What's Hard About Boolean Functional Synthesis?

S. Akshay(✉), Supratik Chakraborty,
Shubham Goel, Sumith Kulal, and Shetal Shah

Indian Institute of Technology Bombay,
Mumbai, India
akshayss@cse.iitb.ac.in

Abstract. Given a relational specification between Boolean inputs and outputs, the goal of Boolean functional synthesis is to synthesize each output as a function of the inputs such that the specification is met. In this paper, we first show that unless some hard conjectures in complexity theory are falsified, Boolean functional synthesis must generate large Skolem functions in the worst-case. Given this inherent hardness, what does one do to solve the problem? We present a two-phase algorithm, where the first phase is efficient both in terms of time and size of synthesized functions, and solves a large fraction of benchmarks. To explain this surprisingly good performance, we provide a sufficient condition under which the first phase must produce correct answers. When this condition fails, the second phase builds upon the result of the first phase, possibly requiring exponential time and generating exponential-sized functions in the worst-case. Detailed experimental evaluation shows our algorithm to perform better than other techniques for a large number of benchmarks.

Keywords: Skolem functions · Synthesis · SAT solvers
CEGAR based approach

1 Introduction

The algorithmic synthesis of Boolean functions satisfying relational specifications has long been of interest to logicians and computer scientists. Informally, given a Boolean relation between input and outupt variables denoting the specification, our goal is to synthesize each output as a function of the inputs such that the relational specification is satisfied. Such functions have also been called *Skolem functions* in the literature [23,29]. Boole [8] and Lowenheim [27] studied variants of this problem in the context of finding most general unifiers. While these studies are theoretically elegant, implementations of the underlying techniques have been found to scale poorly beyond small problem instances [28]. More recently, synthesis of Boolean functions has found important applications in a wide range of contexts including reactive strategy synthesis [4,19,40], certified QBF-SAT solving [7,21,31,34], automated program synthesis [35,37], circuit

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 251–269, 2018.
https://doi.org/10.1007/978-3-319-96145-3_14

repair and debugging [22], disjunctive decomposition of symbolic transition relations [39] and the like. This has spurred recent interest in developing practically efficient Boolean function synthesis algorithms. The resulting new generation of tools [3,17,23,29,33,34,38] have enabled synthesis of Boolean functions from much larger and more complex relational specifications than those that could be handled by earlier techniques, viz. [20,21,28].

In this paper, we re-examine the Boolean functional synthesis problem from both theoretical and practical perspectives. Our investigation shows that unless some hard conjectures in complexity theory are falsified, Boolean functional synthesis must necessarily generate super-polynomial sized Skolem functions, thereby requiring super-polynomial time, in the worst-case. Therefore, it is unlikely that an efficient algorithm exists for solving all instances of Boolean functional synthesis. There are two ways to address this hardness in practice: (i) design algorithms that are provably efficient but may give "approximate" Skolem functions that are correct on only a fraction of all possible input assignments, or (ii) design a phased algorithm, wherein the initial phase(s) is/are provably efficient and solve a subset of problem instances, and subsequent phase(s) have worst-case exponential behaviour and solve all remaining problem instances. In this paper, we combine the two approaches while giving heavy emphasis on efficient instances. We also provide a sufficient condition for our algorithm to be efficient, which indeed is borne out by our experiments.

The primary contributions of this paper can be summarized as follows.

1. We start by showing that unless $\mathsf{P} = \mathsf{NP}$, there exist problem instances where Boolean functional synthesis must take super-polynomial time. Moreover, if the non-uniform exponential time hypothesis [14] holds, there exist problem instances where Boolean functional synthesis must generate exponential sized Skolem functions, thereby also requiring at least exponential time.
2. We present a new two-phase algorithm for Boolean functional synthesis.
   (a) Phase 1 of our algorithm generates candidate Skolem functions of size polynomial in the input specification. This phase makes polynomially many calls to an NP oracle (SAT solver in practice). Hence it directly benefits from the progess made by the SAT solving community, and is efficient in practice. Our experiments indicate that Phase 1 suffices to solve a large majority of publicly available benchmarks.
   (b) However, there are indeed cases where the first phase is not enough (our theoretical results imply that such cases likely exist). In such cases, the first phase provides good candidate Skolem functions as starting points for the second phase. Phase 2 of our algorithm starts from these candidate Skolem functions, and uses a CEGAR-based approach to produce correct Skolem functions whose size may indeed be exponential in the input specification.
3. We analyze the surprisingly good performance of the first phase (especially in light of the theoretical hardness results) and show a sufficient condition on the structure of the input representation that guarantees correctness of the first phase. Interestingly, popular representations like ROBDDs [11] give

rise to input structures that satisfy this condition. The goodness of Skolem functions generated in this phase of the algorithm can also be quantified with high confidence by invoking an approximate model counter [13], whose complexity lies in $\mathsf{BPP}^{\mathsf{NP}}$.

4. We conduct an extensive set of experiments over a variety of benchmarks, and show that our algorithm performs favourably vis-a-vis state-of-the-art algorithms for Boolean functional synthesis.

Related Work. The literature contains several early theoretical studies on variants of Boolean functional synthesis [6,8,9,16,27,30]. More recently, researchers have tried to build practically efficient synthesis tools that scale to medium or large problem instances. In [29], Skolem functions for $\mathbf{X}$ are extracted from a proof of validity of $\forall\mathbf{Y}\exists\mathbf{X}\,F(\mathbf{X},\mathbf{Y})$. Unfortunately, this doesn't work when $\forall\mathbf{Y}\exists\mathbf{X}\,F(\mathbf{X},\mathbf{Y})$ is not valid, despite this class of problems being important, as discussed in [3,17]. Inspired by the spectacular effectiveness of CDCL-based SAT solvers, an incremental determinization technique for Skolem function synthesis was proposed in [33]. In [20,39], a synthesis approach based on iterated compositions was proposed. Unfortunately, as has been noted in [17,23], this does not scale to large benchmarks. A recent work [17] adapts the composition-based approach to work with ROBDDs. For factored specifications, ideas from symbolic model checking using implicitly conjoined ROBDDs have been used to enhance the scalability of the technique further in [38]. In the genre of CEGAR-based techniques, [23] showed how CEGAR can be used to synthesize Skolem functions from factored specifications. Subsequently, a compositional and parallel technique for Skolem function synthesis from arbitrary specifications represented using AIGs was presented in [3]. The second phase of our algorithm builds on some of this work. In addition to the above techniques, template-based [37] or sketch-based [36] approaches have been found to be effective for synthesis when we have information about the set of candidate solutions. A framework for functional synthesis that reasons about some unbounded domains such as integer arithmetic, was proposed in [25].

2 Notations and Problem Statement

A Boolean formula $F(z_1,\ldots z_p)$ on p variables is a mapping $F:\{0,1\}^p\rightarrow\{0,1\}$. The set of variables $\{z_1,\ldots z_p\}$ is called the *support* of the formula, and denoted $\mathsf{sup}(F)$. A *literal* is either a variable or its complement. We use $F|_{z_i=0}$ (resp. $F|_{z_i=1}$) to denote the positive (resp. negative) cofactor of F with respect to z_i. A *satisfying assignment* or *model* of F is a mapping of variables in $\mathsf{sup}(F)$ to $\{0,1\}$ such that F evaluates to 1 under this assignment. If π is a model of F, we write $\pi\models F$ and use $\pi(z_i)$ to denote the value assigned to $z_i\in\mathsf{sup}(F)$ by π. Let $\mathbf{Z}=(z_{i_1},z_{i_2},\ldots z_{i_j})$ be a sequence of variables in $\mathsf{sup}(F)$. We use $\pi{\downarrow}\mathbf{Z}$ to denote the projection of π on $\mathbf{Z}$, i.e. the sequence $(\pi(z_{i_1}),\pi(z_{i_2}),\ldots\pi(z_{i_j}))$.

A Boolean formula is in *negation normal form (NNF)* if (i) the only operators used in the formula are conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$), and (ii) negation is applied only to variables. Every Boolean formula can be converted

to a semantically equivalent formula in NNF. We assume an NNF formula is represented by a rooted directed acyclic graph (DAG), where internal nodes are labeled by $\wedge$ and $\vee$, and leaves are labeled by literals. In this paper, we use AIGs [24] as the initial representation of specifications. Given an AIG with t nodes, an equivalent NNF formula of size $\mathcal{O}(t)$ can be constructed in $\mathcal{O}(t)$ time. We use $|F|$ to denote the number of nodes in a DAG representation of F.

Let α be the subformula represented by an internal node N (labeled by $\wedge$ or $\vee$) in a DAG representation of an NNF formula. We use $lits(\alpha)$ to denote the set of literals labeling leaves that have a path to the node N representing α in the DAG. A formula is said to be in *weak decomposable NNF*, or wDNNF, if it is in NNF and if for every $\wedge$-labeled node in the DAG, the following holds: let $\alpha = \alpha_1 \wedge \ldots \wedge \alpha_k$ be the subformula represented by the internal node. Then, there is no literal l and distinct indices $i, j \in \{1, \ldots k\}$ such that $l \in lits(\alpha_i)$ and $\neg l \in lits(\alpha_j)$. Note that wDNNF is a weaker structural requirement on the NNF representation vis-a-vis the well-studied DNNF representation, which has elegant properties [15]. Specifically, every DNNF formula is also a wDNNF formula.

We say a *literal* l is *pure* in F iff the NNF representation of F has a leaf labeled l, but no leaf labeled $\neg l$. F is said to be *positive unate* in $z_i \in \mathsf{sup}(F)$ iff $F|_{z_i=0} \Rightarrow F|_{z_i=1}$. Similarly, F is said to be *negative unate* in z_i iff $F|_{z_i=1} \Rightarrow F|_{z_i=0}$. Finally, F is *unate* in z_i if F is either positive unate or negative unate in z_i. A function that is not unate in $z_i \in \mathsf{sup}(F)$ is said to be *binate* in z_i.

We also use $\mathbf{X} = (x_1, \ldots x_n)$ to denote a sequence of Boolean outputs, and $\mathbf{Y} = (y_1, \ldots y_m)$ to denote a sequence of Boolean inputs. The *Boolean functional synthesis* problem, henceforth denoted BFnS, asks: given a Boolean formula $F(\mathbf{X}, \mathbf{Y})$ specifying a relation between inputs $\mathbf{Y} = (y_1, \ldots y_m)$ and outputs $\mathbf{X} = (x_1, \ldots x_n)$, determine functions $\mathbf{\Psi} = (\psi_1(\mathbf{Y}), \ldots \psi_n(\mathbf{Y}))$ such that $F(\mathbf{\Psi}, \mathbf{Y})$ holds whenever $\exists \mathbf{X} F(\mathbf{X}, \mathbf{Y})$ holds. Thus, $\forall \mathbf{Y}(\exists \mathbf{X}\, F(\mathbf{X}, \mathbf{Y})) \Leftrightarrow F(\psi, \mathbf{Y})$ must be rendered valid. The function ψ_i is called a *Skolem function* for x_i in F, and $\mathbf{\Psi} = (\psi_1, \ldots \psi_n)$ is called a *Skolem function vector* for $\mathbf{X}$ in F.

For $1 \leq i \leq j \leq n$, let $\mathbf{X}_i^j$ denote the subsequence $(x_i, x_{i+1}, \ldots x_j)$ and let $F^{(i-1)}(\mathbf{X}_i^n, \mathbf{Y})$ denote $\exists \mathbf{X}_1^{i-1} F(\mathbf{X}_1^{i-1}, \mathbf{X}_i^n, \mathbf{Y})$. It has been argued in [3,17,20,23] that given a relational specification $F(\mathbf{X}, \mathbf{Y})$, the BFnS problem can be solved by first ordering the outputs, say as $x_1 \prec x_2 \cdots \prec x_n$, and then synthesizing a function $\psi_i(\mathbf{X}_{i+1}^n, \mathbf{Y})$ for each x_i such that $F^{(i-1)}(\psi_i, \mathbf{X}_{i+1}^n, \mathbf{Y}) \Leftrightarrow \exists x_i F^{(i-1)}(x_i, \mathbf{X}_{i+1}^n, \mathbf{Y})$. Once all such ψ_i are obtained, one can substitute ψ_{i+1} through ψ_n for x_{i+1} through x_n respectively, in ψ_i to obtain a Skolem function for x_i as a function of only $\mathbf{Y}$. We adopt this approach, and therefore focus on obtaining ψ_i in terms of $\mathbf{X}_{i+1}^n$ and $\mathbf{Y}$. Furthermore, we know from [20,23] that a function ψ_i is a Skolem function for x_i iff it satisfies ${\Delta_i}^F \Rightarrow \psi_i \Rightarrow \neg {\Gamma_i}^F$, where ${\Delta_i}^F \equiv \neg \exists \mathbf{X}_1^{i-1}\ F(\mathbf{X}_1^{i-1}, 0, \mathbf{X}_{i+1}^n, \mathbf{Y})$, and ${\Gamma_i}^F \equiv \neg \exists \mathbf{X}_1^{i-1} F(\mathbf{X}_1^{i-1}, 1, \mathbf{X}_{i+1}^n, \mathbf{Y})$. When F is clear from the context, we often omit it and write Δ_i and Γ_i. It is easy to see that both Δ_i and $\neg \Gamma_i$ serve as Skolem functions for x_i in F.

3 Complexity-Theoretical Limits

In this section, we investigate the computational complexity of BFnS. It is easy to see that BFnS can be solved in EXPTIME. Indeed a naive solution would be to enumerate all possible values of inputs $\mathbf{Y}$ and invoke a SAT solver to find values of $\mathbf{X}$ corresponding to each valuation of $\mathbf{Y}$ that makes $F(\mathbf{X}, \mathbf{Y})$ true. This requires worst-case time exponential in the number of inputs and outputs, and may produce an exponential-sized circuit. Given this, one can ask if we can develop a better algorithm that works faster and synthesizes "small" Skolem functions in all cases? Our first result shows that existence of such small Skolem functions would violate hard complexity-theoretic conjectures.

Theorem 1. *1. Unless* $\mathsf{P} = \mathsf{NP}$*, there exist problem instances where any algorithm for* BFnS *must take super-polynomial time*[1]*.*

2. Unless the non-uniform exponential-time hypothesis (or $\mathsf{ETH}_{\mathsf{nu}}$) *fails, there exist problem instances where any algorithm for* BFnS *must generate Skolem functions of size exponential in the input size.*

A consequence of the second statement is that, under the same hypothesis, there must exist an instance of BFnS for which any algorithm must take EXPTIME time. The exponential-time hypothesis ETH and its strengthened version, the non-uniform exponential-time hypothesis $\mathsf{ETH}_{\mathsf{nu}}$, are unproven computational hardness assumptions (see [14,18]), which have been used to show that several classical decision, functional and parametrized NP-complete problems (such as p-Clique) are unlikely to have sub-exponential algorithms. $\mathsf{ETH}_{\mathsf{nu}}$ states that there is no family of algorithms (one for each family of inputs of size n) that can solve 3-SAT in subexponential time. In [14] it is shown that if $\mathsf{ETH}_{\mathsf{nu}}$ holds, then *p-Clique, the parametrized clique problem*, cannot be solved in sub-exponential time, i.e., for all $d \in \mathbb{N}$, and sufficiently large fixed k, determining whether a graph G has a clique of size k is not in $\mathsf{DTIME}(n^d)$.

Proof. We describe a reduction from p-Clique to BFnS. Given an undirected graph $G = (V, E)$ on n-vertices and a number k (encoded in binary), we want to check if G has a clique of size k. We encode the graph as follows: each vertex $v \in V$ is identified by a unique number in $\{1, \ldots n\}$, and for every $(i, j) \in V \times V$, we introduce an input variable $y_{i,j}$ that is set to 1 iff $(i, j) \in E$. We call the resulting vector of input variables $\boldsymbol{y}$. We also have additional input variables $\boldsymbol{z} = z_1, \ldots z_m$, which represent the binary encoding of k ($m = \lceil \log_2 k \rceil$). Finally, we introduce output variables x_v for each $v \in V$, whose values determine which vertices are present in the clique. Let $\boldsymbol{x}$ denote the vector of x_v variables.

Given inputs $\mathbf{Y} = \{\boldsymbol{y}, \boldsymbol{z}\}$, and outputs $\mathbf{X} = \{\boldsymbol{x}\}$, our specification is represented by a circuit F over $\mathbf{X}, \mathbf{Y}$ that verifies whether the vertices encoded by $\mathbf{X}$ indeed form a k-clique of the graph G. The circuit F is constructed as follows:

[1] Since the submission of this paper, we have obtained a sharper complexity result. Details of this can be found in [2].

1. For every i, j such that $1 \leq i < j \leq n$, we construct a sub-circuit implementing $x_i \wedge x_j \Rightarrow y_{i,j}$. The outputs of all such subcircuits are conjoined to give an intermediate output, say $\mathsf{EdgesOK}$. Clearly, all the subcircuits taken together have size $\mathcal{O}(n^2)$.
2. We have a tree of binary adders implementing $x_1 + x_2 + \ldots x_n$. Let the $\lceil \log_2 n \rceil$-bit output of the adder be denoted $\mathsf{CliqueSz}$. The size of this adder is clearly $\mathcal{O}(n)$.
3. We have an equality checker that checks if $\mathsf{CliqueSz} = k$. Clearly, this subcircuit has size $\lceil \log_2 n \rceil$. Let the output of this equality checker be called SizeOK.
4. The output of the specification circuit F is $\mathsf{EdgesOK} \wedge \mathsf{SizeOK}$.

Given an instance $\mathbf{Y} = \{\boldsymbol{y}, \boldsymbol{z}\}$ of p-Clique, we now consider the specification $F(\mathbf{X}, \mathbf{Y})$ as constructed above and feed it as input to any algorithm A for solving BFnS. Let $\boldsymbol{\Psi}$ be the Skolem function vector output by A. For each $i \in \{1, \ldots n\}$, we now feed ψ_i to the input y_i of the circuit F. This effectively constructs a circuit for $F(\boldsymbol{\Psi}, \mathbf{Y})$. It is easy to see from the definition of Skolem functions that for every valuation of $\mathbf{Y}$, the function $F(\boldsymbol{\Psi}, \mathbf{Y})$ evaluates to 1 iff the graph encoded by $\mathbf{Y}$ contains a clique of size k.

Using this reduction, we can complete the proofs of both our statements:

1. If the circuits for the Skolem functions $\boldsymbol{\Psi}$ are super-polynomial sized, then of course any algorithm generating $\boldsymbol{\Psi}$ must take super-polynomial time. On the other hand, if the circuits for the Skolem functions $\boldsymbol{\Psi}$ are always poly-sized, then $F(\boldsymbol{\Psi}, \mathbf{Y})$ is polynomial-sized, and evaluating it takes time that is polynomial in the input size. Thus, if A is a polynomial-time algorithm, we also get an algorithm for solving p-Clique in polynomial time, which implies that $\mathsf{P} = \mathsf{NP}$.
2. If the circuits for the Skolem functions $\boldsymbol{\Psi}$ are sub-exponential sized in the input n, then $F(\boldsymbol{\Psi}, \mathbf{Y})$ is also sub-exponential sized and can be evaluated in sub-exponential time. It then follows that we can solve any instance p-Clique of input length n in sub-exponential time – a violation of $\mathsf{ETH}_{\mathsf{nu}}$. Note that since our circuits can be different for different input lengths, we may have different algorithms for different n. Hence we have to appeal to the non-uniform variant of ETH. □

Theorem 1 implies that efficient algorithms for BFnS are unlikely. We therefore propose a two-phase algorithm to solve BFnS in practice. The first phase runs in polynomial time relative to an NP-oracle and generates polynomial-sized "approximate" Skolem functions. We show that under certain structural restrictions on the NNF representation of F, the first phase always returns exact Skolem functions. However, these structural restrictions may not always be met. An NP-oracle can be used to check if the functions computed by the first phase are indeed exact Skolem functions. In case they aren't, we proceed to the second phase of our algorithm that runs in worst-case exponential time. Below, we discuss the first phase in detail. The second phase is an adaptation of an existing CEGAR-based technique and is described briefly later.

4 Phase 1: Efficient Polynomial-Sized Synthesis

An easy consequence of the definition of unateness is the following.

Proposition 1. *If $F(\mathbf{X}, \mathbf{Y})$ is positive (resp. negative) unate in x_i, then $\psi_i = 1$ (resp. $\psi_i = 0$) is a correct Skolem function for x_i.*

All omitted proofs, including that of the above, may be found in [2]. The above result gives us a way to identify outputs x_i for which a Skolem function can be easily computed. Note that if x_i (resp. $\neg x_i$) is a pure literal in F, then F is positive (resp. negative) unate in x_i. However, the converse is not necessarily true. In general, a semantic check is necessary for unateness. In fact, it follows from the definition of unateness that F is positive (resp. negative) unate in x_i, iff the formula η_i^+ (resp. η_i^-) defined below is unsatisfiable.

$$\eta_i^+ = F(\mathbf{X}_1^{i-1}, 0, \mathbf{X}_{i+1}^n, \mathbf{Y}) \wedge \neg F(\mathbf{X}_1^{i-1}, 1, \mathbf{X}_{i+1}^n, \mathbf{Y}). \quad (1)$$

$$\eta_i^- = F(\mathbf{X}_1^{i-1}, 1, \mathbf{X}_{i+1}^n, \mathbf{Y}) \wedge \neg F(\mathbf{X}_1^{i-1}, 0, \mathbf{X}_{i+1}^n, \mathbf{Y}). \quad (2)$$

Note that each such check involves a single invocation of an NP-oracle, and a variant of this method is described in [5].

If F is binate in an output x_i, Proposition 1 doesn't help in synthesizing ψ_i. Towards synthesizing Skolem functions for such outputs, recall the definitions of Δ_i and Γ_i from Sect. 2. Clearly, if we can compute these functions, we can solve BFnS. While computing Δ_i and Γ_i *exactly* for all x_i is unlikely to be efficient in general (in light of Theorem 1), we show that polynomial-sized "good" approximations of Δ_i and Γ_i can be computed efficiently. As our experiments show, these approximations are good enough to solve BFnS for several benchmarks. Furthermore, with access to an NP-oracle, we can also check when these approximations are indeed good enough.

Given a relational specification $F(\mathbf{X}, \mathbf{Y})$, we use $\widehat{F}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y})$ to denote the formula obtained by first converting F to NNF, and then replacing every occurrence of $\neg x_i$ ($x_i \in \mathbf{X}$) in the NNF formula with a fresh variable $\overline{x_i}$. As an example, suppose $F(\mathbf{X}, \mathbf{Y}) = (x_1 \vee \neg(x_2 \vee y_1)) \vee \neg(x_2 \vee \neg(y_2 \wedge \neg y_1))$. Then $\widehat{F}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y}) = (x_1 \vee (\overline{x_2} \wedge \neg y_1)) \vee (\overline{x_2} \wedge y_2 \wedge \neg y_1)$. The following are easy to see.

Proposition 2. *(a) $\widehat{F}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y})$ is positive unate in both $\mathbf{X}$ and $\overline{\mathbf{X}}$.*
(b) Let $\neg\mathbf{X}$ denote $(\neg x_1, \ldots \neg x_n)$. Then $F(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \widehat{F}(\mathbf{X}, \neg\mathbf{X}, \mathbf{Y})$.

For every $i \in \{1, \ldots n\}$, we can split $\mathbf{X} = (x_1, \ldots x_n)$ into two parts, $\mathbf{X}_1^i$ and $\mathbf{X}_{i+1}^n$, and represent $\widehat{F}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y})$ as $\widehat{F}(\mathbf{X}_1^i, \mathbf{X}_{i+1}^n, \overline{\mathbf{X}}_1^i, \overline{\mathbf{X}}_{i+1}^n, \mathbf{Y})$. We use these representations of $\widehat{F}$ interchangeably, depending on the context. For $b, c \in \{0, 1\}$, let $\mathbf{b}^i$ (resp. $\mathbf{c}^i$) denote a vector of i b's (resp. c's). For notational convenience, we use $\widehat{F}(\mathbf{b}^i, \mathbf{X}_{i+1}^n, \mathbf{c}^i, \overline{\mathbf{X}}_{i+1}^n, \mathbf{Y})$ to denote $\widehat{F}(\mathbf{X}_1^i, \mathbf{X}_{i+1}^n, \overline{\mathbf{X}}_1^i, \overline{\mathbf{X}}_{i+1}^n, \mathbf{Y})|_{\mathbf{X}_1^i = \mathbf{b}^i, \overline{\mathbf{X}}_1^i = \mathbf{c}^i}$ in the subsequent discussion. The following is an easy consequence of Proposition 2.

Proposition 3. *For every $i \in \{1, \ldots n\}$, the following holds:*
$\widehat{F}(\mathbf{0}^i, \mathbf{X}_{i+1}^n, \mathbf{0}^i, \neg\mathbf{X}_{i+1}^n, \mathbf{Y}) \Rightarrow \exists \mathbf{X}_1^i F(\mathbf{X}, \mathbf{Y}) \Rightarrow \widehat{F}(\mathbf{1}^i, \mathbf{X}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{X}_{i+1}^n, \mathbf{Y})$

Proposition 3 allows us to bound Δ_i and Γ_i as follows.

Lemma 1. *For every $x_i \in \mathbf{X}$, we have:*

(a) $\neg\widehat{F}(\mathbf{1}^{i-1}0, \mathbf{X}^n_{i+1}, \mathbf{1}^i, \neg\mathbf{X}^n_{i+1}, \mathbf{Y}) \Rightarrow \Delta_i \Rightarrow \neg\widehat{F}(\mathbf{0}^i, \mathbf{X}^n_{i+1}, \mathbf{0}^{i-1}1, \neg\mathbf{X}^n_{i+1}, \mathbf{Y})$

(b) $\neg\widehat{F}(\mathbf{1}^i, \mathbf{X}^n_{i+1}, \mathbf{1}^{i-1}0, \neg\mathbf{X}^n_{i+1}, \mathbf{Y}) \Rightarrow \Gamma_i \Rightarrow \neg\widehat{F}(\mathbf{0}^{i-1}1, \mathbf{X}^n_{i+1}, \mathbf{0}^i, \neg\mathbf{X}^n_{i+1}, \mathbf{Y})$

In the remainder of the paper, we only use under-approximations of Δ_i and Γ_i, and use δ_i and γ_i respectively, to denote them. Recall from Sect. 2 that both Δ_i and $\neg\Gamma_i$ suffice as Skolem functions for x_i. Therefore, we propose to use either δ_i or $\neg\gamma_i$ (depending on which has a smaller AIG) obtained from Lemma 1 as our approximation of ψ_i. Specifically,

$$\begin{aligned} &\delta_i = \neg\widehat{F}(\mathbf{1}^{i-1}0, \mathbf{X}^n_{i+1}, \mathbf{1}^i, \neg\mathbf{X}^n_{i+1}, \mathbf{Y}),\ \gamma_i = \neg\widehat{F}(\mathbf{1}^i, \mathbf{X}^n_{i+1}, \mathbf{1}^{i-1}0, \neg\mathbf{X}^n_{i+1}, \mathbf{Y}) \\ &\psi_i = \delta_i \text{ or } \neg\gamma_i, \text{ depending on which has a smaller AIG} \end{aligned} \tag{3}$$

Example 1. Consider the specification $\mathbf{X} = \mathbf{Y}$, expressed in NNF as $F(\mathbf{X}, \mathbf{Y}) \equiv \bigwedge_{i=1}^n ((x_i \wedge y_i) \vee (\neg x_i \wedge \neg y_i))$. As noted in [33], this is a difficult example for CEGAR-based QBF solvers, when n is large.

From Eq. 3, $\delta_i = \neg(\neg y_i \wedge \bigwedge_{j=i+1}^n (x_j \Leftrightarrow y_j)) = y_i \vee \bigvee_{j=i+1}^n (x_j \Leftrightarrow \neg y_j)$, and $\gamma_i = \neg(y_i \wedge \bigwedge_{j=i+1}^n (x_j \Leftrightarrow y_j)) = \neg y_i \vee \bigvee_{j=i+1}^n (x_j \Leftrightarrow \neg y_j)$. With δ_i as the choice of ψ_i, we obtain $\psi_i = y_i \vee \bigvee_{j=i+1}^n (x_j \Leftrightarrow \neg y_j)$. Clearly, $\psi_n = y_n$. On reverse-substituting, we get $\psi_{n-1} = y_{n-1} \vee (\psi_n \Leftrightarrow \neg y_n) = y_{n-1} \vee 0 = y_{n-1}$. Continuing in this way, we get $\psi_i = y_i$ for all $i \in \{1, \ldots n\}$. The same result is obtained regardless of whether we choose δ_i or $\neg\gamma_i$ for each ψ_i. Thus, our approximation is good enough to solve this problem. In fact, it can be shown that $\delta_i = \Delta_i$ and $\gamma_i = \Gamma_i$ for all $i \in \{1, \ldots n\}$ in this example. □

Note that the approximations of Skolem functions, as given in Eq. (3), are efficiently computable for all $i \in \{1, \ldots n\}$, as they involve evaluating $\widehat{F}$ with a subset of inputs set to constants. This takes no more than $\mathcal{O}(|F|)$ time and space. As illustrated by Example 1, these approximations also often suffice to solve BFnS. The following lemma partially explains this.

Theorem 2. *(a) For $i \in \{1, \ldots n\}$, suppose the following holds:*

$$\begin{aligned} \forall j \in \{1, \ldots i\}\ \ \widehat{F}(\mathbf{1}^j, \mathbf{X}^n_{j+1}, \mathbf{1}^j, \overline{\mathbf{X}}^n_{j+1}, \mathbf{Y}) \Rightarrow\ & \widehat{F}(\mathbf{1}^{j-1}0, \mathbf{X}^n_{j+1}, \mathbf{1}^{j-1}1, \overline{\mathbf{X}}^n_{j+1}, \mathbf{Y}) \\ & \vee\ \widehat{F}(\mathbf{1}^{j-1}1, \mathbf{X}^n_{j+1}, \mathbf{1}^{j-1}0, \overline{\mathbf{X}}^n_{j+1}, \mathbf{Y}) \end{aligned}$$

Then $\exists\mathbf{X}^i_1 F(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \widehat{F}(\mathbf{1}^i, \mathbf{X}^n_{i+1}, \mathbf{1}^i, \neg\mathbf{X}^n_{i+1}, \mathbf{Y})$.

(b) If $\widehat{F}(\mathbf{X}, \neg\mathbf{X}, \mathbf{Y})$ is in wDNNF, then $\delta_i = \Delta_i$ and $\gamma_i = \Gamma_i$ for every $i \in \{1, \ldots n\}$.

Proof. To prove part (a), we use induction on i. The base case corresponds to $i = 1$. Recall that $\exists\mathbf{X}^1_1 F(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \widehat{F}(1, \mathbf{X}^n_2, 0, \neg\mathbf{X}^n_2, \mathbf{Y}) \vee F(0, \mathbf{X}^n_2, 1, \neg\mathbf{X}^n_2, \mathbf{Y})$ by definition. Proposition 3 already asserts that $\exists\mathbf{X}^1_1 F(\mathbf{X}, \mathbf{Y}) \Rightarrow \widehat{F}(1, \mathbf{X}^n_2, 1, \neg\mathbf{X}^n_2, \mathbf{Y})$. Therefore, if the condition in Theorem 2(a) holds for $i = 1$, we then have

$\widehat{F}(1, \mathbf{X}_2^n, 1, \neg\mathbf{X}_2^n, \mathbf{Y}) \Leftrightarrow \widehat{F}(1, \mathbf{X}_2^n, 0, \neg\mathbf{X}_2^n, \mathbf{Y}) \vee F(0, \mathbf{X}_2^n, 1, \neg\mathbf{X}_2^n, \mathbf{Y})$, which in turn is equivalent to $\exists \mathbf{X}_1^1 F(\mathbf{X}, \mathbf{Y})$. This proves the base case.

Let us now assume (inductive hypothesis) that the statement of Theorem 2(a) holds for $1 \leq i < n$. We prove below that the same statement holds for $i+1$ as well. Clearly, $\exists \mathbf{X}_1^{i+1} F(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \exists x_{i+1} \left(\exists \mathbf{X}_1^i F(\mathbf{X}, \mathbf{Y})\right)$. By the inductive hypothesis, this is equivalent to $\exists x_{i+1} \widehat{F}(\mathbf{1}^i, \mathbf{X}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{X}_{i+1}^n, \mathbf{Y})$. By definition of existential quantification, this is equivalent to $\widehat{F}(\mathbf{1}^{i+1}, \mathbf{X}_{i+2}^n, \mathbf{1}^i 0, \neg\mathbf{X}_{i+2}^n, \mathbf{Y}) \vee \widehat{F}(\mathbf{1}^i 0, \mathbf{X}_{i+2}^n, \mathbf{1}^{i+1}, \neg\mathbf{X}_{i+2}^n, \mathbf{Y})$. From the condition in Theorem 2(a), we also have

$$\widehat{F}(\mathbf{1}^{i+1}, \mathbf{X}_{i+2}^n, \mathbf{1}^{i+1}, \overline{\mathbf{X}}_{i+2}^n, \mathbf{Y}) \Rightarrow \widehat{F}(\mathbf{1}^i 0, \mathbf{X}_{i+2}^n, \mathbf{1}^{i+1}, \overline{\mathbf{X}}_{i+2}^n, \mathbf{Y}) \vee \widehat{F}(\mathbf{1}^{i+1}, \mathbf{X}_{i+2}^n, \mathbf{1}^i 0, \overline{\mathbf{X}}_{i+2}^n, \mathbf{Y})$$

The implication in the reverse direction follows from Proposition 2(a). Thus we have a bi-implication above, which we have already seen is equivalent to $\exists \mathbf{X}_1^{i+1} F(\mathbf{X}, \mathbf{Y})$. This proves the inductive case.

To prove part (b), we first show that if $\widehat{F}(\mathbf{X}, \neg\mathbf{X}, \mathbf{Y})$ is in wDNNF, then the condition in Theorem 2(a) must hold for all $j \in \{1, \ldots n\}$. Theorem 2(b) then follows from the definitions of Δ_i and Γ_i (see Sect. 2), from the statement of Theorem 2(a) and from the definitions of δ_i and γ_i (see Eq. 3).

For $j \in \{1, \ldots n\}$, let $\zeta(\mathbf{X}_{j+1}^n, \overline{\mathbf{X}}_{j+1}^n, \mathbf{Y})$ denote the formula $\widehat{F}(\mathbf{1}^j, \mathbf{X}_{j+1}^n, \mathbf{1}^j, \overline{\mathbf{X}}_{j+1}^n, \mathbf{Y}) \wedge \neg\left(\widehat{F}(\mathbf{1}^{j-1}0, \mathbf{X}_{j+1}^n, \mathbf{1}^{j-1}1, \overline{\mathbf{X}}_{j+1}^n, \mathbf{Y}) \vee \widehat{F}(\mathbf{1}^{j-1}1, \mathbf{X}_{j+1}^n, \mathbf{1}^{j-1}0, \overline{\mathbf{X}}_{j+1}^n, \mathbf{Y})\right)$. Suppose, if possible, $\widehat{F}(\mathbf{X}, \neg\mathbf{X}, \mathbf{Y})$ is in wDNNF but there exists j ($1 \leq j \leq n$) such that $\zeta(\mathbf{X}_{j+1}^n, \overline{\mathbf{X}}_{j+1}^n, \mathbf{Y})$ is satisfiable. Let $\mathbf{X}_{j+1}^n = \sigma$, $\overline{\mathbf{X}}_{j+1}^n = \kappa$ and $\mathbf{Y} = \theta$ be a satisfying assignment of ζ. We now consider the simplified circuit obtained by substituting $\mathbf{1}^{j-1}$ for $\mathbf{X}_1^{j-1}$ as well as for $\overline{\mathbf{X}}_1^{j-1}$, σ for $\mathbf{X}_{j+1}^n$, κ for $\overline{\mathbf{X}}_{j+1}^n$ and θ for $\mathbf{Y}$ in the AIG for $\widehat{F}$. This simplification replaces the output of every internal node with a constant (0 or 1), if the node evaluates to a constant under the above assignment. Note that the resulting circuit can have only x_j and $\overline{x}_j$ as its inputs. Furthermore, since the assignment satisfies ζ, it follows that the simplified circuit evaluates to 1 if both x_j and $\overline{x}_j$ are set to 1, and it evaluates to 0 if any one of x_j or $\overline{x}_j$ is set to 0. This can only happen if there is a node labeled $\wedge$ in the AIG representing $\widehat{F}(\mathbf{X}, \neg\mathbf{X}, \mathbf{Y})$ with a path leading from the leaf labeled x_j, and another path leading from the leaf labeled $\neg x_j$. This is a contradiction, since $\widehat{F}(\mathbf{X}, \neg\mathbf{X}, \mathbf{Y})$ is in wDNNF. Therefore, there is no $j \in \{1, \ldots n\}$ such that the condition of Theorem 2(a) is violated. □

In general, the candidate Skolem functions generated from the approximations discussed above may not always be correct. Indeed, the conditions discussed above are only sufficient, but not necessary, for the approximations to be exact. Hence, we need a separate check to see if our candidate Skolem functions are correct. To do this, we use an *error formula* $\varepsilon_{\Psi}(\mathbf{X}', \mathbf{X}, \mathbf{Y}) \equiv F(\mathbf{X}', \mathbf{Y}) \wedge \bigwedge_{i=1}^n (x_i \leftrightarrow \psi_i) \wedge \neg F(\mathbf{X}, \mathbf{Y})$, as described in [23], and check its satisfiability. The correctness of this check depends on the following result from [23].

Theorem 3 ([23]). *$\varepsilon_{\mathbf{\Psi}}$ is unsatisfiable iff $\mathbf{\Psi}$ is a correct Skolem function vector.*

Algorithm 1. BFSS

Input: $\widehat{F}(\mathbf{X}, \mathbf{Y})$ in NNF (or wDNNF) with inputs $|\mathbf{Y}| = m$, outputs $|\mathbf{X}| = n$,
Output: Candidate Skolem Functions $\mathbf{\Psi} = (\psi_1, \ldots, \psi_n)$

1 **Initialize**: Fix sets $U_0 = U_1 = \emptyset$;
2 **repeat**
3 `// Repeatedly checks for Unate variables`
4 **for** *each* $x_i \in \mathbf{X} \setminus (U_0 \cup U_1)$ **do**
5 **if** $\widehat{F}$ *is positive unate in* x_i `// check` x_i `pure or` η_i^+ `(Eq 1) SAT ;`
6 **then**
7 $\widehat{F} := \widehat{F}[x_i = 1]$, $U_1 = U_1 \cup \{x_i\}$
8 **else if** $\widehat{F}$ *is negative unate in* x_i `//` $\neg x_i$ `pure or` η^- `(Eq 2)SAT ;`
9 **then**
10 $\widehat{F} := \widehat{F}[x_i = 0]$, $U_0 = U_0 \cup \{x_i\}$
11 **until** *F is unchanged* `// No Unate variables remaining;`
12 Choose an ordering $\preceq$ of $\mathbf{X}$ `// Section 6 discusses ordering used;`
13 **for** *each* $x_i \in \mathbf{X}$ *in* $\preceq$ *order* **do**
14 **if** $x_i \in U_j$ *for* $j \in \{0, 1\}$ `// Assume` $x_1 \preceq x_2 \preceq \ldots x_n$;
15 **then**
16 $\psi_i = j$
17 **else**
18 ψ_i is as defined in (Eq 3)
19 **if** *error formula* $\epsilon_{\mathbf{\Psi}}$ *is UNSAT* **then**
20 terminate and output $\mathbf{\Psi}$
21 **else**
22 call Phase 2

We now combine all the above ingredients to come up with algorithm BFSS (for *Blazingly Fast Skolem Synthesis*), as shown in Algorithm 1. The algorithm can be divided into three parts. In the first part (lines 2-11), unateness is checked. This is done in two ways: (i) we identify pure literals in F by simply examining the labels of leaves in the DAG representation of F in NNF, and (ii) we check the satisfiability of the formulas η_i^+ and η_i^-, as defined in Eqs. 1 and 2. This requires invoking a SAT solver in the worst-case, and is repeated at most $\mathcal{O}(n^2)$ times until there are no more unate variables. Hence this requires $\mathcal{O}(n^2)$ calls to a SAT solver. Once we have done this, by Proposition 1, the constants 1 or 0 (for positive or negative unate variables respectively) are correct Skolem functions for these variables.

In the second part, we fix an ordering of the remaining output variables according to an experimentally sound heuristic, as described in Sect. 6, and compute candidate Skolem functions for these variables according to Eq. 3. We then

check the satisfiability of the error formula ϵ_{Ψ} to determine if the candidate Skolem functions are indeed correct. If the error formula is found to be unsatisfiable, we know from Theorem 3 that we have the correct Skolem functions, which can therefore be output. This concludes phase 1 of algorithm BFSS. If the error formula is found to be satisfiable, we move to phase 2 of algorithm BFSS – an adaptation of the CEGAR-based technique described in [23], and discussed briefly in Sect. 5. It is not difficult to see that the running time of phase 1 is polynomial in the size of the input, relative to an NP-oracle (SAT solver in practice). This also implies that the Skolem functions generated can be of at most polynomial size. Finally, from Theorem 2 we also obtain that if F satisfies Theorem 2(a), Skolem functions generated in phase 1 are correct. From the above reasoning, we obtain the following properties of phase 1 of BFSS:

Theorem 4. *1. For all unate variables, phase 1 of* BFSS *computes correct Skolem functions.*
2. If $\hat{F}$ is in wDNNF, phase 1 of BFSS *computes all Skolem functions correctly.*
3. The running time of phase 1 of BFSS *is polynomial in input size, relative to an* NP*-oracle. Specifically, the algorithm makes $\mathcal{O}(n^2)$ calls to an* NP*-oracle.*
4. The candidate Skolem functions output by phase 1 of BFSS *have size at most polynomial in the size of the input.*

Discussion: We make two crucial and related observations. First, by our hardness results in Sect. 3, we know that the above algorithm cannot solve BFnS for all inputs, unless some well-regarded complexity-theoretic conjectures fail. As a result, we must go to phase 2 on at least some inputs. Surprisingly, our experiments show that this is not necessary in the majority of benchmarks.

The second observation tries to understand why phase 1 works in most cases in practice. While a conclusive explanation isn't easy, we believe Theorem 2 explains the success of phase 1 in several cases. By [15], we know that all Boolean functions have a DNNF (and hence wDNNF) representation, although it may take exponential time to compute this representation. This allows us to define two preprocessing procedures. In the first, we identify cases where we can directly convert to wDNNF and use the Phase 1 algorithm above. And in the second, we use several optimization scripts available in the ABC [26] library to optimize the AIG representation of $\widehat{F}$. For a majority of benchmarks, this appears to yield a representation of $\widehat{F}$ that allows the proof of Theorem 2(a) to go through. For the rest, we apply the Phase 2 algorithm as described below.

Quantitative guarantees of "goodness". Given our theoretical and practical insights of the applicability of phase 1 of BFSS, it would be interesting to measure how much progress we have made in phase 1, even if it does not give the correct Skolem functions. One way to measure this "goodness" is to estimate the number of counterexamples as a fraction of the size of the input space. Specifically, given the error formula, we get an approximate count of the number of models for this formula *projected on the inputs* $\mathbf{Y}$. This can be obtained efficiently in practice with high confidence using state-of-the-art approximate model counters, viz. [13], with complexity in $\mathsf{BPP}^{\mathsf{NP}}$. The approximate count thus obtained, when divided

by $2^{|\mathbf{Y}|}$ gives the fraction of input combinations for which the candidate Skolem functions output by phase 1 do not work correctly. We call this the *goodness ratio* of our approximation.

5 Phase 2: Counterexample-Guided Refinement

For phase 2, we can use any off-the-shelf worst-case exponential-time Skolem function generator. However, given that we already have candidate Skolem functions with guarantees on their "goodness", it is natural to use them as starting points for phase 2. Hence, we start off with candidate Skolem functions for all x_i as computed in phase 1, and then update (or refine) them in a counterexample-driven manner. Intuitively, a counterexample is a value of the inputs $\mathbf{Y}$ for which there exists a value of $\mathbf{X}$ that renders $F(\mathbf{X}, \mathbf{Y})$ true, but for which $F(\mathbf{\Psi}, \mathbf{Y})$ evaluates to false. As shown in [23], given a candidate Skolem function vector, every satisfying assignment of the error formula $\varepsilon_{\mathbf{\Psi}}$ gives a counterexample. The refinement step uses this satisfying assignment to update an appropriate subset of the approximate δ_i and γ_i functions computed in phase 1. The entire process is then repeated until no counterexamples can be found. The final updated vector of Skolem functions then gives a solution of the BFnS problem. Note that this idea is not new [3,23]. The only significant enhancement we do over the algorithm in [23] is to use an almost-uniform sampler [12] to efficiently sample the space of counterexamples almost uniformly. This allows us to do refinement with a diverse set of counterexamples, instead of using counterexamples in a corner of the solution space of $\varepsilon_{\mathbf{\Psi}}$ that the SAT solver heuristics zoom down on.

6 Experimental Results

Experimental methodology. Our implementation consists of two parallel pipelines that accept the same input specification but represent them in two different ways. The first pipeline takes the input formula as an AIG and builds an NNF (not necessarily wDNNF) DAG, while the second pipeline builds an ROBDD from the input AIG using dynamic variable reordering (no restrictions on variable order), and then obtains a wDNNF representation from it using the linear-time algorithm described in [15]. Once the NNF/wDNNF representation is built, we use Algorithm 1 in Phase 1 and CEGAR-based synthesis using UniGen [12] to sample counterexamples in Phase 2. We call this ensemble of two pipelines as bfss. We compare bfss with the following algorithms/tools: (*i*) parSyn [3], (*ii*) Cadet [33], (*iii*) RSynth [38], and (*iv*) AbsSynthe-Skolem (based on the BFnS step of AbsSynthe [10]).

Our implementation of bfss uses the ABC [26] library to represent and manipulate Boolean functions. Two different SAT solvers can be used with bfss: ABC's default SAT solver, or UniGen [12] (to give almost-uniformly distributed counterexamples). All our experiments use UniGen.

We consider a total of 504 benchmarks, taken from four different domains: (a) forty-eight *Arithmetic benchmarks* from [17], with varying bit-widths (viz.

32, 64, 128, 256, 512 and 1024) of arithmetic operators, (b) sixty-eight *Disjunctive Decomposition benchmarks* from [3], generated by considering some of the larger sequential circuits in the HWMCC10 benchmark suite, (c) five *Factorization benchmarks*, also from [3], representing factorization of numbers of different bit-widths (8, 10, 12, 14, 16), and (d) three hundred and eighty three *QBFEval benchmarks*, taken from the Prenex 2QBF track of QBFEval 2017 [32][2]. Since different tools accept benchmarks in different formats, each benchmark was converted to both `qdimacs` and `verilog/aiger` formats. All benchmarks and the procedure by which we generated (and converted) them are detailed in [1]. Recall that we use two pipelines for BFSS. We use "balance; rewrite -l; refactor -l; balance; rewrite -l; rewrite -lz; balance; refactor -lz; rewrite -lz; balance" as the ABC script for optimizing the AIG representation of the input specification. We observed that while this results in only 4 benchmarks being in wDNNF in the first pipeline, 219 benchmarks were solved in Phase 1 using this pipeline. This is attributable to specifications being unate in several output variables, and also satisfying the condition of Theorem 2(a) (while not being in wDNNF). In the second pipeline, however, we could represent 230 benchmarks in wDNNF, and all of these were solved in Phase 1.

For each benchmark, the order $\preceq$ (ref. step 12 of Algorithm 1) in which Skolem functions are generated is such that the variable which occurs in the transitive fan-in of the least number of nodes in the AIG representation of the specification is ordered before other variables. This order ($\preceq$) is used for both BFSS and PARSYN. Note that the order $\preceq$ is completely independent of the dynamic variable order used to construct an ROBDD of the input specification in the second pipeline, prior to getting the wDNNF representation.

All experiments were performed on a message-passing cluster, with 20 cores and 64 GB memory per node, each core being a 2.2 GHz Intel Xeon processor. The operating system was Cent OS 6.5. Twenty cores were assigned to each run of PARSYN. For RSYNTH and CADET a single core on the cluster was used, since these tools don't exploit parallel processing. Each pipeline of BFSS was executed on a single node; the computation of candidate functions, building of error formula and refinement of the counterexamples was performed sequentially on 1 thread, and UNIGEN had 19 threads at its disposal (idle during Phase 1).

The maximum time given for execution of any run was 3600 s. The total amount of main memory for any run was restricted to 16GB. The metric used to compare the algorithms was *time taken to synthesize Boolean functions*. The time reported for BFSS is the better of the two times obtained from the alternative pipelines described above. Detailed results from the individual pipelines are available in [2].

Results. Of the 504 benchmarks, 177 benchmarks were not solved by any tool – 6 of these being from arithmetic benchmarks and 171 from QBFEval.

Table 1 gives a summary of the performance of BFSS (considering the combined pipelines) over different benchmarks suites. Of the 504 benchmarks, BFSS

[2] The track contains 384 benchmarks, but we were unsuccessful in converting 1 benchmark to some of the formats required by the various tools.

Table 1. BFSS: Performance summary of combined pipelines

Benchmark domain	Total benchmarks	# Benchmarks solved	Phase 1 solved	Phase 2 started	Solved By phase 2
QBFEval	383	170	159	73	11
Arithmetic	48	35	35	8	0
Disjunctive decomposition	68	68	66	2	2
Factorization	5	5	5	0	0

was successful on 278 benchmarks; of these, 170 are from QBFEval, 68 from Disjunctive Decomposition, 35 from Arithmetic and 5 from Factorization.

Of the 383 benchmarks in the QBFEval suite, we ran BFSS only on 254 since we could not build succinct AIGs for the remaining benchmarks. Of these, 159 *benchmarks were solved by Phase 1 (i.e., 62% of built QBFEval benchmarks)* and 73 proceeded to Phase 2, of which 11 reached completion. On another 11 QBFEval benchmarks Phase 1 timed out. Of the 48 Arithmetic benchmarks, *Phase 1 successfully solved* 35 *(i.e.,* $\sim$ 72%*)* and Phase 2 was started for 8 benchmarks; Phase 1 timed out on 5 benchmarks. Of the 68 Disjunctive Decomposition benchmarks, *Phase 1 successfully solved* 66 *benchmarks (i.e., 97%)*, and Phase 2 was started and reached completion for 2 benchmarks. For the 5 Factorization benchmarks, Phase 1 was successful on all 5 benchmarks.

Recall that the goodness ratio is the ratio of the number of *counterexamples remaining* to the *total size of the input space* after Phase 1. For all benchmarks solved by Phase 1, the goodness ratio is 0. We analyzed the goodness ratio at the beginning of Phase 2 for 83 benchmarks for which Phase 2 started. For 13 benchmarks this ratio was small (< 0.002), and Phase 2 reached completion for these. Of the remaining benchmarks, 34 also had a small goodness ratio (< 0.1), indicating that we were close to the solution at the time of timeout. However, 27 benchmarks in QBFEval had goodness ratio close to > 0.9, indicating that most of the counter-examples were not eliminated by timeout.

We next compare the performance of BFSS with other state-of-art tools. For clarity, since the number of benchmarks in the QBFEval suite is considerably greater, we plot the QBFEval benchmarks separately.

BFSS vs CADET: Of the 504 benchmarks, CADET was successful on 231 benchmarks, of which 24 belonged to Disjunctive Decomposition, 22 to Arithmetic, 1 to Factorization and 184 to QBFEval. Figure 1(a) gives the performance of the two algorithms with respect to time on the QBFEval suite. Here, CADET solved 35 benchmarks that BFSS could not solve, whereas BFSS solved 21 benchmarks that could not be solved by CADET. Figure 1(b) gives the performance of the two algorithms with respect to time on the Arithmetic, Factorization and Disjunctive Decomposition benchmarks. In these categories, there were a total of 62 benchmarks that BFSS solved that CADET could not solve, and there was 1 benchmark that CADET solved but BFSS did not solve. While CADET takes less time on Arithmetic benchmarks and many QBFEval benchmarks, on Disjunctive Decomposition and Factorization, BFSS takes less time.

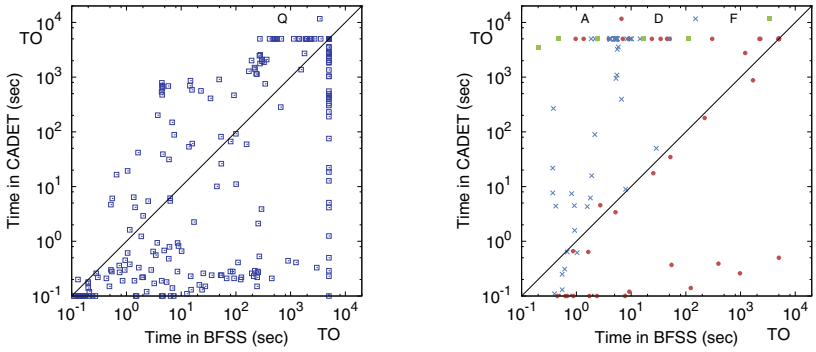

Fig. 1. BFSS vs CADET: Legend: Q: QBFEval, A: Arithmetic, F: Factorization, D: Disjunctive Decomposition. TO: benchmarks for which the corresponding algorithm was unsuccessful.

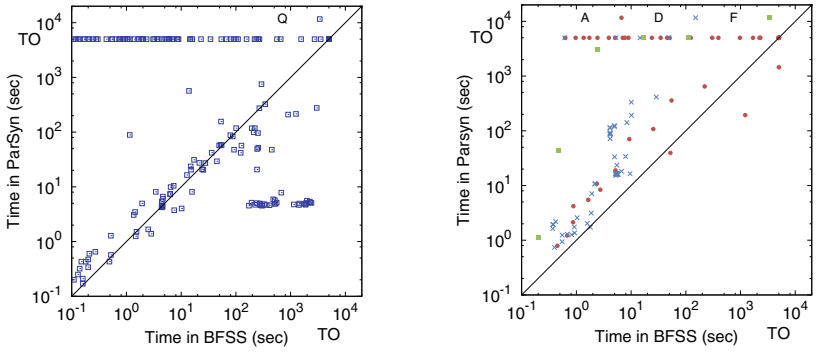

Fig. 2. BFSS vs PARSYN (for legend see Fig. 1)

BFSS vs PARSYN: Fig. 2 shows the comparison of time taken by BFSS and PARSYN. PARSYN was successful on a total of 185 benchmarks, and could solve 1 benchmark which BFSS could not solve. On the other hand, BFSS solved 94 benchmarks that PARSYN could not solve. From Fig. 2, we can see that on most of the Arithmetic, Disjunctive Decomposition and Factorization benchmarks, BFSS takes less time than PARSYN.

BFSS vs RSYNTH: We next compare the performance of BFSS with RSYNTH. As shown in Fig. 3, RSYNTH was successful on 51 benchmarks, with 4 benchmarks that could be solved by RSYNTH but not by BFSS. In contrast, BFSS could solve 231 benchmarks that RSYNTH could not solve! Of the benchmarks that were solved by both solvers, we can see that BFSS took less time on most of them.

BFSS vs ABSSYNTHE-SKOLEM: ABSSYNTHE-SKOLEM was successful on 217 benchmarks, and could solve 31 benchmarks that BFSS could not solve. In contrast, BFSS solved a total of 92 benchmarks that ABSSYNTHE-SKOLEM could not. Figure 4 shows a comparison of running times of BFSS and ABSSYNTHE-SKOLEM.

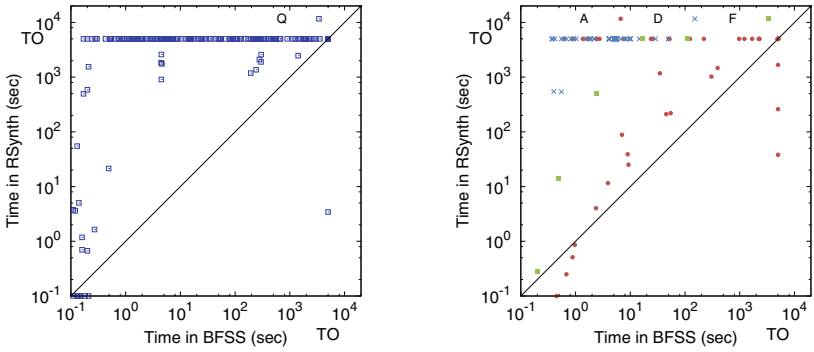

Fig. 3. BFSS vs RSYNTH (for legend see Fig. 1)

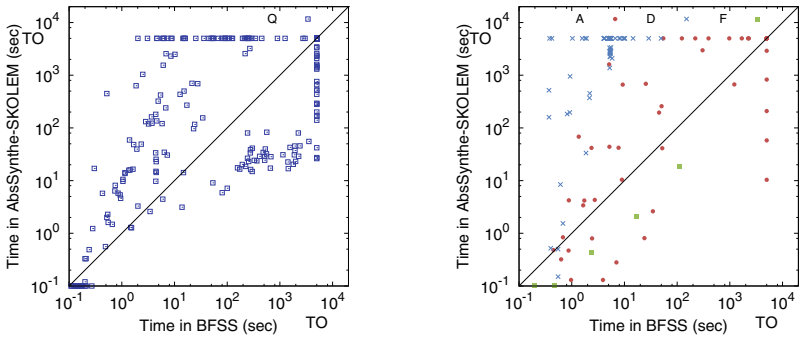

Fig. 4. BFSS vs ABSSYNTHE-SKOLEM (for legend see Fig. 1)

7 Conclusion

In this paper, we showed some complexity-theoretic hardness results for the Boolean functional synthesis problem. We then developed a two-phase approach to solve this problem, where the first phase, which is an efficient algorithm generating poly-sized functions surprisingly succeeds in solving a large number of benchmarks. To explain this, we identified sufficient conditions when phase 1 gives the correct answer. For the remaining benchmarks, we employed the second phase of the algorithm that uses a CEGAR-based approach and builds Skolem functions by exploiting recent advances in SAT solvers/approximate counters. As future work, we wish to explore further improvements in Phase 2, and other structural restrictions on the input that ensure completeness of Phase 1.

Acknowledgements. We are thankful to Ajith John, Kuldeep Meel, Mate Soos, Ocan Sankur, Lucas Martinelli Tabajara and Markus Rabe for useful discussions and for providing us with various software tools used in the experimental comparisons. We also thank the anonymous reviewers for insightful comments.

References

1. Website for CAV 2018 Experiments (2018). https://drive.google.com/drive/folders/0B74xgF9hCly5QXctNFpYR0VnQUU?usp=sharing
2. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard for Boolean functional synthesis. arXiv e-prints (2018). https://arxiv.org/abs/1804.05507
3. Akshay, S., Chakraborty, S., John, A.K., Shah, S.: Towards parallel Boolean functional synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 337–353. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_19
4. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. STTT **7**(2), 118–128 (2005)
5. Andersson, G., Bjesse, P., Cook, B., Hanna, Z.: A proof engine approach to solving combinational design automation problems. In: Proceedings of the 39th Annual Design Automation Conference, DAC 2002, pp. 725–730. ACM, New York (2002). https://doi.org/10.1145/513918.514101
6. Baader, F.: On the complexity of Boolean unification. Technical report (1999)
7. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. Form. Methods Syst. Des. **41**(1), 45–65 (2012). https://doi.org/10.1007/s10703-012-0152-6
8. Boole, G.: The Mathematical Analysis of Logic. Philosophical Library (1847). https://books.google.co.in/books?id=zv4YAQAAIAAJ
9. Boudet, A., Jouannaud, J.P., Schmidt-Schauss, M.: Unification in Boolean rings and Abelian groups. J. Symb. Comput. **8**(5), 449–477 (1989). https://doi.org/10.1016/S0747-7171(89)80054-9
10. Brenguier, R., Pérez, G.A., Raskin, J.F., Sankur, O.: Abssynthe: abstract synthesis from succinct safety specifications. In: Proceedings 3rd Workshop on Synthesis (SYNT 2014) Electronic Proceedings in Theoretical Computer Science, vol. 157, pp. 100–116. Open Publishing Association (2014). http://arxiv.org/abs/1407.5961v1
11. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
12. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 304–319. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_25
13. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9–15 July 2016, pp. 3569–3576 (2016)
14. Chen, Y., Eickmeyer, K., Flum, J.: The exponential time hypothesis and the parameterized clique problem. In: Thilikos, D.M., Woeginger, G.J. (eds.) IPEC 2012. LNCS, vol. 7535, pp. 13–24. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33293-7_4
15. Darwiche, A.: Decomposable negation normal form. J. ACM **48**(4), 608–647 (2001)
16. Deschamps, J.P.: Parametric solutions of Boolean equations. Discret. Math. **3**(4), 333–342 (1972). https://doi.org/10.1016/0012-365X(72)90090-8
17. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based Boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_22

18. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. J. Comput. Syst. Sci. **62**(2), 367–375 (2001)
19. Jacobs, S., Bloem, R., Brenguier, R., Könighofer, R., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The second reactive synthesis competition (SYNTCOMP 2015). In: Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015, pp. 27–57 (2015)
20. Jiang, J.-H.R.: Quantifier elimination via functional composition. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 383–397. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_30
21. Balabanov, V., Jiang, J.-H.R.: Resolution proofs and Skolem functions in QBF evaluation and applications. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 149–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_12
22. Jo, S., Matsumoto, T., Fujita, M.: Sat-based automatic rectification and debugging of combinational circuits with LUT insertions. In: Proceedings of the 2012 IEEE 21st Asian Test Symposium, ATS 2012, pp. 19–24. IEEE Computer Society (2012)
23. John, A., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: FMCAD, pp. 73–80 (2015)
24. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans. CAD Integr. Circuits Syst. **21**(12), 1377–1394 (2002). http://dblp.uni-trier.de/db/journals/tcad/tcad21.html#KuehlmannPKG02
25. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. SIGPLAN Not. **45**(6), 316–329 (2010)
26. Berkeley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/
27. Lowenheim, L.: Über die Auflösung von Gleichungen in Logischen Gebietkalkul. Math. Ann. **68**, 169–207 (1910)
28. Macii, E., Odasso, G., Poncino, M.: Comparing different Boolean unification algorithms. In: Proceedings of 32nd Asilomar Conference on Signals, Systems and Computers, pp. 17–29 (2006)
29. Marijn Heule, M.S., Biere, A.: Efficient Extraction of Skolem Functions from QRAT Proofs. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014, pp. 107–114 (2014)
30. Martin, U., Nipkow, T.: Boolean unification - the story so far. J. Symb. Comput. **7**(3–4), 275–293 (1989). https://doi.org/10.1016/S0747-7171(89)80013-6
31. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 430–435. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_33
32. QBFLib: QBFEval (2017). http://www.qbflib.org/event_page.php?year=2017
33. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 375–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_23
34. Rabe, M.N., Tentrup, L.: CAQE: a certifying QBF solver. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, 27–30 September 2015, pp. 136–143 (2015)
35. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)

36. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005, pp. 281–294 (2005)
37. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT **15**(5–6), 497–518 (2013)
38. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 124–131 (2017)
39. Trivedi, A.: Techniques in Symbolic Model Checking. Master's thesis, Indian Institute of Technology Bombay, Mumbai, India (2003)
40. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 1362–1369 (2017)