1 \lor \neg x_2 \lor x_3), (x_1 \lor \neg x_2 \lor \neg x_3), (x_1 \lor x_2 \lor x_3), (x_1 \lor x_2 \lor \neg x_3)\} \). Note that each clause is falsified exactly by one of the 7 satisfying assignments of \( (x_1 \lor x_2 \lor \neg x_3) \). \( S_\tilde{S}^1 \) and \( S_\tilde{S}^2 \) are constructed in an analogous way, and \( S_\tilde{S} = S_\tilde{S}^1 \cup S_\tilde{S}^2 \cup S_\tilde{S}^3 \).

Next, the set of the constructed output clauses is \( S_\tilde{y} = \{y_1, (\neg y_2 \lor \neg y_3), y_2, (\neg y_1 \lor \neg y_3), y_3, (\neg y_1 \lor \neg y_2)\} \).

Then, we have that \( F(x_1, x_2, x_3, y_1, y_2, y_3) = (x_1 \lor x_3,(\neg x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor \neg x_3) \). \( S_\tilde{y} \) and \( S_\tilde{S} \) are constructed in an analogous way, and \( S_\tilde{S} = S_\tilde{S}^1 \cup S_\tilde{S}^2 \cup S_\tilde{S}^3 \).

Finally, set \( L = (M_{\tilde{x}}^1, M_{\tilde{x}}^2, M_{\tilde{x}}^3) \) to be the list of MSS, where \( M_{\tilde{x}}^1 = \{y_2, (\neg y_1 \lor \neg y_3), y_3, (\neg y_1 \lor \neg y_2)\} \), \( M_{\tilde{x}}^2 = \{y_1, (\neg y_2 \lor \neg y_3), y_3, (\neg y_1 \lor \neg y_2)\} \) and \( M_{\tilde{x}}^3 = \{y_1, (\neg y_2 \lor \neg y_3), y_3, (\neg y_1 \lor \neg y_2)\} \). Then, every satisfying assignment to \( \phi \) has a corresponding MFS \( M_{\tilde{x}} \) that is not covered by \( L \). For example, take the
satisfying assignment \((x_1 = 1, x_2 = 0, x_3 = 1)\). Then, \(\tau_1 = \tau_2 = (x_1 = 1, x_2 = 0, x_3 = 1)\) and \(\tau_3 = (x_1 = 1, x_2 = 0)\). This gives us \(\mathcal{M}_1 = \{(-x_1 \lor x_2 \lor -x_3), (-x_1 \lor x_2)\}\). Note that \(\mathcal{C}(\mathcal{M}_1) = \{y_1, (\neg y_2 \lor \neg y_3), y_2, (\neg y_1 \lor \neg y_3), y_3, (\neg y_1 \lor \neg y_2)\}\), which is not covered by \(L\).

7.3 Input-clause-driven decision lists as synthesis solutions

In general, a decision list implementing a specification \(F\) has the form `if \(f_1(\vec{x})\) then \(\vec{y}_1\) else if \(f_2(\vec{x})\) then \(\vec{y}_2\) else \(\ldots\)` else \(\vec{y}_n\), where the functions \(f_i\) can be arbitrary functions of the inputs \(\vec{x}\). A closer inspection of Algorithm 1, however, reveals that each \(f_i\) in a decision list output by the algorithm is of a very special form: it is a conjunction of input clauses of the original CNF specification. This suggests that a syntactically restricted class of decision lists suffices to serve as implementations of realizable specifications.

To formalize the above intuition, we say that a decision list implementation of a realizable CNF specification \(F(\vec{x}, \vec{y})\) is input-clause driven if every formula \(f_i(\vec{x})\) in the decision list is a conjunction of input clauses of \(F\). For example, if \(F(\vec{x}, \vec{y}) = (\neg x_1 \lor \neg x_2 \lor y_1) \land (\neg x_1 \lor \neg y_2)\), then the set of input clauses is \(\{(\neg x_1 \lor \neg x_2), (\neg x_1)\}\). Hence the decision list `if \((\neg x_1)\) then \((y_1 := 1; y_2 := 1)\)` else if \((\neg x_1 \lor \neg x_2)\) then \((y_1 := 0; y_2 := 0)\)` else \((y_1 := 1; y_2 := 0)\)` else \((y_1 := 0; y_2 := 1)\)` is input-clause driven. However, `if \((x_1 \lor x_2)\)` then \((y_1 := 1; y_2 := 0)\)` else \((y_1 := 0; y_2 := 1)\)` is not input-clause driven.

**Lemma 5** Every realizable CNF specification has an input-clause driven decision list implementation, and the Back-and-Forth algorithm (Algorithm 1) always outputs one such implementation.

**Proof** In Algorithm 1, suppose the list \(L\) of MSS obtained at the end of all iterations is \(\{\mathcal{M}_1, \ldots, \mathcal{M}_L\}\). The construction of the decision list output by the algorithm is described in the proof of Corollary 2. Specifically, for each \(i \in \{1, \ldots, t\}\), \(f_i(\vec{x})\) is defined to be \(\bigwedge_{C_i \in \mathcal{L} \setminus \mathcal{M}_i}(\mathcal{C}_i)\). Hence, Algorithm 1 always outputs an input-clause driven decision list. The correctness of Algorithm 1, guaranteed by Corollary 3, now proves the lemma. \(\square\)

We believe that input-clause driven decision list implementations lend themselves to better interpretation by the user, since the decisions in the list are simply conjunctions of input conditions already present in the clauses of the specification. Therefore, such implementations may be preferable in situations where the implementation must be independently audited by domain experts for correctness. To the best of our knowledge, the Back-and-Forth algorithm is the only Boolean functional synthesis algorithm that can produce input-clause driven decision list implementations.

Since decision lists can represent arbitrary Boolean functions, it is easy to see that for every implementation of a CNF specification, there is a semantically equivalent decision list implementation. An interesting question to ask is whether there always exists an `input-clause driven` decision list implementation that is semantically equivalent to an arbitrary implementation of a CNF specification. As we show below, this turns out not to be the case. Therefore, input-clause driven decision lists represent not only a syntactic subclass, but also a strict semantic subclass, of all implementations of a CNF specification, in general.
**Lemma 6** There exists a realizable CNF specification $F(\vec{x}, \vec{y})$ and a specific implementation $g(\vec{x})$ of $F$ such that there is no input-clause driven decision list that is semantically equivalent to $g(\vec{x})$.

**Proof** We prove this by giving a specific example of $F(\vec{x}, \vec{y})$ and $g(\vec{x})$.

Let $F(x_1, x_2, y_1, y_2) \equiv (\neg x_1 \lor x_2 \lor y_1) \land (x_1 \lor \neg x_2 \lor y_2)$. This is a realizable specification since $(y_1 := x_1, y_2 := \neg x_1)$ implements $F$. Let us call this implementation $g(x_1, x_2)$.

A semantically equivalent decision list is: $\text{if } (x_1) \text{ then } (y_1 := 1, y_2 := 0) \text{ else } (y_1 := 0, y_2 := 1)$. However, this is not an input-clause driven decision list. Indeed, the input clauses of $F$ are $C_{1|z} \equiv (\neg x_1 \lor x_2)$ and $C_{2|z} \equiv (x_1 \lor \neg x_2)$, and the only formulas that can be obtained by conjoining a non-empty subset of $\{C_{1|z}, C_{2|z}\}$ are $(\neg x_1 \lor x_2), (x_1 \lor \neg x_2)$ and $(\neg x_1 \lor x_2) \land (x_1 \lor \neg x_2)$.

Suppose there is an input-clause driven decision list implementation of $F$ that is semantically equivalent to $g(x_1, x_2)$. Since the values of $y_1$ and $y_2$ given by $g(x_1, x_2)$ are not constants, the decision list implementation must have the form $\text{if } (a) \text{ then } (y_1 := a; y_2 := b) \text{ else } \ldots$, where $a$ is one of $(\neg x_1 \lor x_2), (x_1 \lor \neg x_2)$ or $(\neg x_1 \lor x_2) \land (x_1 \lor \neg x_2)$, and $a, b \in \{0, 1\}$.

Suppose $a = (\neg x_1 \lor x_2)$. For $(x_1 = 0, x_2 = 0)$, since $a$ evaluates to true, we must have $y_1 := a$ and $y_2 := b$. From $g(x_1, x_2)$, we also know that $y_1 := x_1 (= 0)$ and $y_2 := \neg x_1 (= 1)$ for the current input assignment. Therefore, we must have $a = 0$ and $b = 1$. However, for $(x_1 = 1, x_2 = 1)$, $a$ evaluates to true again, and this time, we must have $y_1 := x_1 (= 1)$ and $y_2 := \neg x_1 (= 0)$. Therefore, we must have $a = 1$ and $b = 0$, contradicting what we inferred above. Hence, $a$ cannot be $(\neg x_1 \lor x_2)$.

Using a similar argument as above, we can now show that $a$ cannot be $(x_1 \lor \neg x_2)$ or $(\neg x_1 \lor x_2) \land (x_1 \lor \neg x_2)$ either. This contradicts our requirement that $\alpha$ must be one of $(\neg x_1 \lor x_2), (x_1 \lor \neg x_2)$ or $(\neg x_1 \lor x_2) \land (x_1 \lor \neg x_2)$, and $a, b \in \{0, 1\}$. Therefore, there is no input-clause driven decision list that is semantically equivalent to $g(x_1, x_2)$. \hfill $\square$

Lemmas (5) and (6) show that although input-clause driven decision lists cannot represent all implementations of a specification in general, they are sufficient to implement every realizable CNF specification. As a corollary, this implies that while the Back-and-Forth algorithm is not powerful enough to generate all implementations (modulo semantic equivalence) of a given specification, it can always generate at least one implementation that is amenable to easy interpretation.

**8 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 benchmark suite [32]. 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 view 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 our experimental evaluation, we used only those benchmarks that are realizable, since adjusting the Back-and-Forth algorithm to handle unrealizable benchmarks is beyond the scope of the current work. We used the QBFEVAL results [32] to identify the set of realizable benchmarks. 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.

In order to confirm that the Back-and-Forth algorithm is indeed a more practical approach than the CNF decomposition described in Sect. 4, we first compared its running time with that of RSynth with and without decomposition. We present the results in Sect. 8.1.

Next, to see how the algorithm would fit in the current portfolio of synthesis tools, we compared its performance on these benchmarks with four state-of-the-art tools, as of the time of submission of this paper, that employ different synthesis approaches: the CEGAR-based CAQE [35], the CDCL-based CADET [34], the ROBDD-based RSynth [43], and BFSS [2], which combines unateness checking and CEGAR. Out of these tools, CAQE is the one closest to ours, as it uses techniques that are similar to the clause splitting used in the Back-and-Forth algorithm. However, CAQE targets QBF instances with arbitrary quantifier alternation, requiring additional mechanisms for handling these cases. Furthermore, CAQE does not perform the analysis based on MFS and MSS, as is done in the current work. It is important to note that we compare with the legacy version of CAQE from 2016, since the most recent version of CAQE (as of the date of writing this paper) does not support synthesis of Skolem functions. We present these results in Sect. 8.2.

Since the Back-and-Forth algorithm, CAQE, 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 [14] instead of the parallelized UniGen sampler used in [2]. 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 [6], based on MiniSAT, and the Open-WBO MaxSAT solver [31]. The implementation also used the partitioning described in Sect. 7.1. 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 h.

8.1 Comparison with CNF decomposition

Table 1 shows a comparison of the running time of the Back-and-Forth algorithm, for instances from different families, with that of RSynth applied to the same instances when those are decomposed using the CNF decomposition from Sect. 4. For reference, the table also includes the running time of RSynth applied to the original specification, without decomposition. We show the results only for those benchmarks where RSynth applied to the decomposed specification did not timeout or memout. This includes a single instance of the MutexP family, all 6 instances of the Qshifter family, and 5 instances of the FixpointDetection family.

---

1 Code available at https://github.com/lucasmt/BackAndForthSynthesis
As we can see from the measurements, the running time of the Back-and-Forth algorithm is generally smaller than that of the approach using CNF decomposition, in some cases by several orders of magnitude. Surprisingly, in the Qshifter family, RSynth with decomposition is able to solve the last two instances while the Back-and-Forth algorithm cannot, despite the running time of the latter scaling better than the former. This is because the Back-and-Forth algorithm runs out of memory for those instances. Even so, the overall picture suggests that the Back-and-Forth algorithm is indeed a more practical approach for synthesis based on input-output separation.

Comparing with RSynth applied to the original specification, however, it is not yet clear what concrete benefits our algorithms contributes to the state of the art. To better answer this question, the next section measures the performance of the algorithm across the entire set of benchmarks.

### 8.2 Comparison by family

Figure 2 shows for each family the percentage of problem instances that each tool was able to solve within 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. CAQE performed only slightly better in 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 s (except for the two hardest instances of Qshifter, which timed out), outperforming RSynth or BFSS in many cases, and CAQE in all cases. Even so, CADET still performed the best in these classes, solving all instances faster than our algorithm.

| Instance          | Back-and-Forth | RSynth (orig.) | RSynth (with decomp.) |
|-------------------|----------------|----------------|------------------------|
| mutex-2-s         | 2              | 14             | 242281                 |
| qshifter_3        | 16             | 22             | 8                      |
| qshifter_4        | 168            | 26098          | 168                    |
| qshifter_5        | 2537           | -              | 3408                   |
| qshifter_6        | 43245          | -              | 53838                  |
| qshifter_7        | -              | -              | 1278519                |
| qshifter_8        | -              | -              | 21698082               |
| stmt1_145_146     | 830            | 11             | 165729                 |
| stmt1_20_21       | 244            | 7              | 135310                 |
| stmt24_148_149    | 700            | 7              | 1736630                |
| stmt24_7_8        | 5              | 0              | 2                      |
| stmt44_107_108    | 9              | 2              | 27                     |
Lastly, ReductionFinding and SortingNetworks seem to be the most challenging families for existing tools. CADET is able to solve only three instances in total, RSynth one, and BFSS none. The Back-and-Forth algorithm and CAQE perform the best in these families. The former is able to solve 13 cases in ReductionFinding and 6 in SortingNetworks, while the latter is able to solve 5 cases in ReductionFinding and 10 in SortingNetworks. In the SortingNetworks family, all cases solved by the Back-and-Forth algorithm were also solved by CAQE. On the other hand, in the ReductionFinding family, all cases solved by CAQE were also solved by our algorithm.

In summary, the Back-and-Forth algorithm performed competitively in 5 out of 7 families, and was the best solver for one of these. Going by the number of instances solved, it was strictly superior to CAQE in 3 families, to CADET in 2 families, to RSynth in 4 and to BFSS in 3. Of the 13 cases where an instance was only solved by a single tool, the Back-and-Forth algorithm was responsible for 8, all in the ReductionFinding family. The others were 4 SortingNetworks instances uniquely solved by CAQE and 1 FixPointDetection instance uniquely solved by CADET.

As the above results indicate, the Back-and-Forth algorithm performs well on some families of benchmarks, but not so on others. It is natural to ask if there are specific characteristics of benchmark families that meaningfully explain this variation of performance. We undertook a study to try to find such characteristics and were able to identify some preliminary conclusions. Analyzing the benchmarks in the FixPointDetection family, one of the families where CADET outperforms the Back-and-Forth algorithm, it seems that these formulas largely encode a circuit-like structure, where output variables are defined as the conjunction or disjunction of other (input or output) variables. A similar structure is also present in parts of the RankingFunctions benchmarks. This structure is not immediately obvious when the formula is in CNF, but the incremental determinization algorithm implemented in CADET excels in extracting implications. This likely allows CADET to expose the if-and-only-if structure of the formula and easily synthesize the trivial Skolem functions. In contrast, the circuit-like structure is less amenable to the flat decision-list representation extracted by the Back-and-Forth algorithm, and the dependencies between
output variables make these benchmarks unable to benefit from the partitioning technique described in Sect. 7.1.

The next section contains a few more insights about how the Back-and-Forth algorithm was able to exploit the structure of the Tree, MutexP, and Qshiftf families. Other families, such as ReductionFinding and SortingNetworks, display less obvious structural properties. We reserve a deeper analysis for future work, as there are too many (interfering) factors that affect the performance of the Back-and-Forth algorithm and also of the other tools used in our study. The collection of complex heuristics used in the backend engines (viz. SAT solver, MaxSAT solver, ROBDD library, AIG library etc) of Back-and-Forth and other tools make it difficult in general to attribute differences in performance of these tools to specific characteristics of benchmarks. Consequently, we refrain for now from providing a more detailed benchmark-characteristic-based explanation of the performance of the Back-and-Forth algorithm. Instead, we posit that the Back-and-Forth algorithm serves as a good complement to other state-of-the-art Boolean functional synthesis tools in a portfolio approach to solve practical synthesis problems. Such a portfolio solver may even execute Back-and-Forth in parallel with other algorithms, and terminate as soon as one of them gives an answer.

8.3 Decision-list length

Table 2 shows the length of the decision list for every instance solved by the Back-and-Forth algorithm in each family. Note that in theory the decision list can have length up to $2^n$, where $n$ is the number of input variables. In practice, however, we see that the length is significantly smaller than the worst case, in many cases following a linear or even constant progression. In some cases the length of the decision list is even smaller than the number of input variables.

These results suggest that the Back-and-Forth algorithm is able to effectively exploit the structure of an instance through its analysis of the instance’s MFS and MSS. This is especially noticeable in the MutexP family, where the algorithm was able to determine that all instances could be implemented by a decision list of length 1. In other words, in every case there is a single assignment to the output variables that always satisfies the specification regardless of the input. The reason for this is that instances of this family follow a particular structure, where every clause except one contains an output literal in the negative form. The remaining clause is a unary clause with a single output literal in the positive form. This unary clause can be propagated to identify other output variables that must be set to true, and the remaining variables can all be set to false. The fact that every instance in the family can be solved in such a way would not be obvious without the MFS and MSS analysis performed by the algorithm, which tries to minimize the number of decisions on the input variables. In contrast, all the other tools that were able to solve MutexP failed to recognize this pattern and produced significantly more complex implementations.

Two other families in which the Back-and-Forth algorithm was able to extract a clear pattern are the Qshiftf and Tree families. These families benefited from the partitioning techniques used in Sect. 7.1. In instances from either family, every clause has only a single output literal, either in positive or negative form. Therefore, the instance could be implemented by a collection of decision lists, one for each output variable and all of length 2.

These observations suggest benefits of the Back-and-Forth algorithm that go beyond performance. Exploiting the structure of the problem can produce smaller and simpler implementations. This, coupled with the fact that decision lists are in general easier to reason
Table 2  Length of the decision list generated by the Back-and-Forth algorithm for each of the instances that it was able to solve

| Family     | Instance          | # of input vars. | Decision-list length |
|------------|-------------------|------------------|----------------------|
| MUTEXP     | mutex-2-s         | 8                | 1                    |
|            | mutex-4-s         | 16               | 1                    |
|            | mutex-8-s         | 32               | 1                    |
|            | mutex-16-s        | 64               | 1                    |
|            | mutex-32-s        | 128              | 1                    |
|            | mutex-64-s        | 256              | 1                    |
|            | mutex-128-s       | 512              | 1                    |
| QSHIFTER   | qshifter_3        | 11               | 16                   |
|            | qshifter_4        | 20               | 32                   |
|            | qshifter_5        | 37               | 64                   |
|            | qshifter_6        | 70               | 128                  |
| REDUCTIONFINDING | nxquery_query50_1344n | 54          | 781                  |
|            | query04_query25_1344n | 66          | 345                  |
|            | query05_query31_1344n | 66          | 3832                 |
|            | query31_eequery_1344n | 66         | 23394                |
|            | query31_query26_1344n | 36          | 721                  |
|            | query31_query50_1344n | 66          | 1556                 |
|            | query31_reachqu_1344n | 71          | 7344                 |
|            | query33_query45_1344n | 66          | 1713                 |
|            | query34_query11_1344n | 66          | 26750                |
|            | query36_query25_1344n | 66          | 23394                |
|            | query49_ntrivil_1344n | 90          | 10078                |
|            | query50_query06_1344n | 70          | 4122                 |
|            | query64_query01_1344n | 60          | 19154                |
| SORTINGNETWORKS | sortnetsort5.AE.stepl.003 | 102    | 112                  |
|            | sortnetsort5.AE.stepl.004 | 136     | 272                  |
|            | sortnetsort6.AE.stepl.003 | 165     | 158                  |
|            | sortnetsort6.AE.stepl.004 | 220   | 396                  |
|            | sortnetsort7.AE.stepl.003 | 243   | 207                  |
|            | sortnetsort8.AE.stepl.003 | 336   | 254                  |
| TREE       | tree-exa10-10     | 10               | 18                   |
|            | tree-exa10-15     | 15               | 28                   |
|            | tree-exa10-20     | 20               | 38                   |
|            | tree-exa10-25     | 25               | 48                   |
|            | tree-exa10-30     | 30               | 58                   |
| FIXPOINTDETECTION | stmt1_145_146 | 12          | 109                  |
|            | stmt1_20_21       | 9                | 55                   |
|            | stmt24_148_149    | 12               | 109                  |
|            | stmt24_7_8        | 5                | 9                    |
|            | stmt44_107_108    | 5                | 7                    |

When the instance was partitioned as in Sect. 7.1, the lengths of the individual decision lists were summed. In all cases, the size of the list is significantly smaller than the worst case, which is exponential in the number of input variables.
about by humans than a combinational circuit, means that it is easier for users to understand, trust and debug implementations produced by our algorithm. More than that, the cases described above have shown that our techniques can be used to analyze specifications and identify interesting properties and patterns, thus helping understand better the semantics of these specifications. This understanding might even be useful for explaining the performance of other tools and techniques on a given instance or family. As we mentioned at the end of the previous subsection, analyzing CNF formulas is hard, in large part due to the difficulty of identifying the underlying structure of the formula. The insights obtained from the execution of the Back-and-Forth algorithm can be a very important tool in this regard.

9 Discussion

A recurrent observation in recent evaluations [2, 3, 24, 43] 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 struggle with. 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.

Our results have shown that the combination of the decision-list format and the analysis performed by our algorithm is in fact able to identify and extract interesting patterns from the structure of the specification. These patterns contribute to producing implementations that are simpler and more regular than those constructed by other approaches, which is beneficial for taking maximum advantage of the useful properties of decision lists for validation, as described above. Furthermore, such an output can be valuable as a tool for understanding the underlying structure and semantics of the input CNF formula. This information can then be used, for example, to explain the performance of different algorithms, or decide on the best algorithm to use for a certain class of formulas.
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 for 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 Sect. 5) 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.

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. 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.

Acknowledgements 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, Saurabh Joshi for the help in using the Open-WBO tool, and the anonymous reviewers for their suggestions.

References

1. Akshay S, Arora J, Chakraborty S, Krishna S, Raghunathan D, Shah S (2019) Knowledge compilation for Boolean functional synthesis. In: Formal methods in computer-aided design, FMCAD 2019, pp 161–169
2. Akshay S, Chakraborty S, Goel S, Kulal S, Shah S (2018) What’s hard about Boolean functional synthesis? In: Computer aided verification–30th international conference, CAV 2018, pp 251–269. https://doi.org/10.1007/978-3-319-96145-3_14
3. Akshay S, Chakraborty S, John AK, Shah S (2017) Towards parallel Boolean functional synthesis. In: Tools and algorithms for the construction and analysis of systems–23rd international conference, TACAS 2017, pp 337–353. https://doi.org/10.1007/978-3-662-54577-5_19
4. Alur R, Madhusudan P, Nam W (2005) Symbolic computational techniques for solving games. STTT 7(2):118–128
5. Ansótegui C, Bonet ML, Levy J (2009) Solving (Weighted) partial MaxSAT through satisfiability testing. In: Theory and applications of satisfiability testing–SAT 2009–12th international conference, pp 427–440. https://doi.org/10.1007/978-3-642-02777-2_39
6. Audemard G, Simon L (2009) Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st international joint conference on artificial intelligence, IJCAI 2009, pp 399–404. http://ijcai.org/Proceedings/09/Papers/074.pdf
7. Bacchus F, Järvisalo M, Malik S (2021) Maximum satisfiability. In: Handbook of satisfiability, pp 929–991
8. Balabanov V, Jiang JHR (2012) Unified QBF certification and its applications. Form Methods Syst Des 41(1):45–65. https://doi.org/10.1007/s10703-012-0152-6
9. Balabanov V, Jiang JR (2011) Resolution proofs and Skolem functions in QBF evaluation and applications. In: Computer aided verification–23rd international conference, CAV 2011, pp 149–164. https://doi.org/10.1007/978-3-642-22110-1_12
10. Balabanov V, Widl M, Jiang JR (2014) QBF Resolution systems and their proof complexities. In: Theory and applications of satisfiability testing–SAT 2014–17th international conference, pp 154–169. https://doi.org/10.1007/978-3-319-09284-3_12
11. Boole G (1847) The mathematical analysis of logic. Philosophical Library, New York
12. Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Tran Comput 35(8):677–691. https://doi.org/10.1109/TC.1986.1676819
13. Chakraborty S, Fried D, Tabajara LM, Vardi MY (2018) Functional synthesis via input-output separation. In: Björner N, Gurfinkel A (eds) Formal methods in computer aided design FMCAD 2018. IEEE, New York
14. Eén N, Sörensson N (2003) An extensible SAT-solver. In: Theory and applications of satisfiability testing–SAT 2003–6th International Conference, pp 502–518. https://doi.org/10.1007/978-3-540-24605-3_37
15. Fried D, Legay A, Ouaknine J, Vardi MY (2018) Sequential relational decomposition. In: Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science, LICS 2018, pp 432–441. https://doi.org/10.1145/3209108.3209203
16. Fried D, Tabajara LM, Vardi MY (2016) BDD-Based Boolean functional synthesis. In: Computer aided verification–28th international conference, CAV 2016, pp 402–421. https://doi.org/10.1007/978-3-319-41540-6_22
17. Ganian R, Szeider S (2017) New width parameters for model counting. In: Theory and applications of satisfiability testing–sat 2017–20th international conference, pp 38–52. https://doi.org/10.1007/978-3-319-66263-3_3
18. Golia P, Roy S, Meel KS (2020) Manthan: a data driven approach for Boolean function synthesis. https://arxiv.org/abs/2005.06922
19. Heule M, Seidl M, Biere A (2014) Efficient extraction of skolem functions from QRAT proofs. In: Formal methods in computer-aided design, FMCAD 2014, pp 107–114. https://doi.org/10.1007/FMCAD.2014.6987602
20. Ignatiev A, Morgado A, Planes J, Marques-Silva J (2013) Maximal falsifiability–definitions, algorithms, and applications. In: Logic for programming, artificial intelligence, and reasoning–19th international conference, LPAR-19, pp 439–456. https://doi.org/10.1007/978-3-642-45221-5_30
21. Jacobs S, Bloem R, Bremgier R, Könighofer R, Pérez GA, Raskin J, Ryzhyk L, Sankur O, Seidl M, Tentrup L, Walker A (2015) The second reactive synthesis competition (SYNTCOMP 2015). In: Proceedings fourth workshop on synthesis, SYNT 2015, pp 27–57
22. Jiang JR (2009) Quantifier elimination via functional composition. In: Computer aided verification, 21st international conference, CAV 2009, pp 383–397. https://doi.org/10.1007/978-3-642-02658-4_30
23. Jo S, Matsumoto T, Fujita M (2012) SAT-based automatic rectification and debugging of combinational circuits with LUT insertions. In: Proceedings of the 2012 IEEE 21st Asian test symposium, ATS ’12, pp 19–24. IEEE Computer Society
24. John AK, Shah S, Chakraborty S, Trivedi A, Akshay S (2015) Skolem functions for factored formulas. In: Formal methods in computer-aided design, FMCAD 2015, pp 73–80
25. Kuehlmann A, Paruthi V, Krohm F, Ganai M (2002) Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans Comput Aided Des Integr Circuits Syst 21(12):1377–1394. https://doi.org/10.1109/TCAD.2002.804386
26. Kukula JH, Shiple TR (2000) Building circuits from relations. In: Computer aided verification, 12th international conference, CAV 2000, pp 113–123. https://doi.org/10.1007/10722167_12
27. Lakkaraju H, Bach SH, Jure L (2016) Interpretable decision sets: a joint framework for description and prediction. In: International conference on knowledge discovery and data mining (KDD), pp 1675V–1684
28. Li CM, Manyà F (2009) MaxSAT, hard and soft constraints. In: Handbook of satisfiability, pp 613–631. https://doi.org/10.3233/978-1-58603-929-5-613
29. Lowenheim L (1910) Über die Auflösung von Gleichungen in Logischen Gebietkalkul. Math Ann 68:169–207
30. Macii E, Odasso G, Poncino M (2006) Comparing different Boolean unification algorithms. In: Proceedings of 32nd Asilomar conference on signals, systems and computers, pp 17–29
31. Martins R, Manquinho VM, Lynce I (2014) Open-WBO: a modular MaxSAT solver. In: Theory and applications of satisfiability testing–SAT 2014–17th international conference, pp 438–445. https://doi.org/10.1007/978-3-319-09284-3_33
32. Narizzano M, Pulina L, Tacchella A (2006) The QBFEVAL web portal. In: Logics in artificial intelligence, pp 494–497. Springer, Berlin
33. Niemetz A, Preiner M, Lonsing F, Seidl M, Biere A (2012) Resolution-based certificate extraction for QBF - (tool presentation). In: Theory and applications of satisfiability testing–SAT 2012–15th international conference, pp 430–435
34. Rabe MN, Seshia SA (2016) Incremental determinization. In: Theory and applications of satisfiability testing–SAT 2016–19th International Conference, pp 375–392. https://doi.org/10.1007/978-3-319-40970-2_23
35. Rabe MN, Tentrup L (2015) CAQE: A certifying QBF solver. In: Formal methods in computer-aided design, FMCAD 2015, pp 136–143
36. Rivest RL (1987) Learning decision lists. Mach Learn 2(3):229–246. https://doi.org/10.1007/BF00058680
37. Silva JPM, Lynce I, Malik S (2009) Conflict-driven clause learning SAT solvers. In: Handbook of satisfiability, pp 131–153. https://doi.org/10.3233/978-1-58603-929-5-131
38. Solar-Lezama A (2013) Program sketching. STTT 15(5–6):475–495
39. Solar-Lezama A, Rabbah RM, Bodik R, Ebcioğlu K (2005) Programming by sketching for bit-streaming programs. In: Proceedings of the ACM SIGPLAN 2005 conference on programming language design and implementation, pp 281–294
40. Srivastava S, Gulwani S, Foster JS (2013) Template-based program verification and program synthesis. STTT 15(5–6):497–518
41. Stockmeyer LJ (1976) The polynomial-time hierarchy. Theor Comput Sci 3(1):1–22
42. Tabajara LM (2018) BDD-based Boolean synthesis. Master’s thesis, Rice University
43. Tabajara LM, Vardi MY (2017) Factored Boolean functional synthesis. In: Formal methods in computer aided design, FMCAD 2017, pp 124–131. https://doi.org/10.23919/FMCAD.2017.8102250
44. Zhu, S., Tabajara LM, Li J, Pu G, Vardi, MY (2017) Symbolic LTLf synthesis. In: Proceedings of the 26th international joint conference on artificial intelligence, IJCAI 2017, pp 1362–1369 . https://doi.org/10.24963/ijcai.2017/189

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.