Pushing The Envelope for Boolean Functional Synthesis

S. Akshay\textsuperscript{1}, Supratik Chakraborty\textsuperscript{1}, Shubham Goel\textsuperscript{1}, Sumith Kulal\textsuperscript{1}, and Shetal Shah\textsuperscript{1}

Indian Institute of Technology Bombay, India

Abstract. Boolean functional synthesis using AIGs and CEGAR has been recently proposed as a promising alternative to synthesis based on BDDs or on DPLL+CDCL style clausal reasoning. In this paper, we delve deeper into the AIG+CEGAR approach and propose techniques that significantly improve the performance of Boolean functional synthesis, sometimes by orders of magnitude vis-a-vis state-of-the-art tools. This is achieved by a combination of algorithmic modifications resulting from an improved theoretical analysis of the AIG+CEGAR approach. Our approach also harnesses the power of recently proposed efficient almost-uniform samplers to improve the performance of synthesis. Empirical evaluation on a suite of benchmarks shows that our approach outperforms existing state-of-the-art Boolean functional synthesis tools on most of these benchmarks.

1 Introduction

The algorithmic synthesis of Boolean functions satisfying relational specifications has long been of interest to logicians and computer scientists. Boole [5] and Lowenheim [19] 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 [20]. More recently, synthesis of Boolean functions has found important applications in a wide range of contexts including reactive strategy synthesis [2, 12, 33], certified QBF-SAT solving [14, 26, 4, 23], automated program synthesis [29, 27], circuit repair and debugging [15], disjunctive decomposition of symbolic transition relations [31] and the like. This has spurred recent interest in developing practically efficient Boolean function synthesis algorithms. The resulting new generation of tools [21, 16, 1, 11, 30, 26, 24] have enabled synthesis of Boolean functions from much larger and more complex relational specifications than those that could be handled by earlier techniques, viz. [13, 14, 31, 20]. Among the recently proposed techniques, those based on CEGAR [16, 1] have successfully leveraged the spectacular advances made in propositional satisfiability (SAT) solving over the last 20+ years, while treating the SAT solver as a black-box. This effectively allows these techniques to benefit for free from future
advancements in SAT solving technology. In this paper, we pursue the CEGAR-based approach, and propose several algorithmic improvements that translate to significant improvements in the performance of Boolean functional synthesis. Our approach also allows us to provide probabilistic guarantees on the correctness of partially synthesized functions – an important feature when a problem instance is too hard to be solved completely within given time and/or space.

The Boolean functional synthesis problem asks: given a Boolean formula $F(X, Y)$ specifying a relation between inputs $Y = (y_1, \ldots, y_m)$ and outputs $X = (x_1, \ldots, x_n)$, determine functions $\Psi = (\psi_1(Y), \ldots, \psi_n(Y))$ such that $F(\Psi, Y)$ holds whenever $\exists X F(X, Y)$ holds. The function $\psi_i$ is called a Skolem function for $x_i$ in $F$, and $\Psi = (\psi_1, \ldots, \psi_n)$ is called a Skolem function vector for $X$ in $F$.

The main idea in CEGAR-based Boolean functional synthesis is to start off with initial proposals for Skolem functions for all $x_i$, and then update (or refine) the proposed Skolem functions in a counterexample-driven manner. Intuitively, a counterexample is a value of the inputs $Y$ for which there exists a value of $X$ that renders $F(X, Y)$ true, but for which $F(\Psi, Y)$ evaluates to false. As shown in [16], given a proposed Skolem function vector, the existence of a counterexample can be detected using a single query to a SAT solver. The refinement step uses the satisfying assignment returned by this query to update an appropriate subset of the proposed Skolem functions. 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 Boolean functional synthesis problem.

The above idea was originally used to synthesize Skolem functions from factored specifications in [16]. Subsequently in [1], a compositional variant of this technique was proposed for arbitrary specifications, and the refinement process was partially parallelized, yielding performance gains. In this paper, we take a fresh look at CEGAR-based Boolean functional synthesis, and propose several improvements that contribute to the design of a new parameterized algorithm. As our experiments show, not only does the new algorithm outperform earlier CEGAR-based techniques, but we are also able to solve some problems currently beyond the reach of Boolean functional synthesis tools using other techniques. Like earlier CEGAR-based algorithms [16, 1], we leverage advances in propositional SAT solving by using a SAT solver as a black-box. In fact, we go a step further and use recently proposed almost-uniform samplers of propositional models [9, 8] that build on state-of-the-art SAT solvers, yielding several benefits.

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

1. We define a new progress metric for counterexample elimination and show that eliminating a counterexample $\pi$ amounts to reducing this metric for $\pi$ from $|X|$ to 0. We also propose a parameterized algorithm in which each refinement iteration monotonically reduces this metric for at least one counterexample, without adding any new counterexamples. By suitably tuning the parameters of our algorithm, a wide range of refinement algorithms can be obtained, each of which enjoys the above monotonicity property, and eliminate varying numbers of additional counterexamples.
2. Instead of refinement based on a single counterexample at a time, we propose using an (almost-)uniform sampler [8] to obtain a diverse set of counterexamples together. This allows us to eliminate multiple counterexamples together. In addition, it enables us to refine the Skolem functions $\psi_i$ in an order that reduces potentially redundant refinements with high probability.

3. Finally, we perform an extensive set of experiments that demonstrate the performance advantages resulting from the above algorithmic improvements. We also show briefly how a few important parameters of our algorithm can be tuned to solve difficult benchmarks. This provides the user controllability on the performance of our algorithm.

Related work The literature contains several early theoretical studies on variants of Boolean functional synthesis [5, 19, 10, 6, 22, 3]. More recently, researchers have tried to build practically efficient synthesis tools that scale to medium or large problem instances. In [21], Skolem functions for $X$ are extracted from a proof of validity of $\forall Y \exists X F(X, Y)$. Unfortunately, this doesn’t work when $\forall Y \exists X F(X, Y)$ is not valid. Inspired by the spectacular effectiveness of CDCL-based SAT solvers, an incremental determinization technique for Skolem function synthesis was proposed in [24]. While the original tool works only for specifications for which $\forall Y \exists X F(X, Y)$ is valid, a later unpublished release fixed this limitation. In [13, 31], a synthesis approach based on iterated compositions was proposed. Unfortunately, as has been noted in [16, 11], this does not scale to large benchmarks. A recent work [11] adapts the composition-based approach to work with ROBDDs [7]. For factored specifications, ideas from symbolic model checking using implicitly conjoined ROBDDs have been used to enhance the scalability of the technique further in [30]. In the genre of CEGAR-based techniques, [16] 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 [1]. Our work builds on these, and shows that by a more careful analysis, important algorithmic improvements can be made that eventually translate to significant performance advantages. In addition to the above techniques, template-based [29] or sketch-based [28] 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 [17].

2 Preliminaries and Notations

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 $\text{sup}(F)$. 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 $\text{sup}(F)$ to $\{0, 1\}$ such that $F$ evaluates to 1 under this assignment. If $\pi$ is a model of $F$, we write $\pi \models F$ and use $\pi(z_i)$ to denote the
value assigned to \( z_i \in \text{sup}(F) \) by \( \pi \). Let \( Z = (z_{i_1}, z_{i_2}, \ldots, z_{i_k}) \) be a sequence of variables in \( \text{sup}(F) \). We use \( \pi \downarrow Z \) to denote the projection of \( \pi \) on \( Z \), i.e. the sequence \( (\pi(z_{i_1}), \pi(z_{i_2}), \ldots, \pi(z_{i_k})) \). In this paper, we use AIGs, and use \( |F| \) to denote the number of nodes in an AIG representation of \( F \).

We use \( X = (x_1, \ldots, x_n) \) to denote a sequence of Boolean outputs, and \( Y = (y_1, \ldots, y_m) \) to denote a sequence of Boolean inputs. For \( 1 \leq i \leq j \leq n \), we also use \( X^i_j \) to denote the subsequence \( (x_i, x_{i+1}, \ldots, x_j) \). Let \( F^{(i-1)}(X^n_i, Y) \) denote \( \exists X_1^{i-1}F(X_1^{i-1}, X^n_i, Y) \). It has been argued in [16, 11, 1, 13] that given a relational specification \( F(X, Y) \), the Boolean functional synthesis problem can be solved by first ordering the outputs, say as \( x_1 \prec x_2 \prec \cdots \prec x_n \), and then synthesizing a function \( \psi_i(X^n_{i+1}, Y) \) for each \( x_i \) such that \( F^{(i-1)}(\psi_i, X^n_{i+1}, Y) \leftrightarrow \exists x_i F^{(i-1)}(x_i, X^n{i+1}, Y) \). Once all such \( \psi_i \) are obtained, one can substitute \( \psi_{i+1} \) through \( \psi_n \) for \( x_{i+1} \) through \( x_n \) respectively, in \( \psi_1 \) to obtain a Skolem function for \( x_1 \) as a function of only \( Y \). We adopt this approach in this paper, and therefore focus on obtaining \( \psi_i \) in terms of \( X^n_{i+1} \) and \( Y \). Furthermore, we know from [16, 1, 13] that a function \( \psi_i \) is a Skolem function for \( x_i \) iff it satisfies \( \Delta_i^F \Rightarrow \psi_i \Rightarrow -\Gamma_i^F \), where \( \Delta_i^F \equiv \exists X_1^i F(X_1^i, 0, X^n_{i+1}, Y) \), and \( \Gamma_i^F \equiv -\exists X_1^i F(X_1^i, 1, X^n_{i+1}, Y) \). When \( F \) is clear from the context, we often omit it and write \( \Delta_i \) and \( \Gamma_i \). It is easy to see that both \( \Delta_i \) and \( -\Gamma_i \) serve as Skolem functions for \( x_i \) in \( F \).

3 Initial Choice of Skolem Functions

While computing \( \Delta_i \) and \( \Gamma_i \) exactly for all \( x_i \) is computationally expensive in general, we observe that “good” initial approximations of \( \Delta_i \) and \( \Gamma_i \) can often be computed efficiently. We use the approach used in [1] to compute the initial estimates, with several optimizations on top. As our experiments show, these initial approximations are often good enough to solve several Boolean functional synthesis problems.

In the remainder of the paper, we only use under-approximations of \( \Delta_i \) and \( \Gamma_i \), and use \( \delta_i \) and \( \gamma_i \), respectively, to denote them. Recall from Section 2 that both \( \Delta_i \) and \( -\Gamma_i \) suffice as Skolem functions for \( x_i \). Therefore, we propose to use either \( \delta_i \) or \( -\gamma_i \) (depending on which has a smaller AIG) as our initial estimate of \( \psi_i \).

Lemma 1. \( \delta_{1,\text{init}} = \Delta_1 \) and \( \gamma_{1,\text{init}} = \Gamma_1 \).

The proof follows from the definitions and is omitted for lack of space. In general, the initial choice of Skolem functions may not always be good enough. To detect whether \( \Psi = (\psi_1, \ldots, \psi_n) \) is good enough to be correct, we construct an error formula \( \varepsilon_\Psi(X', X, Y) \equiv F(X', Y) \wedge \bigwedge_{i=1}^n (x_i \leftrightarrow \psi_i) \wedge -F(X, Y) \), as described in [16], and check its satisfiability.

Theorem 1 ([16]). \( \varepsilon_\Psi \) is unsatisfiable iff \( \Psi \) is a correct Skolem function vector.

Let \( \pi \) be a satisfying assignment of \( \varepsilon_\Psi \). By definition of \( \varepsilon_\Psi \), when the inputs \( Y \) are set to \( \pi \downarrow Y \), the Skolem functions \( \Psi \) assign values to the outputs \( X \) that
Suppose \( f \) be refined in order to eliminate counterexample for \( \Psi \) in \( F \) and \( \pi \downarrow X \) an evidence for \( \pi \downarrow Y \) being a counterexample for \( \Psi \) in \( F \). When \( \Psi \) and \( F \) are clear from the context, we omit mentioning them.

In the next few sections, we describe a range of algorithms that iteratively update (or refine) \( \psi \) for appropriate values of \( i \in \{1, \ldots n\} \) in order to eliminate counterexamples, until \( \varepsilon_\Psi \) is rendered unsatisfiable. Interestingly, the algorithms in [16, 1] also use the same underlying strategy, although the details are significantly different from those in this paper. All these algorithms effectively expand an under-approximation \( \delta_i \) or \( \gamma_i \), as the case may be, in each iteration of refinement, while maintaining the invariant that the expansion implies (under-approximates or becomes semantically equal to) \( \Delta_i \) or \( I_i \), respectively. Henceforth, we call this class of algorithms expansion-based refinement algorithms.

A crucial remark is that since each \( \Delta_i \) and \( I_i \) is a Boolean function of \((n - i + m)\) variables, their under-approximations can be expanded finitely many times before they become semantically identical to \( \Delta_i \) and \( I_i \). As the Skolem functions \( \Delta_i \) or \( -I_i \) are provably correct [13, 16, 1], it follows that any expansion-based refinement algorithm must terminate if it expands at least one \( \delta_i \) or \( \gamma_i \) in each iteration of refinement. This gives a uniform termination proof for all expansion-based refinement algorithms.

4 Counterexample based Boolean Functional Synthesis

Given a satisfying assignment \( \pi \) of \( \varepsilon_\Psi \), we must refine at least one Skolem function \( \psi_i \) in order to eliminate the counterexample \( \pi \downarrow Y \). We start by identifying a \( \psi_i \) that must necessarily be refined to eliminate this counterexample in any expansion-based refinement. It turns out that this is a crucial notion, which also allows us to prove monotonicity properties of our refinement algorithm. Note that although a satisfying assignment of \( \varepsilon_\Psi (X', X, Y) \) assigns values to \( X', X \) and \( Y \), we will be mostly interested in the values of \( X \) and \( Y \). Hence, we abuse notation and use \( \pi \) to refer to \( \pi \downarrow X, Y \) when there is no confusion.

**Definition 1.** Suppose \( F(X, Y) \) is satisfiable, and let \( \pi \) be an assignment of \( \{0, 1\} \) to variables in \( X \) and \( Y \) such that \( \exists X F(X, \pi \downarrow Y) = 1 \). We define \( k^\pi = 0 \) if \( F(\pi) = 1 \), and \( k^\pi = \min_k \exists X^k \ F(X^k, \pi \downarrow Y) \) otherwise.

For a given choice \( \Psi \) of Skolem functions, if \( \varepsilon_\Psi \) is satisfiable, we define \( k^{max} = \max_{\pi | \pi = \varepsilon_\Psi} k^\pi \) and set it to 0 otherwise. It is easy to see that that if \( \pi \models \varepsilon_\Psi \), then \( 1 \leq k^\pi \leq k^{max} \leq n \). Recall that for \( \pi \models \varepsilon_\Psi \), \( \pi \downarrow X \) is the evidence that \( \pi \downarrow Y \) is a counter-example. However, if \( k^\pi = k \), then since \( F(X^{k-1}, \pi \downarrow X^k, \pi \downarrow Y) \) is already unsatisfiable, we say that \( \pi \downarrow X^k \) is the minimal evidence for counterexample \( \pi \downarrow Y \). It also follows from the definition of \( k^\pi \) that as far as the counterexample \( \pi \downarrow Y \) is concerned, none of the Skolem functions \( \psi_{k+1}, \ldots \psi_n \) give erroneous values of \( x_{k+1}, \ldots x_n \) respectively. However, \( \psi_k \) must necessarily be refined in order to eliminate \( \pi \downarrow Y \). Note that since \( \psi_k \) is either \( \delta_k \) or \( \gamma_k \) depending on whether \( \delta_{k,init} \) or \( \gamma_{k,init} \) had a smaller AIG representation. The
Skolem function $\psi_k$ can therefore be refined by simply expanding $\delta_k$ or $\gamma_k$, as the case may be.

It turns out that computing $k^*$ is straightforward. We use a procedure $\text{ComputeK}(\pi)$ for this purpose. This takes as input $\pi$ and $F$ and uses a binary search to identify the minimum $k$ such that $F(X_1^k, \pi_1X_{k+1}^n, \pi_nY)$ is satisfiable. Clearly, this can be achieved using $O(\log n)$ calls to a SAT solver. On the other hand, computing the maximum over all counter-examples when $k_{\text{max}} \neq 0$ is not easy, and there seems to be no easier way than enumerating all counter-examples in exponential time. In the next section however, we will show that we can do much better, if we want to give probabilistic guarantees on computing this global maximum.

Eliminating counterexamples in one shot

We saw above that refining a Skolem function $\psi_k$ amounts to expanding $\delta_k$ or $\gamma_k$, as the case may be. Clearly, if $\delta_k$ (resp. $\gamma_k$) is made equal to $\Delta_k$ (resp. $I_k$), there can be no further need to refine $\psi_k$. Since $\psi_k^{x+1}, \ldots, \psi_n$ evaluate to correct values for the counterexample $\pi_iY$, if we can ensure that all $\delta_i$ and $\gamma_i$ for $i \in \{1, \ldots, k^*\}$ are made equal to $\Delta_i$ and $\Gamma_i$ respectively, the counterexample $\pi_iY$ must be eliminated. We now give a basic algorithm that does this. We start with a parameterized version of the algorithm.

Algorithm 1: Collapse $(b, c)$

Input: $F(X, Y)$ with $|Y| = m$ and $|X| = n$,
Integers $b, c \in \{1, \ldots, n\}$ with $b \leq c$
Output: For all $1 \leq i \leq n$, initialize $\delta_i, \gamma_i, \psi_i$ using Eqn (?) in Section 3;
1 Initialize: for all $1 \leq i \leq n$, initialize $\delta_i, \gamma_i, \psi_i$ using Eqn (??) in
Section 3;
2 for $i = b$ to $c$ do
3 \[ \delta_i \leftarrow (\delta_{i-1} \wedge \gamma_{i-1})|_{x_i=0}; \]
4 \[ \gamma_i \leftarrow (\gamma_{i-1} \wedge \delta_{i-1})|_{x_i=1}; \]

Theorem 2. Suppose $b = 1$ and $1 \leq c \leq n$. When Algorithm 1 terminates, for all $1 \leq i \leq c$, we have $\delta_i = \Delta_i$ and $\gamma_i = \Gamma_i$. Furthermore, let $\Psi$ be the updated Skolem function vector that results from using the updated $\delta_i$ (resp. $\gamma_i$) obtained above. If $\pi \models \Psi$, then $k^*>c$.

The proof follows from Lemma 1, Definition 1 and by induction on $i$. Clearly, if $c = k_{\text{max}}$, then after the algorithm terminates, no counterexamples exist.

Corollary 1. If $b = 1$ and $c \geq k_{\text{max}}$, when Algorithm 1 terminates, $\Psi$ is a correct Skolem function vector. Furthermore, $|\delta_i|, |\gamma_i| \leq 2^{\min\{i, k_{\text{max}}\}|F|}$.

The sizes of the functions computed above grow exponentially in $n$, and linearly in the size of the specification. This blow-up is similar to that of Jiang et al’s algorithm [13, 14] in the worst-case. Thus, if $k_{\text{max}}$ is small, this algorithm can be efficient; otherwise, it is not. Note that computing $k_{\text{max}}$ requires enumerating all counterexamples, which is not practical in general. We will later show how $k_{\text{max}}$ can be estimated with high confidence by using (almost-)uniform sampling.
Algorithm 2: RefineAtK(π, k) // Assume k = k^π

1 // References and updates δ_k or γ_k, and ψ_k ;
2 // Returns an assignment π’ of X, Y such that π’|X is the updated evidence for counterexample π’|Y;
3 if π(x_k) = 0 then
4 δ_k ← δ_k ∪ {π_Y} // Updates ψ_k if δ_k = ψ_k
5 else
6 γ_k ← γ_k ∪ {π_Y} // Updates ψ_k if ¬γ_k = ψ_k
7 // Update evidence for counter-example π’|Y:
8 π’(x_k) ← ¬π(x_k), π’|X^n_{k+1} ← π|X^n_{k+1}, π’|Y ← π'|Y;
9 for j = k - 1 downto 1 do
10 π’(x_j) = ψ_j(π’|X^n_{j+1}, Y, π’|Y)
11 Return π’;

Eliminating counter-examples one at a time When we have only a “small” number of input variables (which implies that the number of counter-examples is also “small”), it may be easier to eliminate them individually. To see how to eliminate a counter-example, we start by first “improving” the evidence for this counter-example. For instance, we can refine the Skolem function ψ_k by adding the counterexample π|Y and expanding δ_k or γ_k, as described in Algorithm 2.

When Algorithm 2 terminates, we observe that if π |= e_Ψ at the beginning of the algorithm, then at the end of this algorithm, π |− e_Ψ, since we have changed the value of Skolem functions. However, we may not have eliminated the counter-example π|Y. To do this, a naive way would be to repeatedly find the updated value of k^n and refine δ_k or γ_k, as appropriate. However, we may find after refining ψ_k, some future refinement of ψ_j, for j > k, requires us to refine ψ_k again. Indeed, this may lead to an exponential number of refinements to eliminate a single counter-example. A better solution is to refine at the level k^n, in which case we are guaranteed monotonic improvement. This is true since, if we start with k = k^n, we have k^n’ < k^n. Thus, we have monotonically reduced k^n after each call to this algorithm, i.e., a refinement step. This gives us an algorithm, shown as Algorithm 2, to eliminate a single counter-example.

Note that at each iteration of the inner while loop, we get rid of one counter-example and a priori there could be (finite but) exponentially many counter-examples and hence iterations. But now, this exponential is in m, the number of input variables, and NOT in the output variables, n.
Algorithm 3: Eliminate1By1(F)

1. Initialize Skolem functions as in Eqn (??);

2. while $\varepsilon_\Psi$ is satisfiable do

3.  pick $\pi \models \varepsilon_\Psi$  // $\pi \upharpoonright Y$ is a counter-example;

4.  $k \leftarrow \text{ComputeK}(\pi)$  // Use $O(\log(n))$ SAT calls to compute;

5.  while $k \neq 0$  // Eliminate cex $\pi \upharpoonright Y$;

6.  do

7.  $\pi \leftarrow \text{RefineAtK}(\pi, k)$;

8.  $k \leftarrow \text{ComputeK}(\pi)$;

Properties of expansion-based refinement algorithms
The above algorithms are two extremes of the class of expansion-based refinement algorithms. The first removes all counter-examples in a linear number of steps but incurs a possibly exponential memory blowup. The second removes one counter-example at a time efficiently, but has an exponential running time. Our goal is to combine these and find an optimal trade-off. We start by showing some properties of these algorithms.

Lemma 2. For any expansion-based refinement algorithm, the following holds:

1. No new counter-examples are introduced at any stage of the algorithm.

2. To eliminate a cex $\pi$, it is necessary to refine at $k^\pi + 1$.

3. For all $j > k^\pi + 1$, no refinement can change the value of $\Psi_j(\pi \upharpoonright X_{j+1}^\pi, Y)$.

Keeping in mind these impossibility results, we now wish to combine these approaches to obtain a practically scalable approach for removing sets of counter-examples with guarantees of progress.

A combined algorithm processing multiple counter-examples
Since Algorithm 1 is parametrized, we would like to perform it with $b \leq c$ and then continue with Algorithm 3, but at the same time we would also like to process multiple counter-examples simultaneously. As shown in Theorem 2, if $b = 1$, this eliminates all counter-examples till $c$. We will fix a small enough value of $c$ and do this in the first phase of our algorithm (as mentioned earlier, when $c$ is large, this consumes too much memory). Now, before switching to the refinement phase 3, we would still want to apply the Collapse Algorithm for a few steps, from some $b \neq 1$ that depends on the current choice of candidate Skolem functions. For this, given $\pi \models \varepsilon_\Psi$ and given the latest $\delta_b$ and $\gamma_b$ computed, we define $\ell^\pi = \max\{k \mid \pi \models \delta_k \wedge \gamma_k\}$ Note that $\ell^\pi \leq k^\pi$ and further, by a careful analysis, in several cases we can guarantee that collapsing from $\ell^\pi$ to $k^\pi$ will remove multiple counter-examples. This is also substantiated by our experiments. Further, computing this value is also easy (without any need for SAT calls etc) and is done by a procedure ComputeL(\pi, \delta, \gamma), which looks at the current $\delta, \gamma$ and finds the maximum $k$ such that $\pi$ evaluates to true on their conjunction. While we would like the parameter $c_2$ to be $k_2 - \ell_2$, this may be too big in
Algorithm 4: BatchSynthesize($F, c_1, c_2$)

1. Initialize Skolem functions as in Eqn (??);
2. while $\exists \Phi$ is satisfiable do
   3. pick $S \subseteq \{\pi \mid \pi \models \exists \Phi\}$ // pick set of cex using UniGen;
   4. $k \leftarrow \max_{\pi \in S} \text{COMPUTE}_{K}(\pi)$; $k' \leftarrow \min_{\pi \in S} \text{COMPUTE}_{K}(\pi)$;
   5. while $k \neq 0$ // there exists a cex $\pi \not\models Y$ in $S$; do
      6. if $0 \leq k' \leq c_1$ then
         7. collapse $(1, c_1)$ // Phase 1: Collapse from 1 to $c_1$;
         8. $k' \leftarrow -1$ // ensure that Phase 1 is done only once;
         9. $\ell \leftarrow \max_{\pi \in S} \{|\pi \in \text{COMPUTE}_L(\pi, \delta, \gamma) = r\}|$;
            // $\ell$ is the $\ell^s$ that occurs most frequently wrt $\pi \in S$;
      10. collapse $(\ell, \ell + c_2)$ // Phase 2: Collapse $c_2$ steps from $\ell$;
      11. for each $\pi \in S_k \leftarrow \{\pi \in S \mid k^s = k\}$ do
          12. $\pi' \leftarrow \text{RefineAt}_{K}(\pi, k)$ // Phase 3: Refine all cex in $S_k$;
          13. $S \leftarrow S \setminus \{\pi\} \cup \{\pi'\}$ // Update $S$;
      14. $k \leftarrow \max_{\pi \in S} \text{COMPUTE}_{K}(\pi)$; $k' \leftarrow \min_{\pi \in S} \text{COMPUTE}_{K}(\pi)$;

practice and hence, we fix $c_2$ to be a small constant (4), before going onto the refinement step and iterating. This forms the second phase of our algorithm and we repeatedly iterate between this and the refinement phase.

Finally, we perform another important improvement by processing multiple counter-examples. This is a major limitation of Algorithm 3, since in line 3, it processes counter-examples sequentially. Putting all this together gives us our final algorithm, that builds on the above ideas, but also allows us to pick a set of counter-examples and process them together.

As before, this algorithm also always terminates and at termination it provides the correct Skolem functions. Further, it processes the counter-examples in batches of $S$. The parameters $c_1$ and $c_2$ indicate the number of steps that we will collapse, respectively, from the top in the beginning and in between (from $\ell^s$) at each iteration. The remaining question is how to pick $S$, and what guarantees can we give on the performance of the algorithm. It turns out that these are related and are tackled in the next section.

5 Experimental results

Experimental methodology. We have implemented Algorithm 4 using UniGen to sample counterexamples as explained in Section ???. We call this implementation BFSS in this section, and compare it with the following algorithms/tools:


Our implementation of BFSS, makes extensive use of the ABC [18] library to represent and manipulate Boolean functions as AIGs. We make use of two different SAT solvers i)UniGen [8] to give us multiple counterexamples which are (almost) uniformly distributed ii) ABC’s default SAT solver, a variant of
MiniSAT. For the results reported in this paper, UniGen was asked to give a large number of solutions (=110000). The number of counterexamples generated by UniGen and used by bfss is typically much smaller as indicated later in the section. As in [1], we consider benchmarks from different domains:

1. **Disjunctive Decomposition Benchmarks**: These benchmarks were generated by considering some of the larger sequential circuits in the HWMCC10 benchmark suite wherein the problem of disjunctively decomposing each circuit into components was formulated as a problem of synthesizing a vector of Boolean functions. Each generated benchmark is of the form $\exists Y \varphi(X, Y)$ where $\exists X(\exists Y \varphi(X, Y))$ is true.

2. **Arithmetic Benchmarks**: These benchmarks were taken from the work described in [11]. Specifically, the benchmarks considered are floor, ceiling, decomposition, equalization, intermediate, subtraction, min, max.

3. **Factorization Benchmarks**: We considered the integer factorization problem for different bit-widths as in [1].

For each arithmetic benchmark, five variants were generated by varying the bit-width of the arguments of arithmetic operators; specifically, we considered bit-widths of 32, 64, 128, 256 and 1024. Similarly, for the factorization benchmark, we generated five variants, using 8, 10, 12, 14, 16 for the bit-width of the product. Further, for CADET and RSYNTH, we converted each benchmark into qdimacs formats using Tseitin encoding [32]. For each benchmark, the variables are ordered such that the variable which occurs in the transitive fan-in of the least number of nodes in the AIG representation of the specification, appears at the top. We also experimented with an ordering which is the reverse of the above and present results at the end of the section. We used the same ordering for bfss, and parSyn.

**Machine details**: All experiments were performed on a message-passing cluster, where each node had 20 cores and 64 GB main memory, each core being a 2.20 GHz Intel Xeon processor with hyperthreading of 2. The operating system was Cent OS 6.5. As in [1], 20 cores were assigned to each run of parSyn. For RSYNTH and CADET a single core on the cluster was used. bfss was executed on a single node; the initial computation of the functions, building of the error formula and refinement of the counterexamples was performed sequentially on 1 thread and UniGen had 39 threads at its disposal.

The maximum time given for execution was 3600 seconds, i.e., 1 hour. The total amount of main memory (across all cores) was restricted to 16GB. The metrics used to compare the algorithms were time taken to synthesize Boolean functions and maximum/average size of the synthesized functions.

**Results.** We now present the performance of the various algorithms. Our suite consisted of 113 benchmarks overall, with 68 disjunctive decomposition benchmarks, 40 arithmetic benchmarks, 5 factorization benchmarks. With the initial ordering as mentioned above, no tool succeeded on 8 benchmarks; of these 2 belonged to disjunctive decomposition and 6 to arithmetic benchmarks.
**BFSS vs PARSyn:** Figure 1a gives the performance of BFSS versus PARSyn with respect to time of the synthesized functions. Of the 113 benchmarks, with the least fan-in ordering BFSS was successful on 100 benchmarks. PARSyn was successful on 82 of the 113 benchmarks and there was no benchmark on which PARSyn was successful and BFSS was not. BFSS solved 18 more benchmarks than PARSyn, including factorization16, which BFSS took less than 15min to solve (in fact, PARSyn couldn’t solve this even in after 4 days).

On the benchmarks for which both algorithms were successful, there were at least 10 benchmarks on which BFSS took substantially lesser time, in some cases, by a factor of over 100! From Figure 1a, we can see that for some of the smaller benchmarks (time taken less than 1 minute), BFSS takes marginally more time than PARSyn. This is due to two reasons: i) PARSyn solves subproblems in parallel, while in BFSS only counterexamples are generated in parallel ii) if a small number of counterexamples are required then the cost of running UNIGEN is not suitably amortized over the total running time, resulting in BFSS taking more time. This can be addressed in most cases, by asking for a fewer solutions from UNIGEN. For instance, for factorization8, BFSS took 3.37 seconds when it asked for 2200 solutions. This was reduced to 0.94 seconds for 220 solutions. A comparison of average sizes of functions shows that those generated by BFSS are smaller or comparable to those generated by PARSyn. This coupled with the fact that BFSS can solve benchmarks in minutes which PARSyn cannot solve over days indicates its superiority over PARSyn.

**BFSS vs Cadet:** Figures 1b give the performance of BFSS versus Cadet with respect to time and average size of the synthesized functions. Of the 113 benchmarks, Cadet timed out on 71 benchmarks, of which 45 belonged to disjunctive decomposition, 21 belonged to arithmetic and 5, i.e., all the factorization benchmarks whereas BFSS timed out only on 13 benchmarks. Again, there was no benchmark on which Cadet was successful and BFSS was not. The benchmarks on which Cadet reports a time lower than BFSS belong to floor, ceiling and sub-
traction problem instances in the arithmetic benchmark category. However on most of the disjunctive decomposition benchmarks, factorization benchmarks and the decomposition, equalization and min benchmarks, BFSS outperforms CADET. Since, there are tseitin variables in .qdimacs format we compared the maximum sizes of the functions synthesized by CADET and BFSS. We found that for disjunctive decomposition benchmarks, both tools had instances on which they generate smaller skolem functions. For the arithmetic problem instances which were solved by CADET, we found that the maximum size is marginally smaller than that of BFSS.

BFSS vs RSYNTH: We next compare the performance of BFSS with RSYNTH. As shown in Figure 2a. RSYNTH was unsuccessful on 85 benchmarks, timing out on 82 benchmarks and running out of memory on 3 benchmarks. Of these, 57 of which belonged to the disjunctive disjunction category, 2 to factorization and 26 to the arithmetic benchmark category. The benchmarks on which it ran out of memory belong to the disjunctive decomposition category. There were 5 benchmarks which could be solved by RSYNTH but not by BFSS. These belonged to the smaller of the intermediate and max problem instances of arithmetic benchmark category. Since the functions generated by RSYNTH and BFSS are in different formats (BDDs and AIGs), we did not compare their sizes.

Effect of initial abstraction on bfss and number of counterexamples used: Of the 100 benchmarks that BFSS was successful, for a large majority, i.e., 95 of them, the initial skolem functions were indeed correct and did not need any refinements. Of the 5 benchmarks on which refinement was required, 64, 16866 and 4517 were the minimum, maximum and average number of counterexamples.

Effect of a different ordering on bfss: We ran BFSS on the 11 arithmetic benchmarks that it timed out earlier with an ordering which was reverse of the least fan-in ordering. We found that on this ordering BFSS could solve all of intermediate and max benchmarks except the largest ones, taking the count of
benchmarks unsolved by BFSS from 13 to 5. This also indicates that finding a good order is very important and this is part of our future work.

6 Conclusion and future work
We have presented a class of tunable algorithms for boolean functional synthesis which are counter-example based and allow us to trade-off between memory and time, while exploiting recent advances in uniform samplers. Our implementation and experiments show that our algorithm scales well on many medium to large benchmarks. Our algorithm is highly sensitive to the variable ordering and (as many others have observed), obtaining an optimal ordering is critical, and this forms an important direction for our future work. Identifying easy substructures and functional dependencies in the problems and simplifying them in a preprocessing step would also probably help our algorithm. Finally, we believe that by bringing the idea of uniform sampler, our works brings a new dimension to existing work on boolean functional synthesis.
References