Skolem Functions for Factored Formulas

Ajith K. John, HBNI, BARC, India
Shetal Shah, IIT Bombay
Supratik Chakraborty, IIT Bombay
Ashutosh Trivedi, IIT Bombay
S. Akshay, IIT Bombay

***Abstract*—Given a propositional formula $F(x, y)$, a Skolem function for x is a function $\psi(y)$, such that substituting $\psi(y)$ for x in F gives a formula semantically equivalent to $\exists x\ F$. Automatically generating Skolem functions is of significant interest in several applications including certified QBF solving, finding strategies of players in games, synthesising circuits and bit-vector programs from specifications, disjunctive decomposition of sequential circuits etc. In many such applications, F is given as a conjunction of factors, each of which depends on a small subset of variables. Existing algorithms for Skolem function generation ignore any such factored form and treat F as a monolithic function. This presents scalability hurdles in medium to large problem instances. In this paper, we argue that exploiting the factored form of F can give significant performance improvements in practice when computing Skolem functions. We present a new CEGAR style algorithm for generating Skolem functions from factored propositional formulas. In contrast to earlier work, our algorithm neither requires a proof of QBF satisfiability nor uses composition of monolithic conjunctions of factors. We show experimentally that our algorithm generates smaller Skolem functions and outperforms state-of-the-art approaches on several large benchmarks.**

I. Introduction

Skolem functions, introduced by Thoraf Skolem in the 1920s, occupy a central role in mathematical logic. Formally, let $F(x, y)$ be a first-order logic formula, and let $\mathrm{dom}(x)$ and $\mathrm{dom}(y)$ denote the domains of x and y respectively. A *Skolem function* for x in F is a function $\psi : \mathrm{dom}(y) \to \mathrm{dom}(x)$ such that substituting $\psi(y)$ for x in F yields a formula semantically equivalent to $\exists x F(x, y)$, i.e. $F(\psi(y), y) \equiv \exists x F(x, y)$. In this paper, we focus on the case where the formula F is propositional and given as a conjunction of factors. Classically, Skolem functions have been used in proving theorems in logic. More recently, with the advent of fast SAT/SMT solvers, it has been shown that several practically relevant problems can be encoded as quantified formulas, and can be solved by constructing *realizers* of quantified variables. We identify these realizers as specific instances of Skolem functions, and focus on algorithms for constructing them in this paper.

We begin by listing some applications that illustrate the utility of constructing instances of Skolem functions in practice.

1) *Quantifier elimination.* Given a quantified formula $Qx\ F(x, y)$, where $Q \in \{\exists, \forall\}$, the quantifier elimination problem requires us to find a quantifier-free formula that is semantically equivalent to $Qx\ F(x, y)$. Quantifier elimination has important applications in diverse areas (see, e.g. [7], [15], [2] for a sampling). It follows from the definition of Skolem function that eliminating the quantifier from $\exists x F(x, y)$ can be achieved by substituting x with a Skolem function for x. Since $\forall x F(x, y)$ can be written as $\neg \exists x \neg F(x, y)$, the same idea applies in this case too. In fact, the process can be repeated in principle to eliminate quantifiers from a formula with arbitrary quantifier prefix.
2) *Controller Synthesis and Games.* Control-program synthesis in the Ramadge-Wonham [13] framework reduces to games between two players—environment and the controller—such that the optimal strategy of the controller corresponds to an optimal control program. The optimal (or winning) strategy of the controller corresponds to choosing values of variables controlled by it such that regardless of the way the environment fixes its variables, the resulting play satisfies the controller's objective. If the rules of the game are encoded as a propositional formula and if the strategy space for both players is finite, the optimal strategy of the controller corresponds to finding Skolem functions of variables controlled by it. In fact, for a number of two-player games—such as reachability games and safety games [2], tic-tac-toe [5] and chess-like games [3], [2]—the problem of deciding a winner can be reduced to checking satisfiability of a quantified Boolean formula (QBF), and the problem of finding winning or best-effort strategy reduces to Skolem function generation.
3) *Graph Decomposition.* Skolem functions can be used to compute disjunctive decompositions of implicitly specified state transition graphs of sequential circuits [17]. The disjunctive decomposition problem asks the following question: Given a sequential circuit, derive "component" sequential circuits, each of which has the same state space as the original circuit, but only a subset of transitions going out of every state. The components should be such that the complete set of state transitions of the original circuit is the union of the sets of state transitions of the components. Disjunctive decompositions have been shown to be useful in efficient reachability analysis [16].

There are several other practical applications where Skolem functions find use; see, e.g. [12], for a discussion. Hence, there is a growing need for practically efficient and scalable approaches for generating instances of Skolem functions. Large and complex representations of the formula F in $\exists x\ F$ often present scalability hurdles in generating Skolem functions in practice. Interestingly, for several problem instances, the specification of F is available in a *factored* form, i.e., as a conjunction of simpler sub-formulas, each of which depends on a subset of variables appearing in F. Unfortunately, unlike in the case of disjunction, existential quantification does not distribute over conjunction of sub-formulas. Existing algorithms therefore ignore any factored form of F and treat the conjunction of factors as a single monolithic function. We show in this paper that exploiting the factored form can help significantly when generating Skolem functions.

Our main technical contribution is a SAT-based Counter-Example Guided Abstraction-Refinement (CEGAR) algorithm for generating Skolem functions from factored formulas. Unlike competing approaches, our algorithm exploits the factored representation of a formula and leverages advances made in SAT-solving technology. The factored representation is used to arrive at an initial abstraction of Skolem functions, while a SAT-solver is used as an oracle to identify counter-examples that are used to refine the Skolem functions until no counter-examples exist. We present a detailed experimental evaluation of our algorithm vis-a-vis state-of-the-art algorithms [7], [12] over a large class of benchmarks. We show that on several large problem instances, we outperform competing algorithms. Proofs that are omitted can be found in the long version at [9].

Related Work. We are not aware of other techniques for Skolem function generation that exploit the factored form of a formula. Earlier work on Skolem function generation broadly fall in one of four categories. The first category includes techniques that extract Skolem functions from a proof of validity of $\exists X\ F(X,Y)$ [12], [8], [4], [10]. In problem instances where $\exists X\ F(X,Y)$ is valid (and this forms an important sub-class of problems), these techniques can usually find succinct Skolem functions if there exists a short proof of validity. However, in several other important classes of problems, the formula $\exists X\ F(X,Y)$ does not evaluate to true for all values of Y, and techniques in the first category cannot be applied. The second category includes techniques that use templates for candidate Skolem functions [15]. These techniques are effective only when the set of candidate Skolem functions is known and small. While this is a reasonable assumption in some domains [15], it is not in most other domains. BDD-based techniques [14] are yet another way to compute Skolem functions. Unfortunately, these techniques are known not to scale well, unless custom-crafted variable orders are used. The last category includes techniques that use cofactors to obtain Skolem functions [7], [17]. These techniques do not exploit the factored representation of a formula and, as we show experimentally, do not scale well to large problem instances.

II. Preliminaries

We use lower case letters (possibly with subscripts) to denote propositional variables, and upper case letters to denote sequences of such variables. We use 0 and 1 to denote the propositional constants false and true, respectively. Let $F(X,Y)$ be a propositional formula, where X and Y denote the sequences of variables $(x_1,\ldots,x_n)$ and $(y_1,\ldots,y_m)$, respectively. We are interested in problem instances where $F(X,Y)$ is given as a conjunction of factors $f^1(X_1,Y_1),\ldots,f^r(X_r,Y_r)$, where each X_i (resp., Y_i) is a possibly empty sub-sequence of X (resp., Y). For notational convenience, we use F and $\bigwedge_{j=1}^{r} f^j$ interchangeably throughout this paper. The set of variables in F is called the *support* of F, and is denoted $\mathsf{Supp}(F)$. Given a propositional formula $F(X)$ and a propositional function $\Psi(X)$, we use $F[x_i/\Psi(X)]$, or simply $F[x_i/\Psi]$, to denote the formula obtained by substituting every occurrence of the variable x_i in F with $\Psi(X)$. Since the notions of formulas and functions coincide in propositional logic, the above is also conventionally called *function composition*. If X is a sequence of variables and x_i is a variable, we use $X \setminus x_i$ to denote the sub-sequence of X obtained by removing x_i (if present) from X. Abusing notation, we use X to also denote the set of elements in X, when there is no confusion. A *valuation* or *assignment* π of X is a mapping $\pi : X \to \{0,1\}$.

Definition 1. *Given a propositional formula $F(X,Y)$ and a variable $x_i \in X$, a* Skolem function *for x_i in $F(X,Y)$ is a function $\psi(X \setminus x_i, Y)$ such that $\exists x_i\ F \equiv F[x_i/\psi]$.*

A Skolem function for x_i in F need not be unique. The following proposition, which effectively follows from [7], [17], characterizes the space of all Skolem functions for x_i in F.

Proposition 1. *A function $\psi(X \setminus x_i, Y)$ is a Skolem function for x_i in $F(X,Y)$ iff $F[x_i/1] \wedge \neg F[x_i/0] \Rightarrow \psi$ and $\psi \Rightarrow F[x_i/1] \vee \neg F[x_i/0]$.*

The function $F[x_i/0]$ (resp., $F[x_i/1]$) is called the *positive* (resp., *negative*) *cofactor* of F with respect to x_i, and plays a central role in the study of Skolem functions for propositional formulas. In particular, it follows from Proposition 1 that $F[x_i/1]$ is a Skolem function for x_i in F. The above definition for a single variable can be naturally extended to a vector of variables. Given $F(X,Y)$, a *Skolem function vector* for $X = (x_1,\ldots,x_n)$ in F is a vector of functions $\mathbf{\Psi} = (\psi_1,\ldots,\psi_n)$ such that $\exists x_1\ldots x_n\ F \equiv (\cdots(F[x_1/\psi_1])\cdots[x_n/\psi_n])$. A straightforward way to obtain a Skolem function vector $\mathbf{\Psi}$ is to first obtain a Skolem function ψ_1 for x_1 in F, then compute $F' \equiv \exists x_1\, F$ and obtain a Skolem function ψ_2 for x_2 in F', and so on until ψ_n has been obtained. More formally, ψ_i can be computed as a Skolem function for x_i in $\exists x_1 \ldots x_{i-1}\, F$, starting from ψ_1 and proceeding to ψ_n. Note that $\exists x_1 \ldots x_{i-1}\, F$ can itself be computed as $(\cdots(F[x_1/\psi_1])\cdots[x_{i-1}/\psi_{i-1}])$.

Definition 2. *The "**C**an't-**b**e-**1**" function for x_i in F, denoted $\mathtt{Cb1}[x_i](F)$, is defined to be $(\neg\exists x_1\ldots x_{i-1}\, F)\,[x_i/1]$. Similarly, the "**C**an't-**b**e-**0**" function for x_i in F, denoted $\mathtt{Cb0}[x_i](F)$, is defined to be $(\neg\exists x_1\ldots x_{i-1}\, F)\,[x_i/0]$. When X and F are clear from the context, we use $\mathtt{Cb1}[i]$ and $\mathtt{Cb0}[i]$ for $\mathtt{Cb1}[x_i](F)$ and $\mathtt{Cb0}[x_i](F)$, respectively.*

Intuitively, in order to make F evaluate to 1, we cannot set x_i to 1 (resp. 0) whenever the valuation of $\{x_{i+1},\ldots,x_n\}\cup Y$ satisfies $\mathtt{Cb1}[i]$ (resp., $\mathtt{Cb0}[i]$). The following proposition follows from Definition 2 and from our observation about computing a Skolem function vector one component at a time.

Proposition 2. $\mathbf{\Psi}=(\neg\texttt{Cb1}[1],\ldots,\neg\texttt{Cb1}[n])$ *is a Skolem function vector for* X *in* F.

Note that the support of ψ_i in $\mathbf{\Psi}$, as given by Proposition 2, is $\{x_{i+1},\ldots,x_n\}\cup Y$. If we want a Skolem function vector $\mathbf{\Psi}$ such that every component function has only Y (or a subset thereof) as support, this can be obtained by repeatedly substituting the Skolem function for every variable x_i in all other Skolem functions where x_i appears. We denote such a Skolem function vector as $\mathbf{\Psi}(Y)$.

III. A MONOLITHIC COMPOSITION BASED ALGORITHM

Our algorithm is motivated in part by cofactor-based techniques for computing Skolem functions, as proposed by Jiang et al [7] and Trivedi [17]. Given $F(X,Y)=\bigwedge_{j=1}^{r} f^j(X_j,Y_j)$, the techniques of [7], [17] essentially compute a Skolem function vector $\mathbf{\Psi}(Y)$ for X in F as shown in algorithm MONOSKOLEM (see Algorithm 1). In this algorithm, the variables in X are assumed to be ordered by their indices. While variable ordering is known to affect the difficulty of computing Skolem functions [7], we assume w.l.o.g. that the variables are indexed to represent a desirable order. We describe the variable order used in our study later in Section V.

MONOSKOLEM works in two phases. In the first phase, it implements a straightforward strategy for obtaining a Skolem function vector, as suggested by Proposition 2. Specifically, steps 3 and 4 of MONOSKOLEM build a monolithic conjunction F_i of all factors that have x_i in their support, before computing ψ_i. This restricts the scope of the quantifier for x_i to the conjunction of these factors. In Step 6, we use $\neg\texttt{Cb1}[i]$ as a specific choice for the Skolem function ψ_i. After computing ψ_i from F_i, step 7 discards the factors with x_i in their support, and introduces a single factor representing $\exists x_i\, F_i$ (computed as $F_i[x_i/\psi_i]$) in their place. Note that each ψ_i obtained in this manner has $\{x_{i+1},\ldots,x_n\}\cup Y$ (or a subset thereof) as support. Since we want each Skolem function to have support Y, a second phase of "reverse" substitutions is needed. In this phase (see Algorithm 2), the Skolem function $\psi_n(Y)$ obtained above is substituted for x_n in $\psi_1,\ldots,\psi_{n-1}$. This effectively renders all Skolem functions independent of x_n. The process is then repeated with ψ_{n-1} substituted for x_{n-1} in $\psi_1,\ldots,\psi_{n-2}$ and so on, until all Skolem functions have been made independent of $x_1,\ldots,x_n$, and have only Y (or subsets thereof) as support.

MONOSKOLEM can be further refined by combining steps 6 and 7, and directly defining ψ_i in terms of F_i. However, we introduce the intermediate step using $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$ to motivate their central role in our approach. Note that instead of $\neg\texttt{Cb1}[i]$, we could combine $\texttt{Cb1}[i]$ and $\texttt{Cb0}[i]$ in other ways (denoted by COMBINE($\texttt{Cb0}[i],\texttt{Cb1}[i]$) within comments in Algorithm 1) to get ψ_i in Step 6. In fact, Jiang et al [7] compute a Skolem function for x_i in F as an interpolant of $\neg\texttt{Cb1}[i]\wedge\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]\wedge\neg\texttt{Cb0}[i]$, while Trivedi [17] observes that the function $(\neg\texttt{Cb1}[i]\wedge(\texttt{Cb0}[i]\vee g))\vee(\texttt{Cb1}[i]\wedge\texttt{Cb0}[i]\wedge h)$ serves as a Skolem function for x_i in F where h and g are arbitrary propositional functions with support in

Algorithm 1: MONOSKOLEM

```
Input: Prop. formula $F(X,Y)=\bigwedge_{j=1}^{r} f^j(X_j,Y_j)$,
       where $X=(x_1,\ldots,x_n)$
Output: Skolem function vector $\mathbf{\Psi}(Y)$
// Phase 1 of algorithm
1 Factors := $\{f^j : 1\le j\le r\}$;
2 for $i$ in 1 to $n$ do
3     FactorsWithXi := $\{f : f\in$ Factors$, x_i\in$ Supp$(f)\}$;
4     $F_i := \bigwedge_{f\in\text{FactorsWithXi}} f$;
5     Cb0$[i] := \neg F_i[x_i/0]$; Cb1$[i] := \neg F_i[x_i/1]$;
6     $\psi_i := \neg$Cb1$[i]$;
      // Generally, $\psi_i$:=COMBINE(Cb0$[i]$,Cb1$[i]$);
7     Factors := (Factors \ FactorsWithXi) $\cup\{F_i[x_i/\psi_i]\}$;
// Phase 2 of algorithm
8 return REVERSESUBSTITUTE($\psi_1,\ldots,\psi_n$);
```

$X\setminus\{x_i\}\cup Y$. Since computing interpolants using a SAT solver is often time-intensive and does not always lead to succinct Skolem functions [7], we simply use $\neg\texttt{Cb1}[i]$ as a Skolem function in Step 6. Proposition 2 guarantees the correctness of this choice.

Algorithm 2: REVERSESUBSTITUTE

```
Input: Functions
       $\psi_1(x_2,\ldots,x_n,Y),\psi_2(x_3,\ldots,x_n,Y),\ldots,\psi_n(Y)$
Output: Function vector $\mathbf{\Psi}(Y)$
1 for $i=n$ downto 2 do
2     for $k=i-1$ downto 1 do $\psi_k=\psi_k[x_i/\psi_i]$;
3 return $\mathbf{\Psi}(Y)=(\psi_1(Y),\ldots,\psi_n(Y))$;
```

Observe that MONOSKOLEM works with a *monolithic* conjunction (F_i) of factors that have x_i in their support. Specifically, it composes each such monolithic conjunction F_i with a cofactor of F_i in Step 7 to eliminate quantifiers sequentially. This can lead to large memory footprints and more time-outs when used with medium to large benchmarks, as confirmed by our experiments. This motivates us to ask if we can develop a cofactor-based algorithm that does not suffer from the above drawbacks of MONOSKOLEM.

IV. CEGAR FOR GENERATING SKOLEM FUNCTIONS

We now present a new CEGAR [6] algorithm for generating Skolem function vectors, that exploits the factored form of $F(X,Y)$. Like MONOSKOLEM, our new algorithm, named CEGARSKOLEM, works in two phases, and assumes that the variables in X are ordered by their indices. The first phase of the algorithm consists of the core abstraction-refinement part, and computes a Skolem function vector $(\psi_1,\ldots,\psi_n)$, where ψ_i has $\{x_{i+1},\ldots,x_n\}\cup Y$, or a subset thereof, as support. Unlike in MONOSKOLEM, this phase avoids composing monolithic conjunctions of factors, yielding simpler Skolem functions. The second phase of the algorithm performs reverse substitutions, similar to that in MONOSKOLEM.

Before describing the details of CEGARSKOLEM, we introduce some additional notation and terminology. Given propositional functions (or formulas) f and g, we say that f *refines* g and g *abstracts* f iff f logically implies g. Given $F(X, Y)$ and a vector of functions $\mathbf{\Psi^A} = (\psi_1^A, \ldots, \psi_n^A)$, we say that $\mathbf{\Psi^A}$ is an *abstract Skolem function vector* for X in F iff there exists a Skolem function vector $\mathbf{\Psi} = (\psi_1, \ldots, \psi_n)$ for X in F such that ψ_i^A abstracts ψ_i, for every $i \in \{1, \ldots, n\}$. Instead of using $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$ to compute Skolem functions, as was done in MONOSKOLEM, we now use their *refinements*, denoted $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ respectively, to compute abstract Skolem functions. For convenience, we represent $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ as sets of implicitly disjoined functions. Thus, if $\mathtt{r1}[i]$, viewed as a set, is $\{g_1, g_2\}$, then it is $g_1 \vee g_2$ when viewed as a function. We abuse notation and use $\mathtt{r1}[i]$ (resp., $\mathtt{r0}[i]$) to denote a set of functions or their disjunction, as needed.

A. Overview of our CEGAR algorithm

Algorithm CEGARSKOLEM has two phases. The first phase consists of a CEGAR loop, while the second does reverse substitutions. The CEGAR loop has the following steps.

- **Initial abstraction and refinement.** This step involves constructing refinements of $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$ for every x_i in X. Using Proposition 2, we can then construct an initial abstract Skolem function vector $\mathbf{\Psi^A}$. This step is implemented in Algorithm 3 (INITABSREF), which processes individual factors of $F(X, Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$ separately, without considering their conjunction. As a result, this step is time and memory efficient if the individual factors are simple with small representations.
- **Termination Condition.** Once INITABSREF has computed $\mathbf{\Psi^A}$, we check whether $\mathbf{\Psi^A}$ is already a Skolem function vector. This is achieved by constructing an appropriate propositional formula ε, called the "error formula" for $\mathbf{\Psi^A}$ (details in Subsection IV-C), and checking for its satisfiability. An unsatisfiable formula implies that $\mathbf{\Psi^A}$ is a Skolem function vector. Otherwise, a satisfying assignment π of ε is used to improve the current refinements of $\mathtt{Cb1}[i]$ and $\mathtt{Cb0}[i]$ for suitable variables x_i.
- **Counterexample guided abstraction and refinement.** This step is implemented in Algorithm 4: UPDATEABSREF, and computes an improved (i.e., more abstract) refinement of $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$ for some $x_i \in X$. This, in turn, leads to a refinement of the abstract Skolem function vector $\mathbf{\Psi^A}$.

The overall CEGAR loop starts with the first step and repeats the second and third steps until a Skolem function vector is obtained. We now discuss the three steps in detail.

B. Initial Abstraction and Refinement

Algorithm INITABSREF (see Algorithm 3) starts by initializing each $\mathtt{r1}[i]$ and $\mathtt{r0}[i]$, viewed as sets, to the empty set. Subsequently, it considers each factor f in $\bigwedge_{j=1}^{r} f^j(X_j, Y_j)$, and determines the contribution of f to $\mathtt{Cb0}[i]$ and $\mathtt{Cb1}[i]$, for every x_i in the support of f. Specifically, if $x_i \in \mathsf{Supp}(f)$, the contribution of f to $\mathtt{Cb0}[i]$ is $(\neg\exists x_1 \ldots x_{i-1} f)[x_i/0]$, and its contribution to $\mathtt{Cb1}[i]$ is $(\neg\exists x_1 \ldots x_{i-1} f)[x_i/1]$. These contributions are accumulated in the sets $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$, respectively, and x_i is existentially quantified from f. The process is then repeated with the next variable in the support of f. Once the contributions from all factors are accumulated in $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ for each x_i in X, INITABSREF computes an abstract Skolem function ψ_i^A for each x_i in F by complementing $\mathtt{r1}[i]$, interpreted as a disjunction of functions. Note that executing steps 4 through 10 of INITABSREF for a specific factor f is operationally similar to executing steps 1 through 7 of MONOSKOLEM with a singleton set of factors, i.e., $\mathsf{Factors} = \{f\}$. This highlights the key difference between INITABSREF and MONOSKOLEM: while MONOSKOLEM works with monolithic conjunctions of factors and their compositions, INITABSREF works with individual factors, without ever considering their conjunctions. Lemma 1 asserts the correctness of INITABSREF.

Algorithm 3: INITABSREF

```
Input: Prop. formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$,
       where $X = (x_1, \ldots, x_n)$
Output: Abstract Skolem function vector
        $\mathbf{\Psi^A} = (\psi_1^A, \ldots, \psi_n^A)$, and refinements r0[$i$] and
        r1[$i$] for each $x_i$ in $X$
1  for $i$ in 1 to $n$ do
2      r0[$i$] := $\emptyset$; r1[$i$] := $\emptyset$; // Initializing
3  for $j$ in 1 to $r$ do
4      $f := f^j$; // for each factor
5      for $i$ in 1 to $n$ do
6          if $x_i \in \mathsf{Supp}(f)$ then
7              r0[$i$] := r0[$i$] $\cup \{\neg f[x_i/0]\}$;
8              r1[$i$] := r1[$i$] $\cup \{\neg f[x_i/1]\}$;
               // Skolem function for $x_i$ in $f$
9              $\psi_{i,f} := f[x_i/1]$;
10             $f := f[x_i/\psi_{i,f}]$; // $\because f[x_i/\psi_{i,f}] \equiv \exists x_i\, f$
11 for $i$ in 1 to $n$ do
12     $\psi_i^A := \neg$r1[$i$];
       // Interpreting r1[$i$] as a function
13 return $\mathbf{\Psi^A}$=$(\psi_1^A, \ldots, \psi_n^A)$ and r0[$i$], r1[$i$] $\forall x_i \in X$
```

Lemma 1. *The vector $\mathbf{\Psi^A}$ computed by* INITABSREF *is an abstract Skolem function vector for X in $F(X, Y)$. In addition,* $\mathtt{r0}[i]$ *and* $\mathtt{r1}[i]$ *computed by* INITABSREF *are refinements of* $\mathtt{Cb0}[i](F)$ *and* $\mathtt{Cb1}[i](F)$ *for every x_i in X.*

C. Termination condition

Given $F(X, Y)$ and an abstract Skolem function vector $\mathbf{\Psi^A}$, it may happen that $\mathbf{\Psi^A}$ is already a Skolem function vector for X in F. We therefore check if $\mathbf{\Psi^A}$ is a Skolem function vector before refinement. Towards this end, we define the *error formula* for $\mathbf{\Psi^A}$ as $F(X', Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X, Y)$, where $X' = (x_1', \ldots, x_n')$ is a sequence of fresh variables with no variable in common with X. The first term in the error formula checks if there exists some valuation of X that renders $\exists Y F(X,Y)$ true. The second term assigns variables in X to the values given by the abstract Skolem functions, and the third term checks if this assignment falsifies the formula F.

Lemma 2. *The error formula for $\mathbf{\Psi^A}$ is unsatisfiable iff $\mathbf{\Psi^A}$ is a Skolem function vector of X in F.*

The following example illustrates the role of the error formula.

Example 1. *Let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2, y_3\}$ in $\exists x_1 x_2 F(X,Y)$ where $F \equiv (f_1 \wedge f_2 \wedge f_3)$, with $f_1 = (\neg x_1 \vee \neg x_2 \vee \neg y_1)$, $f_2 = (x_2 \vee \neg y_3 \vee \neg y_2)$, $f_3 = (x_1 \vee \neg x_2 \vee y_3)$.*

Algorithm InitAbsRef *gives $\texttt{r1}[1] = (x_2 \wedge y_1)$, $\texttt{r0}[1] = (x_2 \wedge \neg y_3)$, $\texttt{r1}[2] =$ false, $\texttt{r0}[2] = y_3 \wedge y_2$. This yields $\psi_1^A = (\neg x_2 \vee \neg y_1)$, $\psi_2^A =$ true. Now, while ψ_1^A is a correct Skolem function for x_1 in F, ψ_2^A is not for x_2. This is detected by the satisfiability of the error formula $\varepsilon = F(x_1', x_2', Y) \wedge (x_1 = \neg x_2 \vee \neg y_1) \wedge (x_2 = 1) \wedge \neg F(x_1, x_2, Y)$. Note that $\neg F(\neg x_2 \vee \neg y_1, 1, Y)$ simplifies to $(y_1 \wedge \neg y_3)$, and $y_1 = 1, y_2 = 1, y_3 = 0, x_1 = 0, x_2 = 1, x_1' = 0, x_2' = 0$ is a satisfying assignment for ε.*

D. Counterexample-guided abstraction and refinement

Let ε be the error formula for $\mathbf{\Psi^A}$, and let π be a satisfying assignment of ε. We call π a *counterexample* of the claim that $\mathbf{\Psi^A}$ is a Skolem function vector. For every variable $v \in X' \cup X \cup Y$, we use $\pi(v)$ to denote the value of v in π. Satisfiability of ε implies that we need to refine at least one abstract Skolem function ψ_i^A in $\mathbf{\Psi^A}$ to make it a Skolem function vector. Since ψ_i^A is $\neg\texttt{r1}[i]$ in our approach, refining ψ_i^A can be achieved by computing an improved (i.e., more abstract) version of $\texttt{r1}[i]$. Algorithm UpdateAbsRef implements this idea by using π to determine which $\texttt{r1}[i]$ should be rendered abstract by adding appropriate functions to $\texttt{r1}[i]$, viewed as a set.

Before delving into the details of UpdateAbsRef, we state some key results. In the following, we use $\pi \models f$ to denote that the formula f evaluates to 1 when the variables in $\mathsf{Supp}(f)$ are set to values given by π. If $\pi \models f$, we also say f evaluates to 1 under π. We use $\texttt{r0}[i]_{init}$ and $\texttt{r1}[i]_{init}$ to refer to $\texttt{r0}[i]$ and $\texttt{r1}[i]$, as computed by algorithm InitAbsRef. Since UpdateAbsRef only adds to $\texttt{r1}[i]$ and $\texttt{r0}[i]$ viewed as sets, it is easy to see that $\texttt{r0}[i]_{init} \Rightarrow \texttt{r0}[i]$ and $\texttt{r1}[i]_{init} \Rightarrow \texttt{r1}[i]$ viewed as functions (recall these functions are simply disjunctions of elements in the corresponding sets).

Lemma 3. *Let π be a satisfying assignment of the error formula ε for $\mathbf{\Psi^A}$. Then the following hold.*

(a) $\pi \models \neg\texttt{Cb0}[n] \vee \neg\texttt{Cb1}[n]$.
(b) There exists $k \in \{1, \ldots, n-1\}$ s.t., $\pi \models \texttt{r1}[k] \wedge \texttt{r0}[k]$.
(c) There exists no Skolem function vector $\mathbf{\Psi} = (\psi_1, \ldots, \psi_n)$ such that $\psi_j \Leftrightarrow \psi_j^A$ for all j in $\{k+1, \ldots, n\}$.
(d) There exists $l \in \{k+1, \ldots, n\}$ such that $x_l = 1$ in π, and $\pi \models \texttt{Cb1}[l] \wedge \neg\texttt{r0}[l]$.

Algorithm 4 (UpdateAbsRef) uses Lemma 3 to compute abstract versions of $\texttt{r0}[i]$ and $\texttt{r1}[i]$, and a refined version of $\mathbf{\Psi^A}$, when $\mathbf{\Psi^A}$ is not a Skolem function vector. It takes as input the current versions of $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for all x_i in X, and a satisfying assignment π of the error formula for the current version of $\mathbf{\Psi^A}$. Since $\pi \models F(X', Y)$ and $\pi \models \neg F(X,Y)$, and since the value of every x_i in π is given by ψ_i^A, there exists at least one ψ_l^A, for $l \in \{1, \ldots, n\}$, that fails to generate the right value of x_l when the value of Y is as given by π. UpdateAbsRef works by identifying such an index l and refining ψ_l^A. Since $\psi_i^A = \neg\texttt{r1}[i]$, ψ_l^A is refined by updating (abstracting) the corresponding $\texttt{r1}[l]$ set. In fact, the algorithm may, in general, end up abstracting not only $\texttt{r1}[l]$, but several $\texttt{r0}[i]$ and $\texttt{r1}[i]$ as well in a sound manner.

Algorithm 4: UpdateAbsRef

```
Input: $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for all $x_i$ in $X$,
       Satisfying assignment $\pi$ of error formula, i.e.,
       $F(X',Y) \wedge \bigwedge_{i=1}^{n} (x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y)$
Output: Improved (i.e., refined) $\mathbf{\Psi^A} = (\psi_1^A, \ldots, \psi_n^A)$,
        Improved (i.e., abstracted) $\texttt{r0}[i]$ & $\texttt{r1}[i]$, $\forall x_i \in X$
1  $k :=$ largest $m$ such that $\pi$ satisfies $\texttt{r0}[m] \wedge \texttt{r1}[m]$;
2  $\mu_0 :=$ Generalize($\pi$, $\texttt{r0}[k]$);
3  $\mu_1 :=$ Generalize($\pi$, $\texttt{r1}[k]$);
4  $\mu := \mu_0 \wedge \mu_1$;
   // Search for Skolem function among
      $\{\psi_{k+1}^A, \ldots, \psi_n^A\}$ to be refined
5  $l := k + 1$;
6  while true do    // current guess: refine $\psi_l^A$
7      if $x_l \in \mathsf{Supp}(\mu)$ then
8          if $x_l = 1$ in $\pi$ then
9              $\mu_1 := \mu[x_l/1]$;
10             $\texttt{r1}[l] := \texttt{r1}[l] \cup \{\mu_1\}$;
11             if $\pi$ satisfies $\texttt{r0}[l]$ then
12                 $\mu_0 :=$ Generalize($\pi$, $\texttt{r0}[l]$);
13                 $\mu := \mu_0 \wedge \mu_1$;
14             else
15                 break;
16         else
17             $\mu_0 := \mu[x_l/0]$;
18             $\texttt{r0}[l] := \texttt{r0}[l] \cup \{\mu_0\}$;
19             $\mu_1 :=$ Generalize($\pi$, $\texttt{r1}[l]$);
20             $\mu := \mu_0 \wedge \mu_1$;
21     $l := l + 1$ ;
22 $\mathbf{\Psi^A} = (\neg\texttt{r1}[1], \ldots, \neg\texttt{r1}[n])$;
23 return $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for all $x_i$ in $X$, and $\mathbf{\Psi^A}$
```

As shown in Algorithm 4, UpdateAbsRef first finds the largest index k such that $\pi \models \texttt{r0}[k] \wedge \texttt{r1}[k]$. Lemma 3b guarantees the existence of such an index in $\{1, \ldots, n\}$. We assume access to a function called Generalize that takes as arguments an assignment π and a function φ such that $\pi \models \varphi$, and returns a function ξ that generalizes π while satisfying φ. More formally, if $\xi =$ Generalize(π, φ), then $\mathsf{Supp}(\xi) \subseteq \mathsf{Supp}(\varphi)$, $\pi \models \xi$ and $\xi \Rightarrow \varphi$ (details of

GENERALIZE used in our implementation are discussed later). Thus, in steps 2 and 3 of UPDATEABSREF, we compute generalizations of π that satisfy $\texttt{r0}[k]$ and $\texttt{r1}[k]$, respectively. The function μ computed in step 4 is therefore such that $\pi \models \mu$ and $\mu \Rightarrow \texttt{r0}[k] \wedge \texttt{r1}[k]$. Since $\texttt{r0}[k] \wedge \texttt{r1}[k] \Rightarrow \neg\exists x_1 \ldots x_k F$, any abstract Skolem function vector that produces values of $x_1, \ldots, x_n$ (given the valuation of Y as in π) for which μ evaluates to 1, cannot be a Skolem function vector. Since the support of μ is $\{x_{k+1}, \ldots, x_n\} \cup Y$, one of the abstract Skolem functions $\psi^A_{k+1}, \ldots, \psi^A_n$ must be refined.

The loop in steps 6–21 of UPDATEABSREF tries to identify an abstract Skolem function ψ^A_l to be refined, by iterating l from $k+1$ to n. Clearly, if $x_l \notin \mathsf{Supp}(\mu)$, the value of ψ^A_l under π is of no consequence in evaluating μ, and we ignore such variables. If $x_l \in \mathsf{Supp}(\mu)$ and if $x_l = 1$ in π, then $\pi \models \mu[x_l/1]$ and $\mu[x_l/1] \Rightarrow (\neg\exists x_1 \ldots x_{l-1} F)[x_l/1]$. Recalling the definition of $\texttt{Cb1}[l]$, we have $\mu[x_l/1] \Rightarrow \texttt{Cb1}[l]$, and therefore $\mu[x_l/1]$ can be added to $\texttt{r1}[l]$ (viewed as a set) yielding a more abstract version of $\texttt{r1}[l]$. Steps 8–10 of UPDATEABSREF implement this update of $\texttt{r1}[l]$. Note that since $\pi \models \mu[x_l/1]$, we have $\pi \models \texttt{r1}[l]$ after step 10. If it so happens that $\pi \models \texttt{r0}[l]$ as well, then we have $\pi \models \texttt{r0}[l] \wedge \texttt{r1}[l]$, where $\texttt{r1}[l]$ refers to the updated refinement of $\texttt{Cb1}[l]$. In this case, we have effectively found an index $l > k$ such that $\pi \models \texttt{r0}[k] \wedge \texttt{r1}[k]$. We can therefore repeat our algorithm starting with l instead of k. Steps 11–13 followed by step 21 of algorithm UPDATEABSREF effectively implement this. If, on the other hand, $\pi \not\models \texttt{r0}[k]$, then we have found an l that satisfies the conditions in Lemma 3d. We exit the search for an abstract Skolem function in this case (see steps 14–15).

If $x_l = 0$ in π, a similar argument as above shows that $\mu[x_l/0]$ can be added to $\texttt{r0}[l]$. Steps 17–18 of UPDATEABSREF implement this update. As before, it is easy to see that $\pi \models \texttt{r0}[l]$ after step 18. Moreover, since $\pi \models \bigwedge_{i=1}^n (x_i \Leftrightarrow \psi^A_i)$ and $\psi^A_i \equiv \neg\texttt{r1}[l]$, in order to have $x_l = 0$ in π, we must have $\pi \models \texttt{r1}[l]$. Therefore, we have once again found an index $l > k$ such that $\pi \models \texttt{r0}[k] \wedge \texttt{r1}[k]$, and can repeat our algorithm starting with l instead of k. Steps 19–21 of algorithm UPDATEABSREF effectively implement this.

Once we exit the loop in steps 6–21 of UPDATEABSREF, we compute the refined Skolem function vector $\mathbf{\Psi^A}$ as $(\neg\texttt{r1}[1], \ldots \neg\texttt{r1}[n])$ in step 22 and return the updated $\texttt{r0}[i]$, $\texttt{r1}[i]$ for all x_i in X, and also $\mathbf{\Psi^A}$.

Example 1 (Continued)**.** *Continuing with our earlier example, the error formula after the first step has a satisfying assignment* $y_1 = 1, y_2 = 1, y_3 = 0, x_1 = 0, x_2 = 1, x'_1 = 0, x'_2 = 0$. *Using this for* π *in* UPDATEABSREF*, we find that* ψ^A_1 *is left unchanged at* $(\neg x_2 \vee \neg y_1)$*, while* ψ^A_2*, which was* true *earlier, is refined to* $(\neg y_1 \vee y_3)$*. With these refined Skolem functions,* $F(\psi^A_1, \psi^A_2, Y)$ *evaluates to* true *for all valuations of* Y*. As a result, the (new) error formula becomes unsatisfiable, confirming the correctness of the Skolem functions.*

It can be shown that Algorithm UPDATEABSREF always terminates, and renders at least one $\texttt{r1}[i]$ strictly abstract, and at least one ψ^A_i strictly refined, for $i \in \{1, \ldots, n\}$ (see [9] for the proof). The overall CEGARSKOLEM algorithm can now be implemented as depicted in Algorithm 5. From the above discussion and Lemmas 1 and 2, we obtain our main result.

Algorithm 5: CEGARSKOLEM

```
Input: Propositional formula
    $F(X,Y) = \bigwedge_{j=1}^r f^j(X_j, Y_j)$, $X = (x_1, \ldots, x_n)$
Output: Skolem function vector $\mathbf{\Psi}(Y)$ for $X$ in $F$
1 $(\mathbf{\Psi^A}, \{\texttt{r0}[i], \texttt{r1}[i] : 1 \le i \le n\}) :=$ INITABSREF$(\bigwedge_{j=1}^r f^j)$;
2 $\varepsilon := F(X', Y) \wedge \bigwedge_{i=1}^n (x_i \Leftrightarrow \psi^A_i) \wedge \neg F(X,Y)$;
3 while $\varepsilon$ is satisfiable do
4     Let $\pi$ be a satisfying assignment of $\varepsilon$;
5     $(\mathbf{\Psi^A}, \{\texttt{r0}[i], \texttt{r1}[i] : 1 \le i \le n\}) :=$ UPDATEABSREF$(\{\texttt{r0}[i], \texttt{r1}[i] : 1 \le i \le n\}, \pi)$;
6     $\varepsilon := F(X', Y) \wedge \bigwedge_{i=1}^n (x_i \Leftrightarrow \psi^A_i) \wedge \neg F(X,Y)$;
7 $\mathbf{\Psi}(Y) :=$ REVERSESUBSTITUTE$(\neg\texttt{r1}[1], \ldots, \neg\texttt{r1}[n])$;
8 return $\mathbf{\Psi}(Y)$;
```

Theorem 1. *CEGARSKOLEM($F(X,Y)$) terminates and computes a Skolem function vector for* X *in* F*.*

The function GENERALIZE(π, φ) used in UPDATEABSREF can be implemented in several ways. Since $\pi \models \varphi$, we may return a conjunction of literals corresponding to the assignment π, or the function φ itself. From our experiments, it appears that the first option leads to low memory requirements and increased run-time (due to large number of invocations of UPDATEABSREF). The other option requires more memory and less run-time due to fewer invocations of UPDATEABSREF. For our study, we let GENERALIZE(π, $\texttt{r1}[k]$) return one element in $\texttt{r1}[k]$ (viewed as a set) amongst all those that evaluate to 1 under π, such that the support of μ computed in Algorithm UPDATEABSREF is minimized (we had to allow GENERALIZE$(\cdot,\cdot)$ access to μ for this purpose). We follow a similar strategy for GENERALIZE(π, $\texttt{r0}[k]$). This gives us a reasonable tradeoff between time and space requirements.

V. EXPERIMENTAL RESULTS

A. Experimental Methodology

We compared CEGARSKOLEM with (a) MONOSKOLEM (the algorithm based on the cofactoring approach of [7], [17]) and with (b) Bloqqer (a QRAT-based Skolem function generation tool reported in [12]). As described in [12], Bloqqer generates Skolem functions by first generating QRAT proofs using a remarkably efficient (albeit incomplete) preprocessor, and then generates Skolem functions from these proofs.

The Skolem function generation benchmarks were obtained by considering sequential circuits from the HWMCC10 benchmark suite, and by reducing the problem of disjunctively decomposing a circuit into components to the problem of generating Skolem function vectors. Details of how these benchmarks were generated are described in [1]. Each benchmark is of the form $\exists X F(X,Y)$, where $F(X,Y)$ is a conjunction of factors and $\exists Y (\exists X F(X,Y))$ is true. However, for some benchmarks, $\forall Y(\exists X F(X,Y))$ does not evaluate to true. Since Bloqqer can generate Skolem functions only when $\forall Y(\exists X F(X,Y))$ is true, we divided the benchmarks into two categories: a) TYPE-1 where $\forall Y \exists X F(X,Y)$ is true, and b) TYPE-2 where $\forall Y \exists X F(X,Y)$ is false (although $\exists Y \exists X F(X,Y)$ is true). While we ran CEGARSKOLEM and MONOSKOLEM on all benchmarks, we ran Bloqqer only on TYPE-1 benchmarks. Further, since Bloqqer required the input to be in `qdimacs` format, we converted each TYPE-1 benchmark into `qdimacs` format using Tseitin encoding [18]. All our benchmarks can be downloaded from [1].

Our implementations of MONOSKOLEM and CEGARSKOLEM make use of the ABC [11] library to represent and manipulate functions as AIGs. For CEGARSKOLEM, we used the default SAT solver provided by ABC, which is a variant of MiniSAT. We used a simple heuristic to order the variables, and used the same ordering for both MONOSKOLEM and CEGARSKOLEM. In our ordering, variables that occur in fewer factors are indexed lower than those that occur in more factors.

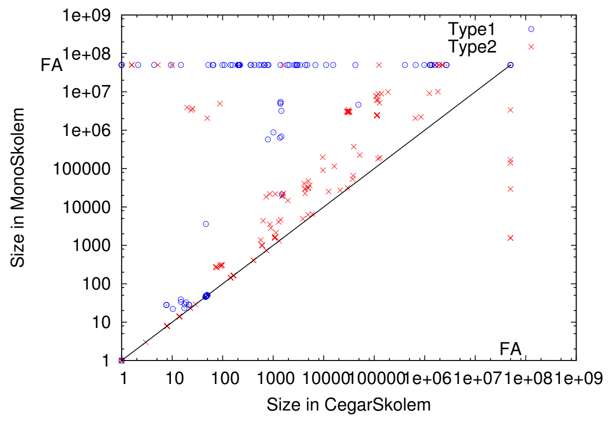
We used the following metrics to compare the performance of the algorithms: (i) average/maximum size of the generated Skolem functions in a Skolem function vector, where the size is the number of nodes in the AIG representation of a function, and ii) total time taken to generate the Skolem function vector (excluding any input format conversion time). The experiments were performed on a 1.87 GHz Intel(R) Xeon machine with 128GB memory running Ubuntu 12.04.4. The maximum time and main memory usage was restricted to 2 hours and 32GB, although we noticed that for most benchmarks, all three algorithms used less than 2 GB memory.

B. *Results and Discussion*

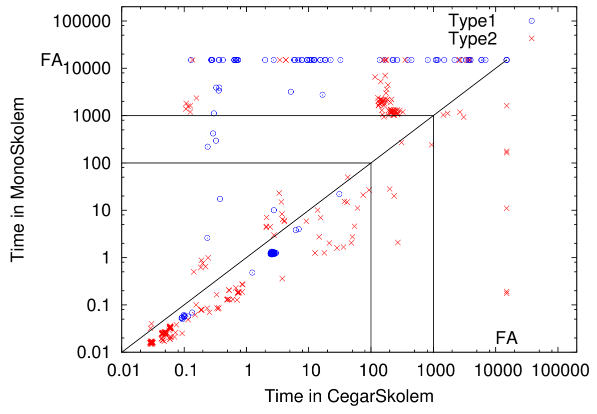
We conducted our experiments with 424 benchmarks, of which 160 were TYPE-1 benchmarks and 264 were TYPE-2 benchmarks. The 424 benchmarks covered a wide spectrum in terms of number of factors, total number of variables, and number of quantified variables (see [9] for details).

1) CEGARSKOLEM *vs* MONOSKOLEM*:* The performance of these two algorithms on all the benchmarks (TYPE-1 and TYPE-2) is shown in the scatter plots of Figure 1, where Figure 1a shows the average sizes of Skolem functions generated in a Skolem function vector and Figure 1b shows the total time taken in seconds. From Figure 1a, it is clear that the Skolem functions generated by CEGARSKOLEM in a Skolem function vector are on average *smaller* than those generated by MONOSKOLEM. *There is no instance on which* CEGARSKOLEM *generates Skolem function vectors with larger functions on average vis-a-vis* MONOSKOLEM.

Due to repeated calls to the SAT-solver, CEGARSKOLEM takes more time than MONOSKOLEM on some benchmarks, but on most of them the total time taken by both algorithms is *less than* 100 seconds (Figure 1b). Indeed, on profiling we found that CEGARSKOLEM spent most of its time on SAT solving. On 38 benchmarks where CEGARSKOLEM took greater than 100 but less than 300 seconds, MONOSKOLEM



(a) Average Skolem function sizes



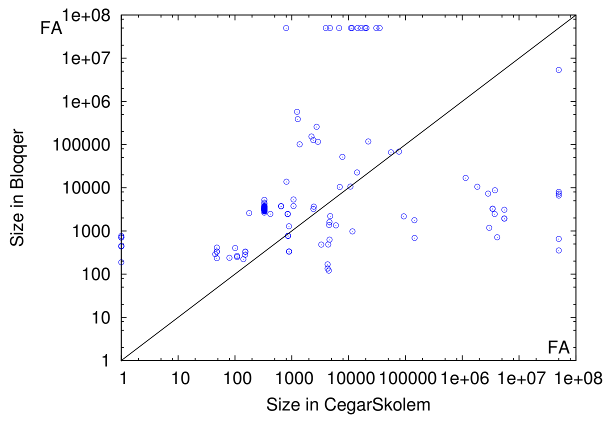
(b) Time taken (in seconds)

Fig. 1: CEGARSKOLEM vs MONOSKOLEM on TYPE-1 & TYPE-2 benchmarks. Topmost (rightmost) points indicate benchmarks where MONOSKOLEM (CEGARSKOLEM) was unsuccessful.

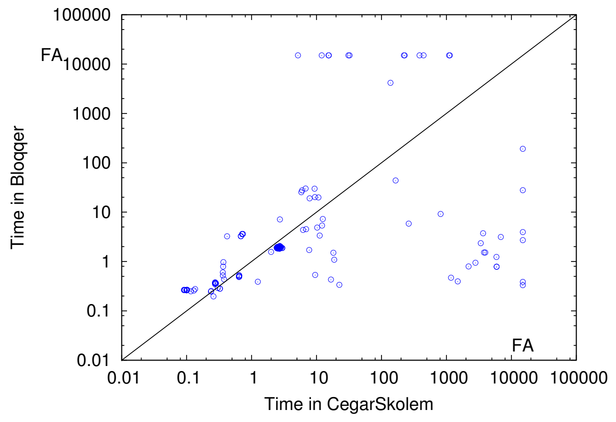
performed significantly worse, taking more than 1000 seconds. We found the degradation of MONOSKOLEM was due to the large sizes of Skolem functions generated (of the order of 1 million AIG nodes) compared to those generated by CEGARSKOLEM (< 8000 AIG nodes). *Large Skolem function sizes clearly imply more time spent in function composition and reverse-substitution.*

For benchmarks where the sizes of Skolem functions generated were even larger (of the order of 10^7 AIG nodes), MONOSKOLEM could not complete generation of all Skolem functions: for 8 benchmarks, the memory consumed by MONOSKOLEM increased rapidly, resulting in memory outs; for 10 benchmarks, it ran out of time; for an overwhelming 83 benchmarks, it encountered integer overflows (and hence assertion failures) in the underlying ABC library. These are indicated by the topmost points (see label "FA" on the axes) in Figure 1. In contrast, CEGARSKOLEM *generated Skolem functions for almost all* (412/424) *benchmarks*. The rightmost points indicate the 12 cases where CEGARSKOLEM failed, of which 10 were time-outs and 2 were memory outs.

2) CEGARSKOLEM *vs* *Bloqqer:* Of the 160 TYPE-1 benchmarks, Bloqqer successfully generated Skolem function



(a) Maximum size of Skolem functions



(b) Time taken (in seconds)

Fig. 2: CEGARSKOLEM vs Bloqqer on TYPE-1 benchmarks. Topmost (rightmost) points indicate benchmarks for which Bloqqer (CEGARSKOLEM) was unsuccessful.

vectors in 148 cases. It gave a `NOT VERIFIED` message for the remaining 12 benchmarks (in less than 30 minutes). These benchmarks are indicated by the topmost points (see label "FA" on the axes) in the scatter plots of Figure 2. Of these, 8 are large benchmarks with 1000+ factors and variables to eliminate (overall, there are 9 such large benchmarks). On the other hand, CEGARSKOLEM was able to successfully generate Skolem functions on 154 benchmarks, including the 9 large benchmarks, on each of which it took less than 20 minutes.

For the 142 benchmarks for which both algorithms succeeded, we compared the times taken in Figure 2b. As earlier, CEGARSKOLEM took more time on many benchmarks, but there were several benchmarks, including the large benchmarks, on which Bloqqer was out-performed. We also compared the maximum sizes of Skolem functions generated in a Skolem function vector (see Figure 2a). We used the maximum (instead of average) size, since Tseitin encoding was needed to convert the benchmarks to `qdimacs` format, and this introduces many variables whose Skolem function sizes are very small, skewing the average. For a majority (108/142) of the benchmarks where both algorithms succeeded, the maximum sizes of Skolem functions obtained by CEGARSKOLEM were *smaller* than those generated by Bloqqer. Hence, *not only does* CEGARSKOLEM *run faster on the large benchmarks, it also generates smaller Skolem functions on most of them.*

3) Discussion: For all benchmarks on which CEGARSKOLEM timed out, we noticed that there were large subsets of factors that shared many variables in their supports. As a result, CEGARSKOLEM could not exploit the factored representation effectively, requiring many refinements. We also noticed that for many benchmarks (197/424), the initial abstract Skolem functions were correct, and most of the time was spent in the SAT solver. In fact, on averaging over all benchmarks, we found that around 33% of the time spent by CEGARSKOLEM was for SAT-solving. This shows that we can leverage improvements in SAT solving technology to improve the performance of CEGARSKOLEM.

VI. CONCLUSION AND FUTURE WORK

We presented a CEGAR algorithm for generating Skolem functions from factored propositional formulas. Our experiments show that for complex functions, our algorithm outperforms two state-of-the-art algorithms. As part of future work, we will explore integration with more efficient SAT-solvers and refinement using multiple counter-examples.

REFERENCES

[1] A. John et al. Disjunctive Decomposition Benchmarks. http://www.cse.iitb.ac.in/~supratik/tools/fmcad_2015_experiments/.

[2] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic computational techniques for solving games. *STTT*, 7(2):118–128, 2005.

[3] Carlos Ansotegui, Carla P Gomes, and Bart Selman. The Achilles' heel of QBF. In *Proc. of AAAI*, volume 2, pages 275–281, 2005.

[4] Marco Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Proc. of CADE*, pages 369–376. Springer-Verlag, 2005.

[5] Christian Bessière and Guillaume Verger. Strategic constraint satisfaction problems. In *Proc. of CP*, pages 17–29, 2006.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003.

[7] J.-H. R. Jiang. Quantifier elimination via functional composition. In *Proc. of CAV*, pages 383–397. Springer, 2009.

[8] J.-H. R. Jiang and V Balabanov. Resolution proofs and Skolem functions in QBF evaluation and applications. In *Proc. of CAV*, pages 149–164. Springer, 2011.

[9] A. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay. Skolem functions for factored formulas. *CoRR*, Identifier: submit/1333056, 2015. (https://arxiv.org/submit/1333056).

[10] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. Wintersteiger. A First Step Towards a Unified Proof Checker for QBF. In *Proc. of SAT*, volume 4501 of *LNCS*, pages 201–214. Springer, 2007.

[11] Berkeley Logic and Verification Group. ABC: A System for Sequential Synthesis and Verification . http://www.eecs.berkeley.edu/~alanmi/abc/.

[12] Martina Seidl Marijn Heule and Armin Biere. Efficient Extraction of Skolem Functions from QRAT Proofs. In *Proc. of FMCAD*, 2014.

[13] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[14] Fabio Somenzi. Binary decision diagrams. In *Calculational System Design, vol. 173 of NATO Science Series F*, pages 303–366. IOS Press, 1999.

[15] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.

[16] D. Thomas, S. Chakraborty, and P.K. Pandya. Efficient guided symbolic reachability using reachability expressions. *STTT*, 10(2):113–129, 2008.

[17] A. Trivedi. Techniques in symbolic model checking. Master's thesis, Indian Institute of Technology Bombay, Mumbai, India, 2003.

[18] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics*, pages 115–125, 1968.